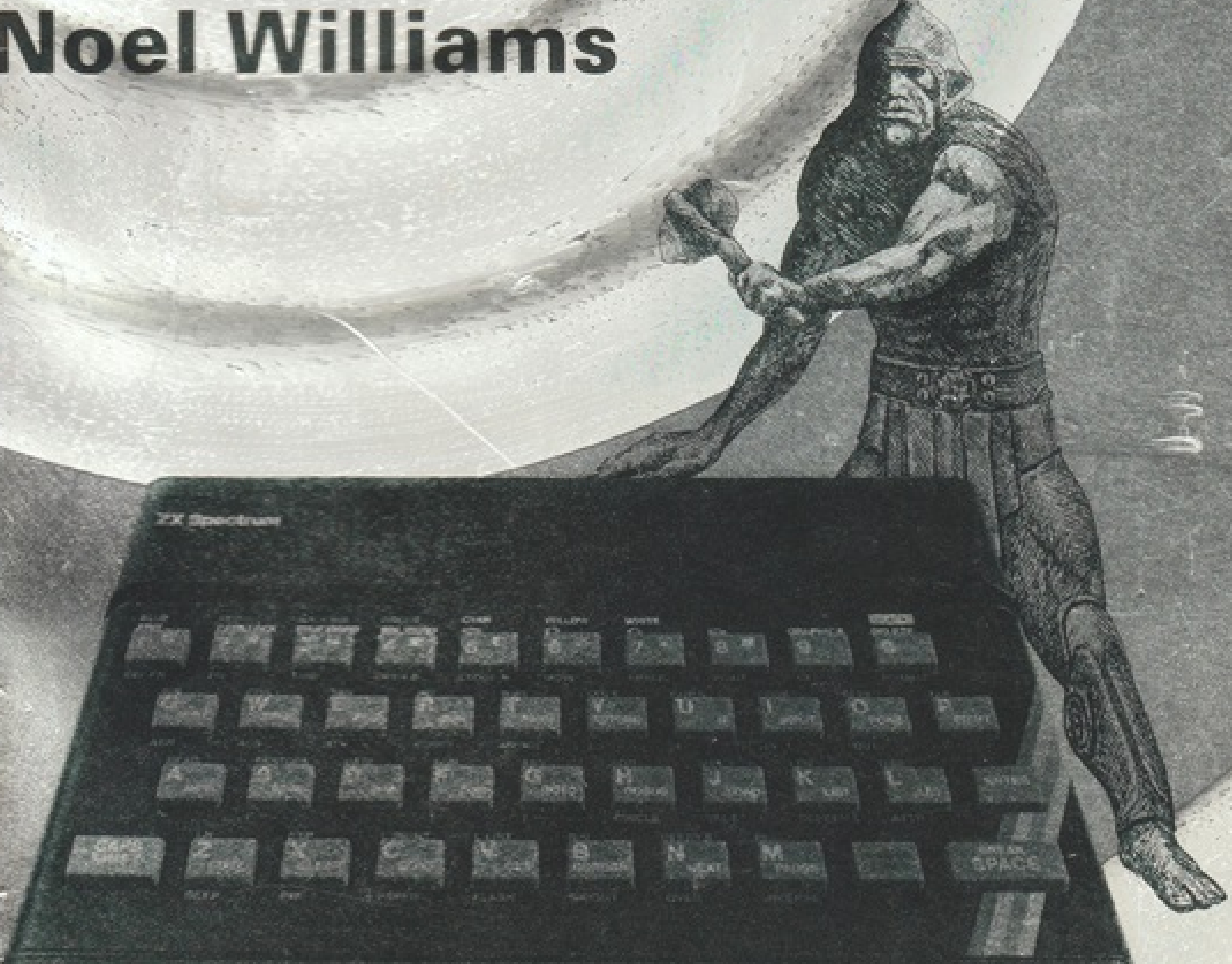
Invent and Write Games Programs for the Spectrum

Noel Williams



INVENT AND WRITE
GAMES PROGRAMS FOR THE SPECTRUM

Invent and Write Games Programs for the Spectrum

NOEL WILLIAMS

McGRAW-HILL Book Company (UK) Limited

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

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

British Library Cataloguing in Publication Data
Williams, Noel
Invent and write games programs
for the Spectrum.
1. Sinclair ZX Spectrum (Computer)–
Programming

I. Title
794 QA76.8.S625

ISBN 0-07-084719-3

Library of Congress Cataloging in Publication Data
Williams, Noel.
Invent and write games programs for the Spectrum.
1. Computer games. 2. Sinclair ZX Spectrum (Computer)–
Programming. I. Title.

GV1469.2.W54 1983 794.8'2 83-16265

ISBN 0-07-084719-3

1 2 3 4 5 CUP 86543

Printed in Great Britain at the University Press, Cambridge

For Carrol, for everything.

Contents

Preface

The aim of this book is simple— to bridge the gap between programming and games playing. This book tries to help games players become better programmers and to help programmers design better games.

The book goes through all the stages of designing and coding a game, from formulating the ideas to debugging. On the way it aims to help those new to programming to become familiar with many of the concepts which, though easy to understand, are usually beset by unfriendly jargon— terms like 'modular programming', 'random access', 'error checking', 'file handling', and 'artificial intelligence'. The terms are used as a games player might need them, not as an arcane code comprehensible only to computing graduates.

This book is for the novice programmer who can just about find his way around most of BASIC and now wants to know what to do with his knowledge— how to go about building a substantial program. Using the adventure game as its main theme and the Spectrum as its major micro it looks at all the ingredients of good games, including structure, display, originality, friendliness, and logic. Whether your micro is a Spectrum or not if you are interested in games you will find a great number of strategies and tips for programming games on any micro. Whether you are interested in adventure games or not you will find many of the principles discussed here are used in designing other kinds of game.

As you work through the book you will take part in designing two complete adventures, both of which are fully listed and documented, as well as over twenty-five game routines. Where useful features of the Spectrum can be exploited these are described, but care has been taken to detail some of the main differences between Spectrum and other micros in order to make adaptation of the routines to other systems easier.

If you own a Spectrum or you want to write an adventure this is the book you need. Processing verbal commands, constructing maps, designing puzzles, storing data, defining graphics, 'intelligent' programs, character and monster variables, plot construction, combat routines, idea generation, program construction, controlling movement, and many specific ideas for adventures are all included, together with many routines in Spectrum BASIC for incorporation into your own games. In particular, a number of the features of Spectrum BASIC which are not well documented elsewhere are used in this book to develop your knowledge of the language.

My grateful thanks are due to Carrol, Kay, and Owen who helped in the preparation of this book in their own special ways, and to Mike Costello of The War Machine for permission to reproduce the Name Generator of Chapter 7.

1 AN INTRODUCTION TO ADVENTURES

You are in a green tunnel with wet walls. On the ground you see:

a scroll
an ebony box

What do you do?

* PICK UP THE BOX

Suddenly a wart-encrusted troll leaps out from a crack in the wall. 'Steal my emeralds would you?', he screams and raises a thick bronze club. Do you want to:

1. RUN
2. STAND AND FIGHT
3. TALK YOURSELF OUT OF IT
4. SMILE SWEETLY

* SMILE SWEETLY

You smile sweetly. The troll smiles back, as his club bounces off your head.

This is an encounter you might find in any adventure game. The adventure could be a board game, a tabletop game with miniatures, a pencil and paper game, or a microgame. Adventure games come in many shapes and sizes and are played in many ways. Almost every micro has at least one adventure written for it, and the mainframes of many a college and business have among their most-used software Zork or Wumpus or just plain Adventure.

This book tells you how to write your own adventures. Because there are many kinds of adventure game and many different micros it cannot cover every aspect, but it does describe ways of designing and programming all the essential features (and some non-essential) in a way that every micro user should be able to use for his own particular machine. Where different approaches are possible, different routines are suggested, so a great variety of games is possible within the broad heading of 'adventure'. Most of the routines are written in one of the most popular BASICs, Sinclair Spectrum BASIC, but where there are important differences between this and other versions of the language, alternatives are given that can be used, especially for Microsoft and BBC BASIC. A cassette of all the major Spectrum routines listed in the book is available separately.

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

So what is an adventure game? The only simple answer to this question is that it is many things, because there are many, many variations. Some are fantasy, some are science fiction, some historical, and some based on novels; some use graphics and some of these are 3D; some are real-time (i.e., like arcade games you have to react to the game immediately— you cannot leave it running while you make a cup of coffee because when you get back you will have lost); some use animation; some involve mazes; some involve problem-solving; some offer real money prizes; some require huge databases while some fit into less than 8K of memory. So a general definition of an adventure game is not going to be worth a lot. However, there are some elements fundamental to all adventure games and we will look at those first. After a general survey we will use these elements as a way first of designing and then of programming several adventure games.

Broadly speaking, an adventure game is a simulation of a particular world. Playing the game means behaving as if you were living in that world, and the sequence of actions that you make in playing would read like a story if written down. Your average adventure game thus has three basic elements: simulation, player behaviour, and story. In the example that began this chapter the simulation is a fantasy world like Tolkien's Middle Earth; the player's behaviour includes picking up the box and smiling sweetly; and the story is 'how I persuaded a troll to part with his emeralds and nearly lived to tell the tale'.

The development of adventure games, though it has happened quite rapidly, has been quite complex. You do not need to know anything about where they have come from to be able to write them, but it is worth knowing that the three main strands have come from several different sources which have combined in many different ways. Almost certainly you can learn something about writing adventure games by knowing more about those sources, but it is equally certain that you will be able to bring something original to the games because of your own backgrounds and interests. One of the main reasons for the flexibility and variety of adventure games is the range of backgrounds of the people who have developed them.

The main sources of adventure games have interrelated inextricably, but four important ones can be identified. Firstly, the rise of the popular fantasy novel, due to authors such as Robert E. Howard (the originator of Conan) and J.R.R. Tolkien, has meant that more and more people are interested in fantasy stories. Secondly, in the late 'sixties war games became a popular hobby, and one type of war game in particular, the fantasy campaign, seems independently to have interested several different players in different places. The most important of these was the game played by Gary Gygax, which gradually became the enormously popular fantasy role-playing game, Dungeons and Dragons. The success of this game has led to dozens of other such games, some set in the Wild West, some on post holocaust Earth, some among eighteenth century pirates, some in outer space, even one called Bunnies and Burrows which is based on *Watership Down*. All of these games can be sources of inspiration for microgames as we shall see in Chapter 2.

Thirdly, at about the same time that fantasy novels and war games were becoming popular important research in computational linguistics (the study of language by using computers) and artificial intelligence (learning about human intelligence by trying to make machines intelligent) was taking place, notably at the Massachusetts Institute of Technology. Here programs were used which analysed and acted on ordinary, natural language; programs investigated the nature of puzzle-solving and problem-solving; and programs were used to manipulate fictitious worlds.

Finally, in the fields of business and education people have for many years used games and simulations for teaching about the real world. It is not possible for students to learn about banking or the Norman Conquest at first hand, but they can learn by taking part in a simulation which makes them act as if it were at first hand. Microcomputers are excellent for such simulations, and simple educational games such as Kingdom (in which the player learns elementary economics) have become popular outside the classroom.

We will look at all of these in later chapters. Fantasy novels can give us ideas for stories; tabletop and board games can suggest game structures and mechanisms; linguistics and artificial intelligence can help in text processing and in making games both more interesting and more life-like; and 'serious' simulations can tell us something about how and why players play. But the main purpose of this book is to help you in game programming. We will look at the different components of the adventure game, such as maps, monsters, and magic, and how these can be programmed. We will look at the advantages and disadvantages of different techniques, such as various ways of storing data. We will look at some of the more common jargon words such as 'random access' and 'menus' to explain them in a straightforward context. We will look at ways of designing games and how those designs can be turned into BASIC. We will also look at ways the microcomputer can be used in assisting adventure games which are not played on a micro. We will do all this by writing our own adventure games as we go along.

The book is organized in the following way. We begin by looking at the different components in the game and different kinds of adventure game. Then comes the main part of the book— seven chapters of programming and design techniques for use on such games, beginning with overall design and working through all the major components. Each section gives one or more routines you can incorporate in your own programs until gradually we build up two complete games, The Throne of Camelot and The Mines of Merlin. Between them these two games illustrate most of the essential principles discussed in each chapter, as well as containing many routines which can be used in other adventures.

2 COMPONENTS OF THE ADVENTURE GAME

2.1 Choice

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

If we want to write a good game, therefore, we must do three things:

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

We will look at the programming implications of these later, but the simple lesson is that the game should play in a straightforward manner although that apparent simplicity should depend on complex hidden balances in the programming.

Adventure games can depend on many kinds of choice. Initially the player may be able to choose the type of character he plays before the game starts, or the configuration of the game (such as its level of difficulty or the number of events it might have), various actions in different situations (e.g., running from, talking to, or fighting an encountered dragon), when to use limited resources, or even when to end the game (as with the SAVE facility incorporated in many puzzle games).

To be a little pedantic for a moment, we could call a player's strategy a series of choices aimed at optimizing rewards in a game. Thus the choices that make a game interesting depend on the kind of rewards the game offers. At the most abstract level a game offers points (as with The Mines of Merlin in Chapter 6). The better the player, the higher his score, i.e., the better his strategy, the series of choices he makes, the more points he will get.

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

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

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

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

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

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

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

So the typical adventure has a series of hurdles to be overcome in order to achieve the final solution. Each hurdle jumped will act as a local reward and temporarily increase REWARD. After a while that temporary increase will be lost, but there will still be an overall increase because the player will know that now he is much nearer to 'the answer'. These hurdles may be problems to solve, objects to find, mazes to get through, monsters to defeat, riddles to answer, and so forth. A simple problem like 'How do I open the door?' can provide enough reward to keep a player going for several hours if he knows that 'the answer' is on the other side.

Global rewards do, however, come under four headings: solutions, points, treasure, and increased abilities. Solutions allow puzzles, problems, riddles, and the like, to be solved. These provide the basis of one main type of adventure game, which I shall call the puzzle game. Essentially a puzzle game is a web of interrelated puzzles making up one total puzzle. In order to find the overall solution each of the separate problems must be solved and in the correct order. At any stage in the game, therefore, Cad will have two objectives: (1) to solve the particular local problem that faces him at the moment and (2) to solve the overall problem. Obviously anything which helps with the former will count as a local reward for him and increase his enjoyment temporarily, whereas anything contributing to the latter will give a permanent increase in REWARD. Winning in this kind of game is like solving Rubik's Cube. There is pleasure in getting each individual coloured square in place, but that is nothing like the overwhelming smugness that comes from being able to put the lot together, every time.

We have already discussed points. Treasure and abilities belong to the second class of adventure game, the combat game. Here the overall task may be very much the same as in the puzzle game— e.g., the finding of a hidden amulet— but the method of achieving it is very different. Instead of a series of problem-solving sessions the player acts like an adventurer in a 'swords and sorcery' novel, meeting beasts, felling them, and then taking any treasure they may have. Accumulating treasure is thus like increasing points— it is an abstract kind of reward. However, it becomes global if treasure can then be used for other purposes, such as buying equipment which helps in other combat or searches, bribing monsters, or magically increasing abilities. In some games the task cannot be achieved until the player's character has developed to a certain level of ability (often called the experience level, after the concept used in Dungeons and Dragons). He may achieve this by success in combat, just as in real life people get better at things by practising them successfully, or he may achieve it by receiving magical increases through particular treasures found.

The interrelations between treasures, ability, points, and some other variables will become clearer in Chapter 6 when we design a combat game. Chapter 5 involves the design of a puzzle game when we shall see how local and global rewards can fit together. The problem we find throughout the book, however, is that the two types of game are difficult to combine. Almost all adventure games are either puzzle games or combat games, and there are hardly any which attempt to combine the two approaches. This is a result partly of memory constraints on micros but mainly because programmers have not been willing to come to grips with the intricacies of mixing the two. Hopefully by the time you have read this book you will understand enough about the workings of both kinds to be able to develop the first of a new kind of adventure— mixing both puzzle and combat.

In the rest of this chapter we will look at some of the fundamental elements of an adventure in outline, before going on to detailed program design and coding in subsequent chapters. You will see from the above discussion, however, that the basic elements in the game are a player persona or character of some kind, a story line made up of a series of choices, and a number of hazards.

2.2 Stories

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

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

An event in a story usually fits the following formula:

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

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

The links between mapped locations may be totally random or totally structured or a combination of both. Similarly with events. The advantage of random determination is that it is easy to program and can give great variety. Its main disadvantage is that a random game rapidly becomes boring because it consists of a series of unpredictable events with no logical relationship, i.e., a series of purely local rewards. A decision made in one location will have no effect on subsequent actions unless the player's character is altered in some way in that location— becoming weaker, perhaps, or finding a laser sword. Adding this kind of alteration to a basically random game is quite easy to do as we shall discover in Chapter 6. It provides a simple way of adding some structure to our game. However, if no such alteration occurs in any location the events are unconnected and we really have a series of small games rather than one large one.

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

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

Random numbers A, B, and C decide whether there should be a monster, object, and action in a particular room; random numbers D, E, and F select the appropriate monster, object, and action if there is one. These would be set up in the program using arrays or subroutines from which random items could be called. Each randomly chosen item would then be a set of variables which potentially modifies the character in some way. Sample monsters, objects, and actions are outlined in other chapters.

The opposite approach, that of a totally structured series of events, is typical of the puzzle game. Here the task is not to survive as many randomly generated events as possible but to discover the puzzle or story and pass through each of the planned events in the correct order. For example, Cad may find that he cannot progress in the game until he has discovered how to open a rusty trapdoor. To discover this, he must bribe a goblin, which means he has to find some money. To obtain the money he must first get past the headless ghost, and so on.

The advantage of this type of game is that it is a real test of the player's abilities, his intelligence, logical power, and imagination, not merely a test of reactions or responses to a random series of accidents. The puzzle adventure game has been likened to the crossword— it demands the same class of skills, including language skills, and all parts must be solved to complete the whole.

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

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

2.3 The player character

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

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

Speed 4
Defence 5
Attack 4

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

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

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

```
8000 REM Set up Sir Jon
8010 DIM A(4)
8020 FOR I=1 TO 4
8030 READ A(I)
8040 NEXT I
9000 REM Sir Jon's data
9010 DATA 2,2,1,1
```

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

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

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

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

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

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

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

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

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

However, we can also add a more abstract points reward for those players who like such things. We can invent an overall score dependent on how well the character is doing. In the case of Sir Jon it seems important that a knight should behave as honourably as possible, so we will give him the honour points depending on how well he does in particular situations. If we make this scale 1 to 100 and record this as A(6) (remember to change those statements), we can also incorporate this rather abstract score into the test for knighthood. Suppose that to be admitted to the Round Table a knight must be very honourable, with an honour's total of over 90. We can add the proviso that this honour is worth more if he has not used his wishes to aid him— he has done it under his own steam without supernatural aid— so that for every wish he has left he scores five honour points. This means that total honour points will be those normally added to A(6) plus 5 * A (5). Our test for knighthood has now become a short routine:

```
7799 REM KNIGHTHOOD TEST
78ØØ LET HP = Ø: REM CLEAR ANY PREVIOUS VALUE
781Ø LET HP = (A(5)*5)+A(6)
782Ø IF A(1) >= 8 AND A(2)> = 8 AND HP>9Ø THEN LET
ROUNDTABLE = 1
783Ø RETURN
```

Line 781Ø has more brackets than some BASICs might think necessary. Different versions of the language have different priorities for evaluating expressions so it is best to keep the expression explicit.

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

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

An array could be used to hold this information just as in the combat game. Each variable in the array will be used as a flag to represent a particular object. If the correct flag is set to one the character possesses that object; if set to zero he or she does not. However, you will see in The Throne of Camelot, when we build it in Chapter 5, that these flags can be used for other purposes to indicate various states of the object in question. After all, any particular variable in a program uses at least one byte of memory, not just one bit. A bit of information is effectively a flag which can only be set to one or zero, i.e., it has only two possible states. In other words, it is binary, and bit stands for Binary InTeger.

However, a byte can have 256 different states, which is why many aspects of BASIC are limited to 256. If you look at the Spectrum character set in Appendix A of the manual you will see that this consists of 256 codes. A byte is made up of eight bits which allows coding of numbers up to 256 using the binary system of counting. Consequently, any variable which can be set to one in a BASIC program can also be set to 255 (the 256th state being zero, which is also a number). So if we use such flags in a puzzle game, we could use a system like the following:

0 means the object is hidden.
1 means the object can be seen.
2 means the player has the object.
3 means he has used it correctly.
4 means he has put it in the correct place.
And so on up to 255.

The system of flags used in The Throne of Camelot is similar to this and is explained in Chapter 5.

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

2.4 Monsters

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

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

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

1. A monster should only have variables and values which are meaningful in terms of the rest of the game. This usually means that those variables are related to the player–character variables, but, in the case of an 'intelligent' monster which can act in the program independently of the character, other variables will be needed.
2. The character should seldom encounter monsters which are extremely powerful, comparatively speaking, unless as the result of a major mistake (otherwise the game becomes 'sudden death').
3. No monster should be invincible, nor too easy to defeat! Actual monsters encountered should either be in a range of powers, all of which can be overcome by the character, or should have their powers related to the current powers of the character.
4. Each monster should be different, not simply by virtue of the magnitude of its variables but also in terms of its overall configuration, i.e., its name, its behaviour, the kind of problem it presents, and consequently the choices or strategies needed for its defeat.

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

Let us work on the random monster, as this is the usual type in the majority of adventures and is the easiest to design. Intelligent monsters will be discussed in Chapter 8.

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

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

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

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

```
1Ø DIM M(3,3)
2Ø FOR I = 1 TO 3
3Ø FOR J = 1 TO 3
4Ø READ M(I,J)
5Ø NEXT J
6Ø NEXT I
7Ø REM BUGBLATTER BEAST
8Ø DATA 1,2,3
9Ø REM OOZLER
1ØØ DATA 2,3,4
```

```
11Ø REM GIANT TURNIP
12Ø DATA 1,3,2
```

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

Arrays are very useful for this type of procedure, a process usually known as random access because any random item randomly selected can be accessed as easily as any other. However, arrays use memory and if there are a large number of monsters in the game an array to hold them all would use a great deal of memory, which would be particularly wasteful if the values were only in a small range, such as the 1 to 4 range above. The array M will use at least 20 bytes on most systems, but nine numbers in the range 1 to 4 can be stored in only five bytes, and could actually be crammed into less.

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

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

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

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

```
DEF FN K$(M$,M,I) = M$(((M–1) *3) +I)
```

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

```
LET J$=FN K$(,2,3)
```

But we would then have to go through the process of turning the character we have just extracted into a number by using VAL, so we might as well make the function more complex in the first place:

```
DEF FN K(M,I)=VAL(M$,((M–1) *3) +I)
```

and use it by writing:

```
LET J=FN K(2,3)
```

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

```
LET M(2,3) = x
```

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

```
DEF FN A$(M$,M,I) = M$(1 TO ((M–1) *3) +I–1) + STR$(X) +
M$(((M–1) *3) +I +1 TO LEN(M$))
```

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

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

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

In order for the functions to work properly the string must stay the same length and the data cannot move position. This means that the original string has to be the maximum length needed by the program and all positions in it must be filled. Thus, if the values of particular items are likely to change from 1 to 20, each slot in the string has to be treated as a two-position slot from the beginning. This is true even if only one item in the string will be two characters long. So our example string would now be: "Ø1Ø2Ø3Ø2Ø3Ø4Ø1Ø3Ø2" and all functions would have to operate on items two characters long rather than one.

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

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

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

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

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

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

```
1Ø1Ø DATA 2,3,4
```

the following short routine finds the attack value:

```
5Ø RESTORE 1Ø1Ø
6Ø FOR I = 1 TO 3
7Ø READ A
8Ø NEXT I
9Ø PRINT "OOZLER'S ATTACK VALUE IS "; A
```

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

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

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

You will find in Chapter 10 a short routine which should stimulate your imagination in this direction. What is important to remember, however, in turning any of these ideas into code is that a balance must be achieved between the amount of program (memory, time, coding) that is required and the effect of the monster on the game. If half the program is used simply to generate one clever monster, found in room 100, the player is unlikely ever to appreciate the intricacies of your design.

2.5 Objects

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

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

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

In terms of the program structure, having objects is one of the simplest ways of adding complexity and variety to our game. After all, a small game of 10 locations and 10 objects could give 100 possible events in the game; a large game of 1,000 locations and 100 objects can give 100,000 events, which is probably more than even the most dedicated Cad would care to solve.

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

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

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

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

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

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

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

3 TECHNIQUES IN PLANNING THE GAME

3.1 Structures

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

The elements will vary slightly according to the particular machine used (e.g., whether it has colour or high-resolution graphics), but at the most general level all adventure games have each of the following: some form of input/output, made up of a textual component (what is written on the visual display unit, such as a description of a room and its contents), a visual component (perhaps a graphical picture of that room or a flashing warning), a sound component (magical explosions, beeps, and burps), together with three structures— the game structure (what the rules of play are and how they are to be carried out), the story or puzzle structures (what the player is meant to be doing in the game), and the program structure (how each of the above elements fits together in a way that a computer can understand). This basic model of a game is shown in Fig. 3.1.

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

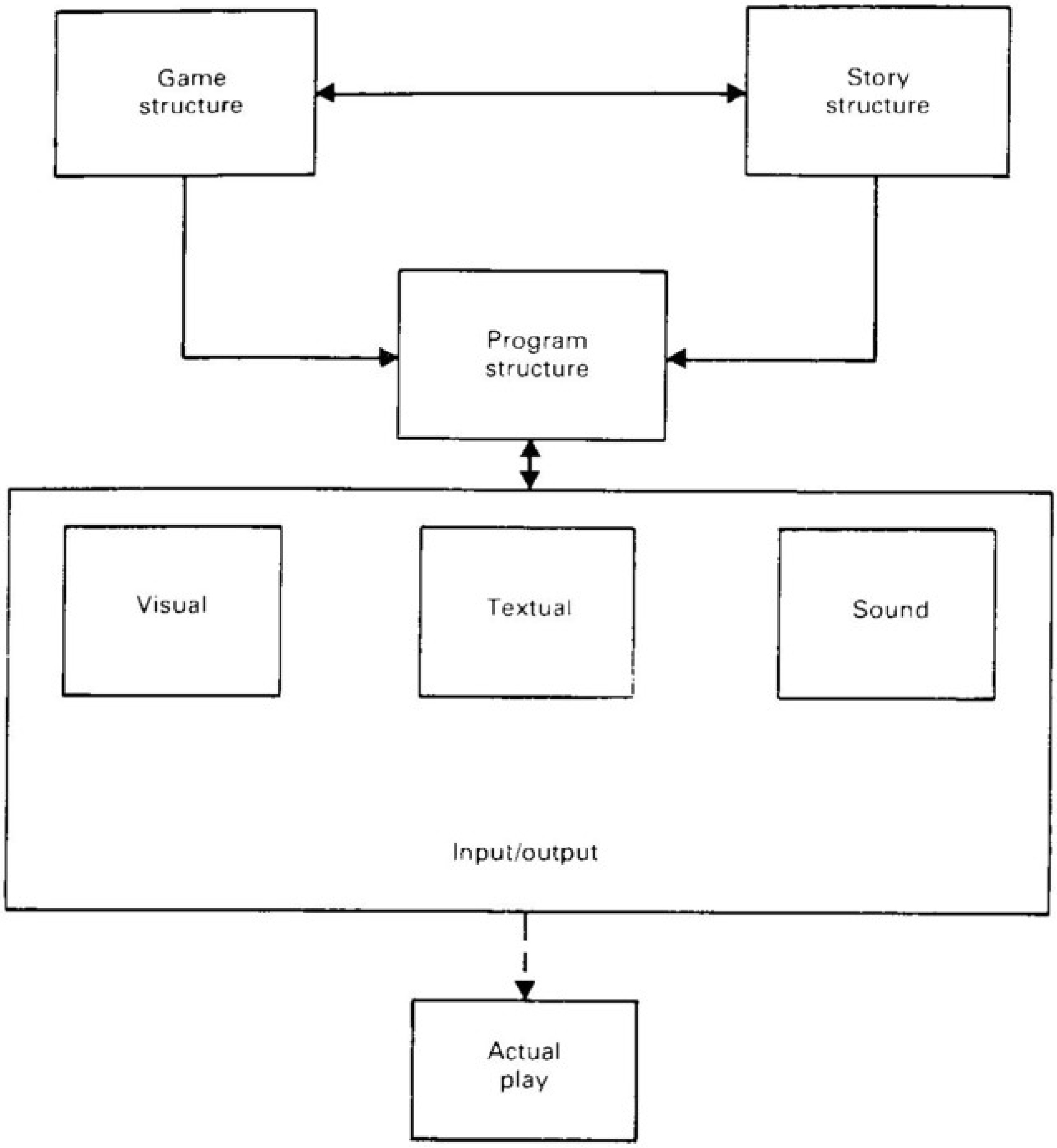


Figure 3.1

strategies that may be used, but first we will make a brief examination of each of the boxes in Fig. 3.1 in order to find the kind of content they may have and the decisions it is necessary to make in filling them.

3.2 Story structure

Many adventure games begin with an idea for a particular story line or scenario. Aha! you think, wouldn't it be a good idea to have a game about climbing down the inside of a volcano to the centre of the earth? How about a game in which players sail the Kontiki across the Pacific? You might wish there was a game where you could travel the universe to piece together the history of an ancient race. However, having had your initial flash of inspiration, you may not know how to turn that into a description which can be coded for the micro. After all, many people have tried writing stories but there are only a few Flemings and Macleans.

You do not need to be a great novelist to write a story, scenario, or puzzle that can become the basis of a program. All you need to remember are two things:

1. A story is a series of linked events in which a character (or group of characters) is changed in some way.
2. An event is made up of a place, a particular time, one or more characters, one or more objects, and some possible actions or consequences.

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

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

To illustrate how we can do this, here is the framework for a simple story.

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

Here are all the elements of our story. We must then decide how to string them together and how to fit them with the six other basic program areas. For example, what are the chances of encountering settlers in the forest? Will there be graphic illustration of any combats? How will the tribe move from place to place?

3.3 Game structure

All games have essentially the same structure— the player makes a move; the consequences of the move are calculated; if the player has won or lost then the game ends; otherwise that player or another player makes another move. In the case of an adventure game there are seldom moves in the traditional sense of, say, a move in Chess. What happens instead is that the player is periodically presented with a series of choices which represent the kinds of actions his character(s) could take in the type of world represented in the story. The choice made is the 'move' and its consequences will be some effect on the character (he is poisoned) or on the world around him (he breaks down the door) or on both (the starship explodes, hurling the character into space).

This can be represented by a simple flowchart, as in Fig. 3.2.

1. Character options are printed.
2. Player chooses an option.
3. The effect of the choice is calculated.
4. The character is modified if necessary.
5. The game world is modified if necessary.
6. If the game is over go to number 7; otherwise go to number 1.
7. End the game.

Figure 3.2

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

1. ATTACK
2. HYPERSPACE
3. RESEARCH

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

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

Such increasing specificity can be continued indefinitely, up to the capacity of the micro used. Many micros only allow a limited number of embedded subroutines which would restrict this kind of nested specificity.

A second game variation involves the effect which the choice has on the player rather than on the character or the world. For example, a request for further instructions or to print the character's current status is a part of the game structure which has no effect on the game. However, it would be perfectly possible for such choices to affect the game. For example, every time a player asks for instructions the intelligence variable of his character could be reduced. This kind of modification makes playing the game more skilful, and that is a primary consideration in designing a game. An enjoyable game is one that is difficult enough to be taxing yet easy enough to understand. Abstract games such as Othello and Chess are very popular for just this reason— they are simple to learn but difficult to play well. Similarly, the game structure of an adventure should make play easy but good play difficult.

This is why there are two broad categories of adventure game— the puzzle game and the combat game. The former builds the game structure around a puzzle or series of puzzles, each of which must be solved in order to win the game. Here intelligence and imagination are tested. The latter type of game is closer to the arcade game or to the table top war game. The game structure depends on the tactics and strategy of combat between the character and monster, with decisions having to be made rapidly, and the consequences such as injury having an effect on subsequent battles. This type of game tests reactions and tactical planning. The ideal adventure game should combine both these approaches, involving player choices which are intellectually challenging in as broad a way as possible, as well as the challenge of quick reflexes and the traditional strategic and tactical skills of board games.

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

In particular, in designing a game structure we should decide on how each option will function. No choice in an adventure game should automatically result in success. Either the choice should bring success only if made together with other correct choices (e.g., you must choose the black bow but only the white arrow) or there should only be a chance of success represented by a function curve of its probability. Such curves are at the heart of many games. For

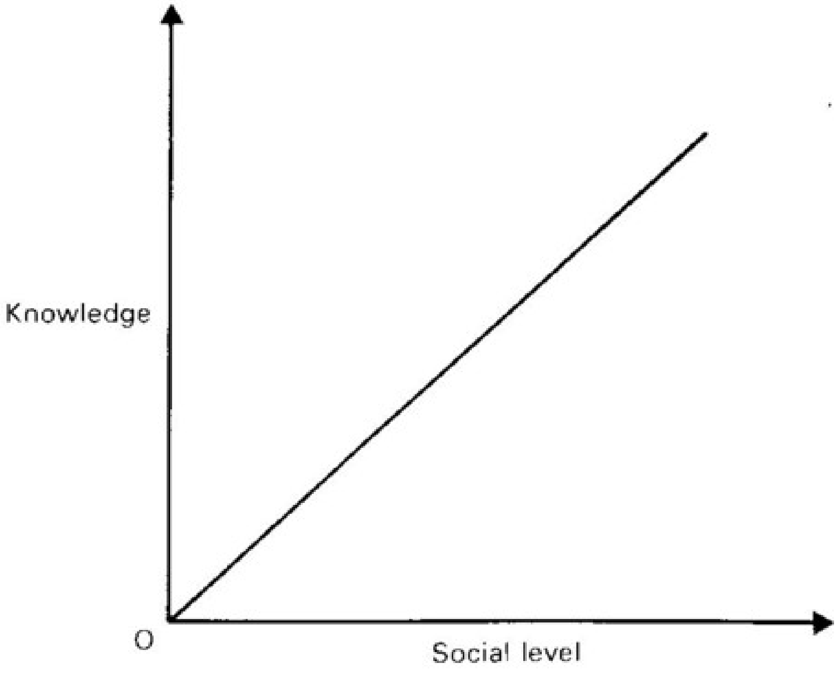


Figure 3.3

example, suppose our game involved the character moving up the social scale from peasant to king and each turn represented a year in his life. As he goes higher up the social ladder his knowledge increases. We could represent this by the curve in Fig. 3.3. However, as he gets older his intelligence decreases, as in Fig. 3.4.

Suppose his chance of passing to the next social level depends equally on knowledge and intelligence. Then the curve for age against chance of increasing social level would be Fig. 3.5.

As increase in age is uniform (one year each turn) but social level is variable so that keeping your chance at 50 per cent depends on previously being successful (which initially you will be only 50 per cent of the time), the actual curve of chance of success in increasing social level as the game progresses will look more like Fig. 3.6, i.e., initially age and social level cancel each other out in determining the chance of increasing social level, but as time goes on age becomes more and more important.

Every probability in a game can be given a curve like this, and the chance of winning the game will depend on the curve which summarizes the combination of all the curves. This can involve some very complex mathematics. However, it is not necessary to go so far as to compute all the curves of all the functions in our game, especially if luck (the random factor) plays a large part. What we have to ensure is that we choose functions, formulae, and algorithms which (1) make sense in the world we are creating and (2) never produce 100 per cent certainties. We must work out the consequences of a few sample

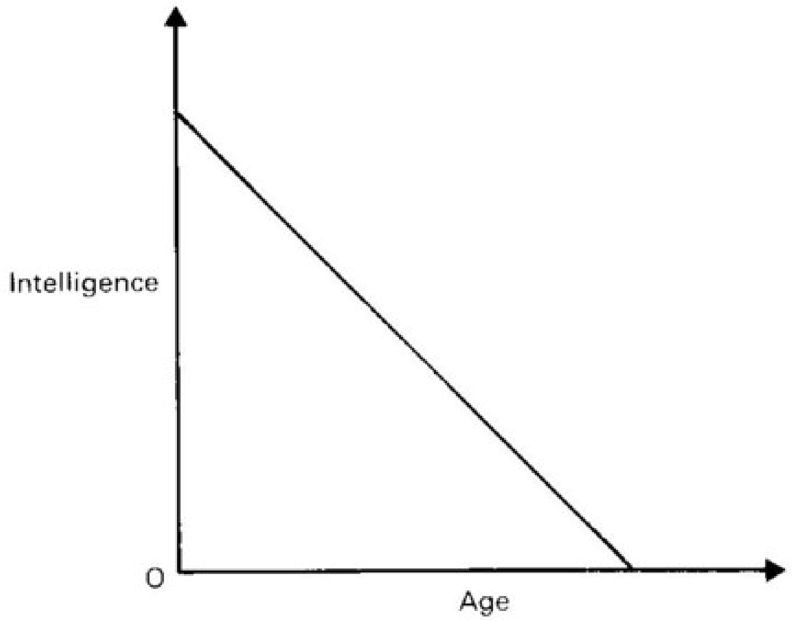


Figure 3.4

choices and play-test the most frequently used functions in our game before we finally decide to incorporate them into the game. It is always possible to adjust the formulae later on, but it is much better to have a balanced structure worked out from the beginning so that we do not have to resort to tinkering which could upset the whole game.

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

3.4 Input/output

SOUND

Many micros have no sound facility and others have very poor ones, so you may not wish to incorporate sound into your game. It is certainly less worth while for adventure than for arcade games, so is by no means essential. However, some micros have excellent sound facilities and there are certain uses of sound which can enhance an adventure game (see Chapter 8), so if we decide to use sound we should do so early on in the design process to maximize effective use, as with every other feature of the game.

The main decision we must make initially is whether sound is to be an integral part of the game, giving information which is not available in any other way (such as using musical clues as part of the puzzle), or simply an enhancement containing no essential features (e.g., a tune played at the beginning and end of each game). Naturally

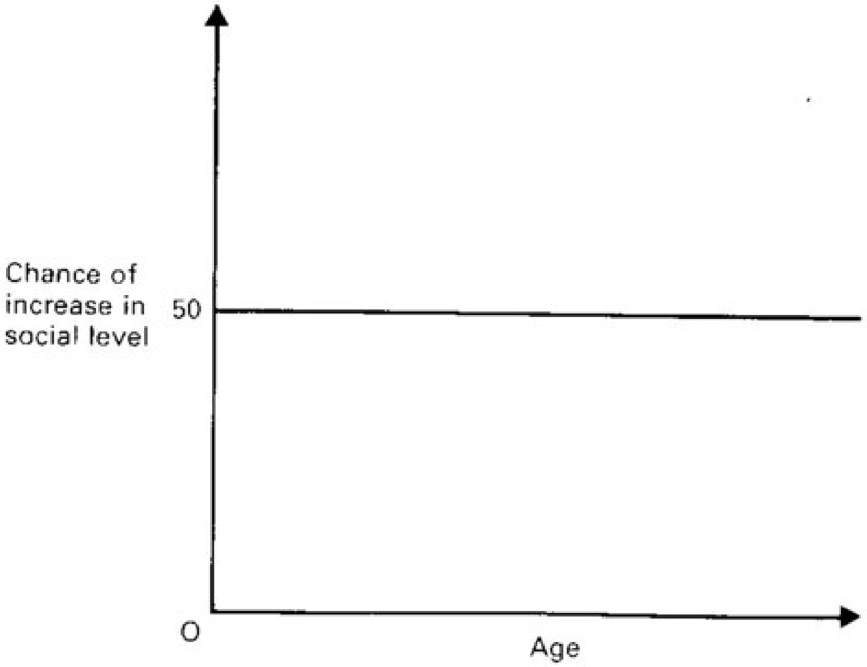


Figure 3.5

sound can be used in both ways in one program. In the case of non-essential sound it can always be added during the final stages of coding, when we know how much memory there is to play with and which sections of the game need such enhancement. However, essential sounds must be built into the design as early as all the other essential information so that we can see how to fit it into the overall structure and be able to develop aural effects parallel to the rest of the program rather than tagged on superficially.

VISUAL

Unless your adventure only gives printed hard copy (as in play-by-mail games, discussed in Chapter 11) there will always be a visual component. The information for each turn will be displayed on the visual display unit (VDU). It may be entirely graphical, in which case it will be an arcade-type adventure, or it may be entirely textual. With the current generation of microcomputers it seems rather wasteful to have no graphical output, but on the other hand graphics can use a large amount of memory and adventures generally require as much memory as possible for their logic. The traditional adventure game is entirely textual, but there is an increasing demand for games which use real-time graphics, colour, high resolution, three-dimensional illustration, etc. As memory becomes cheaper such developments will become more practicable.

It is a good idea, therefore, to make your first adventure mainly textual as this simplifies matters, but to set aside one or two kilobytes

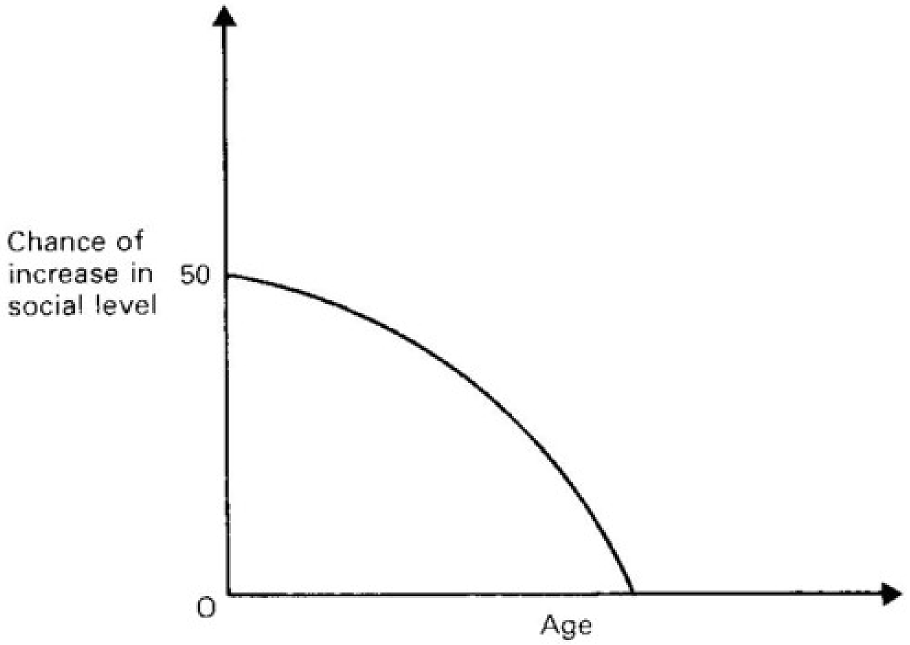


Figure 3.6

of memory for experimentation with graphical enhancement, such as in titles or warnings.

If, on the other hand, we want to illustrate every room and every monster, or to use some form of animation, then we will need much more memory, which will limit the size and scope of the program. There are ways of using files on discs or cassettes which allow storage of graphics outside the random access memory (RAM), thereby giving the best of both worlds (but slowing down execution of the program). For a discussion of this see Chapter 11.

For the moment we need to decide:

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

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

1. Decorative display, as in titles, flashing warnings, decorated text, etc.
2. Illustration, as in a picture of the monster the character is about to grapple or the view seen upon entering a new location
3. Essential information, where the graphics hold information which is essential to the game and not given in any other way, such as when visual clues are used or an arcade encounter is incorporated.

The first of these is easiest to do and can greatly increase the attractiveness of a game, but strictly speaking is unnecessary and so is often neglected. The third is most interesting, but also most difficult, so we will concentrate largely on the second. We should always attempt to use a micro to the limits of its resources. One way to do this is by a clever mixture of sound, graphics, and text which is not merely illustrative and could not be emulated in a board game.

Unfortunately, because of the many different implementations of graphics, it is not possible to go into great detail about how a particular design could be implemented on a number of systems. Later chapters will outline ideas which can be adapted for your particular machine, but most of the graphics in this book use the Spectrum system. Fortunately this has some similarities with other popular systems.

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

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

TEXT

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

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

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

1. Use anything which saves memory.
2. Use anything which speeds execution.

3. Use anything which adds variety to the game.
4. Use anything which makes the program easier to play and more enjoyable for the player.

Because of its fundamental importance it is worth considering text at length in the planning stage. Text processing is often neglected by BASIC programmers despite the fact that many interesting procedures can be carried out. On the one hand we must consider ways of making the game more attractive to the player, in terms of textual variety, correct spelling, easily read displays, etc. On the other hand we must see if we can come up with any original ideas for using text, such as making a pun on the player's name, using jokes in response to player mistakes, displaying text as fragments of parchment found at different places in the game, using cypher routines which change cypher from game to game, generating a series of verbal clues, routines for 'conversation' with encountered monsters, etc. Some of these ideas are developed in Chapters 7 and 8.

3.5 Program structure

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

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

What is meant by modular design? Essentially it is the same as dividing your program up into a number of subroutines, plus a main program which calls each subroutine as it is required. In BASIC a very simple program might look something like Fig. 3.7.

```
1Ø  GOSUB 1ØØØ: REM INSTRUCTIONS
2Ø  GOSUB 2ØØØ: REM INPUT ROUTINE
3Ø  REM COMBAT ALTERNATIVES
4Ø  IF A = 1 THEN GOSUB 3ØØØ
5Ø  IF A = 2 THEN GOSUB 35ØØ
6Ø  IF A = 3 THEN GOSUB 4ØØØ
7Ø  IF A = 4 THEN GOSUB 45ØØ
8Ø  GOSUB 5ØØØ: REM COMBAT
9Ø  GOSUB 6ØØØ: REM IF CHARACTER IS DEAD THIS
    RETURNS D = 1
1ØØ  IF D = 1 THEN GOSUB 7ØØØ ELSE GO TO 1Ø
11Ø  END
1ØØØ  ......................
2ØØØ  ......................
ETC.  ......................
7ØØØ  REM END ROUTINE
```

Figure 3.7

Using a series of subroutines is not the only way to design a modular structure, but it is one of the easiest. Each module or subroutine has a particular task— it may print instructions, or calculate combat, or display spaceship movement. That task may be called once or several times per turn. If called only once, and particularly only once per game, there is no need to put the module in a subroutine. Instead it can be a clearly marked section of the main program. However, to encourage flexibility and variety it is a good idea for as many as possible of the subroutines to be capable of being called up more than once, depending on conditions. For example, the combat subroutine may be called if the player decides to fight, or, later, if the monster decides to fight, or perhaps even if both wish to avoid fighting but the 'gods' (i.e., a random number) decide otherwise.

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

1. The mind map
2. The general flowchart

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

THE MIND MAP

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

Carry on in this way, thinking up new ideas related to one or more of the ideas you already have on the page and writing them down. Then draw in all the major links to ideas already down. Do not pause to evaluate any of the ideas. Just jot them down, as fast as you can, and draw in the links. Carry on until you have definitely run out of ideas. You should end up with something like Fig. 3.8. Here the central idea I started with was 'Mines of Merlin'.

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

You should find that certain ideas have many branches coming from them, whereas others have only one or two. Those with many branches are the main ideas, which will therefore become the major routines in your program; those with only a few branches will be small modules, used less; and those with only one branch do not need to be separate modules at all but can be included in the larger idea they branch from. For example, in Fig. 3.8 the 'weapon' routine and the 'retreat' routine can be included in the 'combat' routine, but the 'death' routine will not as it may be called by another routine.

THE MAJOR FLOWCHART

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

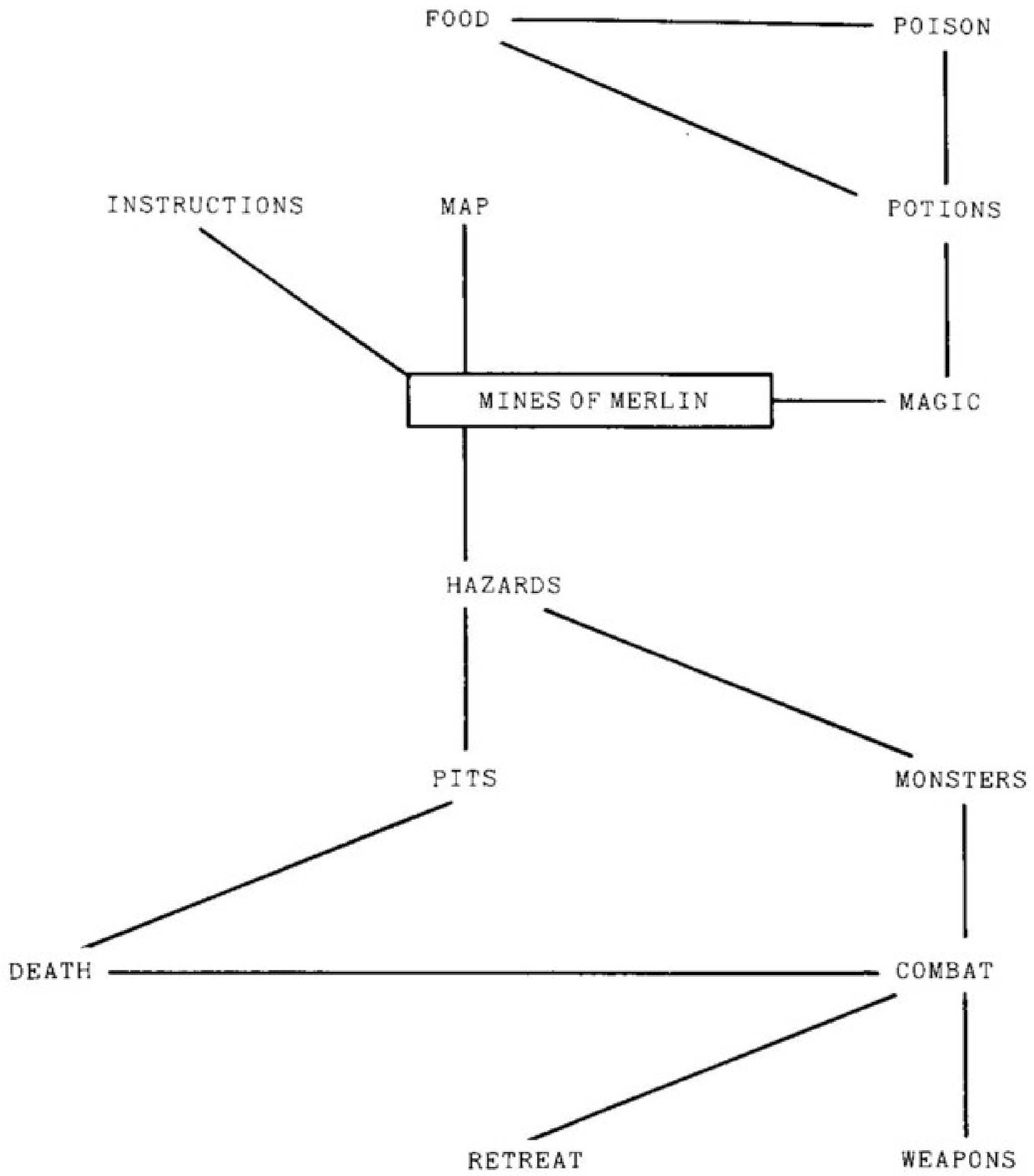


Figure 3.8

There are several examples of flowcharts in this book. Some are quite precise, being quite close to an actual BASIC routine; others are very general, leaving some stages implicit or undeveloped. It is this latter kind you should first aim for in planning your game, giving a broad description of everything that will occur in the program, with some indication of the order in which they will occur. Using the mind map in Fig. 3.8 we might construct a flowchart like the one in Fig. 3.9.

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

```
 1. PRINT INSTRUCTIONS
 2. DISPLAY EXITS
 3. MOVE CHARACTER
 4. IF THE CHARACTER MEETS A MONSTER THEN GO TO 6
 5. IF THE CHARACTER FALLS DOWN A PIT THEN GO TO 9
 6. CALCULATE COMBAT
 7. IF CHARACTER IS DEAD THEN GO TO 10 OTHERWISE GO
    TO 2
 8. DISPLAY THE PIT OUTPUT
 9. CALCULATE THE EFFECT OF THE FALL
10. IF THE PLAYER IS DEAD THEN END THE GAME OTHER-
    WISE GO TO 2
```

Figure 3.9

Work through the 'instructions' module in Fig. 3.9 as an example. Firstly, we decide that there will be written instructions displayed before the game begins. Then we draw a flowchart like Fig. 3.9, putting the instructions block in the correct place. Taking a separate sheet of paper we produce a flowchart of the instructions module, perhaps like Fig. 3.10.

```
1. FLASH UP THE TITLE OF THE GAME TEN TIMES
2. PRINT THE DESCRIPTION OF THE GAME
3. DESCRIBE POSSIBLE PLAYER ACTIONS AND THE AVAIL-
   ABLE CONTROL KEYS
4. PLAYER PRESSES 'S' TO START THE GAME
```

Figure 3.10

Each of the parts of Fig. 3.10 needs further elaboration. Instruction might be redrafted like Fig. 3.11.

```
1. ADD 1 TO COUNTER
2. CLEAR SCREEN
3. PRINT TITLE OF GAME
4. IF THE COUNTER < > 10 THEN GO TO 1
5. CLEAR THE SCREEN
6. GO TO THE NEXT SECTION OF THE PROGRAM
```

Figure 3.11

This looks very much like a BASIC program. It is only a small step from Fig. 3.11 to a section of code like Fig. 3.12.

```
10 FOR I = 1 TO 10
20 CLS
30 PRINT "The Mines of Merlin"
40 NEXT I
50 CLS
```

Figure 3.12

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

However, this procedure cannot on its own ensure a perfect, bug-free design. Other steps have to be taken. One important thing to do is to ensure that the program is amply documented. Do not throw away any design notes, make sure that all thoughts and changes are written down at the time so they are not lost, and keep the program full of REM statements. A good idea is to use increments of 10 lines in writing your BASIC programs, and place all the REM statements on lines ending with 9. In this way each section of code will have its identifier immediately before it and you will know what lines to look for as you scan through your program during development looking for a key section. Then, when the program is finished, if you wish to delete them (to save memory or to make the listing opaque for a user) it is a simple task to go through and delete all lines ending in 9.

3.6 Example plan

Most of the principles discussed in this book will be illustrated through the sample adventure games listed in Chapters 5 and 6, which are also available on a separate cassette. Let us conclude this chapter by showing how I thought up one of these programs, and the design stages it went through before coding began.

Firstly, I decided I wanted to show as many aspects of programming a puzzle game as possible, but in a way which was not too complex to understand. I therefore wanted a reasonably conventional game, so decided on a story based on a knight journeying to Camelot. I wanted the parts to be modular so other programmers could use them if wanted, and I wanted some variety of input, output, and player decisions as well as the usual text.

Because of the need to make the principles clear to someone relatively new to the idea I decided early on that this specification for a program was really too intricate and could better be done by two programs rather than one—one basically a puzzle game and the other a combat game, which is why the two adventures in this book are so different in approach.

At this point I sat down and drew up a number of mind maps, one on all the aspects of games I wished to illustrate, one on all that I knew about Camelot and knights, one on other possible story elements, and one on the game features I wanted to include. As a result of all this I had my first major flowchart, shown in Fig. 3.13.

1. Instructions.
2. Display map.
3. Print description.
4. Input command.
5. Calculate effect.
6. If the player has not won or lost then go to stage 2.

Figure 3.13

At this point I also had some brief notes on the kinds of puzzle I might have and a sketch of a simple screen display.

Only now could I think about coding but I could not go as far yet as writing any of the program. I wrote down a list of all the routines and modules that would be needed and assigned a large block of line numbers to each one, trying to overestimate rather than underestimate the number of lines that would be needed for each. It is best when doing this to remember that, when looking for a subroutine, most machines, including the Spectrum, look through all the lines starting at the lowest number until the right routine is found. Therefore, it is best to put frequently used blocks early in the program and infrequently used blocks towards the end. Typically routines for movement and screen display would go at the start because they are used at almost every turn of play, but instructions and initialization routines, which are only used once in the game, would go at the end. Though this seems slightly illogical, it can make the program run much more quickly. However, if we want very fast execution we need to use machine code or a compiler (which turns BASIC into machine code), but these are beyond the scope of this book.

At this stage, the program design for The Throne of Camelot was as shown in Fig. 3.14, though as you will see later the final program differs slightly from the initial design. The instructions, initialization routine, and DATA statements, which are usually only used once, are at the end, whereas the most used routine are at the beginning.

The main program, lines Ø to 999, is actually no more than a part of the initialization (which I placed here to make it easy to find during program development), three calls to subroutines which in turn print the instructions, complete the initialization, and print the first display, and five lines of loop which do the following: a check is first made to see if the game is over; then the input block is called and if the input is alright the appropriate verb block is called; the verb block will then either call the movement block, which in turn updates the display of map and verbal description, or will call one or more of the routines from the object description and location description blocks; if these do not end the game (by setting a variable called 'dead' to some value other than zero) then the loop is repeated.

BLOCK 1	Ø-999 MAIN PROGRAM
BLOCK 2	1ØØØ-1999 PROCESS INPUT COMMANDS
BLOCK 3	2ØØØ-2999 MOVE PLAYER, PRINT MAP AND PRINT DESCRIPTION
BLOCK 4	3ØØØ-3999 VERB ROUTINES
BLOCK 5	5ØØØ-5999 OBJECT DESCRIPTIONS
BLOCK 6	6ØØØ-6999 LOCATION DESCRIPTIONS
BLOCK 7	8ØØØ-8499 SET UP AND INITIALIZATION
BLOCK 8	85ØØ-8999 INSTRUCTIONS
BLOCK 9	9ØØØ-9499 DATA STATEMENTS
BLOCK 10	99ØØ-9999 END ROUTINE

Figure 3.14

The block diagram represents the flowchart of control which I drew up (Fig. 3.15). The reason there are some gaps in the numbers in the block diagram is simply that I left space for any major addition thought up after the main design. In the event there were no additions, but there is a logical space for any modification that you wish to build into this program. Figure 3.14 shows the modular nature of a program like this and Fig. 3.15 shows how such an adventure is almost completely built around the IF . . . THEN test.

```
1.  INPUT NEW COMMAND
2.  IF THE COMMAND IS A MOVEMENT
    COMMAND THEN GO TO NUMBER 7
3.  IF THE COMMAND REFERS TO AN OBJECT THEN GO TO
    NUMBER 12
4.  IF THE COMMAND IS 'HELP' THEN PRINT A CLUE AND GO
    TO NUMBER 1
5.  IF THE COMMAND IS 'INVENTORY' THEN PRINT THE
    INVENTORY AND GO TO NUMBER 1
6.  IF THE COMMAND IS NONE OF THESE THEN PRINT 'I
    DON'T UNDERSTAND' AND GO TO NUMBER 1
7.  IF THE INTENDED MOVEMENT IS IMPOSSIBLE THEN
    PRINT 'ERROR' AND GO TO NUMBER 1 OTHERWISE GO TO
    NUMBER 8
8.  MOVE THE CHARACTER
9.  PRINT THE NEW DISPLAY
10. UPDATE THE VALUES OF ALL VARIABLES
11. GO TO NUMBER 1
12. IF THE COMMAND IS NOT POSSIBLE THEN PRINT 'I CAN'T
    DO THAT' AND GO TO NUMBER 1 OTHERWISE GO TO
    NUMBER 13
13. CALCULATE THE RESULT OF THE COMMAND
14. PRINT THE RESULT
15. UPDATE THE VALUES OF ALL RELEVANT VARIABLES
16. IF THE GAME IS OVER THEN GO TO NUMBER 17 OTHER-
    WISE GO TO NUMBER 1
17. IF THE PLAYER HAS WON THEN PRINT 'WELL DONE'
    OTHERWISE PRINT 'HARD LUCK'
18. END THE GAME
```

Figure 3.15

3.7 Suggested themes for adventures

FANTASY

1. An underwater world (Atlantis, Mu, Captain Nemo)
2. A world entirely of winter
3. A world in the sky, peopled by winged creatures
4. A world inside a living organism (like The Fantastic Voyage) or a machine (like Tron)
5. The player is to learn the magical skills of a lost race, not through combat but by correctly interpreting a series of magical clues
6. A game based on a series of competing religions or magics, in which similar symbols have differing meanings (hence ambiguous clues)

7. The player designs and plays as a monster of some kind, such as a dragon or vampire
8. A game based on some unusual geographical feature, such as inside a volcano, on a series of floating islands, inside a glacier, among tree tops or entirely on a cliff face

SCIENCE FICTION

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

HISTORICAL

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

4 PLANNING THE DISPLAY

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

4.1 Strategies and menus

The preliminary questions to answer are as follows:

1. Is the display to be purely text, or purely graphics, or a mixture of both?
2. Will there be any real-time interaction and, if so, what will the time interval be for displaying information on the screen?
3. Do we wish to use any of the special features of the machine? For the Spectrum this means deciding if we want to use any of the following:
   flashing colours
   high-resolution graphics
   use of the two-line separate screen
   colour
   user-defined graphics
4. How much information will be shown at a time?

There are two display strategies. The first is successively to add lines to the display so that it is continually scrolling up the screen. This is the easiest method but one of the most untidy and unattractive. It fills the screen with a great deal of information which makes it difficult for the eye to find the exact piece it wants, and most of that information is unwanted at any particular point in the game anyway. Each time it scrolls the whole display moves up in a ragged way and, on the Spectrum, the need to answer the Scroll? request adds an unnecessary complexity and delay to the game.

A slightly more convenient and attractive method is to clear the screen at regular intervals. This can be done each time the screen is filled, which saves the need for continual scrolling, but is better after each block of information is shown. The displayed information is divided into separate classes, each with its own subroutine or procedure, and each class is then called when it is required. To do this we would normally use what is called a menu-driven approach. Roughly speaking a menu is a set of options displayed on the screen from which the user makes his selection. He inputs his selection and the appropriate subroutine is called, clearing the menu from the screen and displaying the appropriate information for that subroutine. For example, the main menu might read:

Option	Select
Combat	1
Refuel	2
Status	3

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

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

Option	Select
Fuel	1
Ammunition	2
Battle damage	3

Selection of one of these could lead to a further menu, and so on, until the nesting limit of the microcomputer is reached. A series of such menus is a way of guiding a user through the complex structure of a program to the actual routine required. It is mainly used in business programs and it is unlikely that you will produce a game as complex as such commercial software. However, it is a useful way of relating displays to each other. When the user has found his way to the routine he wants and has carried out the task he requires by navigating through a series of screens of information (each of which clears the previous screen), the screen will clear again and return to the original menu. This menu is generally known as the 'main menu'.

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

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

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

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

The precise interval this achieves will depend on the microcomputer used, so has to be discovered by trial and error. The Spectrum has the PAUSE command and other micros have commands like INKEY which wait for a specified time or until a key is pressed. For such micros the best method is therefore to instruct the program to wait for a long time using these commands so that plenty of time is given even for the slowest reader. However, faster readers can interrupt at any time simply by pressing a key. So if we decided that it takes 10 seconds to read a particular screen slowly, add a further 5 seconds for good luck, multiply by 50 (the number of television frames that the

PAUSE command waits per second), we can use a routine like the following:

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

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

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

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

It is possible to write a small routine in most dialects of BASIC to perform the same function as the Spectrum's PAUSE, such as:

```
100 FOR I=1 TO 1000
110 GET A$
120 IF A$<>"" THEN I=999
130 NEXT I
```

However, different microcomputers use different forms of GET, GET$, INKEY, and INKEY$, so you should make sure you understand the quirks of your own system thoroughly.

4.2 A look at windows

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

Such a capability is, however, non-standard, so we would normally have to write our own software routines to create such 'windows'. In the case of the Spectrum two windows are already defined. The top 22 lines form one window and the bottom 2 lines another. From BASIC the lower window can usually only be used in INPUT statements, which rather limits its usefulness. As an INPUT statement is in many ways like a PRINT statement in Spectrum BASIC we can use the bottom window to display text. For example, we could write a 'press any key' routine which printed a sentence in the bottom window, the actual sentence being chosen according to circumstances.

Suppose we had three types of advice in our game, held as three strings, namely:

```
10 LET A$ = "I wouldn't do that if I were you"
20 LET B$ = "You can't do that"
30 LET C$ = "You need more strength"
```

We would therefore have routines elsewhere in the program which chose the appropriate warning, depending on the circumstances. The result of the routine would be to assign to the warning string W$, one of these chosen warnings, with a statement like:

```
200 IF A=2 THEN LET W$=B$
```

This could be read as 'If the circumstances are such that the player cannot do what he wants to do then the warning will be "You can't do that". To print the chosen warning in the bottom window we would call our special INPUT subroutine which looks like this:

```
500 INPUT (W$); AT 2,0; "Press any key to continue"; Z$
510 RETURN
```

In other words, we can use the same subroutine to print any sentence we like in the bottom window, providing that sentence has been assigned to W$.

This gives us a hint as to how we can create windows on other parts of the screen. We can write subroutines which print variables at certain positions on the screen and assign to these variables the actual values or strings we want to be printed. This solution is in many ways more elegant and useful than having to put the AT or TAB coordinates in every PRINT statement, because we only have to calculate the necessary coordinates once for each window and then hold them in the subroutine. However, if you prefer you can define text windows so that every kind of PRINT statement must be in a particular place on the screen and must therefore have the appropriate AT or TAB coordinates.

This can be demonstrated easily. Suppose we wanted two small windows for text, one displaying the current strength and magical power of the player's character and the other a list of the things he could see. In the first case this would probably be two numeric variables which would be updated at regular intervals. So the updating routine can also be the PRINTing routine, as follows:

```
100 REM FIRST DO THE CALCULATIONS
........................
........................
........................
140 PRINT AT 1,30; STRENGTH
150 PRINT AT 2,30; POWER
```

This works very well. As the variables STRENGTH and POWER change throughout the game they are updated and immediately PRINTed over the old values.

For our second class of output, a list of items, this might not work so well. Presumably in the whole game there will be a large number of things to be seen, but the character will not be able to see all of them at any one time. This implies that the list will vary from situation to situation. Sometimes there will be no objects and sometimes there will be many. Consequently the techniques of overprinting an exact location will not work, especially as strings tend to be of different lengths. What is needed is a routine which fills as much of the predefined window as is needed with the current text, but also clears the whole of the rest of the window in case the previous text printed in that area took more room.

Let us suppose that no list of items will fill more than three lines of the screen. Therefore we need a three line window, a routine to clear that window, a routine to put together the actual text from the set of possible strings, and a routine to PRINT the chosen text in the correct window. Let us use the last three lines of the Spectrum screen (lines 19, 20, and 21) and suppose that the items are listed as items in the array X$(10). From the set of possible variables the program has selected X$(3), X$(5), and X$(8) which are 'a sledgehammer', 'a fir cone', and 'a green and gold necklace' respectively. This list will be preceded by the phrase "You see:".

Firstly, we produce a list of items using the string concatenator (the plus sign) to turn our separate strings into one string, adding the chosen items to the phrase "You see:". The routine has to know how many items to look for and what they are, so the choosing routines will compile a string (L$) made up of the numbers of the selected items in the array X$. In this case L$ will be "358". The length of the string L$ tells the display routine how many items to look for and to add, as well as the actual item numbers. This is done by lines 6ØØ to 64Ø below. Line 65Ø then clears the selected window by PRINTing three blank lines. Finally line 66Ø PRINTs the new string at the correct position:

```
6ØØ  REM TO PRINT A COLLECTION OF STRINGS AT A
     PARTICULAR POSITION
61Ø  LET P$ ="You see:"
62Ø  FOR I=1 TO LEN(L$)
63Ø  LET P$= P$ + X$(VAL(L$(I)))
64Ø  NEXT I
65Ø  PRINT AT 19,Ø,,,,,,
66Ø  PRINT AT 19,Ø;P$
```

The same procedure can be used whatever the number of lines of screen or the number of items in the list. Simply change the number of commas in line 65Ø and the PRINT AT number. However, suppose we want a window on the left-hand side of the screen rather than the top or the bottom, and a window which is only, say, eight characters wide. We will have to PRINT only eight blank spaces, not complete lines, and each successive eight-character string must be separate. It cannot be printed as one continuous string or it will overwrite a portion of the screen outside the desired window.

In this case we will have to use variables as specifiers of the PRINT position so that the vertical position can be incremented for each new PRINT item. Let us rewrite lines 6ØØ to 67Ø to produce an eight-character by six-line (maximum) window on the left-hand side of the screen, starting five lines down:

```
6ØØ  REM TO PRINT A COLLECTION OF STRINGS IN AN 8 X 6
     COLUMN ON THE LEFT
62Ø  FOR I= 1 TO 6
63Ø  PRINT AT 4+I, Ø;"        ";REM 8 SPACES
64Ø  NEXT I
65Ø  FOR I = 1 TO LEN(L$)
66Ø  PRINT AT 4+I, Ø;X$(VAL(L$(I)))
67Ø  NEXT I
```

From these two examples you should be able to see how text windows can be defined almost anywhere on the screen. Every separate PRINT position is itself a text window, so the maximum number of windows would be the number of these positions (22 × 32 = 704), though you will seldom want more than two or three. The key point is to decide as early as possible how many windows you require and of what size. It makes sense to write the PRINTing subroutines before you have decided on all the text you are to display, but this should be done after you have classified the types of text you will show. It also makes sense in the planning stage to draw a rough sketch of the windows making a full screen display so that you can get an idea of how cluttered or organized the screen will appear to a user.

There are three major advantages to using text windows. Firstly, a clear and attractive display is produced which is pleasant to see, well organized, and easy to understand. Secondly, the process of deciding what to print, where, and when is simplified. And, thirdly, by using text windows a large amount of information can be displayed on the screen at one time without chaos. Several classes of information can be displayed simultaneously without the need for scrolling, menus, or constant screen clearing. Of course, you can also use one window for several purposes, as if it was a miniscreen. If you wish to do this, and the window-cleaning routine is complex, it is often better to have this as a subroutine separate from the PRINTing routine, but called by it. In this way several different types of text can use the same cleaning routine and the same window without the need to duplicate code.

Graphics windows on most machines can be defined in a similar way by using graphics coordinates, which are called the PLOT positions in most systems. The Spectrum has an advantage over several systems because its text and graphics screens are identical. Thus the method used to create graphics windows can be identical to that used to create text. The only crucial differences are that the graphics windows will be defined using PLOT and DRAW rather than PRINT AT, and the reference point or origin is the top left-hand corner for text but the bottom left-hand corner for graphics.

Suppose we want two graphics windows. The first is eight character positions square, starting at the bottom left corner, while the second is ten by four in the top right. Remembering that each character position is an 8 × 8 matrix of pixels (see Chapter 17 of the Spectrum manual) this gives us a window of 64 × 64 pixels and another of 80 × 32. So one window has the PLOTting coordinates Ø,Ø; Ø,64; 64,Ø; 64,64; and the other has 255,153; 175,153; 175,175; 255, 175. These become the maximum and minimum coordinates for PLOTting within. If we wish to turn the second window cyan and draw a circle on it, we could use the following routine:

```
100 FOR X = 175 TO 255 STEP 8
110 FOR Y = 153 TO 175 STEP 8
120 PLOT PAPER 5;X,Y
130 NEXT Y
140 NEXT X
150 CIRCLE 215,164,5
```

Naturally we could print at the same area using a cyan block graphic to achieve the same effect.

5 CONTROLLING MOVEMENT

5.1 Keys

It is usual in adventure games as well as graphic games to control player movement by use of a set of keys. Normally these would be the cursor keys, or the arrow keys (with the arrows representing the chosen direction), or the keys N,S,E,W (for north, south, east, and west), or in some cases the number keys, especially if a numeric keypad is available. In practice any set of keys could be used and it is possible to define a 'graphics keypad', that is to say, a subset of the keyboard made up of nine keys in a square, with the orientation of the external eight keys representing the points of the compass, and the central key either ignored or used for a special action, such as the ubiquitous 'hyperspace'. On a QWERTY keyboard the most likely graphics pad is the sector shown in Fig. 5.1.

Q	W	E
A	S	D
Z	X	C

Figure 5.1

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

5.2 Moving a character

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

5.3 Method A: random movement

Whether our adventure is textual or graphic, we can make the link between successive events purely random. In this case there will be no map as such because returning to the same 'location' may well result in a different event. A typical structure might be as in Fig. 5.2.

1. PLAYER MOVES PIECE
2. GENERATE A RANDOM NUMBER BETWEEN 1 AND 3 = R
3. CHOOSE A SUBROUTINE ACCORDING TO R
4. DO CHOSEN SUBROUTINE

Figure 5.2

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

5.4 Method B: seeded random movement

This method is probably the most economical on memory, which is useful for machines with only a small amount of RAM or for programs where an unusual amount of memory is required for storing data. Put simply, each time the program is run a different map will be generated, but that map remains the same, unalterable, throughout the game. The 'map' is actually a series of numbers, but only an extremely able mathematician would be able to predict the map from the initial randomly chosen number. The method works as in Fig. 5.3.

1. USE A FORMULA TO CREATE A NUMBER WITH A DECIMAL POINT
2. GET RID OF THE INTEGER IN THE NUMBER
3. USE THE FORMULA TO INDICATE POSSIBLE DIRECTIONS
4. USE THE FORMULA TO CALCULATE EVENTS
5. NEXT MOVE

Figure 5.3

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

A Spectrum routine to do this is used in The Mines of Merlin in Chapter 6. The key lines are lines 1ØØØ to 222Ø. Line 22ØØ is where the seeded random function is used for movement and line 8Ø1Ø is where it is defined. Each of the digits in the decimal part of this number can be used to control different events, as, for example, digit 2 is used to control selection of the main event in line 546.

5.5 Full screen movement

This is a graphic method which moves a character, which may be one of the ASCII set or a predefined character, across the screen. If you wish to use a user-defined character it should be defined before the routine is run. Essentially the method works by displaying the character at a certain point at the screen, using PRINT AT, or, if memory-mapped graphics are available, POKEing the screen location. When the character is moved a space will be printed at this location and the character now PRINTed or POKEd at the new location. The routine therefore has the following stages shown in Fig. 5.4.

1. PRINT/POKE CHARACTER AT OLD LOCATION = OL
2. INPUT MOVEMENT
3. CALCULATE THE NEW LOCATION = NL
4. PRINT/POKE SPACE AT OL
5. PRINT/POKE CHARACTER AT NL
6. LET OL=NL
7. GO TO 2

Figure 5.4

A Spectrum routine to do this is Fig. 5.5, which is used in the Treasure Trove program in Chapter 8.

The method becomes more complicated if the screen over which the character is travelling is not itself blank. There is little point in having a blank screen, however, so we will want to PRINT or POKE at select locations on the screen characters representing locations in the game and hence possible encounters. For example, let us have a lake routine and a castle routine in our program. We will represent the lake by O and the castle by * to keep things simple, though user-defined graphics will give better results (see Chapter 8).

Using a visually mapped screen in this way means that we must add two further subroutines to our movement routine. The first must

```
 8ØØ REM MOVES "p" AROUND SCREEN
 89Ø LET x=Ø: LET y=Ø
 9ØØ LET a$=INKEY$: IF a$="" THEN GO TO 9ØØ
 9Ø5 LET p=x: LET q=y
 91Ø IF a$="5"  THEN  LET x=x-1
 92Ø IF a$="6"  THEN  LET y=y+1
 93Ø IF a$="7"  THEN  LET y=y-1
 94Ø IF a$="8"  THEN  LET x=x+1
 95Ø IF y>2Ø  THEN LET y=2Ø
 96Ø IF x>31  THEN  LET x=31
 97Ø IF x<Ø  THEN  LET x=Ø
 98Ø IF y<Ø  THEN  LET y=Ø
 99Ø PRINT AT q,p;" "
1ØØØ PRINT AT y,x;"p"
1Ø1Ø GO TO 9ØØ
```

Figure 5.5

check to see if the position the figure will be moving to (NL) contains the lake or castle. If so it must record the fact (e.g., by setting a flag) and call up the appropriate event. The second routine must use the flag to reprint the castle or lake character after the figure has moved on. So the first would look like Fig. 5.6.

1. IS NEXT LOCATION (NL) A SPACE?
2. IF IT IS A SPACE THEN RETURN TO THE MAIN ROUTINE
3. IF IT ISN'T A SPACE CALL THE APPROPRIATE ROUTINE AND SET THE FLAG TO THE CORRECT NUMBER
4. RETURN TO THE MAIN ROUTINE

Figure 5.6

If the graphics being detected have standard character codes, these codes can be used as the values of the flag and be passed therefore to the second routine, which would look like Fig. 5.7.

1. IF FLAG IS SET TO Ø THEN RETURN TO THE MAIN ROUTINE
2. OTHERWISE SELECT THE CORRECT CHARACTER ACCORDING TO THE FLAG NUMBER
3. PRINT AT THE OLD LOCATION THE SELECTED NUMBER AND RETURN TO THE MAIN ROUTINE

Figure 5.7

5.6 The Ramtop map

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

Into this reserved area is POKEd a series of bytes, each representing one location in the map. These bytes can then be used as the code for a description of the player's current location, or even for direct visual mapping. Let us take the latter first. Suppose the map is a series of rooms arranged within a matrix of 12 × 12 possible locations. Within these possible 144 locations, 48 are rooms and 96 are blank walls. Each of the 48 rooms will be given an identifying number and the blank walls Ø, so that the whole map if drawn looks something like Fig. 5.8.

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

Figure 5.8

Then after each movement the player can be shown his current position on the map by displaying the eight locations around him, together with his current location. This obviously works well in conjunction with the 'graphic keypad' method of controlling movement.

If any of the surrounding locations are Ø they will be displayed as a wall character, such as the graphics block above the Spectrum 8 key. If greater than Ø, the display will be a blank space. Thus, if the player is at location 9,11 in the above matrix, the map of his surroundings would be Fig. 5.9.

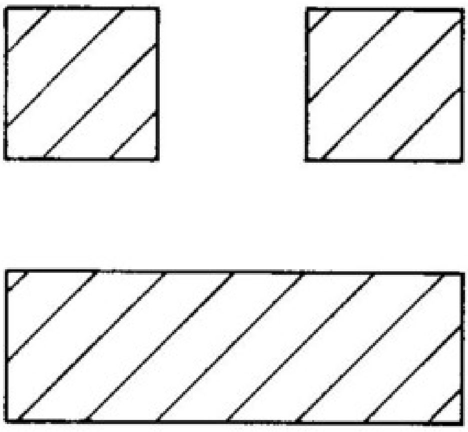


Figure 5.9

Each time the player moves not only will the map be updated but a test will be made beforehand to see if the player can move to that next location, i.e., testing to see if the next byte (location) is greater than Ø. To use such a matrix we would need to reserve 144 addresses and POKE into each in turn the appropriate value. The first 12 addresses will be locations 1,1 to 12,1 on our map; the next 12 addresses will be 1,2 to 12,2, and so on. Thus if the first address is 32120 (decimal), we would POKE Ø into it—similarly with the next address (32121). But 32122 would be given the value 1. An easy way to achieve such a series of POKEs is as follows:

```
5ØØ LET P= 32119: REM ONE LESS THAN THE START
    ADDRESS
51Ø FOR I= 1 TO 144: REM THE NUMBER OF BYTES
52Ø READ A
53Ø POKE P+I,A
54Ø NEXT I
55Ø DATA Ø,Ø,1,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
56Ø DATA Ø,Ø,2,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
57Ø DATA Ø,12,3,4,5,24,Ø,Ø,Ø,Ø,Ø,Ø
ETC.
```

To read a map like this in checking the player's movements and printing the map, we simply PEEK the nine relevant bytes. If X is the horizontal position of the map and Y the vertical, then the locations around him will be x−1,y−1; x,y−1; x+1,y−1; x−1,y; x+1,y;

x−1,y+1; x,y+1; x+1,y+1. However, the map is kept in memory not as a matrix but as a sequence. Each x is 1. Each y is 12. To find the desired byte we must make it x+(y*12). So if the player is located at point 3,2 in the matrix then x=3 and y=2, and we need to read the byte located at 3212Ø plus 3 plus 2*12 = address 32147. The bytes surrounding the position are thus 13 less, 12 less, 11 less, 1 less, 1 more, 11 more, 12 more, and 13 more. We can draw the nine necessary bytes using the following routine:

```
6ØØ LET P=3212Ø +X +(Y*12)
61Ø LET Q= PEEK(P−13): IF Q=Ø THEN PRINT AT (relevant
    screen location) (block graphic)
62Ø LET Q= PEEK(P−12): IF Q=Ø THEN PRINT AT (next screen
    location) (block graphic)
```

etc.

This search can be reduced to a formula as we will see in the next section.

5.7 The puzzle game design

We have spent enough time looking at theory and method; let us now actually create our first game. We will make it a puzzle game to use some of the mapping techniques just discussed, but if we are to have a puzzle game we first need a puzzle. Usually this takes the form of a maze, so we will begin our design with the map of our maze. Because it is a puzzle that has to be solved it will be fixed in form, and does not change from game to game. However, because we are going to use a modular design our method allows the same routines to be used with different maps. The game can be changed totally by changing the map, yet in programming terms this might be no more than changing five or six DATA statements, perhaps only one or two hundred bytes of information. Unfortunately, however, it is usually a little more complex.

We will now design a map, then code it, then look at ways that routine is used in the game and ways that it can be changed for different games.

The maze in an adventure game is a combination of disorientation (as in a real maze like Hampton Court) and logical complexity (something like a crossword). The logical maze can be thought of as an algorithm or flowchart. Cad has to work through all the stages of the algorithm in the correct order to produce the correct result from a program. The algorithm is a series of obstacles to be overcome. Usually each obstacle, each separate problem, will be solved by bringing the necessary object or objects to a particular location and doing the right thing with it/them.

The first stage of design is therefore to list the sequence of obstacles or solutions. We can write each obstacle location as a 'need' location, meaning that you need a particular object to pass through that location, and each solution location as a 'get' location, meaning that it is the place where Cad will find what is needed to overcome a particular difficulty. At this stage we do not need to worry about what the obstacles might be nor what cleverness might be added to the obstacle/object relationship; we simply draw up a map of the sequence of 'gets' and 'needs' required to solve the overall puzzle. For every 'need' location there will be at least one 'get', though we may have some 'gets' without 'needs', i.e., red herrings. We will use the letters of the alphabet to show that two locations are linked in this way. We will also draw up our map from the final location (the player's goal) backwards to the start position.

This game is to be called The Throne of Camelot, so we will make the player's goal the City of Camelot itself. The previous location will be the final obstacle, so this will be NEED Z. Therefore earlier in the program there will be a GET Z square. If we work backwards in this way we will end up with a list like Fig. 5.10.

START
GET J
NEED J
GET K
GET L
NEED K
GET M
NEED M
NEED L
GET N
NEED N
GET R
GET Y
GET P
NEED Y
GET Q
GET H
GET S
NEED H
GET X
NEED X
GET I
NEED P

```
NEED Q
NEED R AND NEED I
GET Z
NEED S
NEED Z
END
```

Figure 5.10

If this was a list of locations it would look like a corridor with a series of rooms. If Cad goes along the corridor from room to room all the obstacles will be found in the right order. This would be pretty pointless. So instead of just a straight line, or a corridor, we have to draw a map which allows the player to travel along many different routes and gives him a number of choices. To do this we can add 'empty' locations, i.e., locations which have no importance in the game, though they may appear to have (such as by containing red herrings). We can draw up a network map in the following way:

1. Firstly draw the starting square and a route out of it.
2. Draw another square with more than one exit route. At each exit route draw another square. In each of the blank squares write E (for empty) or write the first GET or any NEED on your list. In our case they are GET J and NEED J. So our map might look like Fig. 5.11.

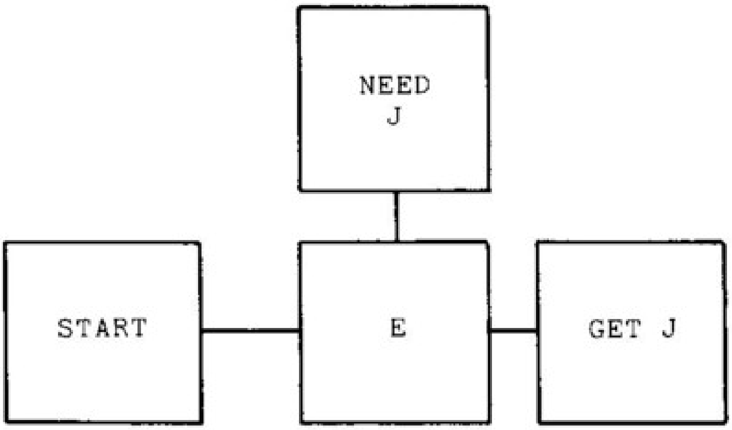


Figure 5.11

3. The player cannot go through a NEED square until he has passed through the relevant GET square. In this case his route would be START—E—GET J—E—NEED J. However, he can pass through GET squares, so now we draw routes out of the GET J square. These cannot lead to any more GET squares because Fig. 5.10, our logical map, says Cad has to pass through NEED J before he receives any other rewards. Thus every route out of the GET J

square must terminate in a NEED square, though the routes could pass through empty squares. The route of the NEED J square must also eventually lead to GET K, so we can add some squares, as in Fig. 5.12.

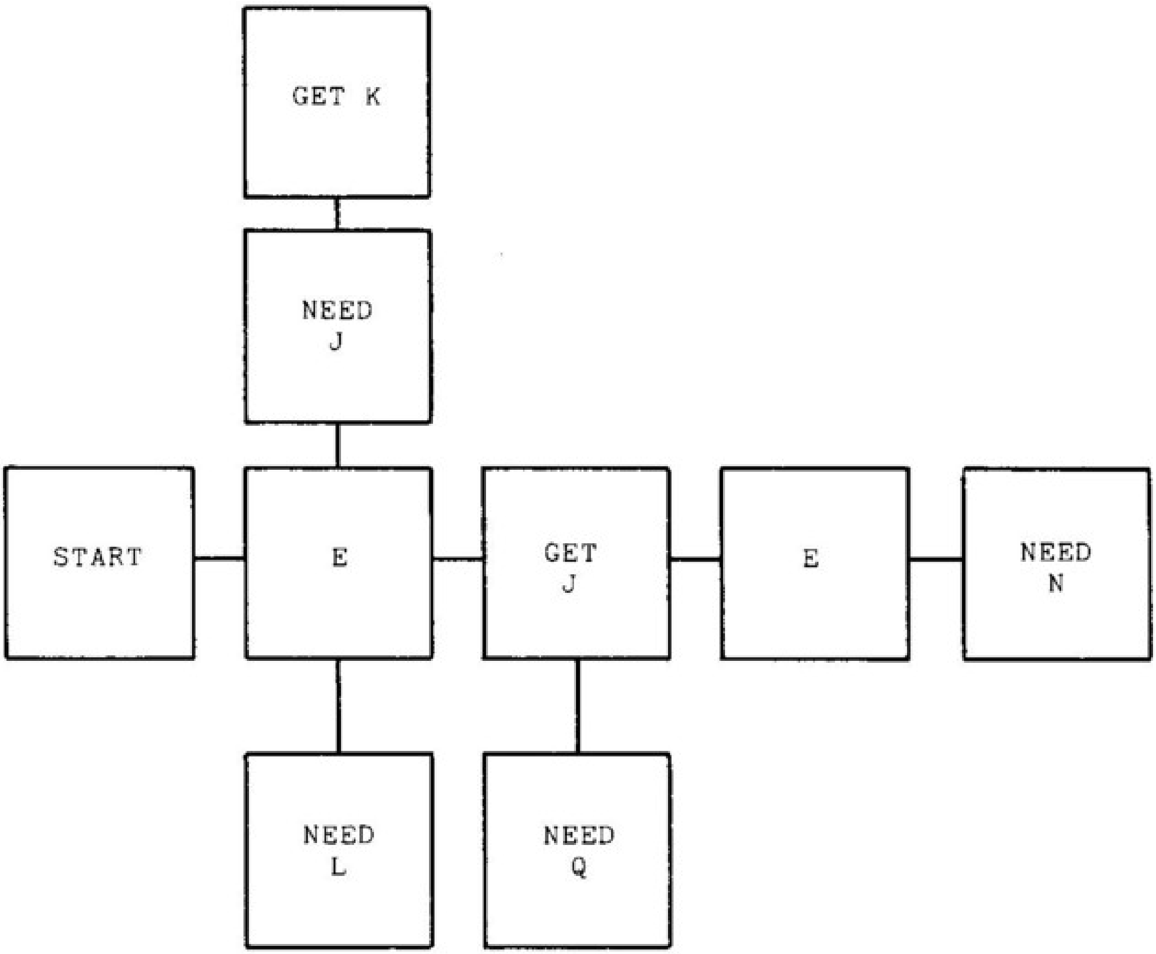


Figure 5.12

4. The next stage after GET K is GET L, so we can add routes out of GET K which lead to GET L. However, since he already has K, the player could be allowed to by-pass L and go straight to NEED K. If he does this, he will discover at a later stage that he needs L, so will have to revise his strategy, but this is what the game is about. The relevant part of the map will look like Fig. 5.13.

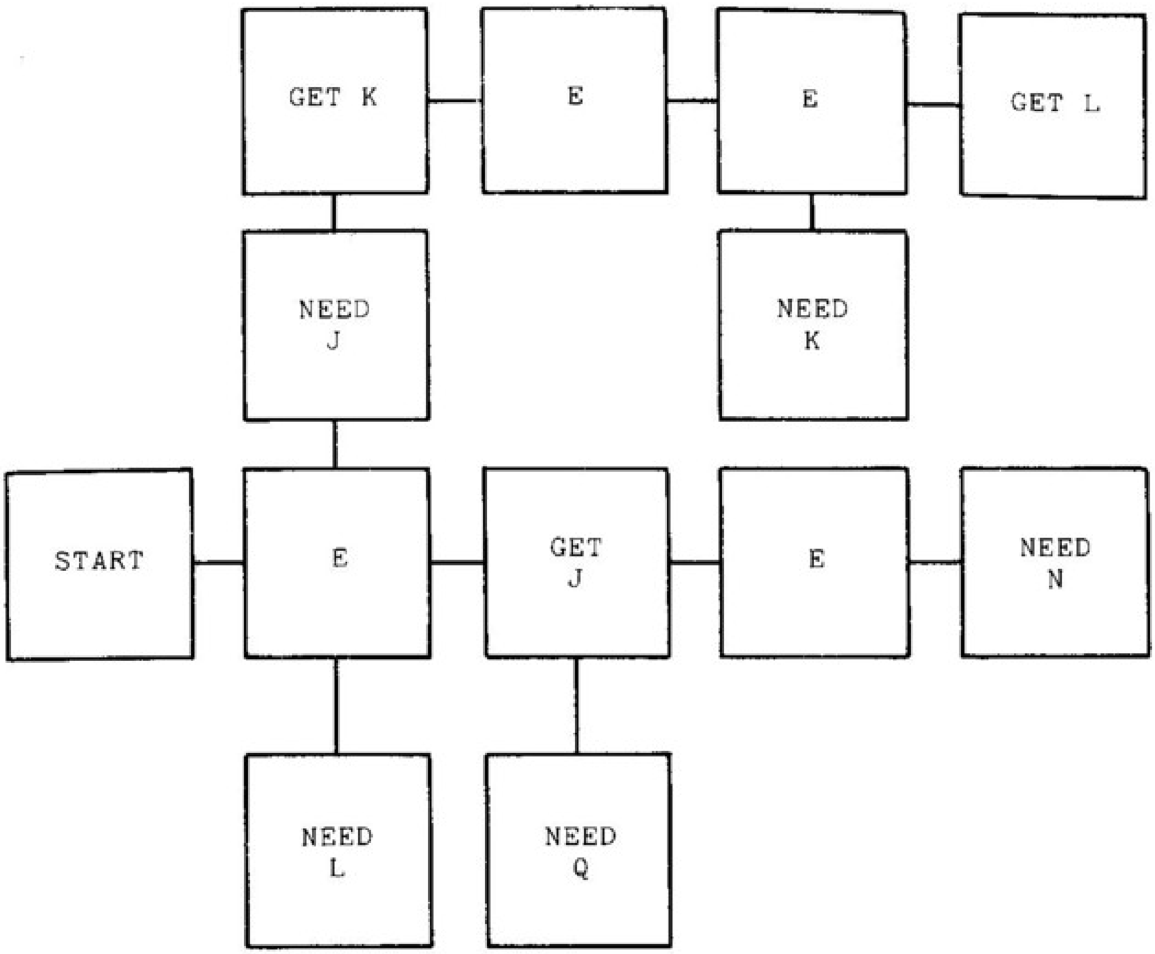


Figure 5.13

5. You will see that one of the routes out of the NEED K square is the next stage on our map, i.e., GET M. Another is to NEED M, which is a dead end until GET M has been reached.
6. We carry on in this way, making sure that there are no routes which make it possible for a player to succeed by by-passing stages in our elementary flowchart. It does not matter if this can be done temporarily, i.e., by 'making a mistake' (as in neglecting to GET L), as long as the player has to go back there eventually. Figure 5.14 shows the network map I used for The Throne of Camelot. If you follow its routes you will see that you have to visit all the squares to get to Camelot, and you have to do most of them in a fixed order, but you also have to retrace your steps several times.

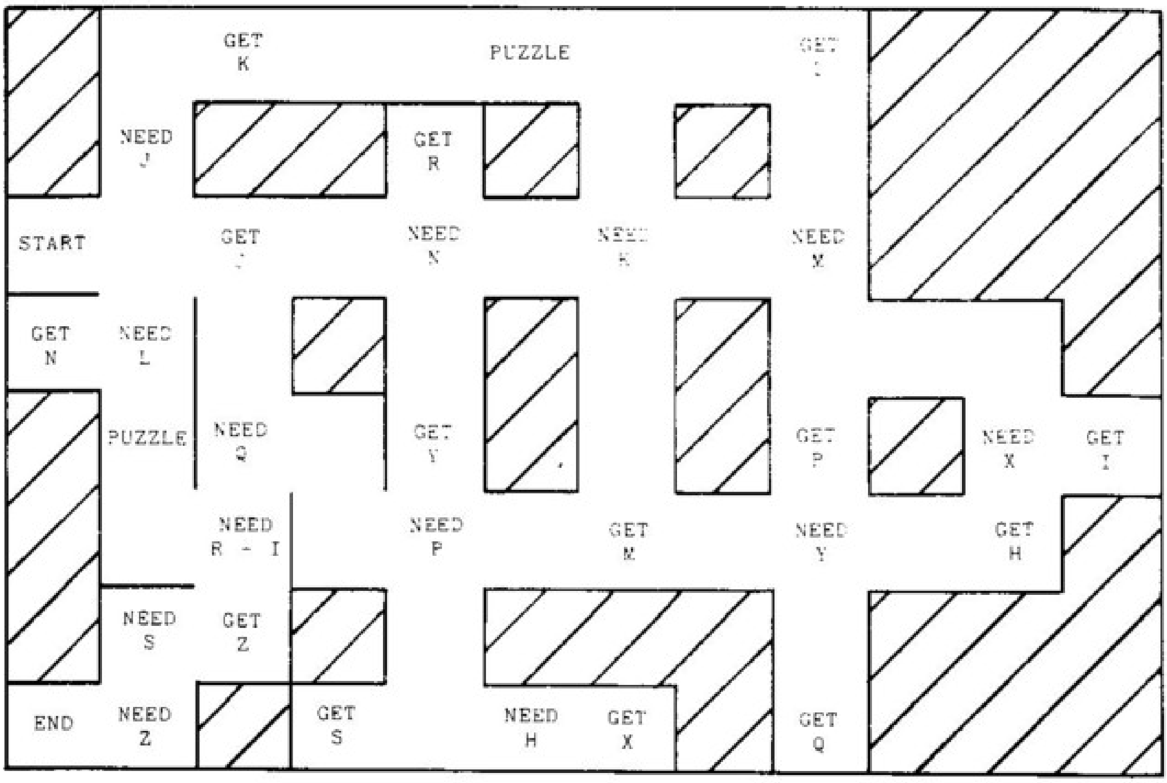


Figure 5.14

We have now completed our basic network map, which combines the logical structure and the maze structure. We can therefore add a little variety to the plan. Various ways to do this are discussed elsewhere in the book, but some simple ones are:

1. To make more than one object necessary to pass through a particular NEED square. You will see that my network includes several of these.
2. To make some objects prevent Cad going through some squares. This can be very frustrating to the player, especially if he finds that an object he has just discovered after a long search causes him to be killed in the very next room.
3. To add extra puzzles or problems which act as NEED squares, i.e., obstacles, but do not need objects or solutions from elsewhere in the game. For example, the player might have to unravel an anagram to proceed further. An anagram routine can be found in Chapter 8. We will add two extra puzzles to Camelot.

When all this is done we can finally draw the map we are going to store in our program. Use gridded paper which gives enough room for all the squares needed in each direction. We must add some space to our map so that the only squares which are adjacent to each other are those which can be travelled to. Our map is not drawn like Fig. 5.14 but like Fig. 5.15, because it is not possible to travel directly from

START to GET N. The reason for adding these gaps, and consequently extra empty rooms, is because of the map display routine we will be using. The routine displays the square currently occupied by the player and its surrounding squares, as described in Sec. 5.6.

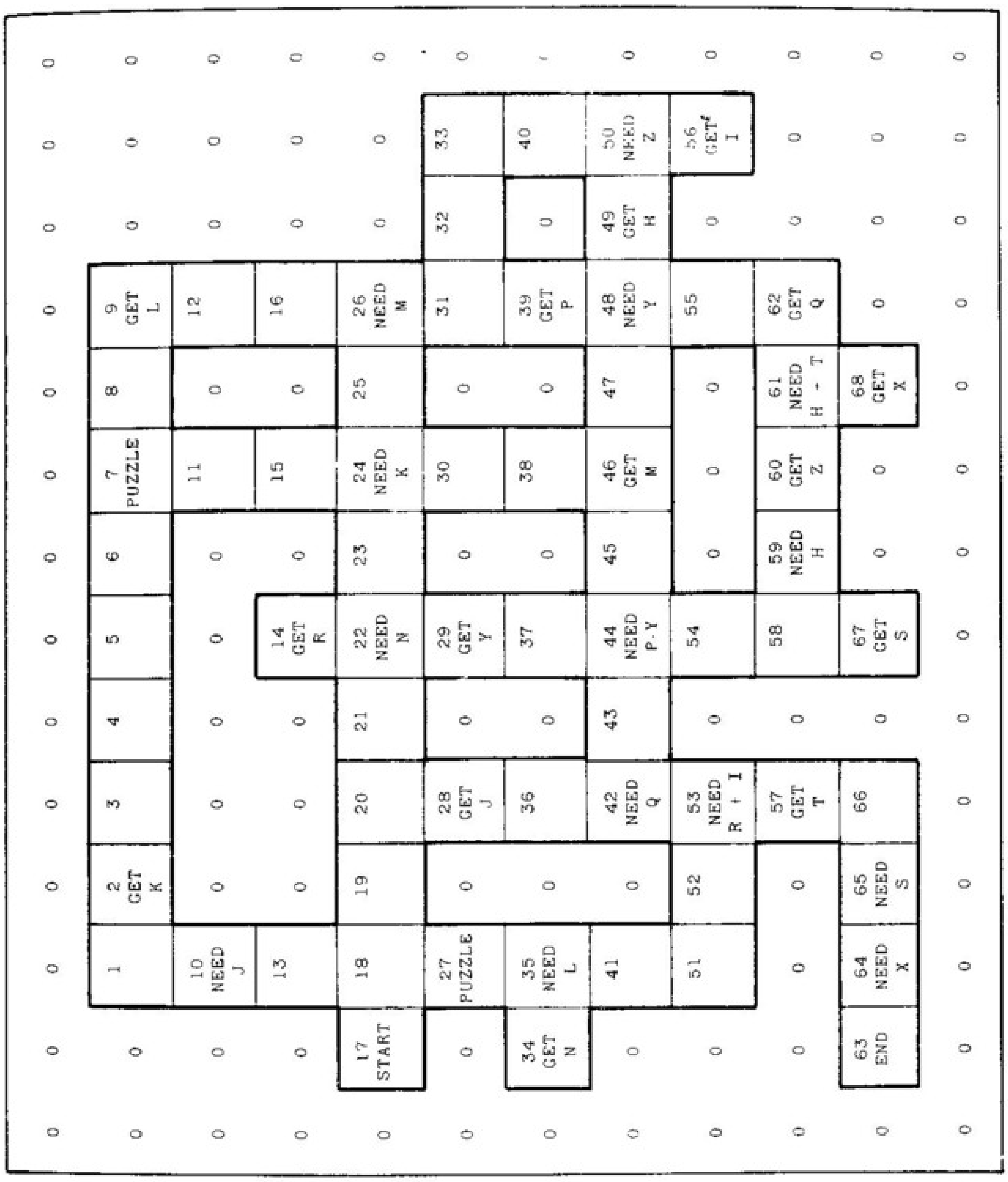


Figure 5.15

Figure 5.15 shows our completed map for Camelot. Note the additions— some more empty rooms, the NEED R square has become NEED R+I, the NEED P square has become NEED P−Y, and there are two puzzle squares.

You will see that the map is on a 12 × 14 grid, giving a 168 square matrix. We could store this in an array, or a series of strings, but for speed of use and convenience we will store it above Ramtop using the method described in Sec. 5.6. You will see that it is easy to work out the necessary DATA statements because of the regular grid used. We could simply give each square a number from 1 to 168 and store these, but there is no point in storing different numbers for all the blank walls, so these become Ø and the only numbered squares are those which the player could possibly travel through. Each DATA statement can be one line of the grid to make it easy to refer to, so the full code of our map will be the DATA statements kept in lines 95ØØ to 96ØØ of the program.

(Note the border of zeros around the map to prevent wandering off its edge.) You will also see that the start location is number 17, the end number 63, and the total number of locations is 68. This may seem rather illogical but it does not affect the nature of the game (and it also makes 'cheating' a little more difficult).

Once the map has been placed in memory using the routine previously described it can be read using the routine in lines 2Ø7Ø to 2165 of Camelot. The DATA statements can always be deleted while the program is running if you want to save memory or to prevent the player from BREAKing into the program and looking at the map, but a machine code program would be needed to do this. As the player moves around the adventure each of the 68 numbers will key the appropriate routine. One way to do this would be to use a series of lines such as:

```
1ØØ  IF LOCATION=1 THEN GOSUB 1ØØØ
11Ø  IF LOCATION=2 THEN GOSUB 1Ø1Ø
12Ø  IF LOCATION=3 THEN GOSUB 1Ø2Ø
etc.
```

but we would need 68 such lines! Some BASICs such as Microsoft BASIC and BBC BASIC have a computed GOSUB which allows statements like:

```
1ØØ  ON LOCATION GOSUB 1ØØØ, 1Ø1Ø, 1Ø2Ø
```

which acts identically to the three lines above, but Spectrum BASIC lacks this function. However, Spectrum BASIC does allow us to use variables in GOTO and GOSUB statements. For example, if all the routines for each of the 68 locations are written in lines 6ØØØ to 6689 of our program and each routine has 10 lines, we could use just one line of BASIC to select the correct routine. Assuming LOCATION is the variable which stores the player's current position, then:

```
1ØØ GOSUB (LOCATION * 1Ø) + 6ØØØ
```

will do the trick. If the player is at the start location, number 17, then this line will send control to a subroutine starting at line 617Ø. If the player is at the end, number 63, control is passed to line 663Ø, and so on. The only thing to remember is that all the location routines should be regularly spaced. If some locations have routines which are longer than others, use the length of the longest routine as the interval between routines. For example, if one routine is 15 lines long but all the others are only 6 lines, they should still be placed 15 lines apart.

The other feature of this method of mapping is the display that results. In order to test the player's memory and intelligence we do not want to display the whole map at once, just the current location and its immediate surroundings. We also want to check that the player does not walk into, or through, walls, i.e., that he can only travel in a permitted direction. Both of these can be done using the kind of map we have just stored in memory.

The display method has already been outlined. If colour is available on your micro then choose two colours for the wall and open blocks which are different from the background colour of the screen. In this way the whole area can be seen clearly as well as the particular type of each block. Choose colours that will also show up on a black and white set. Remember that the relative locations of the surrounding bytes are as held in the following table:

Direction	Address
NW	L−13
N	L−12
NE	L−11
W	L−1
LOCATION	L
E	L+1
SW	L+11
S	L+12
SE	L+13

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

```
1Ø REM M = FIRST ADDRESS OF MAP LOCATIONS
2Ø REM A AND B ARE PRINT COORDINATES
3Ø REM X IS WEST TO EAST
4Ø REM Y IS NORTH TO SOUTH
```

```
 5Ø LET A=2: LET B=2
 6Ø FOR Y=−12 TO 12 STEP 12
 7Ø FOR X=−1 TO 1 STEP 1
 8Ø LET P= PEEK(M+X+Y): REM LOOK AT ADDRESS OF
    LOCATIONS
 9Ø IF P=Ø THEN PRINT AT A,B;"*":GOTO 11Ø
1ØØ PRINT AT A,B;" "
11Ø LET B=B+1: REM MOVE TO NEXT PRINT POSITION
12Ø NEXT X
13Ø LET A=A+1: REM MOVE TO NEXT PRINT LINE
14Ø NEXT Y
```

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

5.8 Filling in the map

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

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

Thus each of the 68 subroutines will involve a description. Some will have additional routines, some of which will be general and others specific to the unique location. For the sake of simplicity we will say that the EMPTY squares are of the first kind, PUZZLE squares are of the second kind, and NEED and GET squares of the third kind.

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

One way to do this is to draw up a list with three columns—the names of the GETS/NEEDS (in our case the letters of the alphabet, with some not used), the objects (the GETS), and what the object is needed for (the NEEDS). A simple example is that NEED J is 'a locked door' and GET J is 'a key'. However, it is best if most of the relationships are not as straightforward as this. In fact many puzzle adventures make a point of being as esoteric as possible, often using puns or long trains of thought to make the thing work. For example, GET K might be 'a duck' and NEED K might be 'a steep cliff'. What is the relationship? Well, in order to go further you have to descend the cliff. How do you do that? It is easy—you get down off the duck. (Not a very good joke perhaps but it can really test the intelligence of the player.)

Once we have a complete list of such relationships we have all the basics of our game. It is a good idea if some sort of theme links them together, but it is not necessary. The list of relationships used in Camelot is in Fig. 5.16, but if you want to play the game first you might want to ignore this figure.

Number	Code	Noun	(GET) square Location	(NEED) square Destination	Solution to Problem
1	K	Monkey	2	24	Give the monk-key
2	L	Sword	9	35	Cut pack of wolves
3	I	Feather	56	53	To write with
4	Y	Crowbar	29	48	To lever fallen tree
5	N	Orange	34	22	Gives juice(deuce)
6	P	Torch	39	44	Makes rock light
7	J	Compass	28	10	Guides through fog
8	Q	Helmet	62	42	Carries water to counterbalance
9	H	Wheel	49	61/59	Has a tyre (attire) Fix cart
10	S	Stocking	67	65	Has a ladder to climb cliff
11	X	Letters	68	64	Provides mail (armour)

Number	Code	Noun	(GET) square Location	(NEED) square Destination	Solution to Problem
12	M	Horn	46	26	Summons Robin Hood
13	Z	Witch-doctor	60	50	Cures witch
14	R	Scroll	14	53	For writing on
15	T	Bananas	57	61	Peeled for slippers
16		Tree			
17		Sign			(These nouns are understood by the program but are not key parts of the problems.)
18		Cupboard			
19		Wolves			
20		Turnstile			
21		Rock			
22		Cage			
23		Jester			
24		Giant			
25		Cart			
26		Goose			
27		Juice			
28		Tyre			
29		Ladder			
30		River			
31		Cliff			
32		Around			
33		Pack			
34		Basket			
35		Mail			
36		Peel			
37		Armour			
38		Counterbalance			
39		Deuce			

Figure 5.16 Objects: nouns, 'gets' and needs'

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

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

In essence anything that can hold a person can be regarded as a location with its own description. Some commonly used places are listed in Fig. 5.17. The description can be as long and involved as memory allows. The larger it is the more the player will need to interpret, but a description which is too long without any possible player interaction will only serve to annoy. The locations should be as interesting as possible.

Try to make the relationships between places of some interest, rather than the straightforward 'You enter another room'. Characters can travel into and out of buildings, up and down hills and cliffs, across ravines and rivers, into secret passages, under bridges, etc. Where possible, extra puzzles can be set by making the entrance to a particular location problematic even if it does not depend on an object to be found. A simple example is to give in the description a choice of routes, only one of which is correct, with the others ending in sudden death; an example would be to describe two treacherous paths down a cliff face, one of which will crumble away. More intricate can be locations which are unreachable unless the correct command is used. For example, to cross a stream the player might try to JUMP, WADE, LEAP, PADDLE and CROSS before he thinks of SWIM. This is used in location 27 of The Throne of Camelot as one of the puzzles.

In writing descriptions of locations we should also make them appear relevant to the objects which are originally located there. This

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

Figure 5.17

means that they make sense to the player (which might not matter if we are designing a nonsense or absurd game such as one based on Alice in Wonderland) and that the important aspect of each location may not at first be apparent, thus adding more problems for the player.

The program will hold the descriptions for Camelot in lines 6ØØØ to 6689. As we have 68 locations this gives 10 lines each. You will notice that I have added some small routines to increase the variety of the game. The player may be fooled into thinking these are important, but they are not really necessary.

5.9 Moving objects

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

Camelot takes care of these problems by using three arrays, namely, O(3Ø,3), O$(15,13), and Y$(15,3Ø). O(x,1) holds the location of object x; O(x,2) holds the length of the name of the object in characters (this is to prevent the Spectrum inserting extra spaces when we read O$); O$(x) holds the name of object x, which is up to 13 characters long; Y$(x) holds the description associated with object x.

During initialization each array must be set up, with O() being given the initial locations of all objects (lines 82ØØ, 824Ø, and 93ØØ). Then the following routines can be implemented:

For player's inventory (routine begins at line 3Ø1Ø):

```
FOR I=1 TO 3Ø
IF O(I,1)=99 THEN PRINT O$(I,1 TO O(I,2))
NEXT I
```

For describing the objects in a room (routine begins at line 5ØØØ):

```
FOR I = 1 TO 3Ø
IF O(I,1) = LOCATION THEN PRINT Y$(I), O$(I,I TO (O(I,2)))
NEXT I
```

To take an object (routine begins at line 3Ø7Ø):

LET O(I,1)=99

To drop an object (routine begins at line 3Ø9Ø):

```
FOR I= 1 TO 3Ø
IF E$ = O$(I,1 TO 3) THEN O(I,1) = LOCATION
NEXT I
```

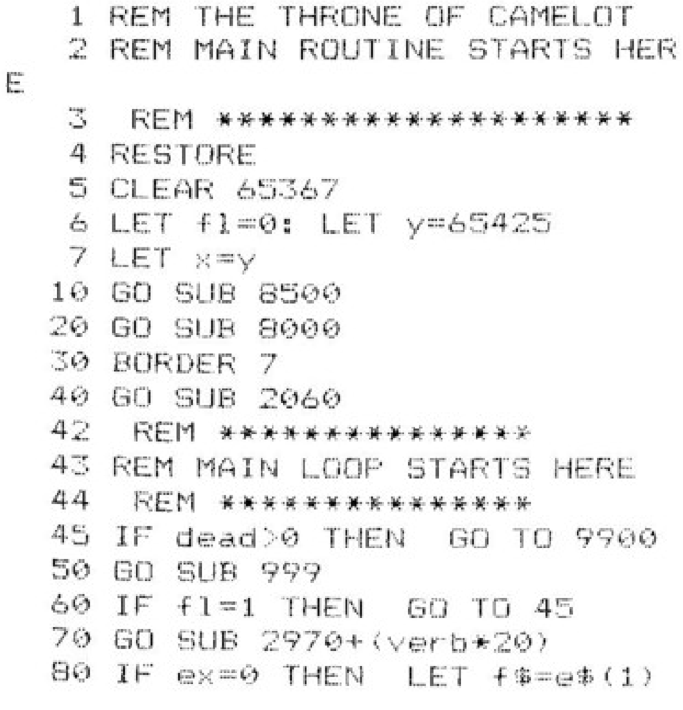
If an object is used or destroyed:

LET O(I,1) = Ø

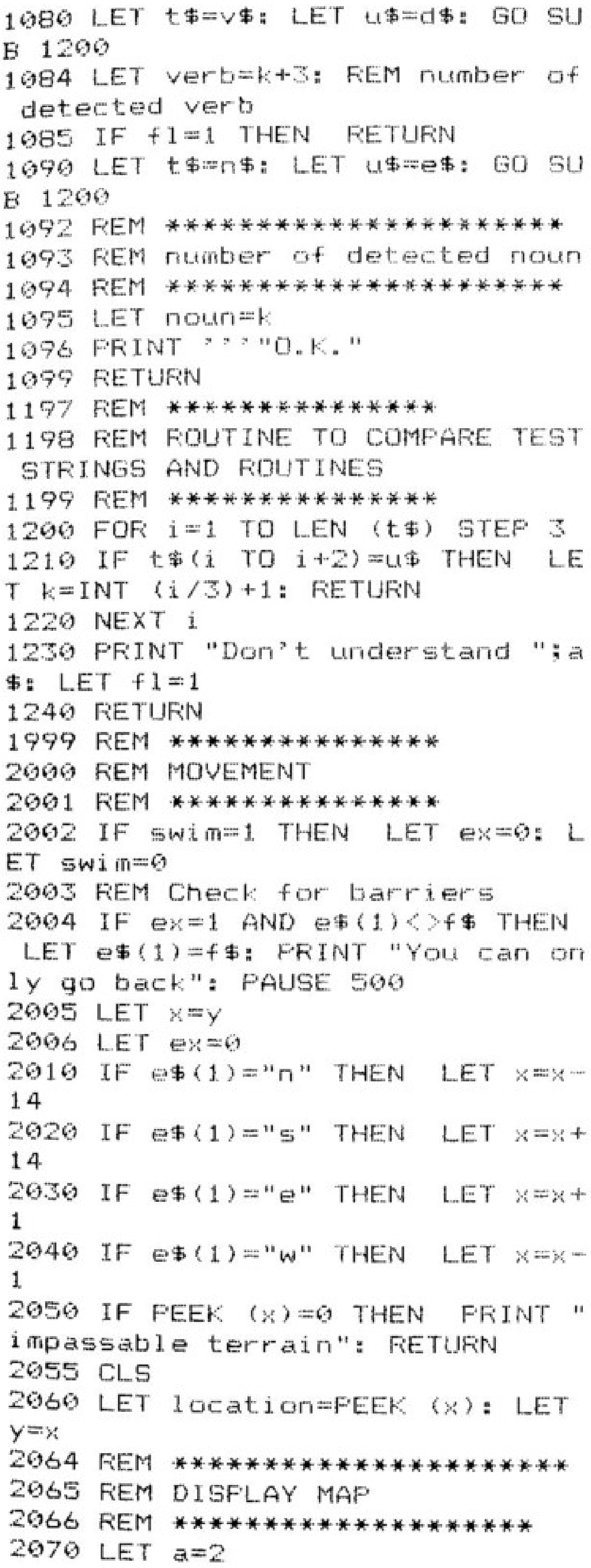
These routines should be self-explanatory once you know that 99 means that the object is carried by the player. The first routine prints a list of all the object names that have the code 99; the second prints all the object names and their descriptions for all objects that have the code of the current location; the third simply sets the code to the possession code; the fourth looks for the object held in the three-letter noun code E$ and sets the object's code to the current location; and the fifth wipes the object from the game by setting it to a number which none of the other routines recognize. This need not be zero, of course, but it is usually the most sensible choice.

Exactly which objects are manipulated and in which ways depends on the verb routines. These are described in Chapter 7.

The Throne of Camelot



```
   1 REM THE THRONE OF CAMELOT
   2 REM MAIN ROUTINE STARTS HER
E
   3  REM ********************
   4 RESTORE
   5 CLEAR 65367
   6 LET f1=0: LET y=65425
   7 LET x=y
  10 GO SUB 8500
  20 GO SUB 8000
  30 BORDER 7
  40 GO SUB 2060
  42  REM ***************
  43 REM MAIN LOOP STARTS HERE
  44  REM ***************
  45 IF dead>0 THEN  GO TO 9900
  50 GO SUB 999
  60 IF f1=1 THEN  GO TO 45
  70 GO SUB 2970+(verb*20)
  80 IF ex=0 THEN  LET f$=e$(1)
  90 GO TO 45
  97 REM ***************
  98 REM END OF MAIN LOOP
  99 REM ***************
 859 PAUSE 1000
 997 REM ***************
 998 REM TWO WORD INPUT
 999 REM ***************
1000 LET a$="      "
1005 INPUT "What next?",a$: IF a
$="" THEN  GO TO 1000
1008 REM ***************
1009 REM DECLARE VARIABLES AS SP
ACES
1010 REM ***************
1011 LET k=0: LET b$="   ": LET
c$="  ": LET d$="   ": LET e$="
   "
1012 IF LEN (a$)<3 THEN  LET a$=
a$+"  "
1013 REM ***************
1014 REM CHECK FOR HELP AND INVE
NTORY
1015 REM ***************
1016 IF a$(1 TO 3)="inv" THEN  L
ET verb=2: RETURN
1017 IF a$(1 TO 3)="hel" THEN  L
ET d$="hel": LET verb=3: RETURN
1019 FOR i=1 TO 12 STEP 3: IF a$
(1 TO 3)=x$(i TO i+2) THEN  LET
dr=1: LET verb=1
1022 NEXT i: IF dr=1 THEN  LET d
r=0: IF verb=1 THEN  LET e$(1)=a
$(1): RETURN
1025 FOR i=1 TO LEN (a$)
1030 IF a$(i)=" " THEN  LET b$=a
$(1 TO (i-1)): LET c$=a$((i+1) T
O LEN (a$)): LET k=k+1
1040 NEXT i
1050 IF k<>1 OR c$="" THEN  PRIN
T "Two words please": GO TO 1000
1065 LET b$=b$+"  ": LET c$=c$+"
   "
1070 LET d$=b$(1 TO 3): LET e$=c
$(1 TO 3)
1075 LET fl=0: LET verb=0: LET n
oun=0
1078 IF d$="go " THEN  LET t$=w$
: LET u$=e$: LET verb=1: GO SUB
1200: IF fl=1 THEN  RETURN
1079 IF d$="go " THEN  LET b$="g
o ": RETURN
```



```
1080 LET t$=v$: LET u$=d$: GO SU
B 1200
1084 LET verb=k+3: REM number of
 detected verb
1085 IF fl=1 THEN  RETURN
1090 LET t$=n$: LET u$=e$: GO SU
B 1200
1092 REM **********************
1093 REM number of detected noun
1094 REM **********************
1095 LET noun=k
1096 PRINT '''"O.K."
1099 RETURN
1197 REM ***************
1198 REM ROUTINE TO COMPARE TEST
 STRINGS AND ROUTINES
1199 REM ***************
1200 FOR i=1 TO LEN (t$) STEP 3
1210 IF t$(i TO i+2)=u$ THEN  LE
T k=INT (i/3)+1: RETURN
1220 NEXT i
1230 PRINT "Don't understand ";a
$: LET fl=1
1240 RETURN
1999 REM ***************
2000 REM MOVEMENT
2001 REM ***************
2002 IF swim=1 THEN  LET ex=0: L
ET swim=0
2003 REM Check for barriers
2004 IF ex=1 AND e$(1)<>f$ THEN
 LET e$(1)=f$: PRINT "You can on
ly go back": PAUSE 500
2005 LET x=y
2006 LET ex=0
2010 IF e$(1)="n" THEN  LET x=x-
14
2020 IF e$(1)="s" THEN  LET x=x+
14
2030 IF e$(1)="e" THEN  LET x=x+
1
2040 IF e$(1)="w" THEN  LET x=x-
1
2050 IF PEEK (x)=0 THEN  PRINT "
impassable terrain": RETURN
2055 CLS
2060 LET location=PEEK (x): LET
y=x
2064 REM **********************
2065 REM DISPLAY MAP
2066 REM ********************
2070 LET a=2
```

```
2080 FOR c=-14 TO 14 STEP 14
2085 LET b=2
2090 FOR d=-1 TO 1 STEP 1
2100 LET p=PEEK (y+c+d)
2110 IF p=0 THEN  PRINT AT a,b;"
■": GO TO 2130
2120 PRINT AT a,b;" "
2130 LET b=b+1
2140 NEXT d
2150 LET a=a+1
2160 NEXT c
2165 PRINT AT 3,3;"$"
2195 PRINT AT 6,0;"You are ";
2199 REM Call location routines
2200 GO SUB (6000+(location*10))
2207 REM **********************
2208 REM Check for objects
2209 REM **********************
2210 GO SUB 5000
2296 RETURN
2297 REM **********************
2298 REM control exits
2299 REM **********************
2300 IF e$(1)="n" THEN  LET f$="
s"
2310 IF e$(1)="s" THEN  LET f$="
n"
2320 IF e$(1)="e" THEN  LET f$="
w"
2330 IF e$(1)="w" THEN  LET f$="
e"
2399 RETURN
2990 GO SUB 2000: RETURN
2997 REM *********************
2998 REM Verbs start here
2999 REM *********************
3010 REM inv routine
3011 REM **********************
3012 PRINT ''
3013 PRINT "You have:";
3015 FOR i=1 TO 15: IF o(i,1)=99
 THEN  PRINT TAB 10;o$(i,1 TO o(
i,2))
3017 NEXT i
3020 IF o(10,3)>=2 THEN  PRINT "
a ladder"
3026 RETURN
3027 REM **********************
3028 REM Help routine
3029 REM **********************
3030 IF location=2 OR location=3
4 THEN  PRINT "The problem is ov
```

```
er my head": RETURN
3Ø31 IF location =35  THEN  PRIN
T "How would you deal with wolve
s?": RETURN
3Ø32 IF location=13 THEN  PRINT
"We need a sense of direction":
RETURN
3Ø33 IF location=24 THEN  PRINT
"The monk is aping your actions"
: RETURN
3Ø34 IF location=26 THEN  PRINT
"If you can't solve this then yo
u've blown it": RETURN
3Ø35 IF location=5Ø THEN  PRINT
"Maybe you should bone up on you
r medecine": RETURN
3Ø36 IF location=61 AND o(15,3)<
>5 THEN  PRINT "Partially dresse
d he's not very appealing. What
about some footwear?": RETURN
3Ø37 IF location=61 AND o(9,3)<>
5 THEN  PRINT "He seems to need
some attire": RETURN
3Ø38 IF location=64 THEN  PRINT
"The armourer never moves from h
is post": RETURN
3Ø39 IF location=53 AND o(14,3)<
2 THEN  PRINT "Right on?": RETUR
N
3Ø4Ø IF location=59 THEN  PRINT
"Where there's a will there's a
way": RETURN
3Ø41 IF location=48 THEN  PRINT
"Itsa besta leava wella lone": R
ETURN
3Ø42 IF location=65 THEN  PRINT
"Oh dear! That's torn it": RETUR
N
3Ø43 IF location=42 THEN  PRINT
"It'll be a weight off my mind w
hen you've solved this": RETURN
3Ø44 IF location=53 THEN  PRINT
"He tells you a tale of a tickli
sh situation": RETURN
3Ø45 IF location=22 THEN  PRINT
"You simply need a card with the
 right number of pips": RETURN
3Ø46 IF location=44 THEN  PRINT
"Sorry I'm in the dark as well":
 RETURN
3Ø48 PRINT "I think you're doing
 fine": RETURN
```

```
3049 RETURN
3050 REM *******************
3051 REM look
3052 REM ********************
3055 IF c$(1 TO 3)="aro" THEN  G
O SUB 5000: RETURN
3056 IF noun>15 THEN  GO TO 3062
3060 IF o(noun,1)=99 OR o(noun,1
)=location THEN  GO SUB 5100+(no
un*5): RETURN
3061 PRINT "I can't see it": RET
URN
3062 GO SUB 5100+(noun*5)
3066 RETURN
3067 REM *********************
3068 REM take
3069 REM **********************
3070 LET g=0: FOR i=1 TO 15: IF
o(i,1)=99 THEN  LET g=g+1:
3071 NEXT i: IF g>4 THEN  PRINT
"You can't carry any more": RETU
RN
3072 IF noun=29 AND o(10,1)=99 T
HEN  LET o(10,3)=2: PRINT "You n
ow have a ladder": RETURN
3073 IF noun=26 THEN  PRINT "She
 won't come": RETURN
3074 IF noun=28 AND o(9,1)=99 AN
D o(9,3)>2 THEN  LET o(9,3)=4: P
RINT "You now have a tyre": RETU
RN
3075 IF noun>15 THEN  PRINT "It
won't move": RETURN
3076 IF o(noun,1)=99 THEN  PRINT
 "You've already got it": RETURN
3077 IF location=59 AND noun=9 T
HEN  LET o(9,3)=3
3078 IF location=o(noun,1) AND o
(noun,3)>0 THEN  LET o(noun,1)=9
9: RETURN
3085 PRINT "Its not here ": RETU
RN
3088 RETURN
3089 REM *********************
3090 REM leave
3091 REM **********************
3092 IF o(noun,1)=99 THEN  LET o
(noun,1)=location: RETURN
3094 PRINT "You don't have it":
RETURN
3106 RETURN
3107 REM *********************
```

```
3108 REM give
3109 REM *********************
3110 IF location=22 AND noun=39
AND o(5,3)=2 THEN  LET o(5,3)=3:
 LET noun=5: LET ex=0: GO SUB 30
90: RETURN
3111 IF location=22 AND noun=27
AND o(5,3)=2 THEN  LET o(5,3)=3:
 LET noun=5: LET ex=0: GO SUB 30
90: RETURN
3112 IF location=61 AND noun=36
AND o(15,3)=2 THEN  LET o(15,3)=
3: LET ex=0: RETURN
3113 IF location=61 AND noun=28
AND o(9,3)=4 THEN  LET o(9,3)=5:
 LET ex=0: RETURN
3114 IF location=53 AND noun=3 T
HEN  LET o(3,3)=2
3115 IF location=53 AND noun=14
THEN  LET o(14,3)=2
3116 IF location=53 AND o(14,3)=
2 AND o(3,3)=2 THEN  LET ex=0
3117 IF location=24 AND noun=1 T
HEN  LET o(1,3)=3: LET ex=0
3119 GO SUB 3090
3128 RETURN
3129 REM *********************
3130 REM light
3131 REM *********************
3132 IF noun<>6 THEN  PRINT "It
won't burn": RETURN
3134 IF o(noun,3)>1 THEN  PRINT
"Its already alight!": RETURN
3135 IF o(noun,1)=99 THEN  PRINT
 "The torch burns brightly": LET
 rock=3: LET o(6,3)=2: IF locati
on=44 THEN  LET EX=0: GO SUB 644
6: RETURN
3148 RETURN
3149 REM **********************
3150 REM peel
3151 REM *********************
3155 IF o(15,1)=99 AND noun=15 T
HEN  PRINT "You have a pile of b
anana peels": LET o(15,3)=2: RET
URN
3157 IF o(5,1)=99 AND noun=5 THE
N  PRINT "You have a handful of
rind": RETURN
3159 PRINT "You've nothing to pe
el": RETURN
3168 RETURN
```

```
3169 REM **********************
3170 REM wear
3171 REM **********************
3172 IF o(11,1)=99 AND noun=11 O
R noun=35 THEN  PRINT "What a sp
lendid fully armed figure": LET
o(11,3)=2: RETURN
3187 RETURN
3188 REM **********************
3189 REM fill
3190 REM **********************
3191 IF location=42 AND o(8,1)=9
9 THEN  GO TO 3212
3192 IF o(8,1)<>99 THEN  PRINT "
You have nothing to fill": RETUR
N
3196 IF location=27 OR location=
41 OR location=35 OR location=36
 AND noun=8 AND o(8,1)=99 THEN
LET o(8,3)=2: RETURN
3208 RETURN
3209 REM *********************
3210 REM empty
3211 REM *********************
3212 IF o(8,1)<>99 THEN  PRINT "
You have nothing to empty": RETU
RN
3213 IF o(8,3)=1 THEN  PRINT "Th
e helmet isn't full": RETURN
3215 IF o(8,3)=2 AND location<>4
2 THEN  PRINT "What a waste": LE
T o(8,3)=1: RETURN
3218 IF o(8,3)=2 AND location=42
 THEN  PRINT "The turnstile open
s": LET o(8,3)=4: LET ex=0: RETU
RN
3229 RETURN
3249 REM *********************
3250 REM cut
3251 REM *********************
3252 IF noun=33 AND location=35
AND o(2,1)=99 THEN  LET o(2,3)=2
: LET EX=0: RETURN
3254 IF o(2,1)<>99 THEN  PRINT "
You've nothing to cut with": RET
URN
3256 PRINT "A waste of strength"
3266 RETURN
3268 REM *********************
3269 REM blow
3270 REM *********************
3271 IF location=26  AND o(12,3)
```

```
=1 THEN  LET o(12,3)=2: GO SUB 6
262: LET o(12,3)=1: LET EX=0: RE
TURN
3273 IF noun<>12 THEN  PRINT "A
waste of breath": RETURN
3274 IF o(12,3)<>1 THEN  PRINT "
You don't have a horn": RETURN
3285 RETURN
3289 REM **********************
3290 REM fix
3291 REM **********************
3295 IF noun=9 AND location=59 A
ND o(9,1)=99 THEN  LET o(9,3)=2:
 LET ex=0: GO SUB 3090: RETURN
3300 PRINT "You can't"
3309 RETURN
3329 REM *********************
3330 REM squeeze
3331 REM *********************
3340 IF o(5,1)=99 THEN  PRINT "Y
ou now have a pocketful"," of ju
ice": LET o(5,3)=2: RETURN
3345 PRINT "A waste of strength"
: RETURN
3348 RETURN
3349 REM *********************
3350 REM swim
3351 REM *********************
3352 IF noun=30  THEN  LET swim=
1: RETURN
3368 RETURN
3369 REM **********************
3370 REM climb
3371 REM **********************
3372 IF noun=16 AND location=2 T
HEN  PRINT s$: LET o(1,3)=1: RET
URN
3373 IF noun=16 AND location=34
THEN  PRINT s$: LET o(5,3)=1: RE
TURN
3375 IF noun=31 AND location=65
THEN  PRINT "You climb halfway,
but its too steep. You fall and.
...splat! Oh dear": LET dead=1:
RETURN
3380 IF noun=29 AND o(10,3)=2  A
ND location=65 THEN  PRINT "You
climb the cliff": LET ex=0: LET
o(10,3)=3
3388 RETURN
3389 REM **********************
3390 REM lift
```

```
3391 REM **********************
3392 IF location=48 AND noun=16
AND o(4,1)=99 THEN  LET o(4,3)=2
: LET ex=0: GO TO 3405
3394 IF noun=21 AND rock=3 THEN
 LET rock=4: LET ex=0: GO TO 340
5
3400 PRINT "It's too heavy": RET
URN
3405 PRINT "You lift it out of t
he way"
3409 RETURN
4997 REM *********************
4998 REM objects at location
4999 REM *********************
5000 FOR i=1 TO 15
5010 IF o(i,1)=location AND o(i,
3)>0  THEN  PRINT "You see ";o$(
i,1 TO (o(i,2)));y$(i): NEXT i:
RETURN
5020 NEXT i
5030 RETURN
5098 REM **********************
5099 REM Looking at objects
5100 REM **********************
5105 PRINT q$: RETURN
5110 PRINT q$: RETURN
5115 PRINT "It has a sharp point
": RETURN
5120 PRINT q$: RETURN
5125 PRINT "Its a full of juice"
: RETURN
5130 IF o(6,3)<2 THEN  PRINT "It
s unlit": RETURN
5131 PRINT "Its alight": RETURN
5135 PRINT q$: RETURN
5140 PRINT "It's bucket shaped":
 RETURN
5145 PRINT "It's a bicycle wheel
": RETURN
5150 PRINT "Its torn": RETURN
5155 PRINT "They've all been pos
ted": RETURN
5160 PRINT q$: RETURN
5165 PRINT "He is casting a spel
l": RETURN
5170 PRINT "Its blank": RETURN
5175 PRINT "They're very appeali
ng": RETURN
5180 IF location=34 AND o(5,1)=3
4 THEN  PRINT "A giant orange ha
ngs from it": RETURN
```

```
5182 IF location=2 AND o(1,1)=2
THEN  PRINT "There's a monkey in
 it": RETURN
5183 IF location=57 THEN  PRINT
"They're full of bananas": LET o
(15,3)=1: RETURN
5184 PRINT q$: RETURN
5185 IF location=5 THEN  PRINT "
It says 'I wouldn't go any "," f
urther if I were you'": RETURN
5187 PRINT q$: RETURN
5190 IF location=22 OR location=
50 THEN  PRINT "The door is open
": RETURN
5192 IF location=14 AND o(14,1)=
14 THEN  LET o(14,3)=1: PRINT "T
here's a scroll in the dust": RE
TURN
5195 PRINT q$: RETURN
5200 IF location=42 AND o(8,3)<3
 THEN  PRINT "There's an empty c
ounterbalance by the lock": RETU
RN
5205 PRINT "Its extremely heavy"
: RETURN
5210 PRINT "It has a goose in it
": RETURN
5215 PRINT q$: RETURN
5220 PRINT "He looks like a scri
be": RETURN
5225 PRINT "One wheel is broken"
: RETURN
5230 PRINT "It has long tail fea
thers": LET o(3,3)=1: RETURN
5235 PRINT q$: RETURN
5240 PRINT q$: RETURN
5245 PRINT "Its very long": RETU
RN
5250 PRINT q$: RETURN
5255 PRINT "It looks unclimbable
": RETURN
5260 PRINT "There's a crumpled s
tocking in it": LET o(10,3)=1: R
ETURN
5265 PRINT q$: RETURN
5270 IF location=67 THEN  PRINT
"There's a crumpled stocking ":
LET o(10,3)=1: RETURN
5285 IF location=62 AND o(8,1)=6
2 THEN  PRINT "The helmet is in
good condition": LET o(8,3)=1: R
ETURN
```

```
5300 IF o(12,3)=0 THEN  LET o(12
,3)=1
5305 IF o(12,1)=46 THEN  PRINT "
There's a horn in the ash "
5310 RETURN
5990 PRINT q$
5998 RETURN
5999 REM **********************
6000 REM location descriptions
6001 REM *********************
6010 PRINT "on a rocky hillside"
," with many loose boulders"
6019 RETURN
6020 PRINT " by a tree on the","
 hillside"
6029 RETURN
6030 PRINT "on a steep hillside"
," with leafless trees"
6039 RETURN
6040 PRINT "on a dirt track"," w
ith many potholes"
6049 RETURN
6050 PRINT "on a dirt track.","
Beside the track is a sign"
6059 RETURN
6060 PRINT "outside a large tent
"
6069 RETURN
6070 PRINT "inside a circus tent
."," A juggler throws a knife","
 at you but misses"
6079 RETURN
6080 PRINT "on a road covered","
 in weeds"
6089 RETURN
6090 PRINT "outside a blacksmith
's."
6099 RETURN
6100 PRINT "on a well-kept road.
": RETURN
6110 PRINT "on a path through","
 a dark forest": RETURN
6120 PRINT "on the north face ",
"of a mountain": RETURN
6130 PRINT "in a thick bank of f
og": IF o(7,1)<>99 THEN  PRINT "
You're lost": LET ex=1: GO SUB 2
300
6132 RETURN
6140 PRINT "in a very dusty"," c
upboard": RETURN
6150 PRINT "in a glade in a fore
```

```
st": RETURN
6160 PRINT "on the south face","
 of the mountain": RETURN
6170 PRINT "on a road": RETURN
6180 PRINT "at a crossroads with
"," a broken signpost": RETURN
6190 PRINT "by a ravine with ","
a narrow bridge across it": RETU
RN
6200 PRINT "on a meandering path
way": RETURN
6210 PRINT "by a ravine with ","
a rope bridge across it": RETURN

6220 PRINT "at The Gambler's Inn
"
6225 IF o(5,3)<3 THEN  PRINT "Yo
u need a playing card"," to go f
urther": LET ex=1: GO SUB 2300
6229 RETURN
6230 PRINT "in the yard of an in
n": RETURN
6240 PRINT "in a monastery"
6242 IF o(1,3)<3 THEN  PRINT "A
monk says"'" 'You need a key to
go further'": LET ex=1: GO SUB 2
300: RETURN
6245 LET ex=0: LET f$=e$(1): PRI
NT "You may pass": RETURN
6250 PRINT "in the porch to a
     monastery": RETURN
6260 PRINT "in Sherwood Forest."
'" Suddenly ten men at arms
   surround you. "
6262 IF o(12,3)=1 AND o(12,1)=99
  THEN  LET ex=1: GO SUB 2300: R
ETURN
6264 IF o(12,3)=2 THEN  PRINT "
Robin Hood to the rescue!"'" The
 armed men run": LET ex=0: RETUR
N
6268 PRINT "They cut off your he
ad": LET dead=1: RETURN
6270 PRINT "by a river with a br
oken bridge across it"
6271 IF swim<>1 THEN  LET ex=1:
GO SUB 2300: RETURN
6280 PRINT "on a beach": RETURN
6290 PRINT "in a stables": RETUR
N
6300 PRINT "in a graveyard": RET
URN
```

```
6310 PRINT "at the edge of a for
est": RETURN
6320 PRINT "by a barred cave": R
ETURN
6330 PRINT "in a very dark caver
n": RETURN
6340 PRINT "in an orange grove":
 RETURN
6350 PRINT "on a riverbank amids
t"," twentyone snarling wolves "
6351 IF o(2,1)<>99 THEN  PRINT "
They tear you to pieces": LET de
ad=1: RETURN
6353 PRINT "They fear your sword
": IF o(2,3)<2 THEN  LET ex=1: G
O SUB 2300: RETURN
6354 LET ex=0: LET f$=e$(1)
6355 RETURN
6360 PRINT "in a rocky bay": RET
URN
6370 PRINT "in a ploughed field"
: RETURN
6380 PRINT "in a cornfield": RET
URN
6390 PRINT "in a farmyard": RETU
RN
6400 PRINT "in a narrow cave"
6401 IF o(13,3)<2 THEN  PRINT "Y
ou hear a scream to the south"
6402 RETURN
6410 PRINT "on a dusty winding r
oad": RETURN
6420 PRINT "in front of"," an ir
on turnstile"
6422 IF o(8,3)<4 THEN  PRINT "It
 blocks your way": LET ex=1: GO
SUB 2300: RETURN
6423 LET ex=0: LET f$=e$(1): RET
URN
6430 PRINT "on a gravelly road":
 RETURN
6440 PRINT "in a passage": IF ro
ck<4 THEN  PRINT " filled with a
 huge rock"'""
6442 IF rock<2 THEN  PRINT "You
can't go any further": LET ex=1:
 GO SUB 2300: RETURN
6444 IF o(6,1)=99 AND o(6,3)=2 T
HEN  PRINT "Your torch makes the
 rock"," much lighter": LET rock
=3: RETURN
6446 IF rock=4 AND o(3,1)<>99 TH
```

```
EN  PRINT "Your way is clear": L
ET ex=0: LET f$=e$(1): RETURN
6448 IF o(3,1)=99 THEN  PRINT "Y
our feather tickles you.","You s
hout out, causing a huge   avala
nche."'"       Suddenly you feel
 rather depressed": LET dead=1:
RETURN
6449 PRINT "You can't go any fur
ther": LET ex=1: GO SUB 2300: RE
TURN
6450 PRINT "in a narrow winding"
," tunnel": RETURN
6460 PRINT "in a farmhouse "
6465 IF o(12,3)=0 THEN  PRINT "w
ith a smoking chimney"
6466 RETURN
6470 PRINT "in a farmyard": RETU
RN
6480 PRINT "on a narrow path"
6481 IF o(4,3)<2 THEN  PRINT "wh
ich is blocked by a fallen"," tr
ee": LET ex=1: GO SUB 2300: RETU
RN
6482 RETURN
6490 PRINT "at a cave entrance":
 RETURN
6500 PRINT "in a damp cave. ","T
here is an old woman moaning.","
Beside her is a huge cage. ","Sh
e says 'If you don't cure"," my
headache I'll turn you"," into a
 toad"
6505 IF o(13,1)<>99 THEN  PAUSE
200: PRINT "Oh dear - you've cro
aked it": LET dead=1: RETURN
6507 PRINT "Thank goodness - a w
itch-doctor": RETURN
6510 PRINT "at a huge gateway":
RETURN
6520 PRINT "on a huge stone"," s
taircase ": RETURN
6530 PRINT "in a huge stone towe
r.","There is a giant"
6531 IF o(3,3)<>2 OR o(14,3)<>2
THEN  PRINT "He says 'Give me so
mething"," to write to King Arth
ur with'": LET ex=1: GO SUB 2300
: RETURN
6532 PRINT " scribbling merrily"
: RETURN
6539 RETURN
```

```
6540 PRINT "in a narrow valley":
 RETURN
6550 PRINT "on a drive to a mans
ion": RETURN
6560 PRINT "in a large cage": RE
TURN
6570 PRINT "in an avenue of tree
s": RETURN
6580 PRINT "outside a laundry":
RETURN
6590 PRINT "on a road ";
6591 IF o(9,1)<>99 OR o(9,3)<2 T
HEN  PRINT "blocked by a  broken
 cart": LET ex=1: GO SUB 2300: R
ETURN
6592 LET ex=0: LET f$=e$(1)
6599 RETURN
6600 PRINT "in a glade ": RETURN

6610 PRINT "in a study. There is
 ","an old man dressed only in",
"a towel.  He shouts 'Have you",
"brought me some clothes?'"
6612 IF o(9,1)<>99 AND o(15,1)<>
99 THEN  PRINT "NO!!! A lightnin
g bolt ","flashes from his finge
rs and ","you suddenly feel a li
ttle ashen": LET dead=1: RETURN
6614 IF o(9,3)<>5 AND o(15,3)<>3
 THEN  PRINT "Give them to me"
6616 IF o(9,3)<>5 OR o(15,3)<>3
THEN  PRINT "What about the rest
?": LET ex=1: GO SUB 2300: RETUR
N
6618 PRINT "O.K. You can pass":
LET ex=0: LET f$=e$(1)
6619 RETURN
6620 PRINT "in an armoury. "''"T
here is a suit of armour ","agai
nst one wall": RETURN
6630 PRINT "at Camelot .Well don
e": LET dead=2: RETURN
6640 PRINT "at the gate to Camel
ot. "
6642 IF o(11,3)<2 THEN  PRINT "T
he gatekeeper shoots at you.","
Oh dear, no armour....": LET dea
d=1: RETURN
6645 PRINT "The gatekeeper lets
you through": RETURN
6650 PRINT "beneath a sheer clif
f": IF o(10,3)<3 THEN  LET ex=1:
 GO SUB 2300: RETURN
6655 LET ex=0: LET f$=e$(1): RET
URN
6660 PRINT "on a golden road": R
ETURN
6670 PRINT "inside a laundry. "'
'"There's a basket in one corner
": RETURN
6680 PRINT "inside a library": R
ETURN
7998 RETURN
7999 REM *********************
8000 REM setup
8001 REM *********************
8004 LET dr=0
8005 LET cart=0: LET head=0: LET
 rock=0: LET tree=0: LET lc=0: L
ET dead=0: LET g=0: LET ex=0: LE
T f$=""
8010 LET v$="lootakleagivligpeew
eafilempwricutblofixdresquswicli
lif"
8015 LET w$="n  s  e  w  norsouw
eseas"
8020 LET n$="monswofeacrooratorc
omhelwhestolethorwitscrbantresig
cupwolturroccagjesgiacargoojuity
rladrivcliaropacbasmaipeearmcoud
euchi"
8030 LET s$="Up you go": LET swi
m=0: LET q$="I see nothing speci
al"
8035 LET x$="n  s  e  w  "
8040 LET verb=0
8101 REM **********************
8102 REM ****PUT MAP IN MEMORY**
**
8103 REM **********************
8104 LET p=65367
8105 FOR i=1  TO 168
8108 READ A
8110 POKE (p+i),A
8150 NEXT i
8200 DIM y$(15,30): DIM o(37,3):
 DIM o$(15,13)
8210 FOR i=1 TO 15: READ y$(i,1
TO 30): NEXT i
8220 FOR i=1 TO 15: FOR j=1 TO 2
: READ o(i,j): NEXT j: NEXT i
8230 FOR i=1 TO 15: READ o$(i,1
TO 13): NEXT i
8240 FOR i=1 TO 15: READ o(i,3):
 NEXT i
8490 PAPER 7: INK 0: CLS
8498 RETURN
8499 REM *********************
8500 REM instructions
8501 REM *********************
8520 BORDER 5: PAPER 1: INK 5: C
LS
8530 DRAW 0,2: DRAW 1,4: DRAW 2,
5: DRAW 4,5: DRAW 5,6: DRAW 6,8:
 DRAW 7,9: DRAW 8,11
8535 DRAW 8,20: DRAW 1,-10: DRAW
 1,70: DRAW 4,-20: DRAW 10,0
8540 DRAW 1,15: DRAW 5,0: DRAW 1
,10: DRAW 0,30
8545 DRAW 0,-40: DRAW 2,-30: DRA
W 10,-15: DRAW 0,-15: DRAW 2,0:
DRAW 0,-20: DRAW 40,-15
8550 DRAW 10,-20: DRAW 40,-10
8560 PLOT 48,70: DRAW 0,8: PLOT
60,99: DRAW 0,6
8590 PRINT AT 10,18; INK 6;"CAME
LOT"
8595 INK 6
8599 PAUSE 1000
8600 BORDER 1: CLS
8605 PRINT AT 2,12; PAPER 5; INK
 1;"CAMELOT"
8610 PRINT AT 5,3;"At Camelot is
 fulsome day "
8611  PRINT AT 8,3;"Your lights
will help you               find t
he way"
8612  PRINT AT 11,3;"Solve the p
uzzles, watch for          clues
 "

8613  PRINT AT 14,3;"Play with w
ords to get                  good
news "
8614  PRINT AT  17,3;"Two word o
rders help me play "
8615  PRINT AT  20,3;"A verb and
 then a noun's O.K. "
8620 PAUSE 500
8621 CLS
8622 PRINT AT 1,1;"You can use t
he single letters","  N, S, E, a
nd W for movement.",,,
8624 PRINT "    And the words 'h
elp' and ","     'inventory' can
be used","            on their own
."",,,            .
```

```
8628 PRINT "     All other comman
ds must be"," two words, and all
 words should ";"        be lowe
r case."
8650 PRINT AT 16,10; PAPER 6; IN
K 1;"INITIALISING"
8652 PRINT AT 18,10; FLASH 1;"Pl
ease wait"
8999 RETURN
9047 REM *********************
9048 REM data for map
9049 REM *********************
9050 DATA 0,0,0,0,0,0,0,0,0,0,0,
0,0,0
9100 DATA 0,0,1,2,3,4,5,6,7,8,9,
0,0,0
9110 DATA 0,0,10,0,0,0,0,0,11,0,
12,0,0,0
9120 DATA 0,0,13,0,0,0,14,0,15,0
,16,0,0,0
9130 DATA 0,17,18,19,20,21,22,23
,24,25,26,0,0,0
9140 DATA 0,0,27,0,28,0,29,0,30,
0,31,32,33,0
9150 DATA 0,34,35,0,36,0,37,0,38
,0,39,0,40,0
9160 DATA 0,0,41,0,42,43,44,45,4
6,47,48,49,50,0
9170 DATA 0,0,51,52,53,0,54,0,0,
0,55,0,56,0
9180 DATA 0,0,0,0,57,0,58,59,60,
61,62,0,0,0
9190 DATA 0,63,64,65,66,0,67,0,0
,68,0,0,0,0
9195 DATA 0,0,0,0,0,0,0,0,0,0,0,
0,0,0
9199 REM Data on objects
9200 DATA "chattering","on the f
loor"," fluttering in the breeze
"
9210 DATA "on the ground",",big
and juicy","","in one corner"
9220 DATA "on the floor"," on th
e ground","nearby ","in a muddle
d heap"
9230 DATA "on the ground"," squa
tting in one corner",", tattered
 and torn","piled on the floor"
9297 REM ********************
9298 REM Initial object location
s and name lengths
9299 REM *********************
```

```
9300 DATA 2,9,9,8,56,9,29,10,34,
9,39,8,28,10,62,9,49,8,67,11,68,
10,46,7,60,13,14,9,57,10
9397 REM *********************
9398 REM Object names
9399 REM *********************
9400 DATA "a monkey","a sword","
a feather","a crowbar","an orang
e","a torch","a compass","a helm
et","a wheel","a stocking","lett
ers","a horn","a witchdoctor","a
 scroll","bananas"
9409 REM *********************
9410 REM data on hidden objects
9411 REM *********************
9420 DATA 0,1,0,1,0,1,1,0,1,0,1,
0,1,0,0
9595 INK 6
9897 REM ********************
9898 REM end
9899 REM ********************
9900 GO SUB 3010
9995 PRINT "Another game?"
9996 IF INKEY$="" THEN  GO TO 99
96
9997 IF INKEY$="y" OR INKEY$="Y"
 THEN  RUN
9999 STOP
```

6 THE COMBAT GAME

The key part of most adventure games is the combat system. Many text-based puzzle games do not have combat, but most of the games with graphics or features other than the central puzzle involve combat. We have already considered some aspects along the way as we have explored the characteristics of the player's character and of the monsters which might be encountered. Most of the beings we build into our game have some combat rating. Indeed, at its simplest you could say that monsters are only combat ratings, existing to test the character/player, to provide a problem to be overcome. It makes sense, therefore, to have a good idea of your combat system before you design the player character or monsters, otherwise you might find your intended system cannot use the values you have set up.

6.1 Values and choice

To see what is involved in a game combat system let us proceed as we have before—start with a simple system and gradually add to it. Suppose therefore that Sir Jon were given just one value, 7, and all monsters were given a value in the range 1 to 9. Combat could then be a simple comparison of the two values, so that whenever Sir Jon met a monster his value was compared with the monster's value and the one with the lower value was killed.

Our flowchart might thus contain the stages:

```
IF Sir Jon's value > monster's value THEN monster dead (= set to
Ø)

IF monster's value >= Sir Jon's value THEN Sir Jon dead (= set to
Ø)
```

We can see at once that problems arise with such a simple system. In the first place, the player does not do anything. Cad has no control over the situation, simply sitting back and watching his hero kill or keel over. Secondly, there is no uncertainty, so no risk or reward. Every time Sir Jon meets a monster of 7 or more, he is killed; otherwise the monster is killed. This would be very boring and little fun to play, even if the graphics involved were exciting.

A simple way to add uncertainty would be to randomize the monster's value. However, if it was totally random we would have the opposite result. There would be no predictability at all. The player would never know whether to fight or run, and the whole game would depend on purely random numbers. It would be very unsatisfactory if a great deal of points were lost because of a single random number.

The answer, therefore, is to restrict the range of values for a particular monster, within which it can be random. Suppose we have three monsters—a goblin could be values 1 to 7; a troll values 3 to 8; and a giant values 6 to 9. This means that Sir Jon's chances of beating any goblin are 6 out of 7; of beating a troll are 2 in 3; and of beating a giant are only 1 in 3. Once the player has learned this through frequent encounters he can take his chances with a sense of the risk and potential reward in each situation.

We can, of course, vary the configuration of monsters in other ways. For example, we could vary the chance of it being of a particular value. The goblin above has a 1 in 7 chance of being value 1, a 1 in 7 chance of being value 7, and a 1 in 7 chance of being value 4 (i.e., equal probability of each value). More realistic, and also more predictable, would be a greater chance of average value, as shown on the graph in Fig. 6.1.

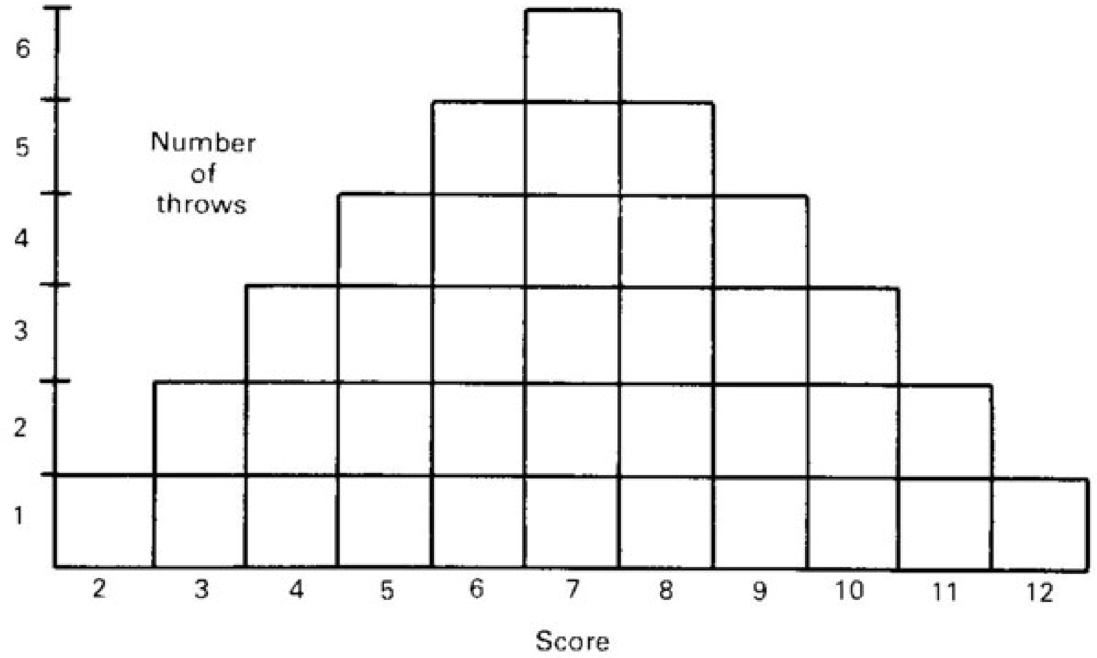


Figure 6.1 Normal distribution curve for throwing two dice.

This is a bar graph which shows 36 throws of two six-sided dice. The resulting curve is what is known as a normal distribution. Out of a total of 36 throws, only 1 would be 2, 2 would be 3, 4 would be 4, 2 would be 11, etc. The extremes are less likely than the middle, but nevertheless possible. We could get such a curve with any range of numbers we liked. To get one in the range 1 to 7 for our goblin we pick a random number in the range 0 to 3 and add it to another in the range 1 to 4. In Spectrum BASIC:

```
100 LET R = INT(RND * 3)
11Ø LET S = INT(RND * 3)+1
12Ø LET MONSTER = R+S
13Ø IF SIRJON > MONSTER THEN LET MONSTER = Ø
```

We can devise formulae to assign just about any probability to any value in any range, which makes for great variety in monsters, though usually linear probability (equal chance) and normal distribution are the only curves used. The important thing to remember is that it should be balanced so that the player has at least some chance of defeating it. Chapter 11 includes a routine to allow different ranges of random number generation for use in non-micro adventures which can easily be adapted for use in micro games themselves.

Let us return to the other problem, that of player interaction. All we have added so far is the possibility that the player, learning from previous games, may choose not to fight a particular monster. This means that he must be given either prior warning of the monster or given the opportunity of retreating when the monster is encountered. It is common for warnings to be available in one of two ways. Either the character has some device for looking a small way ahead, such as a lamp, a magical orb or radar, or a warning is flashed up on the screen that 'a monster is near', but no indication is given of how near or in what direction. A lamp routine is included in the game at the end of this chapter. It is a simple matter, however, to program a warning routine. The stages are as follows:

1. INSPECT THE CONTENTS OF ALL ADJACENT LOCATIONS
2. IF NO LOCATION CONTAINS A MONSTER THEN END THE ROUTINE
3. IF ONE OR MORE LOCATIONS CONTAIN A MONSTER THEN PRINT A WARNING

The warning need not be in words, of course. It could be a flashing danger sign, for example, or it could use the sound facilities of your micro if it has them. If a player has to respond to visual and aural signals as well as words, he has much more to learn and to pay attention to in play.

Retreat can be written into encounters as a combat option. A common mechanism is to program combat in two stages:

1. ASK THE PLAYER WHETHER HE/SHE WISHES TO RETREAT OR NOT
2. IF THE PLAYER ATTACKS PRINT THE VARIOUS OPTIONS AVAILABLE, INCLUDING RETREAT

If the character retreats then the monster probably has a chance of hitting him in the back (what players of war games refer to as a 'free blow') and this chance can be dependent on the relative speeds or dexterities of the character and monster. Retreat might automatically be to the previous location, in which case this might be remembered by the program, usually by setting a flag, or can be in a random direction, in which case the character may blunder into another monster's lair or worse peril.

This gives the player some choice, but not much. Other variables are needed, and these can be added by considering the variables in a real combat. So far we have looked at the relative skill of character and monster, and at the option of attacking or retreating. We can further modify both of these, because fighters generally have different classes of skill. A broad distinction can be made between 'attacking skills' and 'defending skills', and these can be further broken down, e.g., into missile skills, edged-weapon skills, bludgeoning skills, manual dexterity, and so forth. In our games we will stick to attack and defence.

Similarly the attack and retreat options could be supplemented by other options such as the use of magic, the opportunity of talking to the opponent, the possibility of bribery, and so forth. We will see whether all of these options can be included in our combat game, but will not take any of them further for the moment.

In addition to skill and a broad choice of strategy the character may also have a choice of weapons and/or armour. These can be offered at the very beginning when the game is being set up, or may be found or bought en route (as local rewards), or the player may have a permanent selection that he carries with him. We will use sword, spear, flail, and knife as weapon choices, and have different pieces of armour, including a shield. Whereas all the armour can be carried at once, only one weapon can be used at once, so the weapon will be a fluctuating variable whereas the armour will be a cumulative variable giving a cumulative improvement in defence.

All this armour would be rather heavy, so carrying it will also entail a cumulative penalty we can call fatigue. It is also possible to use weapons in different ways, such as slashing, thrusting, piercing, etc. We will allow a simple tactical choice for all weapons, namely 'up' (raises the arm) and 'forward'. We can regard 'up' as essentially defensive but good against large enemies and flying enemies, whereas 'forward' is essentially offensive, being good against smaller enemies and enemies on the ground. Shield up is the best defence but shield forward adds to the attack value.

Finally, certain environments might affect certain weapons or strategies, and certain enemies may well be defended against certain attacks. Let us decide that the monsters encountered will have their own range of armour and weapons as well as different preferred strategies, so the player will have to learn by experience what works best with particular opponents.

There are two reasons for having so many different levels of complexity. The first is that it gives the player a large number of possible tactical options and he must learn which give the best chance against a particular opponent in a particular situation. This means that the game will be a challenge for a long time, with a great deal to learn. The second is that all these variables can be built into other aspects of the game, so that combat is not simply a separate and independent aspect. For example, different magical objects may improve certain skills or weapons, or reduce certain enemy attacks. It may also be possible to lure an enemy into a place where the player knows a tactic is very successful, e.g., out into the open where a bow can be used. The character's chance in a particular combat will depend on whether he has managed to obtain the correct armour or weapon for a particular combat.

Before examining how we might turn all this into BASIC, let us look at monster combat. Monsters could be given a set of choices, weapons, and armour identical to those available to the character. If so, then each moment of combat could be determined by a series of random choices. It is more interesting, however, to give different monsters different strategies, so that trolls, for example, tend to attack with 'flail up', whereas goblins tend to use 'sword forward'. Trolls usually have helmets; goblins usually have hauberks. In such a way no combat can have a certain result. The player can learn that trolls tend to do certain things, but cannot be certain that any particular troll will do that.

It is also a good idea to give monsters a few options which are not normally available to the player. A dragon might be able to breathe fire, a griffon might be able to fly and attack from above, a magician might be able to use a magical wand. If monsters are given wands, weapons, and armour it should be possible for a victorious character to take such objects from the dead. This is an obvious form of direct reward for success.

Now, how do we code all this?

Initially we must decide how much information is available to the player. At one extreme no information might be given, as in a puzzle game, and the player would be left to discover what weapons, skills, and monsters were around and what they did. At the other extreme instructions could list all possible alternatives and which to use when. Both of these seem foolish, the former because the player would become bored before learning sufficiently to be able to play and the latter because there is no discovery, no learning to do. So it would seem best to give some information to the player and make him discover and learn the rest. So in our general instructions we will put the following:

> This game involves a choice of weapon, such as sword or spear and a choice of tactic, such as 'retreat' or 'sword up'. You can improve both your attacking and defensive abilities, especially by collecting pieces of armour. There are many monsters to be encountered. Each monster prefers a particular weapon and tactic, but they do not always use it.

During each fight the screen will display the current weapon, tactic, and armour of both opponents, together with their basic offensive and defensive skills, so that the player will gradually become familiar with the range of options and variations, and gradually be more able to predict what is likely to occur. In broad terms, this is how people learn to fight in reality, by knowing what is possible, what is likely, and what can be done when. However, we will not display the calculations that are being used, so the player can only observe the effects of his actions and not the hidden causes which are buried in the program's functions and routines.

6.2 The Mines of Merlin

Let us now plan a second adventure game which gets rid of some of the faults of the puzzle game, The Throne of Camelot. These problems are:

1. It is fixed in form, so does not change each time it is played.
2. It uses a great deal of RAM.
3. It does not allow much overall strategy and has no combat elements, unlike non-micro adventures.

What we will try to design is a game which uses only a relatively small amount of memory, will vary from game to game, and will allow some strategy and combat.

We will use the seeded random map method because this holds all the information on the adventure in one function, avoiding the necessity to hold a complex map representation separately in memory. We will situate the adventure entirely underground in the conventional way, so we shall want 'up' and 'down' as commands in addition to the normal N, S, E, and W. Let us call this adventure The Mines of Merlin.

The function P=SQR(X*X+Y*Y*Z) will be used to determine the features of any location, where Y is the north–south dimension, X is the east–west dimension, and Z is the up–down dimension. As we want a different game each time we shall start with different values for X and Y (but not Z, because we always start at ground level with Z=1). Each time the player moves the function is used to determine what exits are available from the new location, and only movement in those directions is then allowed. So movement only involves changing X,Y, or Z each time and recalculating the function, a subroutine we will house at lines 1ØØØ to 15ØØ as it will be used quite a lot.

However, the same function can be used to tell us a great deal more about each location. What the function returns is the non-integer part of a number (the part after the decimal point). The Spectrum gives us eight digits after the decimal point. If we take the value of 0.4 as indicating a connection between two adjacent locations, and we define a function accordingly:

```
DEF FNp() = SQR(X*X+Y*Y*Z)—INT(SQR(X*X+Y*Y*Z))
```

then we can use this function with suitably altered parameters to test for available exits, as in lines 1ØØØ to 222Ø of the program. The complete listing is given at the end of this chapter.

However, the formula can also be used to give other information about a particular location. What the function is generating is a pseudo-random number which is nevertheless always the same for given values of X, Y, and Z. We can therefore use it as we would any other random number, to determine if treasures are at that location, or monsters, or if particular events are likely (and what their likelihood is).

For example, suppose there are five chances in a hundred of gold pieces at each location. We take any portion of the eight digits we have at a particular location, turn it into a value between one and a hundred, and see if it is above the limit of five. Thus if we decide on digits one and two of the number and the number is 0.044 856 47 we need to do the following:

1. Multiply by 100 to give 04.485 647.
2. Get rid of the decimal part to give 04, i.e., 4.
3. See if this number is less than 5, which it is.
4. Interpret the result—there are some gold pieces here.

We would then want to find out how many gold pieces there are, for which we might use digits five, six, and seven. So here we do the following:

1. Multiply by 10 000 to give 0448.564 7.
2. Get rid of the integer part to give 0.564 7.
3. Multiply by 1000 to give 564.7.
4. Get rid of the decimal part to give 564.
5. Interpret the result— there are 564 gold pieces here.

Naturally if there are no gold pieces here, digits five, six, and seven could be used for other purposes, e.g., to calculate the strength of an encountered monster or as a variable for sending control to a subroutine. If a digit or set of digits is used for a particular purpose, such as calculating the amount of gold, they can still be used for other purposes, but remember that the two things will be connected. So if the same digits are used to control a routine which turns the player into a dwarf, and the key test is whether digit five is greater than four, each time more than four hundred gold pieces are found the player will be turned into a dwarf. This can be used to advantage, however. For example, suppose we wanted to make the reward for overcoming a monster directly related to the power of the monster; we can use the same digits in the random number for calculating both variables.

The general routine to do this testing and calculation is held at lines 2800 to 2820.

We will keep our game relatively simple. We will not have a map display as there is no map, but will just give a verbal account of the exits from each location. It will thus be a simple random wandering game with a single aim— to win as many points as possible. However, we will employ a number of interrelated variables so that each player decision has to take more than one factor into account. These will be his physical powers, his magical powers, the monsters' physical and magical powers, the weight of items carried, his general constitution, and the value of treasure.

As we are using a fixed formula it will not be possible to move objects around the map because we cannot change the number in each location. We could keep a separate record of objects and locations but our aim is to save memory so we will have to accept a major limitation, namely, that in any given location the same kind of event will recur irrespective of what previously occurred there. So if location eleven has 247 gold pieces on its first visit, it will also have them on all subsequent visits, even if the player takes them!

So we must use the numbers provided by the formula not to give fixed values or routines but conditional ones. For example, if the location is to have a treasure routine we can set it up so that sometimes treasure is found there and sometimes it is lost. Consequently the player will have to decide whether it is worth the risk to retrace his steps.

Let us set up eight types of routine in our game, called by the random number which also maps the dungeon. These are:

1. Monsters
2. Treasure
3. Magical treasure
4. Potions
5. Weapons
6. Accidents
7. Torches
8. Food

This means that the required player variables affected by these possible events must include:

1. Strength	C(1)
2. Skill	C(2)
3. Constitution	C(3)
4. Knowledge	C(4)
5. Food	C(5)
6. Maximum burden	C(6)
7. Money	C(7)
8. Magic bolts	C(8)
9. Potions	C(9)
10. Weapon value	C(10)
11. Armour value	C(11)
12. Burden	C(12)

In addition we want to relate these to each other and give the player the chance of setting up his own configuration of variables as the basis of his play. The usual way to do this is to let the player select three or four basic variables (within set ranges) on which all the other values will depend. Once these have been established the player is allowed to 'buy' equipment according to his idea of the way he will play. So we will start with the variables STRENGTH, SKILL, CONSTITUTION, and KNOWLEDGE. Each can be in the range 1 to 20, and the player will be able to assign 12 of those points in addition to the base value of 8 which each is set to.

Furthermore, the player starts with a hundred silver coins which he can spend on equipment. When this initial configuration has been set up the program will calculate all the values to fit in the player variables according to the player's initial choice.

It should be clear that this is one of the simplest designs for an adventure game, yet it is still fun to play. The rest of this chapter will outline the main blocks of the program in turn and then discuss the kinds of modification that can be made to improve it. You will find other alterations that could be made in other chapters, and the best adventure would actually combine them all, together with some form of puzzle game. The trouble is that every improvement takes memory. At present we are working on a minimal interesting game.

We start with a description of how the player's character variables will be set up. C(7) is straightforward—every silver piece adds to the total. Magical bolts will be a similar total of available spells. Potions will be a similar inventory, each potion when drunk increasing one of the basic four characteristics of the character. Weapon value will be set to the value of the best weapon found so far and will add to or subtract from attack or defence.

The burden is the total of C(7) + C(8) + C(9) + C(10) + C(5) + C(11), while the maximum burden is the product of STRENGTH and CONSTITUTION. Food is a simple number depleted when eaten. Eating raises CONSTITUTION temporarily. The attack value will be a formula derived from STRENGTH, SKILL, tactic, and weapon. Defence value is also based on STRENGTH, SKILL, tactic, and weapon. Magical attack is similarly based on KNOWLEDGE and SKILL.

The game will be divided into a number of turns, monitored by the variable COUNTER, so that exhaustion can be calculated. Every eight turns the player will lose a number of CONSTITUTION points which depend on the amount being carried expressed as a proportion of the weight able to be carried. If CONSTITUTION falls below one the character dies. Every turn all values are updated, including the carriable weight and the weight carried, so the program can check on the current degree of exhaustion. Eating will increase CONSTITUTION, but carrying food will increase the burden. So the player must always balance the value of the articles he or she carries, and their potential value, against the cumulative penalty of carrying them. The basic problem thus turns out to be the same as in the puzzle game—what to carry where and when.

Having sketched the character's configuration we can go on to the eight subroutines. 'Monster' should be easy to design, if not to code. The player will find a location containing a monster. He or she must decide whether to retreat or fight. If retreat is chosen the monster gets a free blow at the back of the character. If fight is chosen, we must work out who gets the first blow (i.e., who has surprise), for which we will use the SKILL variable. Having decided on the order for blows, the player and monster choose in turn whether to use magic or physical combat, each blow being calculated by comparing the appropriate attack value with the appropriate defence value, modified by a random factor.

This means that we need a table of monster variables containing the name of the monster, its attack and defence values, and its SKILL value. Also it will need CONSTITUTION so that we can calculate when it is killed. Finally, as the purpose of the game is to accumulate points, we need a formula for calculating the value of each monster killed. To keep this simple, we will make the formula the sum of all its values before combat begins plus a small random factor.

'Treasure' must also be either positive or negative, so that the player takes chances. We will scatter a number of thieves' dens throughout our dungeon. Sometimes the thieves will be in, in which case they will steal something from the player. At other times they will not be there, so the player can take some of their treasure.

'Magical treasure' will be relatively rare but scattered all over the place. On the positive side it is able to aid magical combat, but some treasure is heavy and some is cursed (i.e., it will help the opponent). It is easy enough for the player to discover the weight of a treasure, but he or she can only find out if it is cursed by attempting to use it in combat.

'Potions' like magical treasure will be found all over the place but may be cursed or advantageous. They will increase or reduce one or more of the four basic abilities.

'Weapons' may also be beneficial or harmful, of different combat values, and may have to be paid for. Sometimes they will be on their own, but often they will be owned by someone who wants to sell them.

'Accidents' will include a miscellaneous set of subroutines which can range from falling down a pit (so losing CONSTITUTION) to breaking a weapon, to falling in love with a sorceress, and to gaining maximum KNOWLEDGE.

The 'torch' routine will simply give the character a torch or deprive him of one. The torch can be used for discovering the nature of the next room before entering it. However, some torches are illusory, so might not reveal the truth.

The 'food' routine will produce locations in which food can be found. It will be of different weights, and some may have a mild poison in it. Other food may have magical effects, which could raise the magical combat value of the player, or reduce one or more values.

Other types of routine like these could be added indefinitely until all available memory is consumed. Each should potentially affect one or more of the player variables, but only in extreme cases should the result be instant death. As such games are fundamentally random the key to making them interesting is to add as many features and variations and interrelations as possible so that there are always several things for the player to think about. Any small modifications which you can invent which have this effect will improve the game. Suggestions are given in later chapters, such as talking and bribery.

The Mines of Merlin

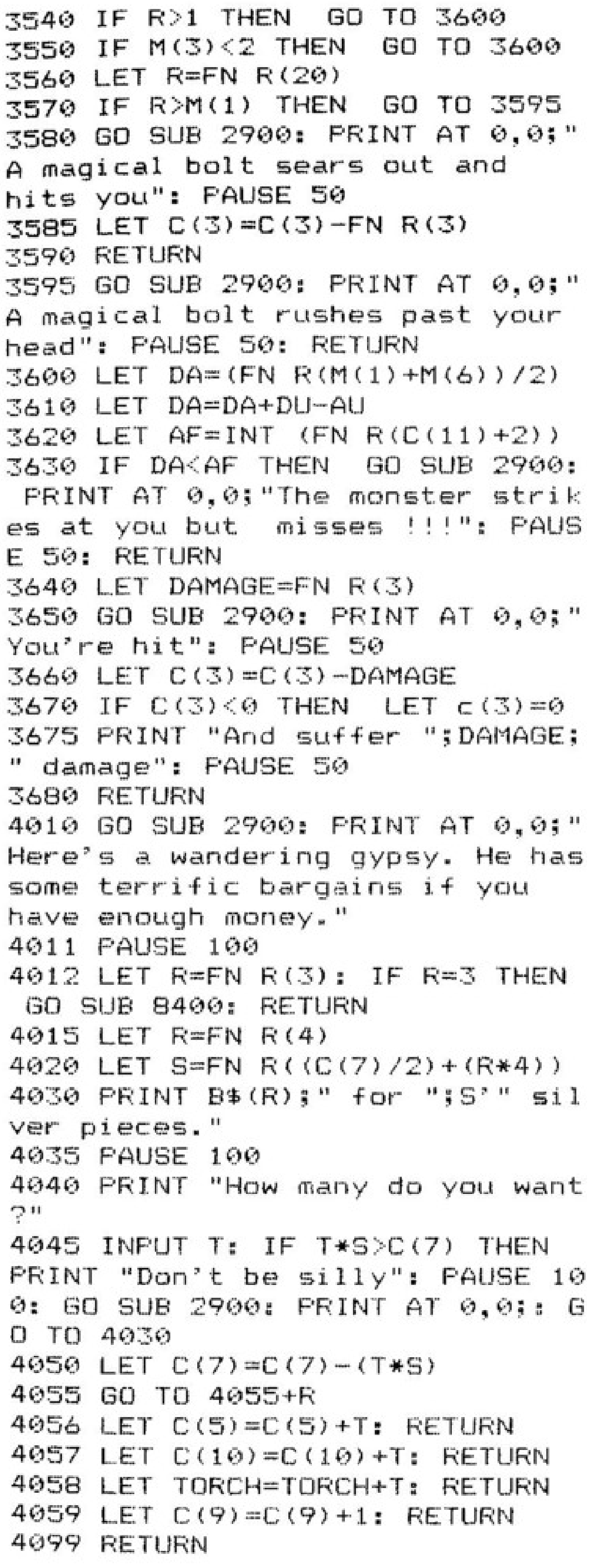
```
   5 REM 2950=validate
   6 REM 2900=cls
  35 REM INSTRUCTIONS
  40 GO SUB 8500
  45 REM INITIALISE
  50 GO SUB 8000
  55 REM START OF MAIN LOOP
  59  REM PRINT UPDATE
  60 LET COUNTER=COUNTER+1
  61 IF DEAD=1 THEN  GO TO 9900
  62 LET TURN=TURN+1
  65 GO SUB 2500
  69 REM INSTRUCTIONS
  70 GO SUB 500
 490 GO TO 60
 491 REM END OF MAIN LOOP
 495 STOP
 499 REM INPUT
 500 GO SUB 2900
 504 REM MOVEMENT
 505 PRINT AT 0,8;"WHAT NOW?"
 508 PRINT  PAPER 6; INVERSE 1;"
D"; INVERSE 0;"rink,"; INVERSE 1
;"E"; INVERSE 0;"at,"; INVERSE 1
;"L"; INVERSE 0;"eave an object,
", INVERSE 1;"M"; INVERSE 0;"ove
,"; INVERSE 1;"R"; INVERSE 0;"es
t or use a "; INVERSE 1;"T"; INV
ERSE 0;"orch?",
 510 PRINT ''"Please type the ap
propriate","initial letter."
 520 LET T$="MRLEDT"
 525 GO SUB 2950
 530 GO TO 466+(CODE A$)
 534 GO SUB 5300: RETURN : REM D
RINK
 535 GO SUB 5200: RETURN : REM E
AT
 542 GO SUB 5110: RETURN : REM L
EAVE
 543 GO SUB 1000: REM MOVE
 544 LET N=1000
 545 GO SUB 2800: IF MOVE=0 THEN
  RETURN
 546 GO SUB 4000+(Q*100): RETURN

 548 GO SUB 5400: RETURN : REM R
```

```
EST
 550 GO SUB 5500: RETURN : REM T
ORCH
 999 REM input commands
1000 REM
1045 LET MOVE=0
1050 IF TORCHFLAG=2 THEN  LET TO
RCHFLAG=0: RETURN
1051 IF MOVE=2 THEN  LET MOVE=0:
 PRINT "SOLID ROCK": RETURN
1052 GO SUB 2900
1055 PRINT AT 0,8;"WHICH WAY?"
1058 LET T$="NSEWUD": GO SUB 295
0
1090 IF A$(1)="N" THEN  LET X=X-
1: GO SUB 2000: IF MOVE=2 THEN
LET X=X+1: GO TO 1050
1100 IF A$(1)="S" THEN  LET X=X+
1: GO SUB 2000: IF MOVE=2 THEN
LET X=X-1: GO TO 1050
1110 IF A$(1)="E" THEN  LET Y=Y+
1: GO SUB 2000: IF MOVE=2 THEN
LET Y=Y-1: GO TO 1050
1120 IF A$(1)="W" THEN  LET Y=Y-
1: GO SUB 2000: IF MOVE=2 THEN
LET Y=Y+1: GO TO 1050
1125 IF A$(1)="U"  AND Z=1 THEN
 PRINT "YOU'RE ON GROUND LEVEL!"
: GO TO 1050
1130 IF A$(1)="U" THEN  LET Z=Z-
1: GO SUB 2000: IF MOVE=2 THEN
LET Z=Z+1: GO TO 1050
1140 IF A$(1)="D" THEN  LET Z=Z+
1: GO SUB 2000: IF MOVE=2 THEN
LET Z=Z-1: GO TO 1050
1998 RETURN
1999 REM MOVEMENT
2000 GO SUB 2200: IF MOVE=2 THEN
  RETURN
2002 IF TORCHFLAG=1 THEN  LET MO
VE=2: LET TORCHFLAG=2: LET N=100
0: GO SUB 2800: RESTORE Q+9201:
READ V$: PRINT "There is ";V$;"
here": RETURN
2005 GO SUB 2900: PRINT AT 0,0;"
EXITS": LET MOVE=0
2010 LET X=X-1: GO SUB 2200: IF
MOVE=1 THEN  PRINT "NORTH"
2020 LET MOVE=0: LET X=X+2: GO S
UB 2200: LET X=X-1: IF MOVE=1 TH
EN  PRINT "SOUTH"
2030 LET MOVE=0: LET Y=Y-1: GO S
```

```
UB 2200: IF MOVE=1 THEN  PRINT "
WEST"
2040 LET MOVE=0: LET Y=Y+2: GO S
UB 2200: LET Y=Y-1: IF MOVE=1 TH
EN  PRINT "EAST"
2050 LET MOVE=0: LET Z=Z+1: GO S
UB 2200: LET Z=Z-1: IF MOVE=1 TH
EN  PRINT "DOWN"
2060 IF z>1 THEN  LET MOVE=0: LE
T Z=Z-1: GO SUB 2200: LET Z=Z+1:
 IF MOVE=1 THEN  PRINT "UP"
2200 LET P=FN P(): IF P<.35 THEN
  LET MOVE=2: RETURN
2210 LET MOVE=1
2220 RETURN
2499 REM PRINT UPDATE
2500 FOR I=1 TO 12
2502 IF C(I)<0 THEN  LET C(I)=0
2505 NEXT I
2510 LET COUNTER=COUNTER+1
2515 LET FIGHT=C(2)+C(1)+C(10)
2516 LET POINT=(MK*5)+C(7)+(TURN
*2)
2520 LET MAGIC=C(4)+C(2)
2525 LET C(6)=C(1)*C(3)*3
2530 LET C(12)=C(7)+(C(8)*5)+C(9
)+(C(10)*5)+(C(5)*2)+(C(11)*4)
2535 IF C(12)>C(6) AND F<>1 THEN
  GO SUB 2900: PRINT AT 0,0;: PR
INT "You're carrying too much. Y
ou   must drop something.": GO S
UB 5100: LET DROP=1
2540 LET FATIGUE=INT (((COUNTER/
8)*(C(12)/12))/(C(1)))
2545 LET C(3)=C(3)-FATIGUE
2546 IF FATIGUE>.5 AND F=0 THEN
 GO SUB 2900: PRINT AT 0,0;"You'
re feeling exhausted"
2550 IF C(3)<1 THEN  PRINT "You
collapse with fatigue and   that
's the end of you.": GO TO 9900
2552 LET C(6)=C(1)*C(3)*3
2553 LET C(12)=C(7)+(C(8)*5)+C(9
)+(C(10)*5)+(C(5)*2)+(C(11)*4)
2554 PRINT AT 15,12;"    ": PRINT
 AT 15,29;"    "
2555 PRINT AT 15,12;INT (TORCH+0
.9)
2556 PRINT AT 15,29;POINT
2557 FOR I=1 TO 4
2558 IF C(I)>20 THEN  LET C(I)=2
0
```

```
2559 NEXT I
2560 FOR I=1 TO 6
2565 PRINT AT 15+I,12;"    "
2570 PRINT AT 15+I,12;C(I)
2575 PRINT AT 15+I,29;"    "
2580 PRINT AT 15+I,29;C(I+6)
2585 NEXT I
2590 IF DROP=1 THEN  LET DROP=0:
 GO TO 2515
2699 RETURN
2799 REM TO SET CONTROLLING INTE
GER
2800 REM MOVES ALL IRRELEVANT IN
TEGERS BEYOND THE DECIMAL POINT
2805 LET Q=(P*(N/10))-INT (P*(N/
10))
2809 REM MOVES THE REQUIRED INTE
GER UP AND GETS RID OF THE REST
2810 LET Q=INT (Q*10)
2820 RETURN
2899 REM CLEAR TOP WINDOW
2900 PRINT AT 0,0
2905 FOR I=0 TO 8
2910 PRINT AT I,0;"
                             ";
2915 NEXT I
2918 IF TOPCLEAR=1 THEN  RETURN
2920 FOR I=9 TO 14
2923 PRINT AT I,0;"
                             ";
2924 NEXT I
2925 RETURN
2949 REM VALIDATE INPUT
2950 LET A$=INKEY$
2952 LET KT=0
2955 IF A$="" THEN  GO TO 2950
2960 IF CODE A$>90 THEN  LET A$=
CHR$ (CODE A$-32)
2964 PRINT AT 11,0;"You've typed
 ";A$
2965 FOR I=1 TO LEN (T$)
2970 IF T$(I)=A$ THEN  LET KT=1
2975 NEXT I
2980 IF KT=1 THEN  RETURN
2985 GO TO 2950
2998 RETURN
2999 REM COMBAT ROUTINES
3010 PAUSE 100: GO SUB 2900
3020 PRINT AT 0,0;"Do you want t
o Retreat, use a   magic Bolt or
 Physical combat?"
3030 LET T$="PBR"
```

```
3040 GO SUB 2950
3050 IF A$(1)="F" THEN  GO TO 33
00
3060 IF A$(1)="B" THEN  GO TO 32
00
3070 LET R=FN R(3)
3080 IF R>1 THEN  GO SUB 2900: P
RINT AT 0,0;"You escape unharmed
": LET RETREAT=1: RETURN
3090 GO SUB 2900: PRINT AT 0,0;"
You can't get past": LET RETREAT
=2
3100 RETURN
3200 IF C(8)<1 THEN  GO SUB 2900
: PRINT AT 0,0;"You have no magi
c bolts": PAUSE 5000: GO TO 3010
3205 LET R=FN R(40)
3210 LET C(8)=C(8)-1
3220 IF R>MAGIC THEN  GO TO 3260
3230 PRINT AT 0,0;"Your magic bo
lt sends the "'M$;" reeling"
3240 LET M(2)=M(2)-FN R(12)
3250 RETURN
3260 GO SUB 2900: PRINT AT 0,0;"
You miss": RETURN
3300 GO SUB 2900: PRINT AT 0,0;"
Weapon Up or Forward?"
3310 LET T$="UF"
3320 GO SUB 2950
3325 LET AU=-1
3330 IF A$="F" THEN  LET AU=1
3340 LET AD=FIGHT-M(1)
3350 LET AD=INT ((FN R(AD))/2)
3360 LET AD=AD+AU-DU
3370 LET DF=INT (FN R(M(5))/2)
3380 IF DF>AD THEN  GO SUB 2900:
 PRINT AT 0,0;"A miss": PAUSE 10
0: RETURN
3390 LET DAMAGE=INT (FN R(C(10)+
C(1))-FN R(M(5)))
3400 IF DAMAGE<1 THEN  LET DAMAG
E=1
3401 LET M(2)=M(2)-DAMAGE
3405 GO SUB 2900: PRINT AT 0,0;"
You cause ";DAMAGE;" damage"
3406 PAUSE 100
3410 RETURN
3499 REM MONSTER COMBAT
3500 LET R=FN R(M(3))
3510 LET S=FN R(M(4))
3520 LET DU=1
3530 IF S>1 THEN  LET DU=-1
```



```
3540 IF R>1 THEN  GO TO 3600
3550 IF M(3)<2 THEN  GO TO 3600
3560 LET R=FN R(20)
3570 IF R>M(1) THEN  GO TO 3595
3580 GO SUB 2900: PRINT AT 0,0;"
A magical bolt sears out and
hits you": PAUSE 50
3585 LET C(3)=C(3)-FN R(3)
3590 RETURN
3595 GO SUB 2900: PRINT AT 0,0;"
A magical bolt rushes past your
head": PAUSE 50: RETURN
3600 LET DA=(FN R(M(1)+M(6))/2)
3610 LET DA=DA+DU-AU
3620 LET AF=INT (FN R(C(11)+2))
3630 IF DA<AF THEN  GO SUB 2900:
 PRINT AT 0,0;"The monster strik
es at you but  misses !!!": PAUS
E 50: RETURN
3640 LET DAMAGE=FN R(3)
3650 GO SUB 2900: PRINT AT 0,0;"
You're hit": PAUSE 50
3660 LET C(3)=C(3)-DAMAGE
3670 IF C(3)<0 THEN  LET c(3)=0
3675 PRINT "And suffer ";DAMAGE;
" damage": PAUSE 50
3680 RETURN
4010 GO SUB 2900: PRINT AT 0,0;"
Here's a wandering gypsy. He has
some terrific bargains if you
have enough money."
4011 PAUSE 100
4012 LET R=FN R(3): IF R=3 THEN
 GO SUB 8400: RETURN
4015 LET R=FN R(4)
4020 LET S=FN R((C(7)/2)+(R*4))
4030 PRINT B$(R);" for ";S'" sil
ver pieces."
4035 PAUSE 100
4040 PRINT "How many do you want
?"
4045 INPUT T: IF T*S>C(7) THEN
PRINT "Don't be silly": PAUSE 10
0: GO SUB 2900: PRINT AT 0,0;: G
O TO 4030
4050 LET C(7)=C(7)-(T*S)
4055 GO TO 4055+R
4056 LET C(5)=C(5)+T: RETURN
4057 LET C(10)=C(10)+T: RETURN
4058 LET TORCH=TORCH+T: RETURN
4059 LET C(9)=C(9)+1: RETURN
4099 RETURN
```

```
4100 LET R=FN R(20)
4101 IF R>SQR (POINT)+5 THEN  GO
 TO 4100
4102 GO SUB 2900
4103 LET TOPCLEAR=1
4105 RESTORE 9249+R
4106 LET KM=R
4110 FOR I=1 TO 6
4115 READ M(I)
4120 NEXT I
4125 READ M$
4130 PRINT AT 9,0;"You see a ";M
$
4131 PAUSE 100
4132 LET RETREAT=0
4134 PRINT AT 10,0;"CONSTITUTION
 ";M(2): PRINT AT 10,17;"SKILL "
;M(1)
4135 LET R=FN R(M(1))
4140 LET S=FN R(C(1))
4145 IF S>=R THEN  GO TO 4155
4146 LET F=1: GO SUB 2500
4147 PRINT AT 11,0;"ARMOUR ";M(5
): PRINT AT 10,0;"CONSTITUTION "
;M(2);" ": PRINT AT 10,17;"SKILL
 ";M(1): PRINT AT 11,17;"WEAPON
";U$(DU+2)
4148 IF M(2)>0 THEN  GO SUB 3500
4149 GO SUB 2500
4150 IF RETREAT=2 THEN  LET RETR
EAT=0: LET F=0: LET TOPCLEAR=0:
RETURN
4155 IF C(3)>0 THEN  GO SUB 3000
4160 IF RETREAT=1 THEN  LET F=0:
 LET RETREAT=0: LET TOPCLEAR=0:
RETURN
4162 LET RETREAT=0
4165 IF C(3)<1 THEN  GO SUB 2900
: PRINT AT 0,0;"Oh dear! You're
dead.": PAUSE 100: LET DEAD=1: L
ET TOPCLEAR=0: RETURN
4170 IF M(2)<1 THEN  GO SUB 2900
: LET TOPCLEAR=0: PRINT AT 0,0;"
You kill him in no uncertain ter
ms": LET F=0: LET MK=MK+KM: LET
TR=M(2)+M(1)+(M(6)*FN R(3)): PRI
NT "He had ";TR;" silver pieces"
: PAUSE 100: LET C(7)=C(7)+TR: R
ETURN
4190 GO TO 4146
4199 RETURN
4205 GO SUB 2900
```

```
4210 PRINT AT 0,0;"You're in a t
hieve's den"
4215 LET N=100000: GO SUB 2800:
LET R=FN R(10)
4220 IF R>5 THEN  PRINT "The thi
ef is in": PAUSE 100: GO TO 4250
4225 PRINT "There's no-one here"
4230 IF Q<3 THEN  PRINT "and no
booty ": PAUSE 100: RETURN
4235 IF Q<6 THEN  LET R=FN R(100
): PRINT AT 0,0;"You find ";R;"
silver pieces": LET C(7)=C(7)+R:
 PAUSE 100: RETURN
4240 PRINT "You find a piece of
armour ": IF C(11)<9 THEN  LET C
(11)=C(11)+1: PAUSE 50: RETURN
4250 LET S=FN R(100): PRINT "He
wants ";S;" silver pieces"
4252 IF S<=C(7) THEN  GO TO 4270
4255 IF S>C(7) THEN  PRINT "But
as you haven't enough he'll take
 something else as well"
4260 IF C(11)>0 THEN  PRINT "a p
iece of armour": PAUSE 150: LET
C(11)=C(11)-1: GO TO 4270
4265 IF C(10)>0 THEN  PRINT "a w
eapon ": PAUSE 150: LET C(10)=C(
10)-1
4270 LET C(7)=C(7)-S: IF C(7)<0
THEN  LET C(7)=0: PAUSE 100: RET
URN
4298 RETURN
4299 REM TREASURE
4310 LET R=FN R(5)
4315 LET S=FN R(15)
4316 GO SUB 2900
4320 IF R=1 THEN  PRINT AT 0,0;"
You find ";INT (S/3)+1;" magic b
olts": LET C(8)=C(8)+INT (S/3)+1
: PAUSE 150: RETURN
4325 IF R=2 THEN  PRINT AT 0,0;"
You find ";S*5;" silver pieces":
 PAUSE 100: LET C(7)=C(7)+S*5: R
ETURN
4330 IF R=3 THEN  PRINT AT 0,0;"
You find a magical shield": PAUS
E 100: LET C(11)=12: RETURN
4335 IF R=4 THEN  PRINT AT 0,0;"
You're surrounded by pigmies who
run off with your weapons": LET
C(10)=0: PAUSE 100: RETURN
4340 PRINT AT 0,0;"You find a si
```

```
lver monster who   eats all your
 armour": PAUSE 100: LET C(11)=0
: RETURN
4398 RETURN
4399 REM POTION
4400 PRINT "POTION"
4410 PRINT "You find a potion"
4420 PRINT "Do you wish to take
it (T)        or drink it (D) or l
eave it       alone (L)?"
4430 LET T$="TDL"
4440 GO SUB 2950
4450 LET POISON=FN R(2)
4460 IF A$(1)="T" THEN  LET C(9)
=C(9)+1
4470 IF A$(1)="D" THEN  LET C(9)
=C(9)+1: GO SUB 5310
4498 RETURN
4499 REM WEAPON
4510 LET R=FN R(3): LET S=FN R(8
)+3
4515 IF R=1 THEN  LET CURSE=1
4520 LET R=FN R(3)+1
4522 LET R=INT (S/R)
4523 GO SUB 2900: PRINT AT 0,0;"
 ";
4525 PRINT "You find a "+W$(R)
4526 GO SUB 5000
4528 IF A$(1)="N" THEN  RETURN
4529 IF C(10)>R THEN  GO SUB 290
0: PRINT AT 0,0;"You already hav
e a better          weapon. Are you
 sure you want   it?": GO SUB 50
00: IF a$="N" THEN  RETURN
4530 IF R=5 THEN  LET C(11)=C(11
)+2: RETURN
4535 LET C(10)=R: RETURN
4598 RETURN
4599 REM ACCIDENT
4605 LET N=100000
4606 GO SUB 2800
4607 LET R=FN R(10)
4610 IF Q<3 THEN  PRINT "An empt
y cave": PAUSE 50: RETURN
4615 IF Q<5 THEN  PRINT "You fal
l down a pit": LET C(3) = C(3)-F
N R(3): PAUSE 100: RETURN
4620 IF R>5 THEN  GO TO 4630
4621 PRINT "You're caught in a w
eb": PAUSE 50: LET R=FN R(3): IF
 R = 3 THEN  PRINT "but escape w
ithout harm": PAUSE 50: RETURN
```

```
4622 IF R=1  THEN  PRINT " and y
ou're injured in escaping": LET
C(3) = C(3)-FN R(5): PAUSE 100:
RETURN
4624 IF R=2 THEN  LET S=FN R(C(1
) * 1.5): IF S<C(1) THEN  PRINT
"You escape but are weakened in
 the struggle": LET C(1)= C(1) -
 FN R(4): PAUSE 100: RETURN
4630 IF O<8 THEN  PRINT "a guill
otine descends from aboveand des
troys one of your        weapons
": PAUSE 100: LET C(10)=C(10)-1:
 RETURN
4650 PRINT "There is a magical w
aterfall     here. A few drops sp
lash you     and...": PAUSE 50
4655 LET R=FN R(3)
4660 IF R=1 THEN  PRINT "you fee
l much stronger": LET STR=STR+3:
 PAUSE 50: RETURN
4665 IF R=2 THEN  PRINT "it diss
olves your armour": LET C(11) =
0: PAUSE 50: RETURN
4670: PRINT "you feel weaker": L
ET C(3) = C(3)-FN R(5): PAUSE 50
: RETURN
4698 RETURN
4700 GO TO 4900
4798 RETURN
4799 REM NOTHING
4800 GO TO 4900
4900 GO SUB 2900: PRINT AT 0,0;"
You find an empty cave."
4998 RETURN
5000 PRINT "Do you wish to take
it?"
5010 LET T$="YN"
5020 GO SUB 2950
5030 RETURN
5100 GO TO 5111
5110 GO SUB 2900: PRINT AT 0,0;
5111 PRINT "Do you want to leave
:"
5112 PRINT "Money  (1)"
5113 PRINT "A weapon  (2)"
5114 PRINT "A magic item (3)"
5115 PRINT "A potion  (4)"
5116 PRINT "Food  (5)"
5117 PRINT "Armour  (6)"
5118 PRINT "Nothing   (7)"
5120 LET T$="1234567"
```

```
5122 GO SUB 2950
5125 GO TO 5125+(VAL (A$)*5)
5130 PRINT "How much?"
5132 INPUT  A: IF A>C(7) THEN  P
RINT Z$: GO SUB 2900: GO TO 5110
5134 LET C(7)=C(7)-A: RETURN
5135 IF C(10)<1 THEN  PRINT Y$:
GO SUB 2900: GO TO 5110
5136 LET C(10)=C(10)-1
5137 IF C(9)=0 THEN  LET CURSE=0
5138 IF CURSE=1 THEN  LET R=FN R
(3): IF R=1 THEN  LET CURSE=0
5139 RETURN
5140 IF C(8)<1 THEN  PRINT Y$: G
O SUB 2900: GO TO 5110
5142 LET C(8)=C(8)-1: RETURN
5145 IF C(9)<0 THEN  PRINT Y$: G
O SUB 2900: GO TO 5110
5147 LET C(9)=C(9)-1: RETURN
5150 IF C(5)<1 THEN  PRINT Y$: G
O SUB 2900: GO TO 5110
5152 LET C(5)=C(5)-1: RETURN
5155 IF C(11)<1 THEN  PRINT Y$:
GO SUB 2900: GO TO 5110
5157 LET C(11)=C(11)-1: RETURN
5160 RETURN
5199 REM EAT
5200 IF C(5)<1 THEN  PRINT "You
can't eat stone.": PAUSE 20: RET
URN
5210 LET C(5)=C(5)-1
5220 LET C(3)=C(3)+INT (FN R((20
-C(3))/2))
5298 RETURN
5299 REM POTION
5305 GO SUB 2900: PRINT AT 0,0;"
 ";
5310 IF C(9)<1 THEN  PRINT "You'
ve no potion": RETURN
5320 LET C(9)=C(9)-1
5330 IF POISON=1 THEN  LET P=FN
R(3)
5340 LET R=FN R(6): LET S=FN R(4
)
5345 IF P=1 THEN  LET POISON=0:
GO TO 5370
5360 GO SUB 2900: PRINT AT 0,0;"
Your ";C$(S);'"goes up ";R;" poi
nts ": LET C(S)=C(S)+R: RETURN
5370 GO SUB 2900: PRINT AT 0,0;"
Whoops!!! It was poisoned ! Your
 "'C$(S);" goes down ";R+1;" poi
```

```
nts": LET C(S)=C(S)-R
5398 RETURN
5399 REM rest
5410 LET COUNTER=1
5420 LET R=FN R(5)
5430 IF R=1 THEN  GO SUB 2900: P
RINT AT 0,0;"Your noisy breathin
g attracts a monster";: PAUSE 10
0: GO SUB 4100
5440 RETURN
5499 REM torch
5510 IF TORCH<0.1 THEN  PRINT Y$
: RETURN
5520 LET TORCH=TORCH-0.1
5530 LET TORCHFLAG=1
5540 GO SUB 1045
7998 RETURN
7999 REM SET UP VARIABLES FUNCTI
ONS AND ARRAYS
8000 RANDOMIZE
8001 LET DROP=0: LET MK=0
8002 LET KM=0: LET TOPCLEAR=0
8003 LET CURSE=0
8004 LET F=0
8005 LET MOVE=0
8006 LET TORCHFLAG=0
8007 LET TORCH=0
8008 LET POISON=0
8009 LET DEAD=0
8010 DEF FN P()= SQR (X*X+Y*Y*Z)
- INT (SQR (X*X+Y*Y*Z))
8011 LET POINT=0
8012 LET TURN=0
8015 LET H$="How many for"
8020 DEF FN R(x)=INT (RND*x)+1
8022 LET X=FN R(10)+10: LET Y=FN
 R(20)+10: LET Z=1
8025 LET Z$="You don't have enou
gh"
8030 LET Y$="You haven't any"
8031 DIM U$(3,7)
8032 LET U$(1)="UP"
8033 LET U$(3)="FORWARD"
8035 DIM W$(5,6)
8036 DIM B$(5,12)
8037 LET DA=0: LET AD=0
8038 LET AU=0: LET DU=0
8039 LET AF=0: LET DF=0
8040 DIM C$(12,12)
8045 DIM C(12)
8046 DIM M(6)
8047 LET M$=""
```

```
8048 LET RETREAT=0
8050 LET J$="Do you want to take
 it?"
8055 LET COUNTER=0: LET FATIGUE=
0: LET FIGHT=0: LET MAGIC=0
8060 FOR I=1 TO 12
8065 READ C$(I)
8070 LET C(I)=0
8075 NEXT I
8080 FOR I=1 TO 5
8085 READ W$(I)
8090 NEXT I
8100 FOR I=1 TO 5
8105 READ B$(I)
8110 NEXT I
8195 CLS
8199 REM sets up character
8200 PRINT "You have 12 points t
o           distribute amongst S
trength,    Skill, Constitution
and         Knowledge"
8205 LET C(4)=12
8206 LET C(7)=100
8210 PRINT H$;" Strength"
8220 INPUT C(1)
8225 IF C(1)>C(4) THEN  PRINT Z$
: LET C(1)=0: GO TO 8210
8226 LET C(4)=C(4)-C(1)
8227 LET C(1)=C(1)+8
8230 PRINT H$;" Skill"
8240 INPUT C(2)
8245 IF C(2)>C(4) THEN  PRINT Z$
: LET C(2)=0: GO TO 8230
8246 LET C(4)=C(4)-C(2)
8247 LET C(2)=C(2)+8
8250 PRINT H$;" Constitution"
8260 INPUT C(3)
8265 IF C(3)>C(4) THEN  PRINT Z$
: LET C(3)=0: GO TO 8250
8266 LET C(4)=C(4)-C(3)
8267 LET C(3)=C(3)+8
8270 PRINT "That leaves ";C(4);"
 for Knowledge"
8275 LET C(4)=C(4)+8
8299 REM SETUP DISPLAY
8300 FOR I=1 TO 6
8310 PRINT AT 15+I,0;C$(I)
8320 PRINT AT 15+I,16;C$(I+6)
8330 NEXT I
8335 PRINT AT 15,0;"TORCHES ": P
RINT AT 15,16;"POINTS "
8340 GO SUB 2510
```

```
8399 REM buying
8400 GO SUB 2500: GO SUB 2900
8402 IF C(7)<8 THEN  GO SUB 2900
: PRINT AT 0,0;"You don't have e
nough money ": RETURN
8405 PRINT AT 0,0;" ";
8410 PRINT "You have ";C(7);" si
lver pieces"
8415 PRINT "You can buy :"
8420 PRINT "ITEM
  SP each"
8425 PRINT "(A) Armour (per piec
e)    15"
8430 PRINT "(F) Food pack
     10"
8435 PRINT "(K) Knife
     10"
8440 PRINT "(M) Magic bolt
     25"
8445 PRINT "(P) Potion
     30"
8450 PRINT "(S) Sword
     30"
8455 PRINT "(W) Torch
     8"
8460 PRINT "(Z) End buying"
8465 LET T$="AFKMPSWZ": GO SUB 2
950
8470 GO TO 8406+CODE A$
8471 IF C(7)<15 THEN  PRINT Z$:
GO SUB 2900: GO TO 8400
8472 LET C(11)=C(11)+1: LET C(7)
=C(7)-15: GO TO 8400
8476 IF C(7)<10 THEN  PRINT Z$:
GO SUB 2900: GO TO 8400
8477 LET C(5)=C(5)+1: LET C(7)=C
(7)-10: GO TO 8400
8481 IF C(7)<10 THEN  PRINT Z$:
GO SUB 2900: GO TO 8400
8482 LET C(10)=1: LET C(7)=C(7)-
10: GO TO 8400
8483 IF C(7)<25 THEN  PRINT Z$:
GO SUB 2900: GO TO 8400
8484 LET C(8)=C(8)+1: LET C(7)=C
(7)-25: GO TO 8400
8486 IF C(7)<30 THEN  PRINT Z$:
GO SUB 2900: GO TO 8400
8487 LET C(9)=C(9)+1: LET C(7)=C
(7)-30: GO TO 8400
8489 IF C(7)<30 THEN  PRINT Z$:
GO SUB 2900: GO TO 8400
8490 LET C(10)=2: LET C(7)=C(7)-
```

```
30: GO TO 8400
8493 IF C(7)<8 THEN  PRINT Z$: G
O SUB 2900: GO TO 8400
8494 LET TORCH=TORCH+1: LET C(7)
=C(7)-8: GO TO 8400
8496 RETURN
8498 RETURN
8499 REM INSTRUCTIONS
8500 PRINT AT 10,8; FLASH 1;"MER
LIN'S MINES"
8505 PAUSE 500: CLS
8510 PRINT "   This is a random
dungeon      containing combat an
d magic.     The aim is to gain a
s many              points as possi
ble"
8520 PRINT AT 6,4;"This is done
by finding        treasures and sl
aying monsters."
8530 PRINT AT 9,0;"But you must
watch yourself. If your Constitu
tion drops below 1,       you're
dead."
8540 PRINT AT 13,0;"You will los
e constitution ","if you forget
to rest, or if","        you are w
ounded.",,," Constitution and st
rength ","determine how much you
 can carry";"skill and strength
help combat,","skill and knowled
ge determine","      magical comb
at."
8550 PRINT ~0;"Press any key to
continue"
8555 PAUSE 1000
8559 CLS
8560 PRINT "  Food increases con
stitution"
8570 PRINT AT 3,2;"Potions may i
ncrease your","   abilities, but
 they may be","            poisone
d. "
8580 PRINT AT 7,2;"You start wit
h eight points","for each of you
r abilities and","   have twelve
 more points to","          distri
bute."
8600 PRINT ~0;"Press any key to
continue"
8610 PAUSE 1000
8620 CLS : PRINT AT 10,8;"INITIA
LISING"
8998 RETURN
```

```
8999 REM DATA
9010 DATA "STRENGTH","SKILL","CO
NSTI'TION","KNOWLEDGE","FOOD","M
AX BURDEN","MONEY","MAGIC BOLTS"
,"POTIONS","WEAPON VALUE","ARMOU
R VALUE","BURDEN"
9020 DATA "KNIFE ","SWORD ","FLA
IL ","SPEAR ","SHIELD"
9030 DATA "Food ","A sword","A p
otion ","A magic bolt"," Torches
"
9201 DATA "a gypsy"
9202 DATA "a monster"
9203 DATA "a thief's lair"
9204 DATA "a treasure"
9205 DATA "a potion"
9206 DATA "a weapon"
9207 DATA "a hazard"
9208 DATA "nothing"
9209 DATA "nothing"
9210 DATA "nothing"
9250 DATA 7,15,3,4,14,2,"snake"
9251 DATA 12,8,2,3,16,4,"demon"
9252 DATA 9,13,5,6,9,5,"gnoblin"
9253 DATA 8,16,6,5,9,8,"svart"
9254 DATA 13,8,3,2,10,4,"grendel
"
9255 DATA 16,6,8,4,12,3,"grim re
aper"
9256 DATA 14,4,5,1,13,6,"troll"
9257 DATA 16,10,4,2,11,6,"zombie
"
9258 DATA 14,12,5,3,16,3,"ghoul"
9259 DATA 10,11,6,3,13,7,"giant"
9260 DATA 12,16,6,4,12,5,"ghost"
9261 DATA 18,10,2,3,14,5,"vampir
e"
9262 DATA 16,9,3,2,14,8,"dragon"
9263 DATA 14,14,5,3,12,9,"banshe
e"
9264 DATA 17,11,8,3,14,7,"gremli
n"
9265 DATA 10,15,2,2,10,"devourer
"
9266 DATA 18,12,6,6,14,8,"bloods
ucker"
9267 DATA 17,14,2,2,15,9,"werewo
lf"
9268 DATA 18,16,2,2,12,10,"clawe
d demon"
9269 DATA 19,15,2,8,14,9,"Beelze
bub"
9900 PRINT "END": STOP
```

7 TEXT

7.1 Input

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

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

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

For single-key entry, numbers are easiest to process, particularly because numeric input is easy to check. For example, if the numbers 2 to 6 are the only allowed input, a routine such as this will do the job:

```
 9Ø PRINT "Please type a number between 2 and 6 and press
    ENTER"
1ØØ INPUT A
11Ø LET A = INT A
12Ø IF A>6 OR A<2 THEN GOTO 9Ø
```

Line 11Ø is necessary to ensure that players do not try clever input involving decimals.

As each key on the keyboard has a code, we can define a range of allowable keys in the same way. The codes for all the Spectrum charact?rs are contained in Appendix A of the manual. For example, lower case letters a to f are codes 97 to 1Ø2 inclusive. To check this input we would change the routine above to read:

```
 9Ø PRINT "Please type a letter from a to f and press ENTER"
1ØØ INPUT A$
11Ø LET A$=A$(1): REM JUST IN CASE MORE THAN ONE
    LETTER IS TYPED
12Ø IF CODE A$<97 OR CODE A$>1Ø2 THEN GOTO 9Ø
```

The problem with this is that the keys we choose are not likely to be easy to remember unless each letter is the first letter of the command it stands for, such as the usual N,S,E,W for the points of the compass. Checking for errors in input as complex as these can itself be quite complicated.

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

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

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

The worst kind of error message is no message at all! The player will have no idea what is going on or what he is meant to be doing.

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

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

A = ATTACK
M = MAGIC

R = RETREAT

whereas on entering a new location the options might be:

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

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

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

```
  8Ø REM FOR COMBAT INPUT
  9Ø PRINT “Please type a single letter (A,M or R)”
 1ØØ LET T$ = “AMRamr”:REM DECLARE TEST STRING
 11Ø GOSUB 1ØØØ
 12Ø PRINT “O.K. That’s valid”
 9ØØ STOP
 999 REM CHECKS FOR VALID CHARACTERS
1ØØØ LET A$ = INKEY$
1Ø1Ø LET T = Ø
1Ø2Ø IF A$ = “” THEN GOTO 1ØØØ
1Ø3Ø FOR I = 1 TO LEN (T$)
1Ø4Ø IF T$(I) = A$ THEN LET T = 1
1Ø5Ø NEXT I
1Ø6Ø IF T = 1 THEN RETURN
1Ø7Ø PRINT “No, you idiot, A, M or R”
1Ø7Ø GOTO 1ØØØ
```

If you follow this routine through you will see that control only returns from the subroutine if one of the correct keys is pressed. T is set to 1 only if one of the correct letters is found and control only returns to the main routine if T is set to 1.

Subroutine 1ØØØ can be called at any time in the game so long as we remember to declare the test string before the call is made. Notice that the declared string controls the subroutine by its contents and length. It is important that we do not include any invalid characters in the string, such as punctuation or spaces, unless these are possible valid input. Remember also that the longer the string T$ the longer the routine will take to run, so it is best to keep T$ limited to five or six characters in each case.

This method of checking is used in The Mines of Merlin, the routine being held at lines 295Ø to 2998. Note two refinements here. Line 296Ø converts any lower case input to upper case, thus making the test strings much shorter and easier to code. Line 2964 tells the player which character he or she has just typed so that any mistakes can be monitored. One of Cad's worst features is the way he blames the program for the way his fingers slip around the keyboard.

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

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

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

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

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

It will now become clear why all these are needed. It would be better if we could use variable names with more mnemonic titles (to make them easier to remember in writing and debugging the program) but Spectrum BASIC only allows single-letter names for string variables. The flowchart we will use is:

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

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

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

We can then compare the first three letters of our input words (D$ and E$) with each three-letter section of this string and, if it matches, we can then branch to the appropriate routine. Let us work through the whole procedure a stage at a time in Spectrum BASIC. As we will want this routine quite frequently it should be put early in our program, at lines 1ØØØ to 15ØØ.

First, input the words:

```
 999 REM TWO WORD INPUT
1ØØØ PRINT "What next?"
1Ø1Ø INPUT A$
```

Then separate A$ into two words:

```
1Ø2Ø LET K = Ø: LET B$="": LET C$="": LET D$="": LET
     E$=""
1Ø3Ø FOR I = 1 TO LEN(A$)
1Ø4Ø IF A$(I)=" " THEN LET B$ = A$(1 TO I–1): LET C$ =
     A$(I+1 TO LEN(A$)); LET K = K+1
1Ø5Ø NEXT I
1Ø6Ø IF K<>1 OR C$ = "" THEN PRINT "Two words please";
     GOTO 1ØØØ
```

K is a counter which marks the number of spaces found, so line 1Ø5Ø can check that there is only one space and that two words have been found.

Next we take the first three letters of each of the two words. The problem here is with words such as GO which have only two letters. We have to add a space to these, which means that V$ must include GO plus space if it is an allowed command, e.g., "LIGTAKGO SWI". Line 1Ø6Ø does this:

```
1Ø6Ø LET B$ = B$ + " ": LET C$ = C$ + " "
1Ø7Ø LET D$ = B$(1 TO 3): LET E$ = C$(1 TO 3)
1Ø75 LET FL = Ø
```

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

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

```
1Ø8Ø LET T$ = V$: LET U$ = D$: GOSUB 12ØØ
1Ø85 IF FL = 1 THEN RETURN
1Ø9Ø LET T$ = N$: LET U$ = E$: GOSUB 12ØØ
1Ø95 IF FL=1 THEN RETURN
1Ø96 PRINT "O.K."
1Ø99 RETURN
1199 REM ROUTINE TO COMPARE TEST STRING AND
     INPUT WORD
12ØØ FOR I = 1 TO LEN(T$) STEP 3
121Ø IF T$(I TO I+2) = U$ THEN RETURN
122Ø NEXT I
123Ø PRINT "I don't know how to"; A$: LET FL = 1
124Ø RETURN
```

Note the flag FL set at lines 1Ø75 and 123Ø. Line 1Ø75 sets it to zero, in case it has already been used. Line 123Ø will set it to one if the first half of the command is invalid so that, on returning to 1Ø85, the routine is not run for the second part. Notice also the STEP command in line 12ØØ, which may be an instruction you have not used before. This simply means that the FOR . . . NEXT loop between lines 12ØØ and 122Ø will increase I by three at a time, thus moving along T$ three characters at a time, which is how our codes are grouped. Normally a FOR . . . NEXT loop will operate in steps of one, which is the default value if STEP is not specified.

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

```
  9Ø PRINT "Please type an instruction from 1 to 9"
 1ØØ INPUT A: LET A = INT(A): IF A<1 OR A>9 THEN GOTO
     1ØØ
 11Ø GOSUB A * 1ØØØ
```

This sends control to subroutines beginning at lines 1ØØØ, 2ØØØ, etc., up to 9ØØØ. However, such elegance is not always possible if the input is initial letters or full words. By careful design we could send control to a subroutine related to the character CODE or the sum of the codes in any three-letter combination, but the design effort is probably not worth while. Consequently, a series of one-line tests is used to direct control to appropriate routines, of the form:

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

For our single-key example appropriate lines might be:

```
13Ø IF A$ = "A" THEN GOSUB 2ØØØ
14Ø IF A$ = "M" THEN GOSUB 21ØØ
15Ø IF A$ = "R" THEN GOSUB 22ØØ
```

and for two-word input appropriate lines might be:

```
13Ø IF B$ = "TAK" THEN GOSUB 2ØØØ
14Ø IF B$ = "SWI" THEN GOSUB 21ØØ
etc.
```

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

To clarify this let us use the example of SWIM. Suppose that the swimming routine begins at line 211Ø. During initialization of the program we can assign a value to the variable called SWI, which also happens to be our three-letter code for swim. When input has been through all the validation checks B$ will hold "SWI" as the first part of the first word. If we can turn the string held by B$ into the value held by the variable with the same name, we can use the input string to direct control.

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

```
   1Ø  LET SWI = 211Ø
.....................
  14Ø  GOSUB VAL(B$)
.....................
.....................
211Ø  REM SWIM ROUTINE
.....................
.....................
```

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

The advantage of this is that it is easy to follow the workings of the program and it saves coding. We need a LET command for every acceptable input word, but can dispense with all the long-winded IF . . . THEN statements. It also allows clever programmers to alter the routine a particular command is running. For example, there may be two locations in which a player might be allowed to swim, namely, a placid stream and a dangerous river. If the player swims in the stream there is no danger of drowning. For the river, everything is the same as for the stream, except that there is the danger of drowning. Therefore the same routine will be used for each, with an extra piece of code for the river. If this extra piece of code was held at line 2ØØ5, with the common routine at 211Ø, then a line like the following could make use of it:

```
 135  IF LOCATION = RIVER THEN LET SWI = 2ØØ5
```

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

As the most frequently used nouns will be those for movement we can speed things up by having two noun strings, one full of objects and the other for directions, i.e., "NORSOUEASWES", and only test the latter when a valid movement verb is detected,i.e., in the case of Camelot the verb GO. In addition, we will need two special commands which do not need nouns, namely, HELP and INVENTORY. The first of these gives help to the player with problems; the second lists all the objects he or she currently has. Before the standard word comparison routine we need three separate tests for these three verbs. As the latter two have no nouns, we test for these before the main routine, i.e., between lines 1Ø15 and 1Ø2Ø:

```
1Ø16 IF A$(1 TO 3) = "INV" THEN GOSUB INV: RETURN
1Ø17 IF A$(1 TO 3) = "HEL" THEN GOSUB HEL: RETURN
```

The test for GO will come before the main verb and noun tests, i.e., before 1Ø8Ø. If W$ is the string "NORSOUEASWES" then lines 1Ø78 and 1Ø79 will do the job:

```
1Ø78 IF D$ = "GO" THEN LET T$ = W$: LET U$ = E$: GOSUB
     12ØØ: IF FL = 1 THEN RETURN
1Ø79 IF D$ = "GO" THEN GOSUB VAL(B$): RETURN
```

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

We must also minimize the effects of faulty input of commands. Likely mistakes involve typing only one word, or words of less than three letters, or extra spaces. Because the checking routine uses the string slicer these mistakes could give the error number 3, 'subscript wrong', for example, because the string is not long enough to be sliced. An easy way to avoid most of these problems is to declare the variables not as empty strings but as strings of spaces and to clear each string to the required number of spaces each time the input loop is used. This is done in lines 1ØØØ and 1Ø11 of the program.

Once the loop has run it will return with a flag marking an error if one has occurred, and input will again be repeated. However, if both input words are all right control will have to be passed to other routines in the program. In the case of Camelot the routines are movement (block 3) or description (blocks 4 and 5). In order that the detected verbs and nouns can be used in such control they have to be given numerical values, and if we do not want to use the method already outlined of declaring variables with the same names as the verb codes and values equal·to the subroutine line numbers we have one further alternative.

For every verb or noun we already have a value, namely, the position of the three-letter code for each word in the relevant string. This is counted by the variable I in the loop which attempts to match D$ or E$ against these codes, but I is used as the variable in several other loops, so monitoring it might become confusing. Consequently, there is a command in line 121Ø which sets K to the same value as I if a match is found in the checking loop. Then K is passed back to the main checking routine where it is turned into the appropriate variable, here called VERB or NOUN.

If there are special words to be detected then VERB and NOUN may have to be set specially. In Camelot there are two such words, INVENTORY and HELP, which are looked for as special cases before the general checker. For these, VERB has to be set specially.

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

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

My first reference sheet is a map of the logical maze with each location numbered as outlined in Chapter 6. Each location which is either a GET or NEED square has G or N written in it, plus the noun or nouns which are there. In addition any special puzzles or features are also noted here, such as the river which has to be swum. Finally, an H is placed in each square where an object is hidden, rather than immediately obvious.

To make things easy I use the variable LOCATION in the program to hold the PEEK code for each location on the map, and this is the control variable for the description routines in block 6 (lines 6ØØØ to 6999). The correspondence is as follows: the location number is on my hand-drawn map, is held above Ramtop in a specific address, and when multiplied by 1Ø and added to 6ØØØ gives us the number of the description routine.

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

Finally, there is the list of nouns. This is probably the most important of the lists. It holds the names of all the objects and the solutions they are part of, as well as the line numbers of the verb routines which use them. The nouns are listed in the order they are held in the arrays o() and o$(), which roughly corresponds to the order the GETs and NEEDs should be encountered in the program to solve it. It is important to have a complete list because the noun routines, being the end points of a series of calls, can seldom be held in a very logical way and will therefore be difficult to locate during debugging. REM statements would help with this problem, but as so many would be needed that the size of the program may well be doubled they would have to be removed from the finished version.

7.2 Talking to monsters

It is all very well killing monsters and stealing all their wordly goods, but not every monster is hostile and some may be too powerful for a lowly adventurer to pit himself against. In such circumstances the wily adventurer is best advised to use sly flattery, gentle persuasion, high-sounding oaths, blood-chilling threats, or any other form of conversation which seems likely to part the monster from his prize. However, this means that the monster should be able to talk back, and talking is a difficult thing to simulate.

Two approaches are possible on a micro. We can aim for as full a simulation as possible in which many tests and transformations are made on the language, attempts are made at meaningfulness, and a wide and varied vocabulary is used. Or we can settle for a couple of sleight-of-hand tricks, which appear to allow conversation but actually just do random things with words.

The first approach is very complex, too difficult to give a full account of here, but useful routines can be adapted from the Eliza or Doctor routines found in many books of programs for micros. These routines are based on two procedures— in the processing of input the program checks for keywords by comparing each input word against a dictionary of recognized words and, according to the match that is found, will compile an output string with an appropriate meaning; and in the processing of that output some changes are made to the structure to make it fit grammatically with the input string.

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

This requires a great deal of thought. The least that is required of a good routine to do this is that it should have a large and appropriate vocabulary, that it should produce reasonably grammatical sentences, that it should allow input of strings longer than two words, and that the response should be connected to the input. This means we would have to think of all the words likely to occur and find some appropriate response for each, which might demand much more work than the program justifies. However the essential design is easy to describe (Fig. 7.1).

1. ACCEPT INPUT STRING
2. SEPARATE STRING INTO COMPONENT WORDS
3. HOLD ALL WORDS IN MEMORY
4. COMPARE ALL INPUT WORDS AGAINST THE DICTIONARY OF KEYWORDS
5. IF A KEYWORD IS FOUND THEN EXECUTE THE APPROPRIATE SUBROUTINE
6. IF THE SUBROUTINE USES THE ORIGINAL STRING THEN DO THE NECESSARY TRANSFORMATIONS ON THAT STRING
7. IF NO KEYWORDS ARE FOUND THEN EXECUTE A DEFAULT OUTPUT
8. IF THE KEYWORD SENDS CONTROL TO ANOTHER ROUTINE THEN EXECUTE THAT ROUTINE, OTHERWISE GO TO THE BEGINNING OF THE CONVERSATION LOOP

Figure 7.1

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

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

Stage 5 is the heart of the program. For each keyword one or more possible outputs should be allowed. These could be randomly selected once control has been directed to them, or they could be motivated in some other way, as discussed in Chapter 8. For our sample string some transformations might also be used. For example, the 'give' routine could take all the words between 'give' and the end of the input string or a conjunction (in this case 'or'), and invert any pronouns found in that substring, in order to get 'give you my jewels'. To this the routine adds 'If I' and 'what will you give me?'. So the output formula is:

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

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

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

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

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

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

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

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

A sample routine for this which could be added to The Mines of Merlin is given below:

```
38ØØ LET R = Ø
38Ø5 PRINT "What do you have to say?"
381Ø LET R = FNR(3) + R
382Ø INPUT A$
3822 IF R < 1 THEN GOTO 395Ø
3825 GOTO 383Ø + (R*5)
383Ø PRINT "I'm fed up with this, I'm going": RETURN :REM
     GOES BACK TO MAIN ROUTINE
3835 PRINT "So that's how you feel is it?": LET R = R + 1: GOTO
     381Ø
384Ø PRINT "I don't quite understand you": LET R = R – 2: GOTO
     381Ø
3845 PRINT "Haven't you anything better to say than that?":
     LET R = R + 1: GOTO 381Ø
385Ø PRINT "So you would like me not to eat you?": LET R = R – 3:
     GOTO 381Ø
3855 PRINT "Do you want to be friendly then?": LET R = R – 5:
     GOTO 381Ø
```

```
386Ø PRINT"I'm getting impatient": LET R = R + 3: GOTO 381Ø
3865 PRINT "I detest adventurers like you": LET R = R + 2
387Ø PRINT "I could give you a reward if you behaved nicely":
     LET R = R - 5: GOTO 381Ø
3875 PRINT"I hate people who say things like that": LET R = R +
     3: GOTO 381Ø
388Ø PRINT"You'd best watch out for my temper": LET R = R - 1:
     GOTO 381Ø
3885 PRINT"You've one more chance to be pleasant": LET R = R
     - 2: GOTO 381Ø
389Ø PRINT "I WARNED YOU!!!": GOSUB COMBAT
     ROUTINE: RETURN
3895 PRINT "Because I think you're so rude I'm taking your
     treasure": LET (treasure variable) = Ø: RETURN
39ØØ PRINT "Stuff this for a lark": LET (treasure variable) = Ø:
     GOSUB COMBAT ROUTINE: RETURN
39Ø5 PRINT"The monster works himself up into an apoplectic fit
     and expires on the floor": LET (treasure variable) =
     (treasure variable) + (monster's treasure): RETURN
395Ø PRINT"You seem so friendly I'm going to let you have all my
     hard earned savings": LET (treasure variable) = (treasure
     variable) + (monster's treasure): RETURN
```

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

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

8 CHROME

8.1 The need for chrome

'Chrome' is a word used by board game designers to refer to the decoration and additional detail that a game might have which has no effect on the structure of the game or its play, but may, even so, be essential to the flavour of the game. In a board game chrome would include the scenario narratives, the attractiveness of counters, the box design, background information, special presentation features, additional variations, etc. Its purpose and effect is very much like the chrome bodywork on a car— it is very attractive, it adds to the price, it adds nothing to the performance of the car, yet it may very well be the thing which attracts the buyer most. In our case chrome is to attract a player. It makes the game unique and intriguing, irrespective of what actual play is like. So, though not strictly necessary, chrome may be essential to the feel of a game and its overall effect on the players, even if the shape and design of the game itself is unremarkable.

The success of a microgame such as Star Trek depends as much on its appearance and theme as on its playability and design. The player can take on the role of Captain Kirk and watch screen displays just like those of a real starship captain (whatever that is!).

Poorly designed games pay little attention to chrome. As far as the designer is concerned it is the game structure which is important, not what the structure is meant to represent. This has led to a host of poor quality abstract arcade-type games for the micro which, apart from their packaging, have little to offer in the way of stimulating intelligence or imagination. Programmers are not necessarily the best people to write games. Many excellent programs are dull to play because there is nothing original on the surface to catch the player's attention. On the other hand, there are some very poorly written games which, because they are well presented and lavishly boxed, sell much better than their programming deserves.

The aim of the good designer is to choose a game structure which is not only interesting to play but is also a fair representation of a reality. It uses detail, effects, graphics, sound, display format, language, and documentation to create an atmosphere within which the logical structure of the game program is a coherent and essential part.

So almost anything which enhances the appearance of the game and makes its logic something real rather than abstract (more like a simulation than an abstract game—the difference between Monopoly and Draughts) is likely to make it a better game, especially if it adds variety to the game. We do not want to overburden the game with superficial detail or else players will never actually play. Some board games suffer from having so many rules and counters, so much background detail, documentation, and supportive text, that players never manage to penetrate it and to understand and play the game. Ideally, therefore, the logic of the game should represent the logic of the reality which is being 'simulated', and the chrome should be sufficient to make it clear how the game logic is also the logic of a particular world.

For example, one way of adding interest and variety to a fantasy or science fiction game, whether it is a micro game or not, is by generating new names for people and places in the game. This could be done quite simply by choosing a random sequence of letters from the alphabet and stringing them together to form a new word. However, the resulting word could be something like 'ZXZQXL', which is not much like a real word. In fact it looks like a random series of letters.

If we are talking about English, there are certain rules about letter sequences that make our language what it is, such as the fact that a Q is always followed by a U, or that a word always has at least one vowel. Actually there are a large number of such rules, and it is possible to write a program which makes up English words because such a program can apply all of these rules in stringing letters together. However, we do not need to go to such lengths to create words which are varied, original, fantasy-like, yet sufficiently like English to be regarded as real words and therefore a sufficient representation of reality to make sense. A program to do this could fit into about 2K for most machines, and could probably be squeezed into less. All it needs to do is to take the following rules into account:

1. English words are made up of syllables (usually between one and four).
2. A syllable is a sequence of sounds, being zero to three consonants followed by one or two vowels followed by another zero to three consonants.
3. Certain sounds or letters occur more frequently in normal language than others.
4. Some letters cannot follow other letters (e.g., in English F is not followed by N in the same syllable).

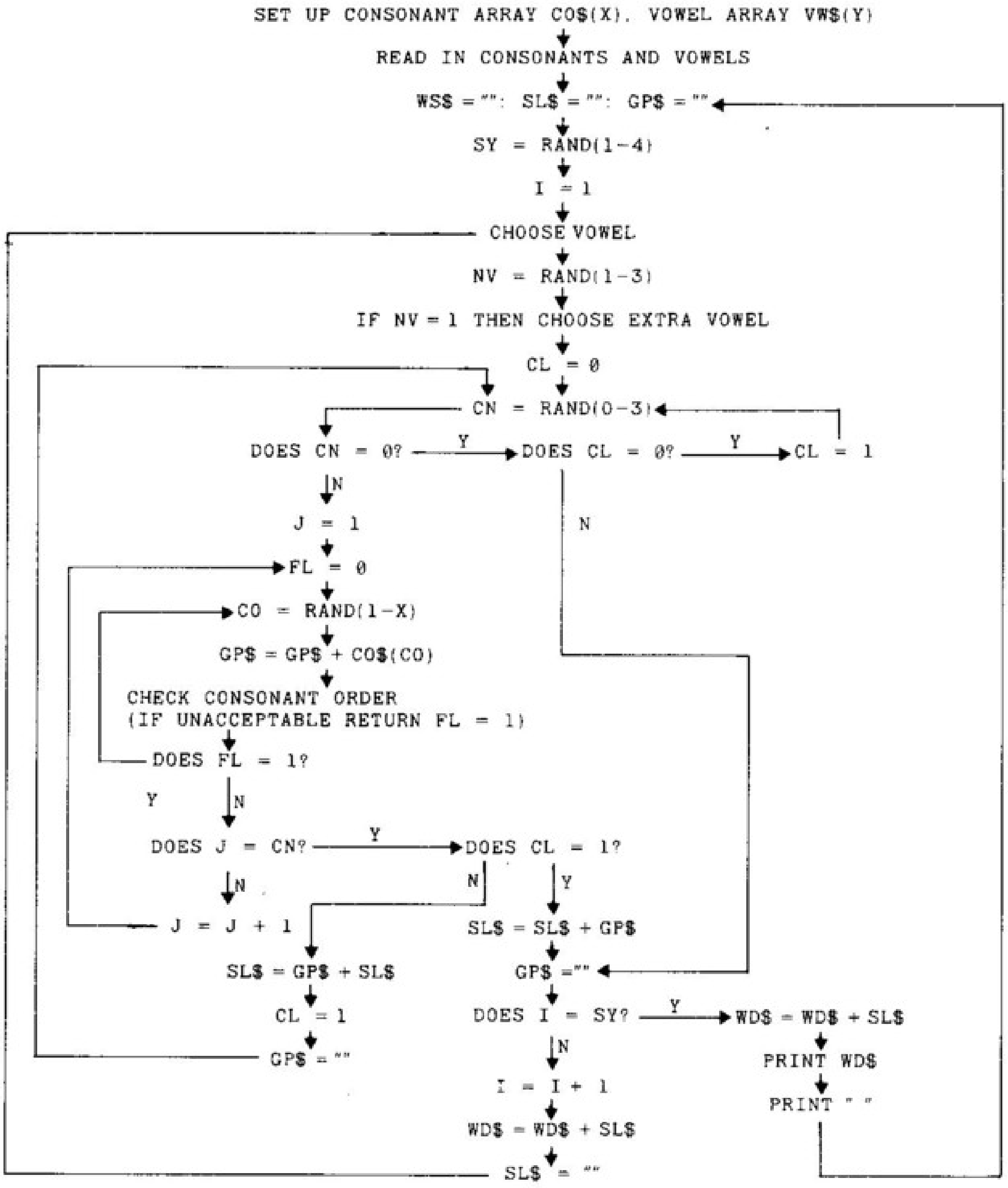
The program should first choose a vowel and then decide how many consonants will precede the vowel and how many will follow. It then chooses a sequence of consonants, checking that they can follow each other legitimately. It does this for both consonant groups and then decides if there will be any more syllables in the word. This gives you an English-like word.

Ways of varying the type of language essentially involve changing one of the rules 1 to 4 above. For example, we could only allow words of more than two syllables, we could change the relative frequency of letters (Fig. 8.1 shows the relative frequency of occurrence in English—this can be controlled by using DATA statements to hold these letters and changing the number of times a particular letter is held in these statements), we could change the number of consonants before or after the vowel(s), and we could change the rules permitting sequences of letters. The latter change might produce some very strange words because there are some letter sequences which it is almost impossible to say without making some other sound between them.

Letter	Relative number of occurrences
Z	1
X	1
J	1
Q	1
K	2
V	3
B	4
P	4
G	4
Y	5
W	5
F	6
M	7
C	7
L	10
D	11
R	15
H	15
S	20
N	20
T	22

Figure 8.1 Relative frequency of consonants in English

The flowchart in Fig. 8.2 outlines a detailed language generating program along the lines just described. I have used variables with



THE VARIABLES IN THE FLOW CHART ARE:

CO$ (NUMBER OF CONSONANTS) = CONSONANT ARRAY
VW$ (NUMBER OF VOWELS) = VOWEL ARRAY
SY NUMBER OF SYLLABLES PER WORD
V CHOSEN VOWEL NUMBER IN ARRAY
NV CONTROLS NUMBER OF VOWELS PER SYLLABLE
CN NUMBER OF CONSONANTS
SL$ VOWEL STRING
GP$ CURRENT CONSONANT STRING
WD$ CURRENT WORD
ZZ$ CURRENT CONSONANT ORDER STRING
FL FLAG FOR CONSONANT ORDER
CL FLAG FOR FIRST OR LAST CONSONANT GROUP

X = NUMBER OF CONSONANTS
TO CHOOSE FROM
Y = NUMBER OF VOWELS
TO CHOOSE FROM

Figure 8.2

long names to make it clearer, though these would have to be changed for Spectrum BASIC. A version of the program for the Spectrum is available on the separate cassette. *The main point is that to make such an idea worth while it has to look like a real word, with all the rules built in. There is no point in bothering with this kind of chrome unless it is done as well as possible.

*The flowchart and description of a language generator program first appeared in Issue 10 of *The War Machine,* August 1982.

8.2 Intelligence in adventure

'Intelligence' means here not player intelligence but machine intelligence. Generating names for use in a program or non-micro game is useful, but it is only a superficial aspect of a game. It can consume memory which might be better used in terms of game structure. However, if you write a program with a degree of intelligence in it, you have provided originality at the fundamental structural game level. A program which can, in some sense, 'think' will provide a degree of variety which is not superficial but fundamental to the game, making each game different but in a significant way.

Such games are just beginning to appear for micros. A notable success for the Spectrum has been The Hobbit, an adventure game in which characters respond to the player's actions but also have lives of their own—e.g., the program 'understands' the concept of 'friend' and 'enemy', so monsters will behave differently according to the way they have been treated by the player's character. Other programmers are currently working on war games in which the player has to combat the computer. The computer makes strategic and tactical decisions which depend on what the player is doing. In other words, it seems to 'understand' what the player is doing and has a large range of possible responses which are chosen according to its understanding of the situation. Most advanced in this area are programs for abstract games such as Chess and Othello, in which the computer can consistently beat even good players because it has a better knowledge of tactical possibilities.

These kind of games employ techniques used in the field of artificial intelligence. Artificial intelligence is really a branch of the study of human intelligence. Its aim is to learn more about the human mind by discovering what we need to know to make a machine act in a way which we would regard as human. However, it has become very important in various other areas such as robotics, missile guidance systems, and medical diagnosis. As intelligence is what is primarily being tested in non-arcade games, and as a game is a self-contained world, games provide a good way of developing distinct artificial intelligence techniques, and these potentially can make a new generation of games which are much more exciting than the currently available selection.

This is because programmed intelligence in a game allows us to overcome (or, at least, to try to overcome) the two major problems we have seen in adventure programming so far. On the one hand, there is the puzzle game. This is a real test of the player's intelligence and imagination, but it has the major defect of being fixed in structure and content, so that once it has been solved it will not be played again. On the other hand, there is the combat game. Rather than being fixed at the outset, this is fundamentally a series of random structures. The player has relatively little to do in the way of developing a consistent and intelligent strategy, and little chance of predicting how the parts of the adventure might fit together. Monsters and treasures are randomly distributed and alterations in the character's variables, abilities, and characteristics are more or less accidental (depending on which locations he went to and when). So though each encounter may be interesting, its fundamentally random nature will eventually lead to boredom.

What is needed is a game which structures itself, so that it is different each time it is played. However, it must have a logical structure, so that player intelligence can be tested and strategies can be developed, and its changes during play must respond to the player's previous decisions not randomly, but intelligently. Artificial intelligence can provide ideas and techniques on how to go about achieving this. Unfortunately, however, the current generation of microcomputers has insufficient memory for large artificial intelligence components, so at present we can only investigate the possibilities and try to find new ways of development.

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

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

For example, let us look at the concept of enemy/friend mentioned above. A simple flowchart for understanding this concept might be as in Fig. 8.3.

```
1. PLAYER ACTS TOWARDS MONSTER
2. IF THE ACTION IS HOSTILE THEN LET ENEMY = ENEMY + 1
3. IF THE ACTION IS NOT HOSTILE THEN LET FRIEND = FRIEND + 1
4. IF FRIEND < ENEMY THEN FIGHT THE CHARACTER
5. IF FRIEND >= ENEMY THEN GIVE THE CHARACTER REWARD
```

Figure 8.3 The concept of friend/enemy

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

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

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

The same is true in writing a game which writes itself. It is quite possible to build a game which is both logically structured and different on each play, though it would be more complex than either of the two game types we have so far examined. For example, a version of the Camelot puzzle game could choose from a set of possible puzzle types each time it was run. Suppose it had the choice of anagram, cypher, and square-root problem. It chooses 'cypher'. It then uses the cypher routine to produce an actual cypher and it holds this in memory as being the particular problem to solve. It then looks at all the empty locations in the game and picks one to hold that cypher. It then chooses another empty location, and sets a flag which indicates 'If the cypher has not been solved, this location is blocked'. Then it selects from a number of possible descriptions the actual description used: 'Your route is blocked by a door with a combination lock'. Finally, it checks that a route is possible between the two locations.

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

It therefore seems unlikely that micros such as the Spectrum will support such elaborate games. However, it would be possible to devise a suite of programs which would together 'write' such original games. For example, all the above eight routines could be held as separate programs, each producing data which were saved to cassette or disc or microdrive and then used as the data for the next program in the sequence. With the speed and capacity of discs or the Spectrum microdrive, such a game-generating suite might be worth developing, because the suite could be run by a single command as if it was one program, and the result would be an original structured game, as if the game was writing itself.

Similarly the problem of holding a number of potential routines which might not be used, in making characters and computer responses more 'intelligent', can also be reduced by using discs. If the least used routines are held on disc or microdrive they do not need to use any of the computer's RAM except when called and loaded. Thus the same RAM could be used for different routines at different times in the game. However, only expensive micros like the Apple presently have such adventures, and even these slow the response time of a game down quite noticably.

Even so, it is possible to improve existing types of game substantially, even within the limits of the current generation of micros. One way to do this is to combine both the puzzle and combat types. There are one or two games which attempt this, but some programs which claim to have been successful are really several separate games masquerading as one.

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

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

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

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

Intelligence could also be built in to a routine like that described in Sec. 7.3 for talking between monster and character. For example, suppose your program keeps track of the number of goblins the player kills and the number he tries to bribe or use some other tactic. This can be used as an index of 'friendliness/hostility to goblins'. Suppose that the player meets a very powerful goblin and does not want to fight it. He does not have any money, so he cannot bribe it, and he needs the treasure which the goblin is guarding. His only option is to talk. In the program's database will be a number of keywords to look for in the talk routine and each will have a value, such as the following:

Word	Value
evil	−3
nasty	−2
ugly	−1
nice	+1
intelligent	+2

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

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

Phrase number	Phrase	Value for hostility of output
1	I'M GLAD YOU'VE COME	+7 TO +9
2	I HATE ELVES	−6 TO −9
3	BECAUSE	−9 TO +9
4	CLEAR OFF	−5 TO −9
5	AND	−9 TO +9
6	BUT	−9 TO +9
7	I LIKE ELVES	+6 TO +9

8	YOU LOUSY ELF	−4 TO −9
9	TELL ME MORE	−7 TO +7
10	I'LL SKIN YOU ALIVE	−8 TO −9
11	I'VE SOMETHING FOR YOU	−3 TO +4
12	I'M GOING TO LET YOU HAVE IT	−5 TO +3

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

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

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

```
 99 REM ROUTINE FOR LYING MONSTERS
1ØØ LET R = INT(RND*3) +1
11Ø IF R= 1 THEN LET L = treasurelocation: LET T$ = treasure-
    name$: GOTO 14Ø
12Ø LET R = INT(RND*4)+1: LET L = INT(RND * 2Ø)+1
13Ø FOR I = 1 TO R: READ T$: NEXT I
14Ø PRINT "If you spare my life I'll tell you where the ";T$;" is"
15Ø REM (player spares monsters life)
16Ø PRINT "The ";T$;" is at";L
18Ø RETURN
19Ø DATA silver crown, gold torque, ruby, emerald
```

Line 1ØØ decides whether to lie or not. There is a one in three chance that the monster will tell the truth. (Of course, some monsters might be more truthful than others, in which case 3 will be replaced by the truthfulness variable.) Line 11Ø sets variables T$ and L to the treasure's name and true location respectively and then sends control to the output lines.

Line 12Ø selects the lie, choosing one of four treasures and one of twenty locations. Line 13Ø reads the name of the treasure into T$, using the DATA statement at 18Ø, which will be the same DATA statement used to originally set up the map. Lines 14Ø to 17Ø output the information. Here seven lines of new program add a whole new dimension to the game; they could be reduced by, for example, letting the random number which determines if a lie is to be told also determine the actual lie to be told.

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

8.3 Sound

Sound, like graphics, is one of those features which varies enormously from machine to machine. The Spectrum really has rather limited sound resources. Even so there are a number of things that can be done with it to enhance a game.

The simplest use of sound is as a reinforcer; that is to say, it is a redundant signal, telling a player what he would already know from some other information available to him. Many games BEEP every turn to tell the player that his turn is over; or BEEP if the player is moving into a wall or other obstacle; or BEEP to show that the game has finished; and so on. Many players are only irritated by this, so if we choose to build such redundant signals into our game it is a good idea to give the player the option of switching the sound off.

This can be done easily. Include in the setting up of the game a question asking if sound is wanted or not and use the answer to set a flag. In every BEEP statement in the program include a test to see if the flag is set; if it is not then make the BEEP or enter the sound subroutine. The test statements would look like this:

```
IF FLAG = Ø THEN BEEP Ø.1,2Ø
```

The other superfluous use of sound which can be irritating is the decorative use. Some micros have sound facilities which, though they make tunes possible, are not very attractive to the ear. It is therefore not a good idea to use them for a decorative effect. However, others have very good multi-channel sound and can be turned into passable synthesizers. The Spectrum is nearer the former than the latter, but tunes played on it can be quite pleasant. The irritation comes when the tune has been played for the three hundredth time!

Tunes can be used to introduce and end games. At the end the game can always be interrupted if the player is fed up with it. However, they should be used within the game with care. Perhaps such tunes are acceptable if used as victory fanfares or as interludes while the program is setting up its DATA. Even here, if the tune lasts a long time it will only be regarded by the player as an unnecessary delay.

If we want music in our game we must integrate it in a sensible way. One way in puzzle games is to make music one of the puzzles. Tunes can be used as clues, just as messages of other kinds can. The clue could be in the title of the piece (Problem: How do I cross the river? Clue: *Yellow Submarine*), in the construction of the music (Problem: How do I attract the Wizard's attention? Clue: *The 1812 Overture,* i.e., make a loud noise), or in the lyrics of a song, for which the micro just gives the tune (Problem: Why does this valley keep changing? Clue: *The Sound of Music,* i.e., 'the hills are alive'). If we wanted to be very nasty we could encode a message in musical notation, so that the message 'beg' could be conveyed by playing the notes B, E, and G.

In combat games these kinds of use are seldom important. What is useful, however, is a signal which tells a player something he might not otherwise know, such as a warning that something is about to happen or is happening. So when a monster is in the next room, or when the Klingons are merely 50 parsecs away, or when the player's lifeblood is about to dribble away, a well-timed noise will be anything but irritating. It conveys exactly the information needed in a recognizable form. In fact research has shown that people tend to pay more attention to audible warnings than to most other kinds— one of the reasons why Space Invaders seemed so compelling was the irresistible quickening of its rhythm.

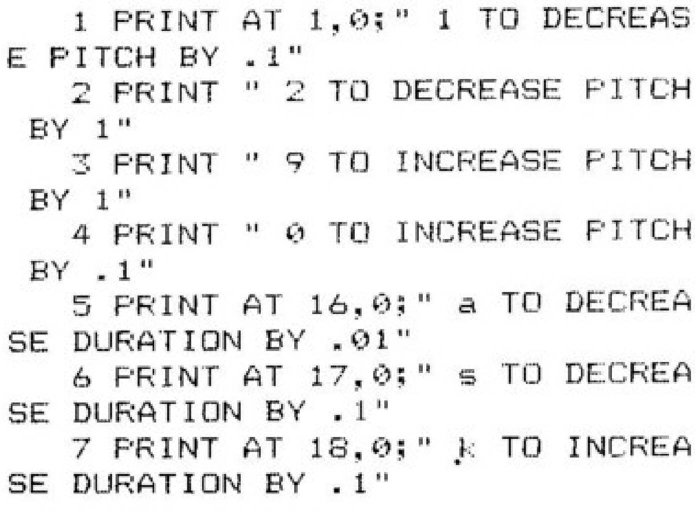
On the Spectrum one such use is to sound a BEEP whenever a key is pressed, because the keyboard is such that players may doubt if their touch has registered, especially if input is not immediately displayed on the screen. This can be done by the command POKE 236Ø9,N, where N is the length of the interval we want the BEEP to sound for.

If you want music in your program it is very tedious to keep writing BEEP plus the values for each note, especially if certain notes or durations are repeated. It is better to hold the parameters for BEEP as data in any of the forms we have explored in previous chapters— as simple variables, as arrays, as DATA statements, or as strings. Remember if you use a string that the time taken to process the string might alter the duration of the notes played. Remember also that BEEP interrupts the normal timing of the Spectrum, holding up everything else, so if it occurs in the middle of a real-time routine allowance must be made for the interruption.

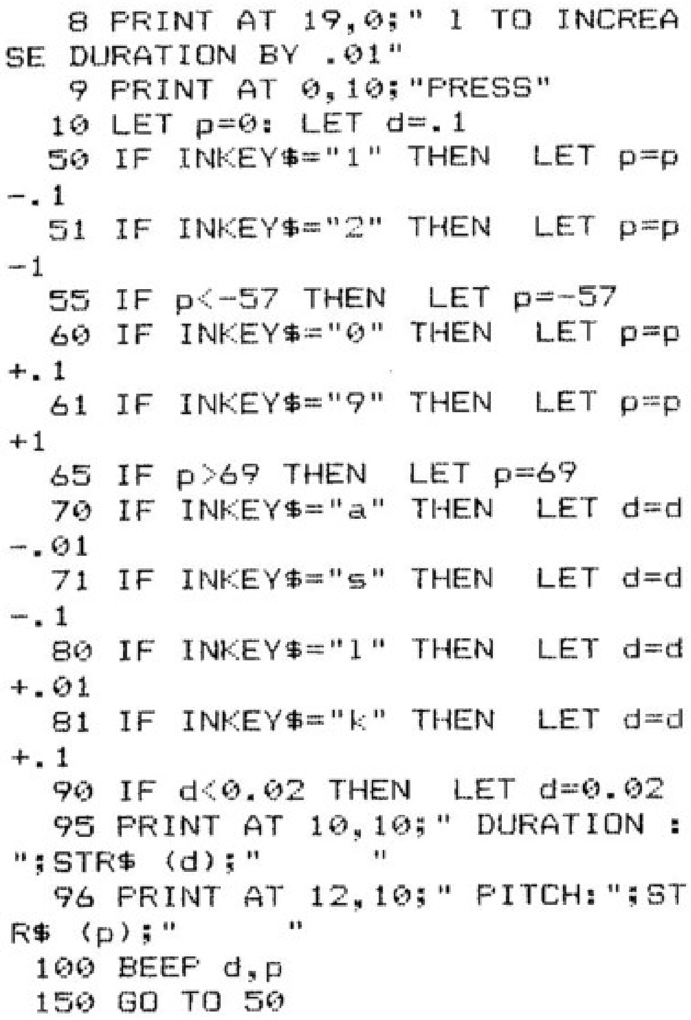
This latter feature can be used to advantage, of course. If we want a delay in our program we can always use PAUSE, but if we do, then nothing is going on at the same time. BEEP acts very much like PAUSE, except that it cannot be interrupted by the user but does something else at the same time. In addition to playing a tune or producing a particular effect the length of the tune also gives the player an idea of the amount of time that has elapsed.

To produce effects with the Spectrum sound you will need to experiment and keep notes of what you do and what the results are like. However, some simple effects are easy to discover. The four simplest to program are to increase or decrease the pitch and to increase or decrease the duration. An increase in pitch with a decrease in duration sounds like acceleration; a decrease in pitch with a decrease in duration sounds like deceleration; pitch which regularly increases and decreases sounds like a siren; a slow increase in pitch followed by a rapid fall mimics a climbing then falling object; and so forth.

To demonstrate some of these, the program in Fig. 8.4 allows the pitch and duration values to be changed as the program runs and plays the result.



```
   1 PRINT AT 1,0;" 1 TO DECREAS
E PITCH BY .1"
   2 PRINT " 2 TO DECREASE PITCH
 BY 1"
   3 PRINT " 9 TO INCREASE PITCH
 BY 1"
   4 PRINT " 0 TO INCREASE PITCH
 BY .1"
   5 PRINT AT 16,0;" a TO DECREA
SE DURATION BY .01"
   6 PRINT AT 17,0;" s TO DECREA
SE DURATION BY .1"
   7 PRINT AT 18,0;" k TO INCREA
SE DURATION BY .1"
```



```
   8 PRINT AT 19,0;" 1 TO INCREA
SE DURATION BY .01"
   9 PRINT AT 0,10;"PRESS"
  10 LET p=0: LET d=.1
  50 IF INKEY$="1" THEN  LET p=p
-.1
  51 IF INKEY$="2" THEN  LET p=p
-1
  55 IF p<-57 THEN  LET p=-57
  60 IF INKEY$="0" THEN  LET p=p
+.1
  61 IF INKEY$="9" THEN  LET p=p
+1
  65 IF p>69 THEN  LET p=69
  70 IF INKEY$="a" THEN  LET d=d
-.01
  71 IF INKEY$="s" THEN  LET d=d
-.1
  80 IF INKEY$="l" THEN  LET d=d
+.01
  81 IF INKEY$="k" THEN  LET d=d
+.1
  90 IF d<0.02 THEN  LET d=0.02
  95 PRINT AT 10,10;" DURATION :
";STR$ (d);"       "
  96 PRINT AT 12,10;" PITCH:";ST
R$ (p);"      "
 100 BEEP d,p
 150 GO TO 50
```

Figure 8.4

9 PUZZLES, TRICKS, AND TRAPS

This chapter is an unashamed collection of bits and pieces, puzzles and problems that can be built into a game to add to its variety and challenge. We will start by looking at Cad's favourite command, HELP!, and then look at some of the things he might like help with.

9.1 The HELP command

At any stage in an adventure the player may be stuck, either because he or she does not know what to do next (especially in a puzzle game) or because he or she has forgotten some vital piece of information, such as the spells that can be used in the combat game. The HELP command can serve two functions: it can give the stuck player clues and the forgetful player reminders. Both functions can be used in the same game, but usually only one is available. To use both, it is probably best to have two commands—HELP for clues and INSTR(uctions) for instructions. If you are using full-word input, then INSTR need only call up the same instruction routines as the initial set-up or a subset of these. It would be wasteful to use a different instruction routine unless it is discovered that allowed instructions are actually part of the game.

For example, in a space game the starship may be travelling around gathering artifacts from an ancient civilization, and each artifact might allow a different action by the ship (such as faster travel, better detection, new weapons, etc.). We might not wish the player to know that such possibilities were available until the appropriate artifact had been found. One way to do this would be to give each artifact a code number and keep a record of all the numbers possessed by the ship/player (using a method like that of object-gathering in Camelot). Then, when the INSTR command was input, a number of separate instruction routines would be called, one for each artifact.

The HELP command can be more complicated. The kind of clues offered may depend on the current location, abilities, and possessions of the character. Its simplest form is often found in the puzzle game, where a single clue is available at each location, the clues not changing at all. Thus if the player finds a stuck door, typing HELP may get the response 'The hinges are rusty'. This should be enough of a clue to suggest that oil is needed. In some cases no clue might be available, so a non-committal response is needed as the default response, such as 'You seem to be doing fine' or 'I don't think you deserve any help'.

This means holding a series of strings, one for each location that needs a clue and one default string for all other locations. Typing HELP would send control to a routine which prints the string which is appropriate to the current location. The easiest way to obtain such conditional access to strings is by using an array holding all the strings, where the subscript of the array is the number of the current location. The drawback with this is that locations without clues will still get reserved space in the array, which is very wasteful in memory. So here we have a case for a conditional GOTO statement (which the Spectrum does not have). The statement in Microsoft or BBC BASIC would be (if there were five possible locations):

```
ON LOCATION GOTO 5ØØ5, 5Ø3Ø, 5Ø15, 5Ø3Ø, 5Ø25
```

where 5Ø3Ø is used twice because it is the non-committal response. Each of the statements 5ØØØ to 5Ø3Ø would PRINT the appropriate clue.

In Spectrum BASIC we have to be more crafty. We use a GOTO with a variable derived from the location number. If the room has no clue it has to pass through two GOTO statements, thus:

```
 8ØØ GOSUB 5ØØØ
4999 REM CLUE ROUTINES
5ØØØ GOTO 5ØØØ + (LOCATION *1Ø)
5Ø1Ø PRINT "The hinges are rusty": RETURN
5Ø2Ø GOTO 5Ø6Ø
5Ø3Ø PRINT "Locks need keys": RETURN
5Ø4Ø GOTO 5Ø6Ø
5Ø5Ø PRINT "What kind of herb needs waiting for?": RETURN
5Ø6Ø PRINT "You're doing fine": RETURN
```

The kinds of clues given should not make the solution too easy or else the puzzle will be solved without much effort. They should aim to set the player in the right direction without explicitly giving the answer. An example would be line 5Ø5Ø above, "What kind of herb needs waiting for?". The answer is 'thyme' but the clue does not tell the player that. It forces him or her to do some thinking, though not as much as the problem it helps with.

We should bear in mind when writing the HELP commands that different sets of conditions might be true in one location at different times. For example, in a jail the player may have to go through the following stages to escape:

1. Obtain a meal.
2. Find the file hidden in the meal.
3. File through his handcuffs.
4. Break a leg off the bed.
5. Use the leg to bend the window bars apart.

This would probably be marked by a flag being incremented from 0 to 5; when it reaches 5 the player could escape. So if HELP was typed while the player was at that location five possible helping hands might be necessary. These would have to be accessed through a combination of two tests, one sending control to the location routine, the other within that location routine picking the correct clue from the five. Thus:

```
5ØØØ GOTO 5ØØØ + (LOCATION *1Ø)
5Ø1Ø REM JAIL
5Ø11 GOTO 5Ø11 + FLAG
5Ø12 PRINT "I feel hungry": RETURN
5Ø13 PRINT "I don't like lumpy cake": RETURN
5Ø14 PRINT "A file is a handy object": RETURN
5Ø15 PRINT "Now to unmake the bed": RETURN
5Ø16 PRINT "Just a leg through the window and we're free":
     RETURN
```

We might even wish to make the HELP commands more specific, so that tests are made to see whether the player has the objects needed for particular actions; if not, remind him of the fact. For example, we could check that Cad had all the objects needed to pass through the next stage of the adventure before entering that stage, and the HELP command could indicate what might be missing.

9.2 An anagram puzzle

A simple puzzle routine which does not take up much room yet can be used over and over again in a game is the anagram puzzle. The routine listed in Fig. 9.1 uses words read from DATA statements and it would be quite possible to use statements which already exist in the program for some other purpose to feed this routine, providing they were strings and organized in the correct way. If you have memory to spare you can add especially complex vocabulary, but it is not necessary.

To use existing DATA statements proceed as follows. Firstly, ensure that they have been READ properly for their primary purpose (e.g., if they are to be READ into string arrays). Then RESTORE the DATA pointer to the beginning of the string DATA which will be used by the anagram routine. If they are the first DATA statements in your program, you need only RESTORE at the beginning of the anagram routine: if the relevant statements are not the first, then put RESTORE n, where n is the line number of the first relevant DATA statement. However, if you use DATA statements anywhere else in the program make sure that you RESTORE the data pointer to the beginning of the set of DATA that you want.

The anagram routine works as follows. Firstly, a word is selected from the data. This chosen word is held in W$ and in V$. Line 2Ø then generates a random whole number, R, which is in the range 1 to the length of the chosen word. Line 39Ø adds to the string Y$ (which is initially an empty string) the Rth character of the input word, V$. Line 4Ø then removes the Rth letter from V$. Line 45 checks that there are characters left in V$ and if so repeats the random choosing process. Lines 5Ø to 9Ø ask for an answer and continue asking until the correct answer is found. This would have to be changed in a full game.

This is only an illustrative routine and should be modified for any serious use. The player should be allowed only a limited number of guesses and some option to end the routine. As it stands numbers and punctuation will all be treated as if they were words or parts of words. This might be useful (e.g., we might want the computer to generate a code of four digits as the combination to a safe and then scramble them using the anagram routine to give a partial clue to the player), but generally anagrams are only words, so we would want to check that all the selected characters were codes between 65 and 90 or between 97 and 122 (upper and lower case respectively). This would be even more important if the game depended on keyboard input for its chosen word.

However, these are all simple adaptations. The problems that can be created for a player by using such a routine are well worth the work the routine might need.

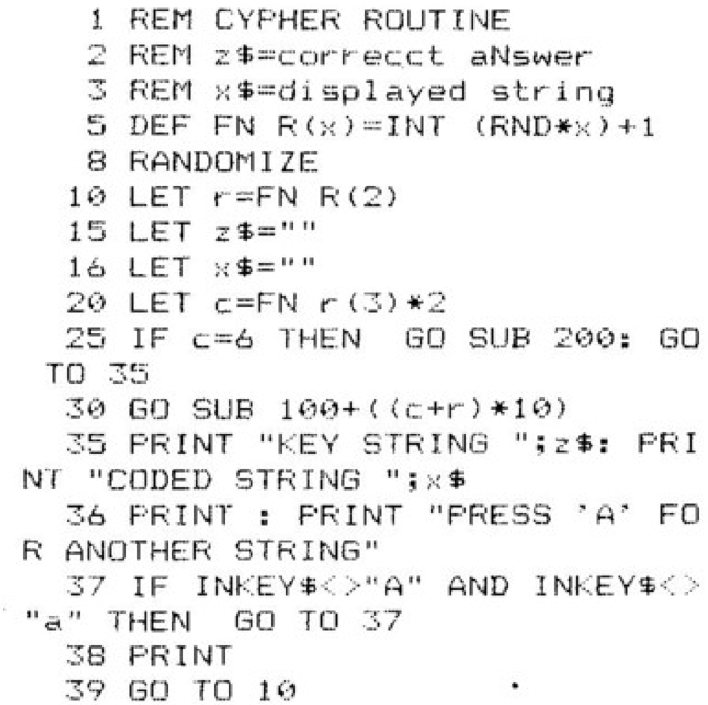
There are many puzzles of similar types which can be incorporated as elements of large adventure games. In a sense all adventures are really a series of small games linked together— a number of puzzles and/or tests of strategy and reaction which are all linked together. Camelot incorporates tests of logic, tests of imagination (or, to put it more bluntly, puns) and tests of verbal skill. Other small games which can be included involve mathematical puzzles, tests of general knowledge, cypher tests (like the Mastermind game), or tests of language (such as synonyms, acronyms, or abbreviations).

```
  5 LET V$ ="": LET Y$ =""
 1Ø LET R = (RND * 1Ø)+1
 15 FOR I = 1 TO R: READ V$: NEXT I
 2Ø LET W$ = V$
 25 LET R = INT (RND * (LEN(V$))) + 1
 3Ø LET Y$ = Y$ + V$(R)
 4Ø LET V$ = V$(1 TO R−1) + V$(R + 1 TO LEN(V$))
 45 IF V$ <> "" THEN GOTO 2Ø
 5Ø PRINT Y$; " is an anagram of what word?"
 6Ø PRINT "Please type your answer"
 7Ø INPUT A$
 8Ø IF A$ = W$ THEN PRINT "Correct": GOTO 1ØØ
 9Ø PRINT "Wrong. Try again": GOTO 5Ø
1ØØ STOP
11Ø DATA "boomerang", "hyphen", "catastrophe", "elephant",
    "waterfall", "cormorant", "computer", "adventure",
    "programmer", "analogue"
```

Figure 9.1 Anagram routine

9.3 A cypher routine

Elsewhere we looked at the possibility of using a routine which could generate new cyphers for each game. The program in Fig. 9.2 is a very simple routine which randomly selects one of five ways to produce a string for such a code.



```
   1 REM CYPHER ROUTINE
   2 REM z$=correcct aNswer
   3 REM x$=displayed string
   5 DEF FN R(x)=INT (RND*x)+1
   8 RANDOMIZE
  10 LET r=FN R(2)
  15 LET z$=""
  16 LET x$=""
  20 LET c=FN r(3)*2
  25 IF c=6 THEN  GO SUB 200: GO
 TO 35
  30 GO SUB 100+((c+r)*10)
  35 PRINT "KEY STRING ";z$: PRI
NT "CODED STRING ";x$
  36 PRINT : PRINT "PRESS 'A' FO
R ANOTHER STRING"
  37 IF INKEY$<>"A" AND INKEY$<>
"a" THEN  GO TO 37
  38 PRINT
  39 GO TO 10
  40 STOP
 129 REM four random integers
 131 FOR i=1 TO 4
 132 LET r=FN r(9)
 133 LET z$=z$+STR$ (r)
 134 NEXT i
 135 LET x$=z$
 138 RETURN
 139 REM four random letters
 140 FOR i=1 TO 4
 141 LET r=FN r(26)
 142 LET z$=z$+CHR$ (64+r)
 143 NEXT i
 144 LET x$=z$
 148 RETURN
 149 REM four random integers en
coded
 151 LET r=FN r(8)+1
 152 GO SUB 130
 153 LET x$=""
 154 FOR i=1 TO LEN (z$)
 155 LET x=VAL (z$(i))+r
 156 IF x>9 THEN  LET x=x-9
 157 LET x$=x$+STR$ (x)
 158 NEXT i
 159 RETURN
 160 REM four random letters enc
oded
 161 GO SUB 140
 162 LET r=FN r(26)
 163 LET x$=""
 164 FOR i=1 TO LEN (z$)
 165 LET x=CODE (z$(i))+r
 166 IF x>90 THEN  LET x=x-26
 167 LET x$=x$+CHR$ (x)
 168 NEXT i
 169 RETURN
 200 REM encode a real word
 201 LET r=FN r(20)
 202 REM the number in brackets
= the total possible words
 205 RESTORE 260
 210 FOR i=1 TO r
 220 READ z$
 230 NEXT i
 240 GO SUB 162
 250 RETURN
 260 DATA "DOOR","CLAW","SHOT","
FILL","CART"
 265 DATA "HAND","HEAD","SLAP","
BRAG","LION"
 270 DATA "GOAT","TRIP","JUMP","
```

```
HEAL","CRAG"
 275 DATA "HELP","MOON","SWAY","
CHIP","LEAD"
```

Figure 9.2

It produces two strings, X$ and Z$, printed out at line 35. Each time the program is RUN, Z$ will be assigned a chosen string and X$ will hold a version of Z$ which will be displayed at some point in the rest of the game. In two of the five choices X$ and Z$ are the same. Therefore they could be used as 'key' and 'lock'. A NEED square holds Z$ and the equivalent GET square, giving the information needed to pass the NEED square, will hold X$. These are very simple codes, each being a string of either four letters or four numbers.

In two other cases a random string is chosen to fill Z$ and then the string is manipulated before being held in X$. In the case of the numbers, each digit has a value added to it (the same value for each of the four) so the player, on discovering X$, must also discover the value that has to be subtracted from each digit to give the correct answer. A key line here is line 156 which tests to see if the addition of the chosen number and the value is greater than nine; if it is, it divides by 10 and gets rid of the decimal part of the number. This makes the guessing more tricky. The player should therefore be made to pay for each guess at the required value, perhaps having a reduction in intelligence each time he fails.

The manipulated character string is similarly transformed. Each letter in the chosen string is replaced by another letter a fixed number of letters away in the alphabet. If the letters go beyond Z they begin again at A, so that X plus 4 gives B. The player has a more difficult task than with the digits as there are 26 possible answers to the chosen value as well as a larger number of possible combinations of characters in the original string.

The fifth choice takes a real word from one of the DATA statements and manipulates it in the same way as the randomly chosen character string. The player now has a better chance because he can use two pieces of information—the letters are all encoded by other letters a fixed distance away and the complete string makes sense. If other DATA statements are added then the range of the random number in line 2Ø1 needs to be changed to the total number of possible words.

As usual with the use of DATA, any DATA statements in the program could be used. One possibility is to hold some form of clue in the DATA, such as the whereabouts of a treasure, split up into component words. The player can then discover the clue one word at a time until he has sufficient to be able to guess the whole thing. Such a use of this puzzle would thus have four stages: finding the original encoded string(s), finding the 'money' needed to pay for guesses, making the correct guess, and piecing all the words together to make a coherent clue. Then, of course, the clue itself may be cryptic. . .

Using the same type of design it would be quite possible to devise complete mystical alphabets for players to decipher during the game (e.g., in a game like the Egyptian game described in Chapter 2). Instead of using the conventional alphabet as the encoding string a string of user-defined graphics can be used, so that the player will initially have no idea what they represent nor how they fit with normal language.

It is also possible to build string functions to act as simple encoding devices which can be applied at any point in the game. Suppose we have one routine which allows Sages to give clues as to the whereabouts of valuable artifacts. A function such as the one below could be applied every time such a clue is uttered so that the player is given his information, providing he can decode it:

```
DEF FN A$(B$) = B$((LEN(B$)−3) TO LEN(B$)) + B$(1 TO 4) +
B$(5 TO (LEN (B$)−3))
```

where B$ is the clue string and must be more than eight characters long. This function simply splits it up into three unequal portions and recombines it, so that the actual string given out is a garbled version of the original.

9.4 Teleport

A common trap in many adventures is the warp or teleporrter which transports the character to some unknown destination in the game world. In many puzzle games this cannot usually be used because the player has to get through all locations in a more or less fixed order, but in some it can form a way of sending the player into the next stage of the game (so that it appears he has become lost because the surroundings are unfamiliar, but in actuality he is on the correct route through the logical structure of the game) or a limited trap which sends the player back to an earlier stage. This may perhaps create a problem, depending on the objects carried by the character when the transportation occurs.

However, the random transporter is usually only used in combat games where a maze of some kind is involved. The transporter routine calculates a random location in the maze and then places the character in that location. It would be possible therefore for a routine such as this in the Merlin game to locate the player in a section of the caverns which has no surface exit (though the formula used makes this unlikely). We must either use the device only in dungeons where we can be certain there is a possible exit, namely, fixed map combat games, or we must test to ensure that there is an exit from the location the routine produces. The latter choice can be difficult. The easy compromise is to incorporate the inadequacy as an apparent part of the design by PRINTing a comment which laments the loss of the player.

If we wanted to incorporate such a routine into The Mines of Merlin we would thus have to make it rare (so the player was not arbitrarily trapped too often) and to try to maximize the possibility of an exit as in the routine below:

```
65ØØ REM TRANSPORTER
651Ø LET N = 1ØØ: GOSUB 28ØØ
652Ø IF Q <> 1 THEN RETURN
653Ø LET Z = FNR(3) + 1
654Ø LET X = FNR(2Ø)
655Ø LET Y = FNR(2Ø)
656Ø GOSUB 1ØØØ
657Ø IF MOVE = Ø OR MOVE = 2 THEN GOTO 653Ø
658Ø PRINT "Aaaaaaaaaaah!! You've stepped into a teleporter"
659Ø PRINT "You may never get out of here now"
6599 RETURN
```

Note the use of FNR(x) in this routine. This is the same function as that defined in The Mines of Merlin in line 8Ø2Ø, a single function which can be used to generate any random number in any range within a program. The number in brackets in the function call defines the range of the potential numbers.

Lines 651Ø to 652Ø decide if there is a transporter at that location, using the built-in random number. The routine assumes that it has been called by another routine, probably the 'hazard' routine, and simply returns there if Q is any value other than one. Lines 653Ø to 655Ø set the new value for the location, X and Y being in a large range but Z being restricted to the second to fourth levels only, so the player will not be sent to a remotely deep level with no hope of rediscovering the surface. Subroutine 1ØØØ calculates the new movement and, if movement there is not possible, loops back to do the whole thing again. The final lines PRINT an appropriate message and return control to the calling routine.

A similar trap can create an apparent maze without the need for a complete maze-building program. The puzzle game can use this best but it could be incorporated into a combat game with slight modification. The idea is to give the illusion of a maze without actually creating one. This is simple to do. We create an endless loop which accepts all movement commands but continually PRINTs a message like 'You are lost in the maze's endless corridors'. Obviously there would not be much point to this on its own, so there must be some way out of the maze (using the correct object in the correct way, or typing a suitably clever command) or some kind of limit set (such as using a very long loop which nevertheless has an end, or always letting one direction lead back to the start). Normally players entering such a maze will persist for 10 or 20 moves, until they think they can go no further, so if we set a counter in the loop such that after 25 moves an object is found we reward persistence and make the maze appear varied. If Cad finds the object he may well believe that the maze actually exists and simply has to navigate around it in the usual way.

Here is a simple version of this:

```
66ØØ REM A PSEUDO-MAZE
661Ø PRINT "You are lost in the corridors of an endless maze"
662Ø GOSUB (command routine)
663Ø IF (command is 'I am not amazed') THEN PRINT "You have
     found the exit": GOTO 669Ø
664Ø IF (move is West) THEN LET WESTCOUNT = WEST-
     COUNT + 1
665Ø LET COUNTER = COUNTER + 1
666Ø IF COUNTER > 25 THEN PRINT "You see an emerald
     statue": GOSUB (object routine): LET COUNTER = Ø
667Ø IF WESTCOUNT > 1Ø THEN PRINT "There is a little old
     man in the corridor": GOSUB (little old man): LET WEST-
     COUNT = Ø
668Ø GOTO 661Ø
669Ø RETURN
```

Again this routine should be called by another. It is an endless loop containing three tests. Test 1, at line 663Ø, leaves the loop if the player has found the correct cryptic phrase. Test 2, at line 666Ø, reveals an object if the player has gone sufficiently far into the maze. Test 3 reveals a character if the player has gone far enough west. Note that the two counters are reset to zero by these lines, so the player will find the statue every 25 turns. Consequently, an additional test has to be built in to see if the statue is still there, so the 'object' routine will have to set an appropriate flag if it is taken or destroyed and line 666Ø must also ensure that this flag is not set. A similar test structure would be needed for the little old man.

9.5 Bribery and gambling

In The Mines of Merlin we only allowed a simple response to monsters— either Cad runs away from them or he or she fights them.

As the encounters with monsters form the main events in such a game it is worth considering some additions that can make it more interesting. Two such additions are bribery and gambling.

Both of these involve the same kind of exchange with probably the same result— the player loses money and so is less likely to be able to buy what he needs in other locations. Bribery consists of a request from the monster for money or some other gift followed by the player continuing on his way without harm if the monster gets what he wants. The bribe function can be incorporated in one of two ways: either the player has a free choice of it as one of his strategies or before he is allowed to make his choice the monster will ask for a bribe. Fig. 9.3 shows a way of constructing the routine to fit with our game.

```
 11Ø PRINT "Do you want to:"
 12Ø PRINT "(F) Fight"
 13Ø PRINT "(R) Retreat"
 14Ø PRINT "(B) Bribe"
 15Ø INPUT A$: REM CAN USE INKEY$ AND VALIDATION
     ROUTINE IN MINES OF MERLIN
 16Ø LET G = Ø
 17Ø IF A$ = "B" THEN GOSUB 35ØØ: IF G = 1 THEN RETURN
 18Ø IF G = 2 THEN GOSUB (fight routine)
 19Ø CLS: GOTO 11Ø
35ØØ REM BRIBES
351Ø LET R = FN R(money)
352Ø PRINT "He says he will accept"; R "silver pieces"
353Ø PRINT "Will you pay?"
354Ø INPUT A$
355Ø IF A$(1) = "N" THEN RETURN
356Ø IF money < R THEN PRINT "Try to swindle me, would
     you?": LET G=2: RETURN
357Ø LET MONEY = MONEY-R
358Ø LET G = 1
359Ø RETURN
```

Figure 9.3

The variable 'money' represents the current total of silver pieces that the character has. In The Mines of Merlin this is C(7). The routine can be improved by making the monster ask for weapons or other items that the character is carrying. This could be the fall-back request if the character has not enough money or it could be randomly determined before the routine is run. Alternatively, the monster could steal all of the player's money if he decides not to bribe.

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

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

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

```
1ØØ PRINT "How much would you like to bet?"
11Ø INPUT A
115 IF A >money THEN PRINT "I'm afraid your calculator needs
    new batteries": CLS: GOTO 1ØØ
12Ø LET money = money−A
125 PRINT "What would you like to multiply your ";A;" silver
    pieces by?"
13Ø PRINT "Please type a number between 2 and 2Ø"
14Ø INPUT B: IF B<2 OR B>2Ø THEN PRINT "Perhaps you
    should learn some maths before you play with the big boys?":
    CLS: GOTO 12Ø
15Ø PRINT "That's a cool";A*B; "you could win"
16Ø LET R = INT(RND *B) + 1
17Ø IF R = 1 THEN PRINT "And you win it!!!"; LET money =
    money + R: RETURN
18Ø PRINT "Oh what a terrible shame. You lose. Never mind, it's
    only money"
19Ø IF money > Ø THEN PRINT "Another go?"
2ØØ INPUT A$
21Ø IF A$(1) = "Y" THEN GOTO 1ØØ
```

Any of the micro gambling games, such as fruit machine or roulette or dice, can be adapted to fit in as a subroutine in a larger game, having the advantages of more complex and varied odds and often interesting graphics. A simple way to test the player's knowledge of probabilities is to ask him to bet on the total of two six-sided dice, giving either a flat doubling of the bet if he wins (in which case he is best advised to guess 'seven' every time) or linking the winnings to the odds. For example, throwing two dice will give an average of one 12 in every 36 throws, so we can safely offer odds of 3Ø to 1 on a 12 and be sure that the player will lose in the long run.

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

One other option to use in monster encounters is the TALK routine discussed in Chapter 7. This can also be linked to bribing and gambling so that if keywords on these topics are found the program switches to the program in the appropriate routine.

10 GRAPHICS

There is no reason why graphics should be used in an adventure game, particularly those games such as puzzle games where most of the information has to be textual. Using graphics for the sake of it may consume memory that might be better used for maps or data, but if there is memory left after you have written your masterpiece of a main program you can improve attractiveness greatly by the addition of graphics. However, if you do intend to use some form of graphics in your game you should try to make the decision as early as possible in the design. This ensures that your graphics are an integral functioning part of the game rather than something added on which contributes little, and makes sure you keep an eye on memory consumption throughout your design.

10.1 Graphics and the Spectrum

There are so many popular microcomputer graphic systems that it is impossible to consider many aspects here. If detailed and complex graphics are wanted in your games you should read a book on implementing graphics for your system. Some aspects are quite common, and the Spectrum contains several graphic functions which are similar to those available on other machines. The key graphical functions available on the Spectrum are:

1. Block graphics (using PRINT)
2. User-defined graphics (using PRINT)
3. High-resolution pixel graphics (using PLOT, DRAW, and related commands)

Some form of block graphics are available on most microcomputers. If we imagine the VDU screen divided into squares, or character positions, each block will occupy one such position. On the Spectrum the character screen is 22 lines by 32 columns, giving 704 squares. Each of these squares is an 8 × 8 array of dots and each block graphic (including the alphabet, numbers, and numeric signs) is a particular combination of some of those 64 dots. If you do not understand this you should reread Chapters 15, 16, and 17 of the Spectrum manual.

The easiest way to understand this is to think of each little 8 × 8 block as a blank page. On this page are a number of dots; if you join any combination of dots then, just like a child's join-the-dots puzzle, a pattern will appear. The only difference is that instead of joining dots the micro produces patterns by switching dots on and off. The dots which are switched on or lit give the pattern (the INK colour of Spectrum graphics); those which are switched off give the background (the PAPER colour of the Spectrum).

The BIN statement of the Spectrum allows us to control all of the dots in a particular square, so we can join up our own dots, i.e., define a new pattern, a new character. If, for example, we wanted a small triangle, we could plot it on a grid of 8 × 8 pixels as in Fig. 10.1. This drawing can be represented as the series of BIN statements next to that figure. You will see that there is a direct correspondence between the pattern of pixels which are used and the pattern of ones in the BIN statement. This is because each row in the character block is held in memory as a byte (i.e., eight bits of binary information). The BIN statement allows us to address each bit in each of these bytes, saying whether it is on or off (one or zero). Eight such bytes make up one character.

```
BIN 00000001
BIN 00000011
BIN 00000111
BIN 00001111
BIN 00011111
BIN 00111111
BIN 01111111
BIN 11111111
```

Figure 10.1

Other machines use different sytems to do the same thing. For example, the user-defined graphics on the BBC use a number between 0 and 255 to represent each row. You will remember that this is the range of numbers that can be held by one byte (or eight bits), so it can be seen that each of the numbers 0 to 255 will mean a different combination of offs and ons for the row. Eight such numbers form a character.

On the Spectrum we can have up to 21 user-defined graphics stored at addresses USR "A" to USR "U". These graphics can be printed in the same way as the normal letters A to U using the graphics mode of the Spectrum (the graphics mode is obtained by pressing Caps Shift and 9). Twenty-one may seem quite a lot, but they can be used up quite quickly in a graphics adventure. This is because to create interesting new graphics one user-defined square is generally not enough. Although one square can provide simple shapes, faces, spaceships, cars, cannon, etc., it cannot give enough detail for drawing things like trolls or dragons or elaborate buildings and artifacts.

To get over this problem we have to draw our design not on one 8 × 8 grid but on several adjacent grids. If our object is to have a long vertical axis put two grids on top of each other; if it is long horizontally put the grids side by side. Let us draw a cyclops and a dragon. The cyclops will be tall so we align the grids as shown in Fig. 10.2.

```
BIN ØØØ11ØØØ
BIN ØØ1111ØØ
BIN Ø11ØØ11Ø
BIN Ø111111Ø
BIN ØØ1111Ø1
BIN ØØØ11ØØ1
BIN ØØ1111Ø1
BIN Ø1Ø11Ø11
BIN Ø1Ø11ØØ1
BIN 11Ø11ØØØ
BIN ØØ1111ØØ
BIN ØØ1ØØ1ØØ
BIN ØØ1ØØ1ØØ
BIN Ø11ØØ11Ø
BIN ØØØØØØØØ
BIN ØØØØØØØØ
```

Figure 10.2 A cyclops

You can see that for the cyclops we have 16 BIN statements— two sets of eight. The first set is the head and shoulders of the monster, the second set his torso and legs. In order to put these two characters together to make one monster we must remember which letters each part is stored under. We will make the top part ‘A’ and the bottom part ‘B’.

We can control the printing of the monster by using the PRINT AT statement. If we PRINT graphics A at 10,10, then we will have to PRINT graphics B at 11,10. This means careful program design when using such graphics. The best way is to use a short routine for printing each monster, even if the routine is only one line. The PRINT AT parameters in the routine would have to be variables passed to the subroutine by the main program. So the routine for printing one cyclops might be like this:

```
8999 REM CYCLOPS
9ØØØ PRINT AT Y,X; "(graphics A)": PRINT AT Y+1,X;
     "(graphics B)"
9Ø1Ø RETURN
```

For drawing the dragon we use three blocks of eight and print them on the same line, one after the other, thus: PRINT AT Y,X; "(graphics C)"; PRINT AT Y,X+1; "(graphics D)"; PRINT AT Y,X+2; "(graphics E)". The same instruction can be written as PRINT AT Y,X; "(graphics C)(graphicsD)(graphics E)", so it is easier to print horizontal than vertical designs. The dragon is shown in Fig. 10.3.

```
BIN ØØØ11111    BIN 11ØØØØØØ    BIN ØØØØØØØØ
BIN ØØØØØØ11    BIN 11111ØØØ    BIN ØØØØØØØØ
BIN Ø1ØØØØØØ    BIN 111111ØØ    BIN ØØ1ØØØØØ
BIN 111ØØØØØ    BIN ØØ11111Ø    BIN ØØ11ØØ1Ø
BIN Ø1ØØØØØØ    BIN 11111111    BIN 11111111
BIN Ø11ØØØØ1    BIN 11ØØØ1ØØ    BIN 1ØØ111ØØ
BIN ØØ11ØØ11    BIN 11ØØØ1ØØ    BIN ØØØØØØØØ
BIN ØØØ11111    BIN ØØØØ1111    BIN ØØØØØØØØ
```

Figure 10.3

The drawback with these graphics should now be obvious. In order to get a figure that is at all life-like, two, three, or even more blocks must be used, and this rapidly reduces the available graphics. We have used one-quarter of the available blocks to create two monsters, so this would suggest that the average game could only get about eight or nine worth while graphic elements.

There is a way out of this. We can redefine our graphics according to their current purpose. Suppose we have used all of the user-defined slots and now wanted to use the dragon in the program. Providing we do not want all the other graphics and the dragon displayed at the same time, we can select three of the user-defined graphics which are not being used when the dragon is wanted and read the dragon data into these slots. Then, when the dragon is no longer needed we can reread the original data back in.

To do this we will need a general routine for reading the BIN statements which also reassigns the letter being used, all the separate BIN statements, and careful planning. If we do not plan well all the graphics will become confused and displays will end up with cyclodragotrolls all over them. It makes sense, therefore, to include before each set of BIN statements a REM line reminding us what the next eight BIN lines represent, and which letter (USR address) they are held in. The first cyclops block might thus be stored in a program as:

```
9499 REM CYCLOPS HEAD
95ØØ DATA "a"
951Ø DATA BIN ØØØ11ØØØ
952Ø DATA BIN ØØ1111ØØ
953Ø DATA BIN Ø11ØØ11Ø
etc.
```

To read this pattern into an existing graphic block we first have to RESTORE the DATA pointer to the relevant line. To make things easy we can point to the REM line, as the RESTORE statement causes the DATA to be read from the next line after the pointer which has DATA in it; in this way we will find it much easier to remember what each restore number means. The read-in routine would thus be:

```
 999 REM TO READ DIFFERENT GRAPHICS
1ØØØ RESTORE 9499
1Ø1Ø READ M$ : REM USR ADDRESSED LETTER
1Ø2Ø FOR I = Ø TO 7
1Ø3Ø READ N
1Ø4Ø POKE USR M$ +I,N
1Ø5Ø NEXT I
1Ø6Ø RETURN
```

However, because we want the RESTORE pointer to be a variable as well, it is usually better to have a variable previously declared which RESTORE will use. If we changed line 1ØØØ to:

```
1ØØØ RESTORE monster
```

the variable 'monster' would have been set by the main program which passes control to this read-in routine. For example, suppose the monster that was used depended on the current strength of the player's character. Then the calling routine could be:

```
5ØØ LET monster = 9ØØØ + (STRENGTH *1ØØ)
51Ø GOSUB 1ØØØ
```

Remember that the monster BIN statements can themselves be variables, and we can use zero in place of BIN ØØØØØØØØ. We might wish to use variables in these DATA statements if a group of graphics had the same basic design. For example, we might use graphics to show what the player was currently carrying or wearing, rather than displaying an inventory. In such a case we could use a set-up with five graphic blocks, like Fig. 10.4.

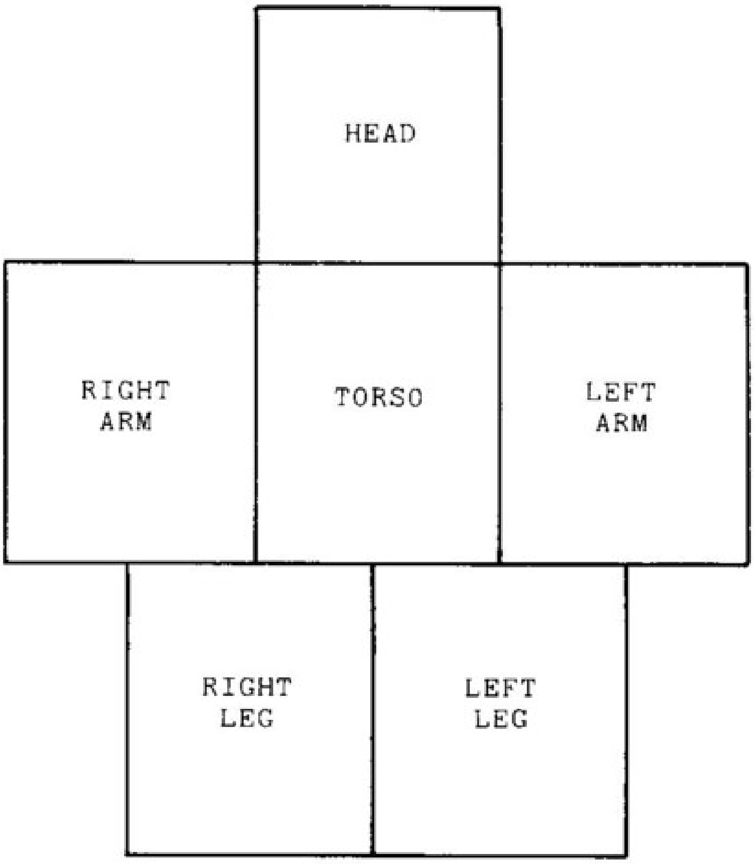


Figure 10.4

Each of these five blocks could change according to the character's current status. He or she could be wearing leg armour, gauntlets, a breastplate, boots, or nothing; could be carrying a shield, any number of weapons, a torch, a sack, or any of a multitude of objects. Suppose we wanted to show that the torch was lit and currently held in the left hand. The graphic block, and hence the BIN statements, would be exactly the same for the lit and unlit torches, except that a flame would be added. By using one set of variables for 'left arm' and another for 'torch' and a third for 'flame', we could make the required graphic blocks simply by combining the required sets of variables.

We can use the graphics we have defined in exactly the same way as ordinary characters or block graphics. Thus they can be repeatedly PRINTed to make elaborate designs; they can be animated or moved around the screen, as we will see in the Treasure Trove game below; they can be combined with the standard character set for complex effects; they can be added to block graphic designs; and they can be combined in the same PRINT position using OVER. This means that two (or more) user-defined graphics can be overprinted on the same character position so that, for example, a figure can be shown walking through a doorway by PRINTing the doorway, then PRINTing the figure over it, then PRINTing the doorway again.

We can also use a single graphics character in different ways by giving it different attributes. This is an economical way of creating 'different' characters. For example, if we define a simple graphic man using BIN, we can then make him a goblin by PRINTing him in blue or a skeleton in white, a magician by using BRIGHT, a ghost walking through walls by using OVER, a magical creature by using FLASH, and so forth. Similarly, if we are designing a graphical adventure with a printed map the same sign on the map can stand for different things by using different colours. A series of irregular triangles can be hills, mountains, pyramids, or sea; vegetation could be swamp, forest, scrub, etc., simply by changing the colour of the sign.

There is one problem with the Spectrum's graphics, including the user-defined graphics. The function SCREEN$(x,y) will normally return the character found at position x,y on the screen. However, it does not recognize characters with CODEs below 32 or above 127. Consequently, we cannot use SCREEN$ to check on the screen status if it is full of user-defined graphics, or even block graphics. Thus a game which moves a graphic adventurer across the screen looking for monsters and treasures will not work if the program is designed to depend on the displayed symbols for these things.

There are three simple ways to get round this problem. One is to use non-graphic characters for functions where screen detection is required and reserve the other graphics solely for decorative effect. This seems a waste of the Spectrum's excellent graphic resources.

The second method is to hold in memory an exact map of what is displayed on the screen and use this representation for collision detection rather than the displayed screen map. This is essentially the method used in The Throne of Camelot. However, for a large map and/or screen display this can be very expensive on memory and may be slow to process. It also seems rather inelegant to hold two representations of the same information simultaneously if it can be avoided.

The third method is to use the attributes of screen locations as indicators of the adventurer's status. With eight INK colours, eight PAPER colours, two flashing states, and two brightness states there are 256 possible combinations that can be tested for (do you recognize the magic number?).

For example, suppose the displayed map is of a wilderness containing forests, mountains, a lake, and cities. We can use the user-defined graphics to draw this map, making it visually as interesting as possible. If, in PRINTing it, we ensure that forests use green INK, mountains magenta, lakes blue, and cities black then we can use the attribute of the current screen position of the adventurer to control the program. Just as in The Throne of Camelot the main loop moved the player (changed the variable called 'location') and then used the 'attributes' of that location which had been coded in the program to determine which routines ran next, so we can continually move the player's persona across the screen and use its attributes to govern the rest of the program.

This is done using ATTR(x,y). This function returns a number encoding the current status of any screen location— what colour INK it has, what colour PAPER it has, whether it is BRIGHT or not, and whether it is FLASHing or not. In the present example we are interested in the colour of the INK, i.e., the range of numbers 0 to 7. Consequently, to decode the number returned by ATTR we must reduce the number to below eight. We must therefore do the following calculation:

1. IF THE NUMBER IS GREATER THAN 128, SUBTRACT 128
2. IF THE RESULTING NUMBER IS GREATER THAN 64, SUBTRACT 64
3. IF THE RESULTING NUMBER IS GREATER THAN 32, SUBTRACT 32
4. IF THE RESULTING NUMBER IS GREATER THAN 16, SUBTRACT 16
5. IF THE RESULTING NUMBER IS GREATER THAN EIGHT, SUBTRACT EIGHT.

(As we will see in Treasure Trove we do not need to go to such lengths in one special case.)

This leaves us with a number between 1 and 7 which holds the INK colour. We then ask which of the four colours possible on our map that number is and send control accordingly, using lines like:

```
IF N = Ø THEN GOSUB CITY
IF N = 1 THEN GOSUB SEA
IF N = 2 THEN GOSUB MOUNTAIN
IF N = 3 THEN GOSUB FOREST
GOTO BEGINNING OF MAIN LOOP
```

The tedious part of using the user-defined graphics is actually designing them in the first place and working out the BIN statements and how they are to be printed. There now follows a listing of a number of predefined graphics for use in your own programs. Each is preceded by a REM statement explaining how to PRINT it.

```
9ØØØ REM A SACK OF MONEY: 1 BLOCK
9Ø1Ø BIN ØØØØØØØØ
9Ø2Ø BIN ØØØØØØØØ
9Ø3Ø BIN ØØ111ØØØ
9Ø4Ø BIN ØØØ1ØØØØ
9Ø5Ø BIN ØØ111ØØØ
9Ø6Ø BIN Ø11111ØØ
9Ø7Ø BIN Ø1111ØØØ
9Ø8Ø BIN ØØ11ØØØØ

9ØØØ REM A TREE: 1 BLOCK
9Ø1Ø BIN Ø11Ø111Ø
9Ø2Ø BIN 11111111
9Ø3Ø BIN ØØ111Ø11
9Ø4Ø BIN ØØØ1111Ø
9Ø5Ø BIN ØØØØ11ØØ
9Ø6Ø BIN ØØØØ1ØØØ
9Ø7Ø BIN ØØØ111ØØ
9Ø8Ø BIN Ø111111Ø

9ØØØ REM A GIANT FIGHTER: BLOCK 1 OF 4: PRINT THIS
     AT Y,X
9Ø1Ø BIN ØØØ11111
9Ø2Ø BIN ØØØ1ØØ11
9Ø3Ø BIN ØØØØØØ11
9Ø4Ø BIN ØØØØØØ11
9Ø5Ø BIN ØØØØØØØ1
9Ø6Ø BIN ØØØØØØ11
9Ø7Ø BIN ØØØØØ111
9Ø8Ø BIN ØØØØØ111

9Ø9Ø REM A GIANT FIGHTER: BLOCK 2 OF 4: PRINT AT Y,
     X+1
91ØØ BIN ØØØØØØØØ
911Ø BIN 1ØØØØØØØ
912Ø BIN 11ØØØØØØ
913Ø BIN 1ØØØØØØØ
914Ø BIN ØØØØØ1Ø1
915Ø BIN 11ØØ1111
916Ø BIN 111ØØ1Ø1
917Ø BIN 1Ø11Ø1ØØ

918Ø REM A GIANT FIGHTER: BLOCK 3 OF 4: PRINT AT
     Y+1,X
919Ø BIN ØØØØ1111
92ØØ BIN ØØØØ1111
```

```
9210 BIN 00011111
9220 BIN 00011111
9230 BIN 00111110
9240 BIN 01111100
9250 BIN 01111100
9260 BIN 11111110

9270 REM A GIANT FIGHTER: BLOCK 4 OF 4: PRINT AT Y+1,
     X+1
9280 BIN 10011100
9290 BIN 10000000
9300 BIN 11000000
9310 BIN 11100000
9320 BIN 01100000
9330 BIN 01100000
9340 BIN 01100000
9350 BIN 01111000

9000 REM AN ORNAMENTAL CROSS: BLOCK 1 OF 2: PRINT
     AT Y,X
9010 BIN 00010000
9020 BIN 00111000
9030 BIN 01010100
9040 BIN 11111110
9050 BIN 01010100
9060 BIN 00111000
9070 BIN 00010000
9080 BIN 00010000

9090 REM AN ORNAMENTAL CROSS: BLOCK 2 OF 2: PRINT
     AT Y+1,X
9100 BIN 00010000
9110 BIN 00010000
9120 BIN 00111000
9130 BIN 00111000
9140 BIN 01111100
9150 BIN 11111110
9160 BIN 11111110

9000 REM CASTLE: BLOCK 1 OF 2: PRINT AT Y,X
9010 BIN 01010000
9020 BIN 01110000
9030 BIN 01010000
9040 BIN 01110000
9050 BIN 01110101
9060 BIN 01111110
```

```
9Ø8Ø BIN 1111111Ø

9Ø9Ø REM CASTLE: BLOCK 2 OF 2: PRINT AT Y,X+1
91ØØ BIN ØØØ1Ø1ØØ
911Ø BIN ØØØ111ØØ
912Ø BIN ØØØ1Ø1ØØ
913Ø BIN ØØØ111ØØ
914Ø BIN Ø1Ø111ØØ
915Ø BIN 111111ØØ
916Ø BIN 1111111Ø
917Ø BIN 11111111

9ØØØ REM RUINED TOWER: BLOCK 1 OF 2: PRINT AT Y,X
9Ø1Ø BIN Ø1ØØØØØØ
9Ø2Ø BIN Ø1ØØØØØØ
9Ø3Ø BIN Ø11ØØØØØ
9Ø4Ø BIN Ø11ØØØ1Ø
9Ø5Ø BIN Ø11ØØ11Ø
9Ø6Ø BIN Ø11Ø111Ø
9Ø7Ø BIN Ø1111Ø1Ø
9Ø8Ø BIN Ø1111Ø1Ø

9Ø9Ø REM RUINED TOWER: BLOCK 2 OF 2: PRINT AT Y+1,X
91ØØ BIN Ø1Ø1111Ø
911Ø BIN Ø1Ø1111Ø
912Ø BIN Ø111111Ø
913Ø BIN Ø111111Ø
914Ø BIN 1111111Ø
915Ø BIN 1111111Ø
916Ø BIN 11111111
917Ø BIN 11111111

9ØØØ REM UNICORN: BLOCK 1 OF 3: PRINT AT Y,X
9Ø1Ø BIN ØØØØØØØØ
9Ø2Ø BIN 1ØØØØØØØ
9Ø3Ø BIN Ø1ØØØØØØ
9Ø4Ø BIN ØØ1ØØØØØ
9Ø5Ø BIN ØØ111ØØØ
9Ø6Ø BIN Ø11111ØØ
9Ø7Ø BIN 1111111Ø
9Ø8Ø BIN ØØØØ111Ø

9Ø9Ø REM UNICORN: BLOCK 2 OF 3: PRINT AT Y+1,X
91ØØ BIN ØØØØ1111
911Ø BIN ØØØØ1111
912Ø BIN ØØØ11111
913Ø BIN ØØ1Ø1ØØØ
```

```
9140 BIN 00101000
9150 BIN 00101000
9160 BIN 00101000
9170 BIN 00001000
9180 REM UNICORN: BLOCK 3 OF 3: PRINT AT Y+1, X+1
9190 BIN 11110000
9200 BIN 11111000
9210 BIN 11111100
9220 BIN 01010110
9230 BIN 10010111
9240 BIN 10010001
9250 BIN 10010000
9260 BIN 00010000
```

10.2 Minigames

Arcade games have their attractions and there is no doubt that one of the major reasons for the growth of the home computer industry has been the popularity of fast-action screen games. It is not possible in BASIC to write very fast arcade-type games—the language is just too slow, because it is an interpreted language. If you have a compiler or understand machine code, you will not have this problem, but the majority of us have to make do with the slowness of BASIC.

However, speed is not everything. The growing popularity of adventures shows this. It might be worth while putting some arcade-type action into our adventure, even if it is not of the highest quality, to add spice of a different kind to what is primarily a game of intellect.

If we are going to include a mini graphic game of some kind we must make sure that it connects with the main game, and that it does not use a disproportionate amount of space. The arcade element must be a bonus over and above the expected element in the game that it represents. One solution would be to make all the monster combats real-time graphics, though I have yet to see a reasonable attempt at this which does not waste all the other resources available to an adventure programmer.

Much easier, though using essentially the same principles, is a treasure-gathering routine. A graphical version of monster combat must hold and manipulate a large number of variables, and attempt to simulate real combat with limited resources. In BASIC the level of animation that this requires really is not possible. However, if we simplify our graphics by using block graphics or user-defined graphics, and reduce our aims, we can nevertheless produce quite satisfactory arcade interludes which are testing in a limited way, pleasant to see, and function in a meaningful way in the context of the overall game.

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

It is easy to see why this slows a game down. If we have 40 invaders, 1 gun, 5 missiles, and 20 bombs on the screen at the same time, then each 'turn' in the game involves deleting 66 objects, calculating their next positions, checking for contacts at those positions, calculating the effects of any contacts and displaying the results, and printing 66 objects at their new positions. For all this to happen and still look natural it must literally take place in the blink of an eye. For BASIC it is more like 'forty winks'.

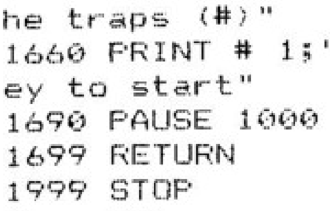
We will now look at a simple version in which only one object moves (a little man representing the player) and only one position needs testing for collisions each turn—the position the man will next move to. The idea behind the routine is simple. Instead of the player merely being told 'You have found forty-three silver pieces', the treasure routine involves him actually collecting the money himself. He has a fixed number of moves, so can only collect a limited amount, and he does so by racing round the room picking up jewels. This is the game Treasure Trove, listed as Fig. 10.5.

Treasure Trove

```
  1 CLEAR
  2 CLS
  3 LET y=10: LET x=10
  4 LET money=0
  5 INK 7
 10 LET x=0: LET y=0
 15 BORDER 0
 20 PAPER 0
 30 GO SUB 1500
 35 CLS
 40 DEF FN r(x)=INT (RND*x)
 45 FOR j=1 TO 2
 46 READ a$
 50 FOR i=0 TO 7
 55 READ a
 60 POKE USR a$+i,a
 70 NEXT i
 80 NEXT j
```

```
  81 FOR i=1 TO 20
  82 LET x=FN r(21)+1
  83 LET y=FN r(31)+1
  86 PRINT AT x,y; INK 7;"#"
  88 NEXT i
 100 FOR i=1 TO 12
 110 LET x=FN r(21)+1
 120 LET y=FN r(31)+1
 130 PRINT AT x,y; INK 2;"●"
 140 NEXT i
 150 FOR i=1 TO 5
 160 LET x=FN r(21)+1
 170 LET y=FN r(31)+1
 180 PRINT AT x,y; INK 5;"●"
 190 NEXT i
 200 FOR i=1 TO 30
 210 LET x=FN r(21)+1
 220 LET y=FN r(31)+1
 230 PRINT AT x,y; INK 4;"●"
 240 NEXT i
 490 DATA "C"
 500 DATA BIN 00011000
 510 DATA BIN 01111110
 520 DATA BIN 01111110
 530 DATA BIN 11111111
 540 DATA BIN 11111111
 550 DATA BIN 01111110
 560 DATA BIN 01111110
 570 DATA BIN 00011000
 580 DATA "p"
 590 DATA BIN 00111000
 600 DATA BIN 00111000
 610 DATA BIN 00111000
 620 DATA BIN 00010000
 630 DATA BIN 01111100
 640 DATA BIN 00010000
 650 DATA BIN 00101000
 660 DATA BIN 0100100
 890 LET y=10: LET x=10
 895 FOR i=1 TO 100
 900 LET a$=INKEY$: IF a$="" THE
N  GO TO 900
 905 LET p=x: LET q=y
 910 IF a$="5" THEN  LET x=x-1
 920 IF a$="6" THEN  LET y=y+1
 930 IF a$="7" THEN  LET y=y-1
 940 IF a$="8" THEN  LET x=x+1
 950 IF y>21  THEN  LET y=21
 960 IF x>31 THEN  LET x=31
 970 IF x<0 THEN  LET x=0
 980 IF y<1 THEN  LET y=1
 985 IF SCREEN$ (y,x)="#" THEN
```

```
LET x=p: LET y=q
 990 PRINT AT q,p;" "
 995 PRINT AT 0,1; INVERSE 1;"Mo
ney ";money
 996 PRINT AT 0,19; FLASH 1;"Tim
e Left ";100-i
1010 IF ATTR (y,x)=5 THEN  LET m
oney=money+10: FOR b=1 TO 5: BEE
P .001,b*5: NEXT b
1020 IF ATTR (y,x)=2 THEN  LET m
oney=money+5: FOR b=5 TO 9: BEEP
 .01,b*3: NEXT b
1025 IF ATTR (y,x)=4 THEN  LET m
oney=money+1: FOR b=5 TO 10: BEE
P .01,b*3: NEXT b
1026 PRINT AT y,x; INK 3;"🯅"
1027 BEEP .01,1
1030 NEXT i
1100 FOR J=1 TO 100
1105 INK FN R(7)
1106 PAPER FN R(7)
1107 PRINT AT 10,9;"OUT OF TIME"
1110 NEXT J
1120 PAPER 0: INK 7
1199 STOP
1500 LET A$="TREASURE TROVE"
1510 FOR I=1 TO LEN (A$)
1520 LET R=FN R(7)
1525 LET S=FN R(7): IF S=R THEN
 GO TO 1525
1530 PRINT AT I+2,I+6; INK R; PA
PER S;A$(I)
1550 NEXT I
1560 PAUSE 100
1570 CLS
1600 PRINT AT 2,2;" You must gat
her the jewels","     to build u
p your wealth"
1610 PRINT AT 5,4;"Blue jewels a
re worth","      10 silver piece
s"
1620 PRINT AT 8,4;"Red jewels ar
e worth","       5 silver pieces
"
1630 PRINT AT 11,4;"Green jewels
 are worth","     1 silver piece
 each"
1640 PRINT AT 15,4;"You control
movement","    using the cursor
keys."
1650 PRINT AT 18,2;"It is not po
ssible to pass","      through t
```



```
he traps (#)"
1660 PRINT # 1;"Press a cursor k
ey to start"
1690 PAUSE 1000
1699 RETURN
1999 STOP
```

Figure 10.5

The jewels are worth three different values. This introduces a limited element of strategy—given a limited amount of time, what is the best route to pick up as many as possible of the high-value jewels (cyan) while passing through as many as possible of the large clusters of jewels with smaller denominations? The maximum score would be 140 silver pieces, and an average score around 60. These can easily be varied either by changing the values of the jewels (lines 1Ø1Ø, 1Ø2Ø, and 1Ø25) or by changing the quantities of jewels (lines 1ØØ,15Ø, and 2ØØ).

As a slight complication there are a number of traps, which are simply obstacles that the player cannot travel over. If we wanted to increase the difficulty of the game we could add more traps (line 81) or we could impose a penalty if a trap was hit (either the removal of some treasure or a reduction in the remaining number of turns).

The program works quite simply and is given here as a complete game so it can be typed in and run without incorporation in another game. Firstly, titles and instructions are printed in subroutine 15ØØ. Points to notice are lines 151Ø to 155Ø which print the title as a diagonal series of blocks of random colours. As the program stands the colours are PAPER colours, with the letters in black. However, if the zero in line 153Ø is changed to R the ink on each block will be a random INK colour which is different from the PAPER colour (line 1525 checks that they are different).

Line 156Ø makes sure that the player has time to read the title and lines 16ØØ to 166Ø print the instructions. Notice how the instructions have been placed on the screen to maximize the use of space and make them easy to read. Notice also the command in line 166Ø which PRINTs on the bottom two lines of the screen—normally impossible. This is not generally a good idea, but permissible in this case because it is cleared off the screen before anything else is done.

The subroutine returns to the main routine and then lines 45 to 8Ø READ the DATA for two user-defined graphics, the jewel which is held as graphic 'c' and the man, held as graphic 'p'. Note that not only are the BIN statements held in DATA statements but so are the addresses for each character (lines 49Ø to 66Ø).

Lines 1ØØ to 24Ø now print the display by randomly choosing a PRINT AT position and putting firstly traps and then jewels of different colours on the screen. Lines 895 to 111Ø are then the main loop, run 1ØØ times, which is the 'time limit' of the player. Line 91Ø looks for input from the keyboard, while line 92Ø sets p and q to the man's current position. Line 91Ø sets the new values for the position according to the key pressed (note that, although the player believes he is pressing 'cursor keys', it is actually the numbers 5 to 8 which are tested for). Lines 95Ø to 98Ø ensure that the man does not attempt to go off the screen by checking that x and y, the coordinates the man will be printed at, are not outside the screen boundaries.

Line 985 uses SCREEN$ to check the nature of the position the man wants to move to, in the normal way. If the position is a trap x and y are reset to their old values, held at p and q. The old position is then overprinted with a space by line 99Ø.

Lines 995 and 996 print the current number of turns left and silver pieces acquired. Note that 'time left' flashes in an attempt to increase the tension of the game. Lines 1Ø1Ø to 1Ø25 test for jewels at the current position by the means discussed above for checking the ATTRibutes of the new position. This is because the jewels are user-defined and hence invisible to SCREEN$. This is also why the PAPER colour is black. All kinds of ATTRibute values could be tested for, but as black is zero and so adds nothing to this value, the numbers we are testing for can be small and simple—just the attribute numbers of the INK colours for red, green, and cyan.

If a particular attribute is found then the variable 'money' is increased by the correct amount and a rapid BEEP scale is sounded to indicate the fact. Then the man is PRINTed AT his new position and a rapid BEEP signals that he has moved and hence a turn has elapsed. It has to be a rapid BEEP because the Spectrum sound slows the program up substantially, as it interrupts the running of the program, unlike other systems which process sound without interfering with the running of the main routine.

The loop continues until 100 turns have elapsed, when lines 11ØØ to 111Ø FLASH in random colours the fact that the game is over; line 1199 then STOPs execution.

11 MICROS IN GAMES WITHOUT MICROS

11.1 GAPs

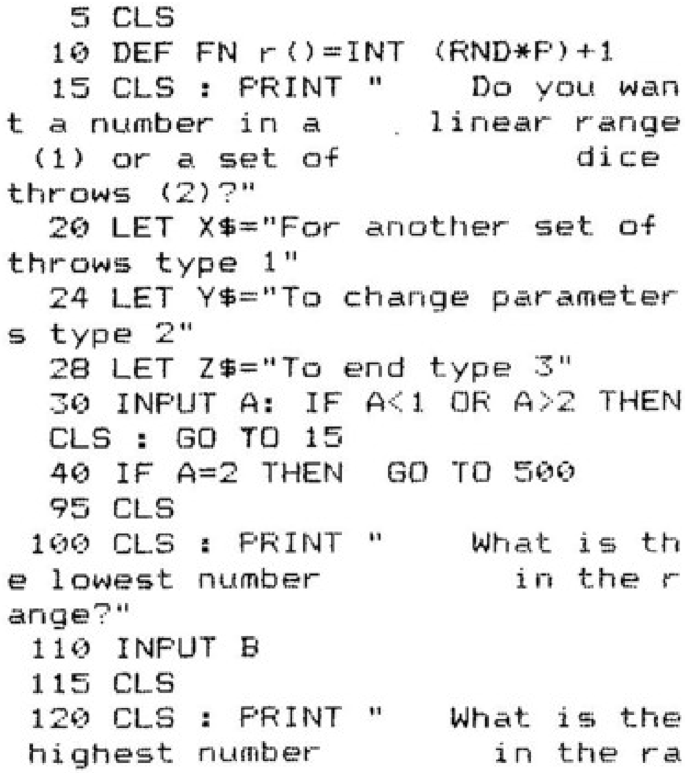
To use a micro in an adventure game it is not necessary to play the game on a micro. As mentioned in Chapter 1, many adventure games are played as board games or tabletop games. The most well known of these are Dungeons and Dragons (a fantasy game) and Traveller (a science fiction game). There are many advantages of playing board or tabletop versions of adventure games: the game can be much more complex and fluid than its micro equivalent (Advanced Dungeons and Dragons has over 1000 pages of 'rules'); miniatures and counters can make the game physically and visually more attractive to play; the game is more of a social activity, with the emphasis on role-playing; and a greater degree of realism and detail can be achieved. It is possible that micro adventure games will eventually replace these games, but this will not be for many years until memory, graphics, interactive control, and 'intelligence' have been greatly enhanced.

However, micros can add a marvellous new dimension to most tabletop and board games. Many games players use their micros to run games assistance programs (usually abbreviated to GAPs). These are programs which handle some of the mechanical, tedious, or complex aspects of the game, enabling the players to enjoy play without worrying about its mechanisms. Examples are GAPs which work out the complex details of combat between two armies in a strategic game or which generate star maps for games such as Traveller.

If you like micro adventure games, you will almost certainly like the original tabletop versions, so this section outlines two GAPs which you might like to incorporate in such games. Naturally any program which can be used in a tabletop game can also be used in an adventure played totally on a micro, so you could always use these ideas as the basis of your own adventure programs. The main drawback is that routines based on these ideas may use a disproportionate amount of memory, because they calculate very detailed aspects of games. Therefore, if you use them in a micro adventure, the game is going to be rather unbalanced as you will not have much room for the other routines. Unless you have a 16 bit machine or a micro like the Newbrain which can address a great deal of memory through paging, you will not have enough memory to use such routines in micro adventures. There is, however, another solution which is discussed in the next section, namely using a suite of programs as one adventure, as in games played by mail.

To be worth the programming time a GAP must either do something rather better than it could be done manually or it must save time. Usually GAPs are written to save time on aspects of games which are boring. Most frequent of these are the routines which generate characters for tabletop role-playing. Normally a role-playing game needs an hour or two beforehand while a number of dice are rolled to set up the variables which form the character or characters playing the game. As such a character may have more than 20 such values, each of which affects or depends on the others, this can take a great deal of tedious work. As the basic process is to generate a series of random numbers and set up a list of values according to how these random values interrelate, it is easy to write a flowchart for this and then code it up.

Every game has its own version of this process, so there is little use in giving a program here. However, as many of these games require a complex series of random numbers to set them up, the following general routine may be useful. It is a standard random number generator which allows selection of either linear probabilities in a chosen range or other probabilistic curves (chosen by selecting a number of dice with any number of sides other than one). Figure 11.1 gives the routine.



```
    5 CLS
   10 DEF FN r()=INT (RND*P)+1
   15 CLS : PRINT "    Do you wan
t a number in a     linear range
 (1) or a set of            dice
throws (2)?"
   20 LET X$="For another set of
throws type 1"
   24 LET Y$="To change parameter
s type 2"
   28 LET Z$="To end type 3"
   30 INPUT A: IF A<1 OR A>2 THEN
   CLS : GO TO 15
   40 IF A=2 THEN  GO TO 500
   95 CLS
  100 CLS : PRINT "    What is th
e lowest number         in the r
ange?"
  110 INPUT B
  115 CLS
  120 CLS : PRINT "   What is the
 highest number         in the ra
```

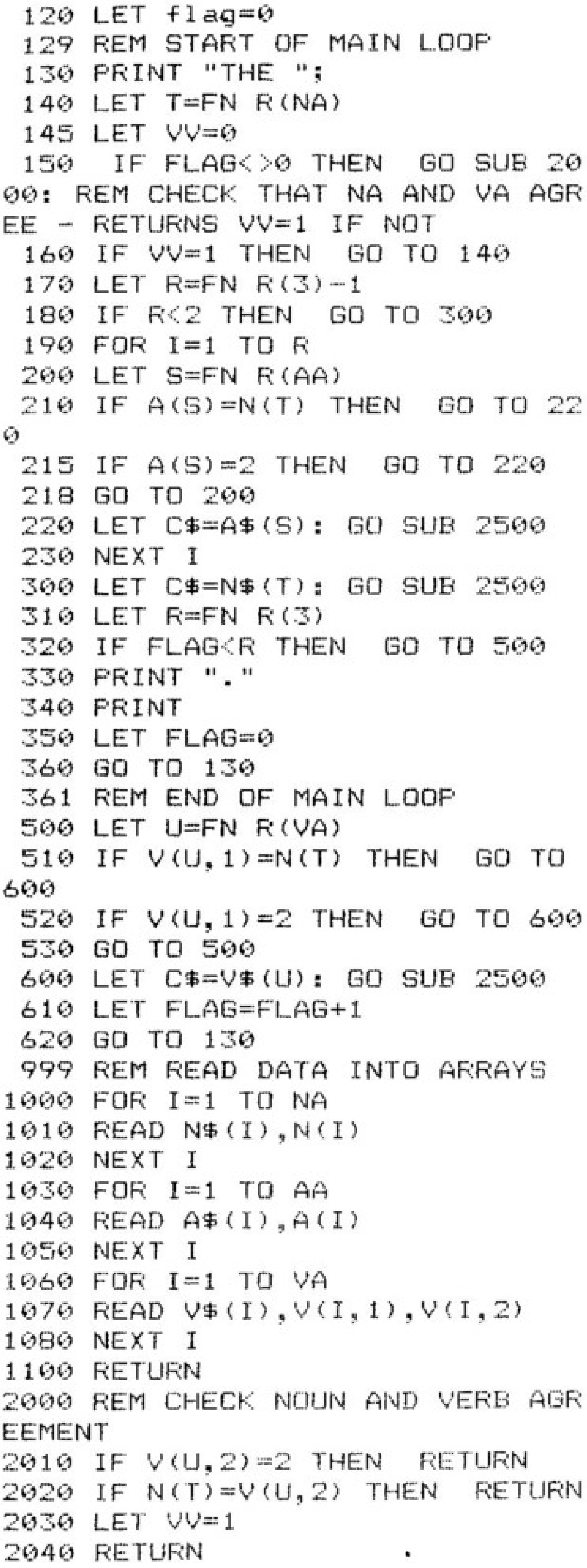
```
nge?"
 130  INPUT C
 135 CLS
 136 PRINT AT 0,9;"LINEAR RANGE"
 137 PRINT AT 2,0;"High
          Low"
 138 PRINT AT 3,2;C: PRINT AT 3,
24;B
 140 PRINT AT 10,6;"The number i
s ";INT (RND*(C+1-B))+B
 150 PRINT AT 16,0;"For another
number type 1"
 160 PRINT Y$
 170 PRINT Z$
 180 INPUT A
 185 IF A<1 OR A>3 THEN  CLS : G
O TO 150
 186 CLS
 190 IF A=1 THEN  GO TO 135
 200 IF A=2  THEN  GO TO 10
 210 IF A=3 THEN  GO TO 999
 500 CLS : PRINT "How many sides
 to your dice?"
 510 PRINT "Please enter a numbe
r between 2 and 100"
 520 INPUT P
 530 LET a=INT (A): IF A<2 OR A>
100 THEN  CLS : GO TO 500
 540 CLS : PRINT "How many throw
s of your ";P;" sided dice?"
 550 INPUT B
 560 LET B=INT (B): IF B<1 OR B>
1000 THEN  PRINT "Only 1 to 1000
 throws allowed ": GO TO 540
 570 CLS : PRINT "Do you want th
e throws ","printed separately (
1)","or added together  (2)?"
 580 INPUT C
 590 LET C=INT (C): IF C<1 OR C>
2 THEN  CLS : GO TO 570
 600 IF C=1 THEN  GO TO 700
 605 LET D=0: LET E=0
 610 FOR I=1 TO B
 620 LET E=FN R()
 630 LET D=D+E
 640 NEXT I
 645 CLS
 650 PRINT "Your number is ";D
 660 PRINT X$
 670 PRINT Y$
 680 PRINT Z$
 690 INPUT A
 692 IF A<1 OR A>3 THEN  CLS : G
O TO 660
 694 IF A=1 THEN  GO TO 605
 695 IF A=2 THEN  GO TO 10
 696 IF A=3 THEN  GO TO 999
 700 LET D=0
 702 CLS
 705 FOR I=1 TO B
 710 LET D=FN R(): PRINT "You th
row ";D
 720 NEXT I
 750 PRINT X$
 760 PRINT Y$
 770 PRINT Z$
 790 INPUT A
 792 IF A<1 OR A>3 THEN  CLS : G
O TO 750
 794 IF A=1 THEN  GO TO 700
 795 IF A=2 THEN  GO TO 10
 796 IF A=3 THEN  GO TO 999
 999 STOP
```

Figure 11.1

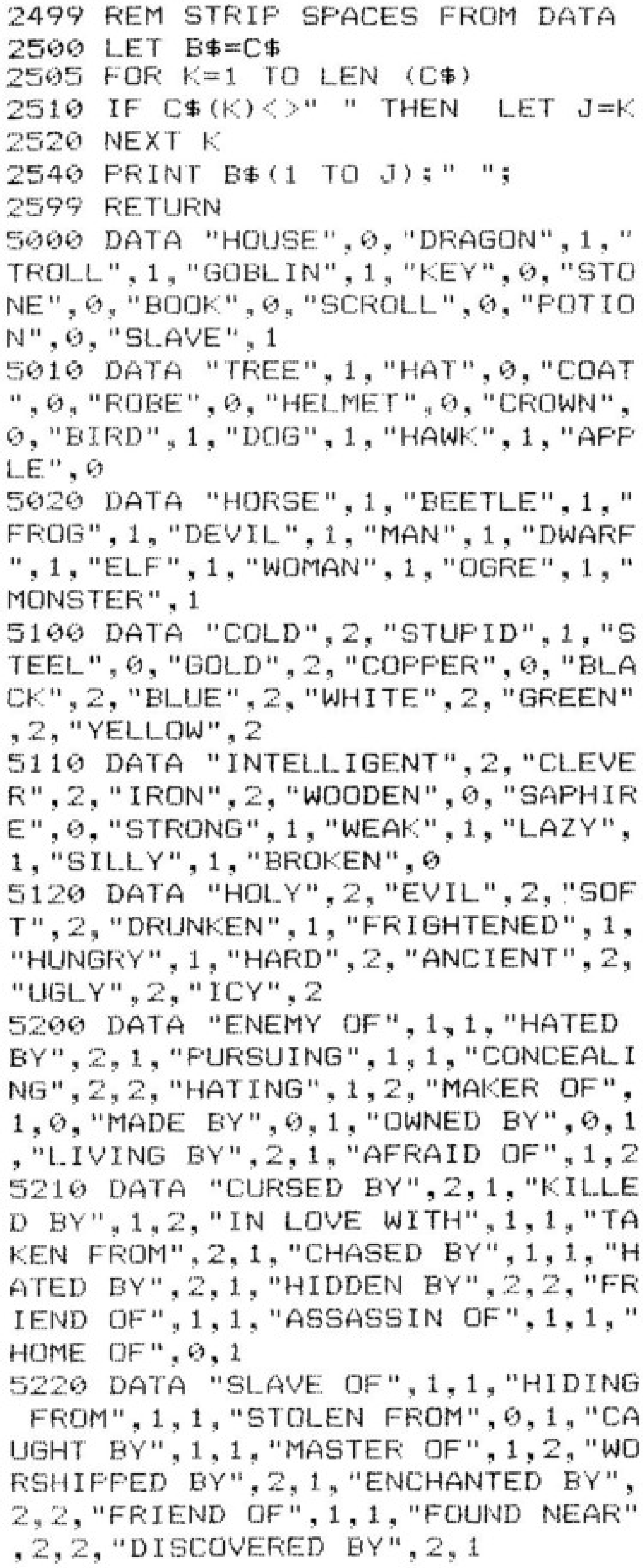
Many other aspects of FRP games are essentially random, though with certain fixed values, so this routine may be useful for many aspects of the game, such as determining the numbers of rooms in a dungeon, the number and types of monsters in a particular location, the spells available to a particular magician, the spaceships in a fleet, the troops available to an army, and so forth. No special programming tricks are needed for these—simply careful structuring. However, the program for any of these will be different for each game system. In general it is better to generate all the random numbers needed first and hold them in an array or file of some kind.

There are some general routines which can be of use to many role-playing games. One example is the name-generating routine given in Chapter 7. Another could be of use not only in game-playing but also for stimulating ideas of all kinds; this is an idea generator which creates new ideas for use as the basis of games or scenarios, such as the one given in Fig. 11.2.

```
   50 REM FANTASY IDEAS PROGRAM
   89 REM INITIALISATION
   90 LET na=30: LET aa=30: LET v
a=30
   95 LET VV=0
  100 DIM n$(na,11): DIM N(NA): D
IM a$(aa,11): DIM A(AA): DIM v$(
va,14): DIM V(VA,2)
  110 GO SUB 1000
  115 DEF FN r(x)=INT (RND*x)+1
```



```
 120 LET flag=0
 129 REM START OF MAIN LOOP
 130 PRINT "THE ";
 140 LET T=FN R(NA)
 145 LET VV=0
 150  IF FLAG<>0 THEN  GO SUB 20
00: REM CHECK THAT NA AND VA AGR
EE - RETURNS VV=1 IF NOT
 160 IF VV=1 THEN  GO TO 140
 170 LET R=FN R(3)-1
 180 IF R<2 THEN  GO TO 300
 190 FOR I=1 TO R
 200 LET S=FN R(AA)
 210 IF A(S)=N(T) THEN  GO TO 22
0
 215 IF A(S)=2 THEN  GO TO 220
 218 GO TO 200
 220 LET C$=A$(S): GO SUB 2500
 230 NEXT I
 300 LET C$=N$(T): GO SUB 2500
 310 LET R=FN R(3)
 320 IF FLAG<R THEN  GO TO 500
 330 PRINT "."
 340 PRINT
 350 LET FLAG=0
 360 GO TO 130
 361 REM END OF MAIN LOOP
 500 LET U=FN R(VA)
 510 IF V(U,1)=N(T) THEN  GO TO
600
 520 IF V(U,1)=2 THEN  GO TO 600
 530 GO TO 500
 600 LET C$=V$(U): GO SUB 2500
 610 LET FLAG=FLAG+1
 620 GO TO 130
 999 REM READ DATA INTO ARRAYS
1000 FOR I=1 TO NA
1010 READ N$(I),N(I)
1020 NEXT I
1030 FOR I=1 TO AA
1040 READ A$(I),A(I)
1050 NEXT I
1060 FOR I=1 TO VA
1070 READ V$(I),V(I,1),V(I,2)
1080 NEXT I
1100 RETURN
2000 REM CHECK NOUN AND VERB AGR
EEMENT
2010 IF V(U,2)=2 THEN  RETURN
2020 IF N(T)=V(U,2) THEN  RETURN
2030 LET VV=1
2040 RETURN
```



```
2499 REM STRIP SPACES FROM DATA
2500 LET B$=C$
2505 FOR K=1 TO LEN (C$)
2510 IF C$(K)<>" " THEN  LET J=K
2520 NEXT K
2540 PRINT B$(1 TO J);" ";
2599 RETURN
5000 DATA "HOUSE",0,"DRAGON",1,"
TROLL",1,"GOBLIN",1,"KEY",0,"STO
NE",0,"BOOK",0,"SCROLL",0,"POTIO
N",0,"SLAVE",1
5010 DATA "TREE",1,"HAT",0,"COAT
",0,"ROBE",0,"HELMET",0,"CROWN",
0,"BIRD",1,"DOG",1,"HAWK",1,"APP
LE",0
5020 DATA "HORSE",1,"BEETLE",1,"
FROG",1,"DEVIL",1,"MAN",1,"DWARF
",1,"ELF",1,"WOMAN",1,"OGRE",1,"
MONSTER",1
5100 DATA "COLD",2,"STUPID",1,"S
TEEL",0,"GOLD",2,"COPPER",0,"BLA
CK",2,"BLUE",2,"WHITE",2,"GREEN"
,2,"YELLOW",2
5110 DATA "INTELLIGENT",2,"CLEVE
R",2,"IRON",2,"WOODEN",0,"SAPHIR
E",0,"STRONG",1,"WEAK",1,"LAZY",
1,"SILLY",1,"BROKEN",0
5120 DATA "HOLY",2,"EVIL",2,"SOF
T",2,"DRUNKEN",1,"FRIGHTENED",1,
"HUNGRY",1,"HARD",2,"ANCIENT",2,
"UGLY",2,"ICY",2
5200 DATA "ENEMY OF",1,1,"HATED
BY",2,1,"PURSUING",1,1,"CONCEALI
NG",2,2,"HATING",1,2,"MAKER OF",
1,0,"MADE BY",0,1,"OWNED BY",0,1
,"LIVING BY",2,1,"AFRAID OF",1,2
5210 DATA "CURSED BY",2,1,"KILLE
D BY",1,2,"IN LOVE WITH",1,1,"TA
KEN FROM",2,1,"CHASED BY",1,1,"H
ATED BY",2,1,"HIDDEN BY",2,2,"FR
IEND OF",1,1,"ASSASSIN OF",1,1,"
HOME OF",0,1
5220 DATA "SLAVE OF",1,1,"HIDING
 FROM",1,1,"STOLEN FROM",0,1,"CA
UGHT BY",1,1,"MASTER OF",1,2,"WO
RSHIPPED BY",2,1,"ENCHANTED BY",
2,2,"FRIEND OF",1,1,"FOUND NEAR"
,2,2,"DISCOVERED BY",2,1
```

Figure 11.2

Such a program has some of the elements of artificial intelligence, i.e., it simulates creativity, producing new ideas just as a human being might (e.g., in generating the mind maps suggested in Chapter 3). However, the program does not evaluate any of the ideas. To do so it would require a huge database of knowledge on what was practical or desirable in the real world. Creativity in human terms is a question of finding new links between existing items and that is what this program does. The user has to evaluate them, deciding which might lead to useful games or scenarios and which are trivial or useless.

The value of a program like this depends on its database. The version included here attempts to create new fantasy ideas, but different databases could be used for other types of plot or scenario, such as Science Fiction or a Western. Other databases, rather more remote from the type usually found in an adventure game, could include different kinds of human social or political relationships, or the kinds of relations typical of television soap opera, or any situation where two elements are linked in some way.

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

If designing your own GAP it is best to begin with one small aspect of the game which can be isolated from the rest and which you understand quite well. You will realize by now that the reason for this is that such programs have many more features than one expects and therefore need very careful planning. The temptation is always to begin with something very complex, such as a game's combat system. While it is true that these systems are generally tailor-made for micro adaptation, it is also true that you will find more problems than you expect, so you are likely to end up abandoning the project half-way through if you bite off more than you can chew.

It is much better therefore to begin with something elementary so you can learn how to do it by practice without becoming frustrated or bored, e.g., a dice-throwing program or a program which generates the armies in a war game. These are relatively self-contained yet could be very useful in a game. Once you have successfully created a simple GAP like this you can begin to work on fuller programs. These must be worked on in sections. Fortunately most games can be described as algorithms or flowcharts, so the nature of the problem is usually not too difficult to specify. However, make sure that you have drawn up all the stages of the description before you begin coding or you will find that you have not allowed for all the intricacies of board games. Even Monopoly is quite complex.

11.2 Play by mail adventure

Another way to play adventure is by post. Perhaps this seems strange, but many games' players now take part in postal campaigns of one kind or another, such as war games, science fiction games, and fantasy adventure games, as well as many other conventional games. Chess, Diplomacy, and even Ludo are played by post.

The advantages and disadvantages of postal play are obvious. On the one hand it is a good way to find keen players; you do not have to arrange meetings or calculate elaborate timetables so that all players can be in the same place at the same time; each player has plenty of time to calculate his or her move; a large number of players can be playing one game (impossible in the normal living room); aspects of campaigns/adventures such as surprise, hidden movement, spying, intelligence, morale, discovery of magic, finding powerful secrets, meeting 'new' characters or races, or travelling an unknown universe can be handled in a realistic rather than an artificial way. On the other hand play by mail (or PBM as it is commonly called) usually means strict deadlines for sending in moves, little direct contact between players, a great deal of administration for the umpire, and a degree of inflexibility. Games may also take months or years rather than hours or days to complete — some people would call this an advantage and others a disadvantage.

You can play such games without a computer. In fact most PBM games do not involve a microcomputer. Computer-moderated PBM is a very recent phenomenon as umpires and games masters have realized that the computer can take on much of the tedious work in such a game and can do much to make such games more interesting and varied. Generally only the umpire or coordinator in a PBM game uses the micro and players communicate with the umpire by post and telephone in the usual way. Each turn every player receives feedback from the umpire on the results of previous moves, then works out the next move, writes it down, and sends it in to the umpire before the deadline. The umpire feeds it into his micro as data for a program or suite of programs, the micro calculates the results, and the umpire then sends those results back to the players. However, it would be possible for such games to work entirely on micros, with moves and results being communicated by exchange of cassette or disc, or even transmitted directly down telephone lines by using acoustic couplers or modems.

We will concentrate on the simpler game with one micro, one umpire, and a number of players. There are two key differences between this set-up and the usual micro adventure game. In the kinds of game described in Chapters 1 to 10 usually only one player plays the game at a time. In PBM games, the program must cope with simultaneous input from a number of players. This causes several programming difficulties.

Secondly, in an interactive micro game speed of response is important. Any delay of more than a second or two while a player's instructions are being processed leads to frustration and boredom with the game as a whole. In PBM the speed of response is not important. Naturally the micro has to calculate all the results well before the deadline to enable the umpire to write back to the players in time, but the running of the program could take hours or days between input and output without causing any problems. This allows several programming techniques not normally useful to the adventure game programmer. These include:

1. Processing of natural language input
2. Production of varied natural language output
3. A high degree of complexity in rule systems (e.g., combat, magic, etc.)
4. Large databases stored on cassette or disc (e.g., adventures with thousands rather than hundreds of locations)
5. Interaction of different kinds of games (e.g., adventures, graphic games, and strategic war games can be combined)
6. Programs split into several stages or sections to avoid microcomputer memory limitations

In other words, PBM adventure gives the maximum scope in design, content, and implementation of an adventure. The programmer is no longer limited to the micro's RAM, nor to the specified forms of input and output interactive games demand, nor to programming with a view to a constantly attractive and updated VDU.

There are some other preliminary considerations, however. In order to run a PBM game efficiently it is best to have a printer (and preferably one with some form of graphics output) and disc drives, or some other form of mass storage. In principle a PBM game can operate without either of these, especially a simple game. However, it would mean that you would have to copy output from VDU to paper by hand, which is a tedious task and rather wasteful of a micro's resources. Cassette storage can be used instead of disc, but is much slower and operates by serial rather than random access. What this means is that a disc allows almost instant use of any data or program stored anywhere on the disc, whereas a cassette has to be searched one byte at a time, from the beginning to the correct point. So if you were in the habit of storing the data for your games on C60s, it might take half an hour to find the correct data, whereas 5 seconds is considered slow for discs.

The Spectrum will of course have the Microdrive, which ought to be generally available by the time this book is published. Though almost certainly a tape-based system the Microdrive will perform to all intents and purposes very like a normal disc drive. If you can afford one disc drive or Microdrive, then it is better to try and afford two. Two drives allow easy copying of discs and the use of program discs and data discs in separate drives. With one drive you would either have to keep all the data for a particular program on the same disc or to swop discs around while the program is running which, though usually possible, is not a good habit.

However, the majority of micro owners cannot yet afford disc drives, so we will assume in the rest of this chapter that the configuration you have is a micro, a cassette recorder, and a printer. For the Spectrum this means also having an RS232 or Centronics interface to drive a sophisticated dot matrix or daisy wheel printer, as the Sinclair printer is not really adequate for attractive output to be read by players, excellent though it is for general use.

The design of your PBM game can be identical to a normal micro adventure, except that output must go to a printer as well as, or instead of, your television screen. The actual programming will depend on the BASIC syntax of the interface you use. If you are using a Sinclair printer, or the interface you have allows the Sinclair commands, it is simply a matter of substituting LPRINT for PRINT. If the game has graphics output or a complex or interesting screen display then you will use COPY, which prints on the printer a copy of what is displayed on the screen. However, this facility might not work on a number of printers, so you may have to redesign the output as a series of LPRINT commands. Although much more tedious, this does give you more control over the output. Consequently, if the printout for each player is essentially the same with a few variations, then the LPRINT command allows these changes to be made on the printed output while screen display can remain the same.

Though it is possible to have a PBM which is exactly like an ordinary micro adventure, it will not make a very satisfactory PBM game. Very few people would like to wait two weeks for a printed copy of a display identical to the one they could get by playing the game on their own micro, especially if it hardly changes from turn to turn. Certainly it would not be much fun to keep trying different commands, such as HIT DOOR, SMASH DOOR, BREAK DOOR, etc., every fortnight only to have a 'nothing happens' result every time!

So the design of a PBM adventure must build in at least two differences from the standard adventure. Firstly, the player must be allowed to input very complex instructions, rather than simple one-word commands, and correspondingly output must be complex and must change substantially each turn. It does not matter if some of the output is unimportant in terms of playing the game, such as elaborate descriptions of locations; what does matter is that it should demand a lot of thought by the player. Part of the fun of a PBM adventure involves extracting from such complex descriptions exactly those bits of information which are important and need to be responded to.

For example, in a normal micro adventure output in a specific location might be:

> You see a room full of cobwebs. There is a great spider on the ceiling.

The description cannot be much longer than this because of limitations on memory. Because such limitations can be removed by a PBM games master, the description could be as elaborate as the following:

> The room is very dusty. In one corner lies an upturned bucket. The ceiling and one wall are covered in cobwebs, and a number of small spiders are scurrying through the strands. There is a pool of greenish liquid in the middle of the room, on which a fragment of parchment is floating. The room smells faintly acid, especially near a large clump of web. Faintly in the distance you hear a sound like pebbles falling off a cliff.

Faced with such a description the player does not know which of these items is important and which is not. He must therefore write orders which take as many as possible of the circumstances into account. We know that there is a huge spider in the main clump of web and that the distant sound is actually a band of orcs scrambling down the cliff because they have heard you coming. However, the player may decide to ignore the web and the sound and instead investigate the pool, the smell, the small spiders, the bucket, the dust, or the parchment. If he or she does, then the spider will leap and the orcs will eventually attack from behind.

Two ways to build such complex output into your game are to develop large descriptive databases or to have 'random description' routines. The first method involves no more than we have already done when we designed the puzzle section of Camelot. For each location we wrote a description and stored it as a routine in RAM to be called when the player came to the appropriate location. The only difference we need implement is to store the description on magnetiic media—in our case, cassette. The Spectrum gives us two ways of doing this. One is to use 'MERGE', the other to use 'SAVE string DATA array name()'.

In the first case we write our program to LPRINT a description using data saved as a separate program. The description could be held as a series of LPRINT or PRINT statements, or as DATA statements which are to be READ and then manipulated, or as strings or string arrays to be manipulated. To take the simplest example, we would have a program which accepts the player's instructions and then has to print a description of the next location. Suppose the program uses a routine starting at line 7ØØØ to print this description. Then the program needs to call the following routine before the routine at 7ØØØ is called:

```
5999 REM ROUTINE TO FIND AND LOAD A DESCRIPTION
6ØØØ REM LOOK FOR RIGHT ROUTINE
6Ø1Ø PRINT"Please place location tape in recorder, and press the
     PLAY key"
6Ø2Ø IF LOCATION = 1 THEN MERGE "CASTLE"
6Ø3Ø IF LOCATION = 2 THEN MERGE "ROOM"
6Ø4Ø IF LOCATION = 3 THEN MERGE "SWAMP"
.............etc.............................................................................
61ØØ GOSUB 7ØØØ
```

The routine asks you to put the correct tape in the cassette recorder; it then searches for and MERGEs the program with the appropriate name. Each of the named programs that it could be looking for will be in lines 7ØØØ to 7999, and the line numbering for all of them must be identical. This is because a program which is MERGEd will only delete lines with numbers identical to those in the MERGEd routine, so errors could result if one description had more line numbers than the others.

A typical description might be like this:

```
7ØØØ REM CASTLE
```

```
7010 LPRINT "You see a large black castle"
7020 LPRINT "A bat is fluttering around one tower"
7030 LPRINT "Over the gate flies a banner bearing the emblem of
     a red raven on a green field"
etc.
```

If such a description has no relevance to the game, being just local colour, then the MERGEd routine need be no more than a list of LPRINT statements. However, each different location will presumably have its own design features, such as the spider and orcs in the above example. Consequently, these MERGEd routines must also pass values to the main routine in the program which has called the description. The kind of information which will be needed is the same as that used in previous chapters to describe monsters, environments, and objects—in short, values for all the variables in the program which characterize that location and which the player's character could interact with. These would include combat values, lists of possible objects and codes for the possible effects, magical values of monsters and the environment, etc.

It is also possible to build subroutines into the configuration of each room, so that different possibilities present themselves on different visits. For example, the castle routine might itself have a routine which determines whether the gate is closed or open (consequently passing a value to the main program); or a routine which varies the description, perhaps depending on the searching abilities of the player; or a routine which randomly determines if the bat attacks the player.

It is also possible for a routine MERGEd in this way to call and MERGE another, either as a subroutine of itself or to replace itself. If we use this method, we must index our tapes very carefully, and ensure that the interactive request in each MERGE call, which asks the games master to insert a cassette, specifies exactly the right cassette in an unambiguous way. Every description requires a unique name, every name should be written on the cassette, each cassette should have its own name or number, and, preferably, there should be an index of the whole lot.

You may be wondering why a location routine such as 'CASTLE' above might want to merge another which wipes it out. An obvious example would be a travelling routine—either an arbitrary transporter or warp routine, or a routine which forces the player to move to a particular location, such as in retreating, for example, or being carried away by a giant eagle to its nest. For example:

```
7200 LET R = INT (RND*3)
```

```
7210 IF R = 1 THEN LPRINT "A giant eagle descends and carries
     you away": PRINT "Please insert tape N1 and press the
     PLAY key": MERGE "NEST"
.........................................................................
7200 REM NEST ROUTINE
7210 LPRINT "You are in a huge, smelly nest"
7220 LPRINT "There are three eggs in the nest"
.........................................................................
```

The second method of storing and loading descriptive locations is less versatile than using MERGE because the programmer is limited to loading data rather than routines. However, if we only need to store descriptions and not subroutines of any kind, then these can be held in string arrays. One major advantage of this method over the MERGEing method is that there is no need for us to worry about exact matching between line numbers. LOADing a string array that has previously been SAVEd only wipes out any other array or string variable in the program with the same name.

This method would be used by micros without MERGE or other cassette file handling routines, and would often be used if the main program contained all the values, variables, and interactive routines. It simply loads a particular array, which we can call L$, with the information that the preexisting routines use for output. In Spectrum BASIC, to SAVE an array we use 'SAVE "CASTLE" DATA L$()', and to LOAD it 'LOAD "CASTLE" DATA L$()'. From an array called L$ this saves a stream of bytes called "CASTLE". Thus we can use virtually identical commands to call our descriptive data into the program, such as:

```
6020 IF LOCATION = 2 THEN LOAD "CASTLE" DATA L$()
```

Remember to ensure that there is sufficient space to load the largest of the SAVEd arrays likely to be needed. However, we do not need to DIMension the array, nor to change the dimensions if the possible arrays replacing L$() are of different sizes. This method is therefore easier to use than complex MERGEing because there are fewer errors to build in. Thus the ease is paid for by the relative lack of versatility.

SAVEing and LOADing data and routines from cassette can be used in ordinary micro adventures, but always involve a long delay. Consequently few interactive adventures use this method, except by dividing the adventure into a series of large separate programs, usually passing data from one to the next. Without any connection between adventures in this way, they can seem to be no different from independent adventures.

The second method of creating elaborate descriptions requires more thought before coding, but less actual coding or code. Consequently it requires less memory, storage, and time. The technique involves mixing the significant descriptions which are location-specific with randomly determined insignificant description. We have seen how this can be done in Chapter 7, where both meaningful and random 'conversation' were mixed.

The problem to overcome is how to fit random descriptions to fixed descriptions in a way that makes sense and also fits with the surrounding plot and environment. It is necessary to devise a system of marking so that all locations are given a type and each type can call up typical random phrases. For example, one type of location may be 'the castle'. Each different castle will have its own unique description and associated routines that can alter the player's character. However, all castles would call routines which established random descriptions of additional detail using units like 'the towers', 'the battlements', 'the type of stone', 'the shape of windows', 'the number of doors', 'kinds of inhabitant', 'degree of dilapidation', etc.

This kind of randomization can also be made more meaningful, such as when arranging production of typical encounters. Just as in an interactive adventure certain locations have certain types of monster and treasure, so in a PBM game sets of routines can be built up which not only add circumstantial description but also add circumstantial narration. Things which are going on around and because of the player's character, even if of no structural significance, will add to the flavour of the game (pleasing those who play the adventure because it feels like being in a fantasy novel), and may give clues, hints, and apparent problems (for those who are more concerned with discovering every aspect of a game's possibilities).

All such systems have three basic parts. There are firstly the descriptive units which make up the basis of the text construction. Secondly, there are the coding systems which translate different types of location into different possible descriptions. And thirdly, there are the mini routines which support such descriptions with appropriate happenings.

To demonstrate how such a system might work let us construct a game with the following simple criteria:

1. There are three location types—underwater, on the water's surface, and on board ship.
2. There are four monster parameters—combat abilities, knowledge of secrets, type of mobility, and mode of communication.

The game will thus have a set of descriptions, coded as suitable for one, two, or all three of the location types; a set of event structures, made up of the components described in Chapters 2 and 3, similarly coded; and a set of routines whose codes match the description and event codes to put together a monster for this location whose values in all parameters fit the chosen code.

Suppose that code 1 was 'under water', code 2 was on the 'surface', and code 4 was 'on board ship'; then descriptions and events could have codes from 1 to 7 indicating which location type they were suited for. Code 3 is both under and on water, code 6 is surface and ship, and so on. All the event structures and monster parameters would be similarly coded. Thus the parameter 'mode of communication' might have the following possible values and codes:

Value	Description	Code
1	talking	6
2	gestures	7
3	slapping water with fin	2
4	singing	6
5	blowing bubbles	1
6	acrobatic movements	5

All other parameters would be coded in a similar way, with all possible values of the parameter representing possible aspects of a monster with an equivalent verbal description, and would be coded to ensure compatability between the chosen location, event, and all monster features. So if we randomly choose 'floating amidst a tangle of seaweed' as our location, this would have code 2, so the monster which was in this place might be able to communicate by any or all of the following—talking, gesture, slapping water with fin, and singing—but not by acrobatic gestures or blowing bubbles.

In this way a location description would be built up which was essentially random, not designed by the programmer, but uniquely configured and meaningful for the player because the elements were all consistent and provided a new event for him or her to respond to. Built in to the descriptive aspect would be possibilities similarly coded. For example, the game may be built on the premise that creatures on the water's surface were generally hostile (hence the variable 'enemy' is set to −6), but that creatures who can talk are generally friendly (hence a chance that the variable 'enemy' will be modified is denoted by a value between 1 and 9). The player will be able to learn these tendencies through previous interaction, so will then be able to react to this new situation with a degree of knowledge but also a degree of uncertainty.

The only constraint on the designer of a PBM game such as this is the time available to him or her for creating the original suite of programs. The more time we have, the more we can build into the game, all parameters can be coded; all parameters can themselves have parameters. The degree of possible complexity is immense and that complexity can be extremely life-like and meaningful—a good compromise between the purely random and purely determined adventures played solely on micros.

Create your own adventures

Games programs are not easy to write. Nor are they easy to invent. But in this book Noel Williams discloses some of the professional programmers' secrets and explains how to create your own computer-world of fantasy and adventure!

Three complete games

In demonstrating the techniques of designing and writing games programs, three complete games are developed, *'Camelot'*, *'Merlin's Mines'*, and *'Treasure Trove'*. You learn about planning the game, text, movement, graphics and 'chrome', that all-important touch of professionalism. . .

Software available

A cassette tape is available separately, containing the games and other major listings—including *'Tongues'*, a progam to generate alien languages! See inside for details.

McGRAW-HILL Book Company (UK) Limited
MAIDENHEAD · BERKSHIRE · ENGLAND
07 084719 3