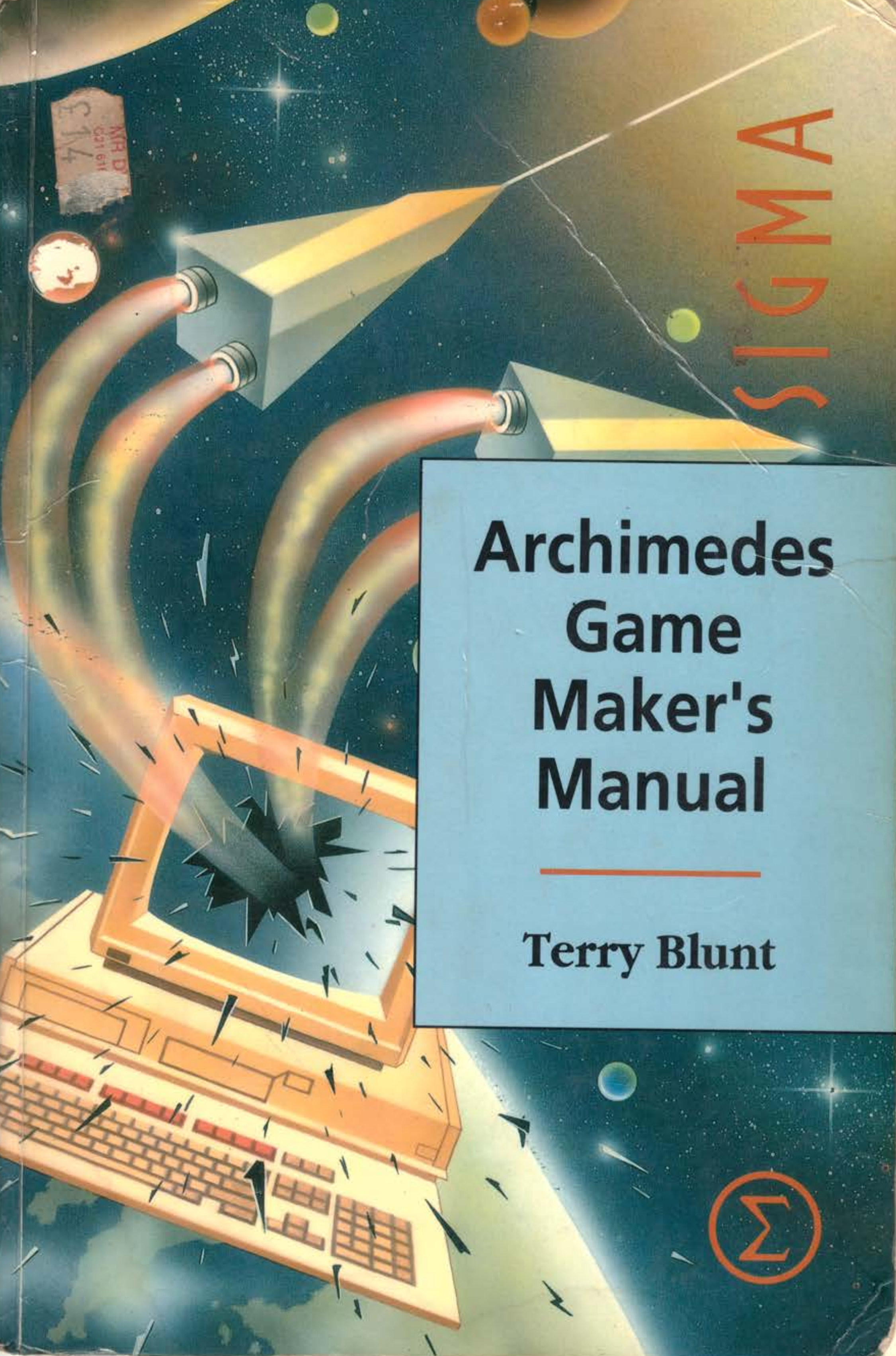

Archimedes Game Maker's Manual

Terry Blunt

Archimedes Game Maker's Manual

Terry Blunt

SIGMA PRESS – Wilmslow, United Kingdom

First published in 1992 by

Sigma Press, 1 South Oak Lane, Wilmslow, Cheshire SK9 6AR, England.

British Library Cataloguing in Publication Data

A CIP catalogue record for this book is available from the British Library.

ISBN: 1-85058-241-6

Typesetting and design by

Sigma Hi-Tech Services Ltd

Printed in Malta by
Interprint Ltd.

Distributed by

John Wiley & Sons Ltd., Baffins Lane, Chichester, West Sussex, England.

Preface

This book is for those new to the Archimedes, and probably new to games programming. Of necessity, some knowledge of programming in Basic is assumed, so with this in mind working with a copy of the BBC Basic Guide to hand is advisable. For those producing promising games, but needing to delve deeper into the Archimedes for more advanced work, serious consideration should be given to the expense of the Programmers' Reference Manual.

The Acorn Gamemaker's Manual can be separated naturally into two parts. The first is a guide to planning and design, as well as programming techniques particularly applicable to games. The second part is three groups of subsections that concentrate on each loosely defined game type. As it is impossible to define precisely a game type these days, reading over several in the same group will probably be necessary. There is a final chapter devoted to using the Basic assembler to produce useful ARM code extensions.

The ideas and methods contained here are intended as a guide rather than a set of absolute rules, so with increasing experience it will be possible to extend and improve on this work. Also, even the very best programmers will admit that there's always a different view of a familiar task, so it pays to look closely at any program listings that come to hand. The ideal solution to a problem could quite easily be buried deep inside something as remote from games as a desktop diary program.

Acknowledgements

I would like to thank Minerva Software for their interest and suggestions while this book was in its early stages, Nick Pelling for some timely help near the book's completion, and Steve Turnbull for encouragement and moral support throughout the development of this, the most complex work I have undertaken to date.

Terry Blunt

CONTENTS

1

Introduction

1.1 About the Archimedes

The Archimedes has been designed very specifically as a fast graphics machine. At the time of its launch, the only really effective Wimp driven systems were the most expensive business machines. Since then, some older, command line driven operating systems, as they are called, have had bits bolted on to try and bring them up to date. This is never very satisfactory however, when two basically incompatible design concepts are being forced together.

Again, many command line driven computers look decidedly clumsy now, often having little by way of programmer support. Hence, there tends to be a wide variety of usage, and little standardisation of keyboard/mouse handling, file and data transfer, or resource sharing. This ultimately makes it harder for the computer user to get the best out of the machine, and is liable to result in programs full of obscure bugs.

There can be very few computer manufacturers in Acorn's enviable position of having designed the processor and its major support chips, the software in the form of Risc Os and many support applications, as well as the overall design of the complete computer system. Thus Acorn have an unusually intimate knowledge of every part of the Archimedes, and can give detailed information with a very high degree of confidence.

While the Archimedes was running under the Arthur operating system, and arguably still effectively in final development, Acorn spent a considerable amount of effort persuading the major programming organisations to adhere to their protocols. This included not only the Wimp environment, but also the low level interfacing with the Risc Os operating system itself. The success of this strategy has resulted in a powerful, integrated system that is readily expandable both from a hardware and a software point of view, a system well suited for either home or commercial use.

It is doubtful whether there has been any computer that hasn't, at least in passing, had an attempt made to use it for playing games, regardless of whether or not the machine is really suitable. However, the speed and graphic capabilities of the Archimedes makes it a fairly natural choice for modern graphically orientated games. Indeed, although always a dangerous statement to make, it is probably currently, the best machine available, being able to deal comfortably with the demands on speed, memory and graphical capability, all at a reasonable cost. In fact, the heart of the machine is currently used by a French arcade game manufacturer as its game *engine*.

An area of computing often overlooked is that of sound production. Acorn have not fallen into this trap, but have provided a sophisticated sound system giving eight channels of high quality stereo sound. The Archimedes is capable of utilising complex sound generating algorithms, as well as sampled sound data, most of the control being handled transparently in the background. This is all the more remarkable when you consider that the Archimedes doesn't have a dedicated sound chip set, but works by software control of a relatively simple A/D converter. This then makes the Archimedes the ideal platform for developing your ideas into fast, colourful, lively games.

Obviously with all the features available under Risc Os, comes considerable complexity, and it is likely that you'll have to spend a fair amount of time getting to know the machine. There is a great temptation to criticise features that appear to be unnecessarily complex, but as you become more familiar with Risc Os you will see that there are very practical reasons for Acorn's compromises, not the least of which, in some cases, is that of maintaining compatibility across an unusually wide range of old and new hardware.

Acorn's long term policy of maintaining compatibility over different models provides a high degree of confidence that, provided you stick to Acorn's guidelines, the game ideas and programming techniques you develop will be valid for a long time, probably far longer that the commercial life of the games themselves. Acorn make the point that, in real terms, there is absolutely no benefit in trying to use the old peek and poke tricks, that were so common in the older machines.

1.2 The Environment

Most of the ideas in this book assume that you are working in Basic. However, Acorn Basic V is such a well structured language that, provided you've developed the habit of using tight, self-contained modules, moving over to another high-level language should present few problems. You may well be tempted to use one of these compiled languages, or even one of the compiled versions of Basic. Sacrificing some accessibility in this way can be an advantage in the form of a small degree of copy protection.

Compiled languages can be quite difficult to disassemble. Added to this, many compilers have facilities to enable you to create relocatable modules, or turn your program into a fully self contained application. On the downside, even the Basic compilers currently available, are more fussy about things like empty loops and procedure exit points. Some features of Basic V, such as the EVAL function can't be realised at all. In general, with compilers of any high-level language, you will need to be more organised in your programming style.

The commonest reason for choosing to compile is to increase execution speed. This will add smoothness and polish in many situations, but if there is a serious speed problem, choosing to compile for this reason alone, may simply mask rather than cure it, and the difficulties may show up again later. If you have such problems it is likely that you should either re-consider your data structures or look again at the practicality of your overall game design. Possibly you should move critical sections into assembled ARM code, which is the most efficient form of all. However, neither compiling nor moving to ARM code will make very much difference where the time delays are mainly caused by intensive screen manipulation through the normal VDU drivers.

There are quite a few sections of ARM code in this book. I make no apology for this, as there are some things which simply are not practical to do in any high level language. The sections of code I've included are so short that you should have no problems using them, even without understanding how they work. In any case ARM code is probably the easiest modern machine language to understand, so once you are familiar with it, you may find you actually prefer working from Basic simply in order to assemble fast efficient ARM code.

There is a chapter specifically on ARM code for direct screen manipulation and the like. This is really aimed at those of you who already have an understanding of ARM code programming. However, those less familiar with this subject, will still be able to make use of the code fragments in this chapter, but may have difficulty expanding the ideas contained. I recommend that you either follow up by reading through the ARM programming series, run by several of the BBC Micro magazines or the ARM programming books now available.

A large number of operating system routines are available to you from Basic via SYS calls. These let you really get into the machine and make full use of the operating system, with a high degree of confidence that your programs will survive any future improvements in the hardware and Risc Os itself. Those of you familiar with the older BBC machines will particularly appreciate the ability, in many cases, to pass parameters in, and get results back from, these calls without the need to set up special parameter blocks.

Acorn use a highly sophisticated system of memory management. One of the results of this is that very few areas of memory are fixed. Where it used to be common to peek and poke memory directly to gain speed advantage, or for various special effects, such techniques are doomed to failure on the Archimedes. Acorn provide reliable *official* entry points to all the routines, and memory areas you could possibly want to access.

1.3 Programming Techniques

Quite a few of the example program sections in this book are rather crude and unfinished. I've done this quite deliberately so that you can see the principle or idea as clearly as possible. In any case, you're supposed to be writing the game not me! I've also used simple drawing routines instead of sprites for some demonstrations. These wouldn't normally be practical as, even on the Archimedes, they are rather slow, but are simpler to program and easier to follow when demonstrating a point.

As this book is more heavily weighted on games aspect rather than pure programming techniques, there are some unexplained details in the program examples. Unfortunately this is necessary to prevent the book becoming unmanageably large and heavy going. The really obscure points are explained, but if you find that some other points seem curious I can only assure you that there are sound reasons for them, and that a little experimentation will probably reveal these. If all else fails, look through the relevant parts of the Basic Guide, or if you have it, the Programmers' Reference Manual.

As a general rule, you will have to take great care with error checking and input range tests, although I haven't done much of this with my examples. By the very nature of the activity, players are likely to be very careless and inaccurate when thoroughly engrossed in a really exciting game. The keyboard and mouse are likely to be sending all sorts of rubbish to your program so your input routines must be really bomb proof.

Be wary of using the Ctrl key for any game functions - you can get some highly undesirable effects when this key is hit at the same time as others. By default, Ctrl+[has exactly the same effect as pressing the Escape key. In fact, it would be wise to completely disable normal Escape key action in the main game loop, and have it tightly controlled with error procedures in the rest of the game. You'll make yourself extremely unpopular if, in the middle of a particularly frantic starfight, your player sees *Escape at line 220*.

As well as accidental errors, you should also make some allowance for deliberate input errors. Mind you, if that is happening it tends to suggest that your players may be getting bored. Occasionally people will enter long strings of digits for numbers, trying to make the game crash. By using string input, and checking the length, the program section below completely eradicates this problem. If the player just presses Return or enters letters, or even control characters, the routine will simply return a zero.

```
REPEAT
   INPUT number$
UNTIL LEN number$<39
number%=VAL number$
```

Similarly you should check for mouse button presses, by masking the bits to avoid the confusion of two buttons being pressed at the same time. You would then use something like the lines:

```
MOUSE x%,y%,b%
   IF (b% AND 2)=2: REM menu pressed
```

Alternatively to trap all eventualities use a case statement like the following snippet:

```
MOUSE x%,y%,b%
CASE b% OF
   WHEN 7:REM select+menu+adjust
   WHEN 6:REM select+menu
   WHEN 5:REM select+adjust
   WHEN 4:REM select
   WHEN 3:REM menu+adjust
   WHEN 2:REM menu
   WHEN 1:REM adjust
   OTHERWISE:REM no buttons
ENDCASE
```

There are many situations requiring nested loop structures, which can be a fine opportunity for mixing up loop counters. You should therefore take a fixed policy towards the names of variables used in these situations, and their order of precedence. This is shown with the following somewhat contrived fragment.

```
FOR K%=4 TO 90 STEP 3 : REM only J & I loops inside K
  J%=21
  REPEAT
    J%-=1 :             REM yes it's a loop counter
    FOR I%=0 TO 4 : REM no loops inside this one
      IF block%(J%,K%,I%)>5 end%=TRUE
    NEXT
  UNTIL J%=0 OR end%=TRUE
  FOR I%=0 TO 5 :       REM not J - it's an inner loop
    IF NOT end% THEN count%+=1
  NEXT
NEXT
```

Following a series like I, J, K, L, M, ensures that you always know what nesting depth you've reached regardless of the order that the loop variables are be used in array indices and the like. To be consistent, you mustn't use these variable names for anything else in your program.

In any case, it's a good idea to make a point of being consistent with your use of variable and procedure names. Personally, I take a policy of using lower case letters for all except the resident integers A% to Z%. These I usually reserve for fast loop variables, passing values into ARM code, or for passing values from one program to another. If you keep all other variables to lower case, keywords stand out more clearly and there is no possibility of accidentally creating hidden embedded keywords in variables, like the *TO* in *TOTAL%*. It is also wise to avoid subscript variables like *a1%, a2%* and *a3%* unless there are clear mathematical reasons for their use. These can become very confusing.

Further confusion can result from the use of the underline character in variable names. Although quite acceptable to Basic, and giving neat looking words in printed listings, it is very easy to put a minus sign in by mistake. Deeply embedded in a nest of genuine mathematical symbols, such an error would be very difficult to find.

Yet another source of confusion is excessive use of multi-dimensional arrays. It's very tempting to try to pack an entire data structure in one array. But this can be very difficult to follow, and in any case is not always the most efficient form either for memory or speed. You are generally better off using a group of arrays dimensioned to the same size, often referred to as parallel arrays. If a single array is unavoidable, then consider using constants for the subscripts rather than numbers. It makes it easier to follow, and update if necessary.

Compare the following:

```
DIM parameters%(3,40)
DIM size%(40),weight%(40),colour%(40),position%(40)
```

Similarly, where you're obliged to use a single array:

```
DIM attrib%(5,1,30)
strength%=0:psi%=1
intelligence%=2:stamina%=3
experience%=4
us%=0:them%=1
```

Then in the main body of the program

```
friend%=23:foe%=9
```

Compare these:

```
try%=attrib%(0,23,1)-attrib%(1,9,1)
try%=attrib%(us%,friend%,psi%)-attrib%(them%,foe%,psi%)
```

Generally I recommend the use of constants. You may want to incorporate new features in your game, involving adding an extra element in an array for example. It becomes extremely tedious when you have to hunt out every FOR-NEXT loop to change the terminating value, where using a constant requires only one change.

Often when designing a screen layout, even from a paper plan, there are small adjustments needed in the positions of objects to make the overall effect look balanced. If you use constants, for things like borders and panels in particular, then you will only need to adjust the constants to re-align all parts of that particular group.

```
MOVE left%+width% DIV 2-halfchar%,bottom%+height%
PRINT "H"
```

Although this looks rather wordy, you can always thin it down later when you are satisfied with the overall appearance. Many of the examples are spread out like this, and significant time and memory savings can be made by compressing the final working program.

Many programmers drop procedures and functions all over the place in a program with no thought given to the overall readability. A better idea is to decide on an order of precedence and stick to it as closely as possible.

A very common form is to list these routines in the order they are called, combined with a nesting level. As this is easier to show than explain, there is a typical program framework below.

```
   REM ------ Main game loop
   REPEAT
      PROCinitialise
      REPEAT
         PROClevel
      UNTIL levelend
      PROCupdate
   UNTIL gameend
   PROCfinalise
   END
   :
   REM ----- 1st level routines
   DEF PROCinitialise
      REM code
   ENDPROC
   :
   DEF PROClevel
      result=FNkey
      REM code
   ENDPROC
   :
   DEF PROCupdate
      PROCinput
      REM code
   ENDPROC
   :
   DEF PROCfinalise
      REM code
   ENDPROC
   :

REM ----- 2nd level routines
   DEF FNkey
      REM code
   =result
   :
   DEF PROCinput
      REM code
   ENDPROC
```

Obviously there is a limit as to how far you can take this idea. You may, for example, have your key press function being called from several routines at different nesting levels. You should then put it at a level below the lowest one that calls it.

If you use the Basic editor to work on your program, which is definitely advisable, you can temporarily move a procedure or function to the very end of the program while you work on it. This means you can get to it with a simple Shift/Down cursor in the editor, and just List, from Basic. You'll find this rather easier than having to search by line numbers, which will of course change as you add to the program. When the routine is reasonably under control you can move it back where it belongs.

1.4 Making Notes

Probably the most repeated and most ignored programming advice is that of documentation. I repeat it here, as I believe it is essential for effective programming. It really is important that you make some form of plan. It is also important to make notes of what ideas you get, as you get them, preferably dated, and including the reasons why you decided on any specific course of action. This last is probably the most important of all. In the past, I have wasted hours trying to understand a complex piece of code that seems to defy logic, a section of code that I wrote only a few months previously.

When I have a new idea to work on, I usually take an easy informal approach at first. I simply keep a separate folder for each project I'm working on, with loose sheets of paper in it. These are dated and dropped in while I think out an overall idea and try to clarify in my mind the overall structure of the idea. At this stage it is the idea I try to structure, not the program that will eventually result. When I come to develop the program properly I read through all my notes before doing anything else. I've learnt from experience to keep these early notes even after I've prepared more organised documentation.

Once you have enough ideas put together to give you an outline of your game, you should start the proper documentation. Initially, using your informal notes, make out a simple list of exactly what the game is and what it does roughly in order. The example below should give you a clearer idea.

❑ This game is a combat game

❑ There are two teams of opponents (goodies and baddies)

❑ Goodies are controlled by the player, baddies by the computer

❑ There is an option for baddies to be taken over by a second player

❑ Fights are between pairs of opponents

❑ Any goodie not attacking a baddie will be a first target for the baddies to attack

❑ Any attacker may fumble and strike itself or a comrade by mistake.

Lots of people throw up their hands in horror as soon as you talk about documentation, flow charts and pseudo code. In fact it's much easier to handle these ideas than you might think. In the first place, taking a simple comparison, nobody's frightened by a list of knitting instructions. This is, of course, the documentation using a specially coded unambiguous set of instructions, for making the garment. In other words, a pseudo language. Your knitting pattern will almost certainly have a set of diagrams showing the order in which each part of the garment is to be made and fitted to the rest. This is a clear example of a flow chart.

In brief, pseudo code can be any set of instructions that are completely unambiguous. It doesn't necessarily have to follow any commercial guidelines, as it's for your benefit only. A flow chart is a diagram that shows the running order of events, and again there is no need to worry about whether you are using all the correct symbols. Just draw boxes, with a two or three word note explaining what the box is, and arrows leading from one to the next.

In your documentation, along with a detailed program outline, you should list all procedures and variables that are used. For procedures and functions, include all parameters passed in, noting which may be modified. Also give a brief description of the purpose of the procedure. Here is a typical example of this.

Name:	PROCsetsprites
Sources:	Spritearea%, spritepointer%(), DATA
Function:	Creates a table of sprites, reading data from the first line after the procedure, using workspace at spritearea% and filling array spritepointer%() with the address of each sprite. Total number of sprites is in spritetotal%
Results:	Spritepointer%(), spritetotal%

This may seem rather tedious, but in a program of any size, that may take months to develop, a little time spent on documentation can save you hours of frustration. This is particularly true of variable names. It is amazingly easy to choose the same variable name for two different jobs, resulting in the most obscure bugs to try to find. You should do this work as you develop the routines. If you leave it till later you will almost certainly have forgotten important points, and will probably find the task of documenting 40 or 50 routines a daunting prospect. If you are fortunate enough to own two computers, you could have the second one set up as a wordprocessor for instant access. Sometimes I use an elderly BBC Model B for this.

Once again, keep your earlier documentation. If flaws develop in your logic, provided you have kept as much detail as possible while developing your design, these problems will be easier to follow back to their source.

There are two distinct schools of thought regarding comments and remarks within program listings. On the one hand you may be advised to sprinkle you programs liberally with remarks, but at the other extreme you'll be told that you should keep all your comments in the main documentation.

If you intend to compile your program you can probably afford to be pretty liberal with comments, although too many on one block of code can, instead of being an aid, become quite confusing. Where you are running interpreted Basic, the situation is quite different. Any extra text in the program both consumes processing time and memory. Therefore it is probably best to make more detailed documentation and keep your programs clear and uncluttered.

2

Early Planning

It's a great shame that, for many people, the rush of enthusiasm for an idea, rapidly dissipates when they are told that they will have to plan out their idea to make use of it. The unfortunate consequence of failing to do so, almost always results in disaster as unexpected problems arise. Often a basically sound idea is then abandoned completely as being impossible.

With this in mind, I've drawn up the planning ideas in this chapter. As with most of this book, the suggestions are not rules etched in stone, but a starting point, which you can build on and alter to suit your needs. I dislike planning ahead as much as anybody else, so I try to keep documentation to the bare minimum without losing important information. Probably the most important point to keep in mind, for both game design and programming in general, is to be consistent in whatever approach you choose.

2.1 Finding Ideas

For a lucky few, ideas come fully formed while languishing in the bath. Unfortunately, for the rest of us, quite a bit of effort is involved in generating ideas. It is important to be clear at the outset, as to whether you intend to write for your own satisfaction, and that of your family and friends, or whether you hope to write games for commercial distribution.

In the former case you only need to follow up comments from your friends, and can get a framework for a game quite easily from them. The standard of gameplay could afford to be lower as well, because much of the attraction of the game will be that it is a *home brew*. However you'll probably get little satisfaction from writing a game you know is below standard, no matter how much some of the family might like it.

If you are determined to get your name in the charts, I'm afraid you'll have to forget the games you personally think are the best, and be a bit more commercially minded. Have a good look at the ones that seem to be selling. As well as playing them yourself, if you get the chance, watch other people playing them. If you have the opportunity go along to some of the computer exhibitions. You'll get a free look at some of the best games on the market, and with the larger shows, you can see what the competition is like on non-Acorn machines. Another point is that at these shows you can chat with a wide cross-section of the computer using public.

You should be looking for the things that are most disliked. These are the features you want to avoid in your games. Human nature is perverse enough to ensure that, out of a dozen features, people will discuss the one point they find objectionable in preference to the other 11 they approve of.

You may find, after looking over a number of existing games, that there seems to be a gap in what's available. Obviously, the more games you see, the better your chance of finding such a gap. If you're lucky you could be on the track of some entirely new action.

It is very easy to get discouraged when you see the slick presentations of some software houses. Keep in mind that most of these have teams of professional programmers with years of experience, but that all the professional expertise in the world will not overshadow a single really good idea. Once you start writing games, you will develop your own tricks and hopefully, end up having other people envying your work.

2.2 Targeting

When forming the outline of your game you need to consider who it is aimed at. Apart from anything else, you need to know what degree of complexity you can reasonably expect your players to cope with. With this in mind, when writing games that have a large number of dynamic controls, such as flight simulators, it is advisable to provide an auto option for as many as possible. In this way players can slowly take over control of the whole game as they become more experienced. I have frequently discarded games that looked promising at first, simply because they demanded too much too quickly, and became a chore instead of fun.

You also need to take into account the of age of your potential players. In general, the peak age for Alien Zappers is about 14, and for Adventurers it's nearer 18. Very young players need games that are brightly coloured, with little detail, low player accuracy, and short concentration times. People in their 20s are more likely to be interested in simulations and strategy games, and beyond 30 or so, your potential players are probably looking for the very long term challenges that can be easily put aside for a while. These may be sophisticated versions of all types, but requiring many hours or even weeks, of concentration. Detail will be extremely important. Financial simulations, for example, are most likely to be played by middle-aged lower management people, or those who think they should be there! Your player is likely to be pretty picky about simulations that refer to long term loans, international exchange rates, interest and the like.

Having said all that, I don't think anybody can resist the occasional session of a few minutes of pure, mindless destruction! I expect there will always be a market for games that simply involve mass annihilation of hoards of beasties.

2.3 Types of Game

It is important to work realistically within your limitations, and to choose the type of game that you understand best. Although possible at a stretch, you will have an uphill struggle if you try to write, say, a flight simulator but know nothing at all about 3D projection. However, you can gain a great deal of new knowledge and satisfaction from developing a game idea where, at first, you only partly understand the basic principles. As you gain experience you can take on progressively more sophisticated ideas.

In the later chapters of this book I've given descriptions and a breakdown of the main game types. The list is not exhaustive and it's quite likely that other people would give you a different list. As if that isn't enough, most modern games contain elements of several basic types. This is particularly true of the so-called graphic adventures, which are part zapp-'em, part story. Initially game types can be subdivided as follows:

- **Arcade**
  - Alien Zapping
  - Rebounds
  - Platforms
- **Role Playing**
  - RPGs
  - Adventures
  - Combat
  - Simulations
- **Strategy**
  - Cards
  - Board/Parlour Games

The first group consists principally of fast response games, hence their attractiveness in the arcades. For these you will need to be able to produce really *tight* programs. You need to be able to see the action very clearly in your mind, from a programming point of view, and be able to circumvent speed bottlenecks. However, you generally won't need a knowledge of mathematics beyond middle school level, and your game will only require cartoon type realism.

The second group requires more of an understanding of relationships and trigonometry rather than absolute speed. Here you will be mainly thinking from the point of view of how a change in one piece of data will affect its neighbours. At the same time, most of these games have a background map structure of some form, which may also interact with other data types. Any graphics are likely to be more detailed, and require greater realism.

In the last group, observed speed of execution is generally relatively unimportant. Understanding algorithms and rule systems is your main target. Your programming here will be more of a hard logic type. You can expect to have to develop quite complex routines for computer generated moves, most of them probably recursive in nature. Presentation is very important. Your player will be gazing for long periods at essentially static displays. Tiny details can develop an irritation factor quite out of proportion to their size.

2.4 Originality

It is pointless writing a game that is more or less a copy of an existing one, unless the one already available is remarkably poorly executed, or you intend to produce a cut-down budget version to compete with a very expensive game. In the latter case you will have to be very careful about copyright. What you really need is an idea that is different to anything else on the market, but not so different that it is hard to understand.

One possibility would be to add a new twist to an old game. A good example of this kind of development is the breakout theme. From a simple bouncing ball knocking out bricks, this has reached the point of multiple balls, varying wall shapes and construction, extra bonus bricks, two player options and even a circular bat movement instead of just side to side. So far, I've not seen a breakout game that is genuinely three dimensional, so there's one possible opening.

Old parlour games give other possibilities. After all, the basic idea behind most modern platform games has a great deal in common with Snakes and Ladders! As well as the familiar games like Chess, others such as

Backgammon, Draughts, Ludo, Reversi and Solitaire have all been successfully transferred to computer environments.

It is also worth looking over real-life sporting events for ideas. Golf, snooker and football simulations are now well developed, but there is plenty of scope for improvement, and for new computer versions of other games, ranging from grouse shooting in Scotland to white-water canoeing in the Grand Canyon. I remember that an incredibly primitive fishing game on the 8-bit BBC used to attract players away from far more sophisticated commercial games, probably because fishing is something that almost everyone can identify with.

Finally there are the real-life working situations that you can draw on for ideas. This, of course, is where flight simulators have their origins. A few simple examples are:

❑ Rail transport system management

❑ Sea defence maintenance

❑ Sheep or cattle droving

❑ Fire fighting

❑ Deep space asteroid mining.

2.5 Addictiveness

The ideal game is one that people want to play indefinitely. The psychology of this is quite complex, but hinges around two basic concepts: Players must always get a reward and feel that at each attempt achieved something better than the previous one.

A reward is not necessarily a successful completion of a level or section. It could also be a well worded failure message, or display. Whatever it is, it should be relevant, it should be encouraging, it should represent progress in either direction, and if at all possible, it should not be repeated.

Varying the way each level ends, goes a long way to giving the player the feeling of progress, but simple random variations will quickly be recognised as that, and may actually serve to reduce player interest. One of the best indicators of progress is time. As players become familiar with a game they almost always react faster. Therefore, on the very first completion of a level, store the time taken, even if it isn't part of your scoring system. On the next completion of the same level, compare the times. If you have say, a five percent improvement then give a more up-beat end message.

Try to avoid absolute time references. What you want to do is to mark improvements against a player's own capabilities, not against those of some lighting fast games freak.

2.6 Desktop Games

It is well worth thinking about arranging your game so that it can multi-task as an application. Challenges like Patience or Solitaire are particularly amenable to this treatment. Puzzle types, or text adventures are another likely group. Your player will then be able to drop in and out of the game easily while, say, writing to a friend, or doing the household accounts. However, this idea is hardly practical with a game that heavily uses the whole screen, or requires all 256 colours in the display.

Apart from any other considerations, your player is quite likely to have the desktop set up for Mode 12, a 16 colour mode. An invaders type zap-'em typifies this problem, and also makes such heavy demands on processor time, it would be unlikely that effective multi-tasking would be possible. Having said that, I have seen a desktop Space Invaders written entirely in Basic, that certainly matches the playability of the earlier 8 bit implementations.

No matter what type they are, initially all your games should be made to run as applications. Shift/Break can't be relied on to work, one reason being that some players may have their machines configured to auto-boot from a hard disk. All games should also be made to return cleanly to the desktop with just an Escape key press, or mouse click on a suitable icon, allowing other suspended tasks to continue where they left off. I know of at least one distributor who won't even look at a game that can't run this way.

2.7 Teamwork

It usually works out that the best programmers are not the best artists or musicians, so it is well worth considering teaming up with a graphic artist, or possibly, if you want to improve the sound content of your games, a musician. On the downside, you lose a measure of control and need to be far more organised. At the same time, you share the workload, and are more likely to spot bugs before they become a real problem. You should also end up with a game that is far better than any of you could produce by yourselves. Most of the best games currently available are produced by two or three people working together.

It is particularly important to have good graphics. So much so, that it is well worth devoting days or even weeks to getting a single screen just right. The game playing public are becoming very fussy about standards of artwork. Pretty scenery and cute monsters can make a rather ordinary game into a best seller. Sound, at present, is less critical, but will get more so as the public become educated as to what is possible, and will therefore expect better sound effects and music from all games.

When, as a team, you have something that you think worth marketing, it is probably as well for one of you to take on the role of front man. It can become very confusing for potential distributors if they have several people to deal with, all saying something very slightly different.

2.8 Some Dos

No matter how wonderful you think your title sequence is, it is a sad fact that most people will get bored with it sooner or later, so you should always provide a means of getting straight to the heart of the game. Wherever possible use a system that enables either a key press or mouse click to move on. This is also relevant for instructions, as once understood, nobody wants to wade through sheet after sheet of text. However you should always supply quite detailed instructions on, for example, a menu click. This is much better than relying on a disk inlay card which has a habit of becoming lost.

If no keys are pressed, it is well worth making the title screen *time out* to a short demo of the main action, then possibly on to a score table, followed by more action in another part of the game, and so on. As soon as anyone touches the mouse or keyboard, the game should instantly go back to the title screen, giving an immediate invitation to play. Let your potential distributors know about this - it may well help to sell the game. Dealers will be more likely to leave it running on a spare showroom machine.

Make sure there's a story line. We all know that in reality it's just a lot of complex calculations, and fancy pixel pushing, but playing games is the way most people suspend reality for a while. The sillier your story is the better. Look at these examples:

Use the mouse to move the spot onto the rectangles to make them disappear.

While out with your friends, hunting Gronks, you are appalled to see that the Zarks, led by your old adversary Thrid, are building a dam across the Wyde river. Instantly you realise that their plan is to inundate the beautiful city of Crystalfire, home of your true love. One of your friends has taken a *flier back to Lonya, to try to muster reinforcements, while the other sadly, has already succumbed to the deadly fire of a Zark pin blaster. Only the lightning responses of a seasoned fighter like yourself can frustrate the Zarks' evil plan, by destroying the dam as fast as they build it. You wonder at the twist of fate that brought you here - the first Lonian to have a flier fitted with the Mk III Brendell portable Laser cannon, a weapon capable of pounding the huge granite mega-bricks into vast, billowing clouds of incandescent dust.*

Now, which game would you prefer to play? A point worth mentioning is that the story line, while including the obligatory bit of mushy romance, is completely non-sexist, and non-racist too (unless you happen to know any Zarks).

If possible give your player the choice of keys, mouse, or joystick control. There are a number of joystick add-ons now, so try to cater for them. The makers will be only too pleased to give you software control information. It is in their interests as much as yours, for new games to be able to use their hardware.

If you make use of sound, which is almost mandatory these days, it is essential that your players have the option of turning it off, or even better, give them full volume control. Complex sound generation algorithms are probably not a good idea. They tend to be very difficult to develop, needing a thorough understanding of the sound system at machine level. Such algorithms tend to be comparatively slow to execute. You are probably better off using sampled sounds. Most sampler software includes voice module making capabilities and permission to use these modules in your own programs.

There's no reason why you shouldn't use public domain routines, sound samples or pictures in your games, but do make sure they really are PD. Check with the author. Apart from anything else, there may be a better version. As a matter of courtesy give acknowledgements for these routines, and if you have presented your game to distributors make sure they know about these inclusions.

If you want to be really squeaky clean, close and kill any modules that your game has loaded, and unset any system variables, File$path, Alias$ and the like. Delete any system sprites that you've defined. All of this gives memory back to the desktop task manager. However, don't kill any modules that you don't own. I was intensely annoyed by a supposedly desktop compatible game that killed the Memalloc module, causing another application to crash because having *RMEnsured* its presence earlier, the application quite reasonably expected it to be still there.

It's doubtful whether anyone is seriously using an unexpanded A305 now, but there will be an enormous number of A310s about and quite a few unexpanded A3000s. Plan your game to work within 1 Mbyte if at all possible. This may mean using text compression or multi-part programs and the like. If you are fortunate enough to have an ARM 3 upgrade, be especially careful to ensure that your game won't slow down noticeably in an older ARM 2 machine. Also, if you have the MEMC1A upgrade, bear in mind that the older version runs about 10% slower, and these were fitted to all the older 310s and 440s.

Always build a set of cheat routines into your games. The following list is fairly typical:

❑ Immortality

❑ Jump to location/level

❑ Disable baddies

❑ Remove baddies

❑ Slow motion/single step

❑ Walk through walls

❑ Undo last move

❑ Go to game end.

Not only will this help you when debugging, you can make this information available to reviewers on pre-release copies. If you want to be really nasty, you can always disable them on final release versions!

2.9 Some Don'ts

Don't use any of the system workspace areas if you can possibly avoid it. The sizes can't be guaranteed. If you can lump all your memory requirements together into your application user space, you will know if there is enough available with a single Wimpslot test in the !Run file.

You should avoid like the plague doing any auto-configuring. There is always a way round the problem. The most common excuse for the need to re-configure, is that of screen size. There are three fairly painless solutions to this. Firstly your game can check the screen size, and if it is inadequate, abort with an instruction to the player to set the screen size from the task manager. As an alternative, you can instruct him or her on how to find the MemAlloc module in the !Lander application on the Apps 2 disk. Then explain how to move it over to your disk. This will only have to be done on one occasion, so is not much of an imposition. Using this your program can then set the screen, font and RMA sizes as required. Finally, you may be able to get permission from Acorn to put the module on your own game disk, with a suitable acknowledgement. Don't assume that you can though. If in doubt ask.

When your game closes down, don't leave the sound system locked into your module so that a VDU 7 beep sounds like space invaders' gunfire. Make sure it restores the sound system to the condition when you started, re-setting the voices, and cleanly detaching and closing down any sound modules belonging to your game.

Avoid topical themes for storylines, unless you can get your game completed and distributed very, very fast. A topical theme will rapidly become dated, and could seriously shorten its commercial life.

Don't put the name of any other title, software house, programmer or distributor anywhere in your product without written permission. If your game is similar to one commercially available take great care that nothing in it can be taken as a direct comparison with the original. There were an awful lot of law suits flying around in the early 80s due to rip-off copies of well known titles. In particular, be careful of computer implementations of well known board games. Many of these are copyright, and already licensed to a software house.

Don't make last minute changes to a game you are about to send off for consideration by a distributor. Be especially wary of invisible time wasters. I nearly fell foul of this with the star program of Chapter 5. I thought I aught to add a mouse buffer flush inside the main loop just to be tidy. The result was that the loop time became too great for one screen refresh, but only when the star was at the top of the screen. Always therefore, check any changes very thoroughly, no matter how simple they may seem.

3

More Planning

3.1 Identifying Major Stages

The secret of managing a large, sophisticated program is not trying to handle it all at once. If you can't deal with a problem break it down into smaller problems and keep on doing so until you've got bits that are small enough to manage. The only real problem then becomes that of deciding where to start.

All properly designed games can be immediately broken down into three principal stages:

1 **Initialisation**
2 **Game loop**
3 **Finalising - Endgame**

Initialisation will, of course, involve setting up the framework of the data structures and loading or defining score-tables and startup values. From the game player's point of view, it will consist of the title screen, instructions, and where relevant, the story line.

The **game loop** is the game proper, where the program spends most of its time.

Finalising would be concerned with storing any sections of data, like score tables, that will be used next time the game is played, also ensuring a tidy exit, preferably to the desktop. From the player's viewpoint, the endgame would have the closing messages, best score congratulations, or otherwise.

As you can see, we've already broken the game down to three much smaller parts, and we don't even know what it is yet! If we break these down further we might get something like the following:

1.1	Title screen
1.2	Set up data structures and major game constants
1.3	Load sprites and other data
1.4	Storyline
1.5	Instructions
1.6	Key/Mouse choice
1.7	Continue saved game request
2.1	Start game loop
2.2	Set up game level
2.3	Play level
2.4	Update scores, lives, level
2.5	If loop conditions OK repeat game loop
3.1	Endgame message
3.2	Position save request
3.3	Close program and return to desktop

If you keep breaking the problem down like this you will soon find that you have it defined almost to program level.

As a point of interest you will see that I've put the bulk of the true initialising after the title screen, but before the rest of the program. This fools players by giving them something to look at so that they don't realise how long the game takes to set up.

When you get down to the level of the main game loop itself, start off by listing every action that takes place in the loop. Don't worry about the order at first, the main thing is to make sure you've not omitted anything. Once you have the list, you can then begin to sort out which actions are dependent on others, and get the ordering right with the appropriate tests. The list below is fairly typical for an arcade game.

Check key press
Check mouse
Move player object
Move enemy object(s)
Move missile(s)
Check collisions
Destroy objects
Create objects
Check time

Obviously not all operations will be taking place at the same time, so you can start by positioning those actions that have to take place every pass. These are highlighted in bold in the example above. This will form the framework that everything else hangs on, hence the importance of making sure you don't omit anything at this stage. Notice that the collision check has to be made every pass to allow for any possible movement, or creation of new objects. Below is yet another fairly typical list. This one is for a draughts game and is in pseudo code form.

```
Game loop start
   Identify which player to move
   If computer move then
      Calculate best move
   Else
      Accept player input
      Validate move
   End If
   Make move
   Check for game end
Repeat Game loop
```

The precise method you use to formulate your ideas and get them down on paper, isn't, in fact, too important. The important points are that, at a later date, you understand what you have written, and that the notes assist you in clarifying your thinking.

3.2 Data Structures

It should be around this point that you start to think about the variables you are going to use. You will probably be manipulating a lot of data in the body of your game, and you will need to work out what form, or structure that data should have. Where you have groups of objects or characters, it is almost inevitable that you will eventually decide to use an array of some sort to contain this data, with a FOR-NEXT index variable picking off individual items. With the powerful array and matrix arithmetic available in Basic V, arrays become particularly attractive for manipulating related data such as three dimensional vectors.

Names are of course most obviously held in strings or string arrays, but numeric values are less obvious. As a general rule integer arrays will usually be most practical. If your game plan seems to need an array of real numbers you should look again, as you may be using unnecessary precision. Bear in mind that in Mode 13 for example, pixel size is four graphics units, both X and Y, so for drawing purposes you don't even need integer accuracy. Apart from this, you can increase your integer accuracy sometimes, by using the Basic barrel shifting operations to multiply up by, for example, 1024 while performing calculations, then shifting back down again afterwards for your actual results.

Where you are expecting a fair amount of ARM code mixed with ordinary Basic, it will probably turn out that the most efficient structures are word aligned blocks of memory. These can be accessed directly as double words in ARM code and as indirected integers in Basic. Indices are simply barrel shifted to get the correct address as shown in the fragments below.

Basic

```
N%=array%!(index%<<2)
```

ARM code

```
LDR R0,[R1,R2,LSL#2]
; R1 contains the array address R2 contains the index
```

It is sometimes best to think of all the individual data structures as a whole entity. Ask yourself whether there are repeating types that can be usefully combined and handled by common procedures or functions. If, for example, you are working on an adventure it might be wise to treat all the data relating to the characters as a single structure. If you call this structure an object, you can define a set of procedures to handle it. Internally they can be quite complex with all the separate arrays handled in appropriate loops. To the rest of the program they are simply closed boxes with just an object number being passed in, possibly with some information being returned. Those of you familiar with other languages will probably recognise this as a primitive record type.

Whatever structures you finally settle on, it is well worthwhile leaving spare elements. There is a good chance that later enhancements, or final trimming of the game, can use these to good effect and you can then slot in the modifications with minimal disturbance to the rest of the program.

3.3 Layouts

If your game is to really attract, it is vital that you get the layout right. This is something that you will need to decide quite early and spend a fair amount of time on. To start off, just make outline sketches on paper. It is quite tempting to use !Draw or an art package of some sort for this, but unless you are exceptionally good at handling them, they will only slow you down at this stage, so you are better off sticking to pencil and paper.

3.3.1 Positioning

Usually, with fast action games, it helps to keep the peak activity slightly below the centre of the play area. The slower the game the more you can afford to spread the action out, without your player losing control.

In most arcade style games the play area is reduced and various bits of relatively static information are put on borders and panels round the edges. Where you need to do this you must be very careful about where on the screen, you put the information. Figure 3.1 shows the most common arrangements for a wide range of games. I've only shown the bare bones of the layout. There would of course be far more information in most games along with static detail in the borders, to make the overall appearance more attractive.

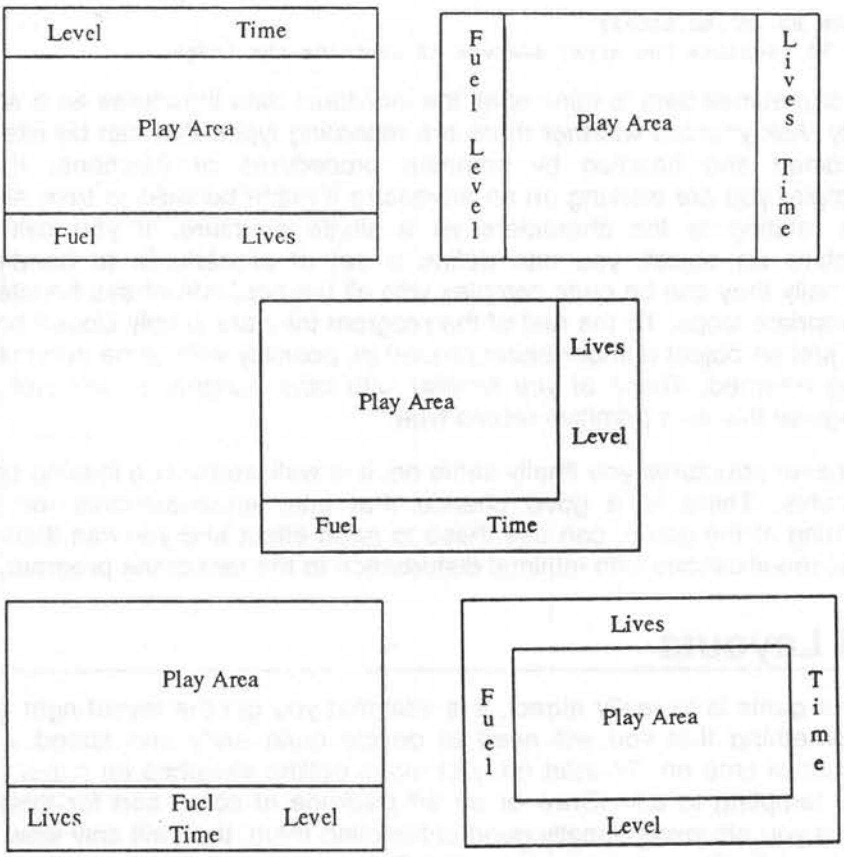

Figure 3.1: Screen layouts

Your player can only really concentrate on one or two areas at the same time, so the one I suggest for most fast games is layout 4. Here you have all the vital information directly under the main game play area, and your player's peripheral vision should be able to pick up the less important stuff either side of this, without any difficulty.

Layout 5 is the worst possible. Having such a spread of information all round the edges of the screen will distract your players, without giving them any real chance of assimilating the information. This will destroy what might otherwise be a good game due to player fatigue. The very last thing you want!

3.3.2 Proportions

The proportions of the different areas of the screen is also important. You will notice how different each of the drawings of Figure 3.1 look, although they contain essentially the same information, and have very similar play areas. As a general rule, rectangular outlines look better than square ones, although this is partly determined by the contents of each area. Drawing 2 would only really look good in either a vertically scrolling game, or if very large circular objects were involved. Where practical you should try to produce rectangles with height-width ratios as close as possible to 1.6 : 1. Artists will recognise this as an approximation of the golden ratio. I confess that don't know the reason why, but it just so happens that these proportions are generally regarded as most pleasing to the eye.

3.3.3 Novelties

A few games use the entire screen as the playing area, so with these it is only the action that you need to concentrate on while in the play mode. Even so, you should give thought to the layout of score tables and message screens that appear at the end of each level. A simple *Game Over* splashed across the middle of the screen looks very amateurish, but is still surprisingly common.

It's a good idea to consider novelties, like the player's character slowly sliding down the screen, or disintegrating. You could make the background fade out, or if you have some of the PD screen fade and dither routines, use these. Try having a different end to each level.

Score tables in particular, can be very static and boring. You can liven them up considerably with a bit of animation. A few of the game sprites running round the edge is one possibility. Or you can have a sequence where the words and letters are being continuously dragged into place by the goodies, only to be shot up by the baddies.

3.4 Preliminary Testing

Before you develop your game too far you will need to check that the overall idea is really viable. The only practical way for doing this is to write as much of the essential parts as possible, with dummy values and deliberate time delaying calculations, so as to reproduce the real game as near as possible,.

As you don't want to waste a lot of time with this, you should just grab random bits of screen for sprites. It is their size and screen mode that is most important, not their content. Similarly, you can use a procedure or function to read key presses and mouse clicks, without actually doing anything with the results.

In this way you can quickly establish whether you are likely to have a lot of trouble with your game concept. If you find you are getting bogged down trying to track down a host of irritating but seemingly minor bugs, you shouldn't be afraid to scrap the program completely. But retain the game idea, take a deep breath and try again with different data structures, screen layouts and control procedures. A thought to keep in mind is that, if it's getting complicated, you're probably doing it wrong. Unfortunately you may find your game isn't viable at all. I'm sure you'll agree that it's better to find this out sooner rather than later.

3.5 Time and Memory

Although the Archimedes is fast, there is still a real need to keep time consuming operations as compact as possible. This usually means some sacrifice in readability. Without getting too cryptic you should reduce variable and procedure names to the minimum readable.

3.5.1 Improving loops

Oddly enough, you can actually make significant speed improvements by increasing the number of variables. The two rather contrived FOR-NEXT loops below do exactly the same job, but the second version works up to 20% faster. This is because there are fewer calculations inside the loop. The more passes there are through the loop, and the more complex the calculations, the greater the difference becomes. For the loops shown, assume that *a%, b%, c%, d%, e%* and *num%()* have been defined in some other part of the program.

```
FOR I%=a%*b% TO c%*d% STEP a%*b%
   e%+=num%(c%*d%-a%*b%)
  NEXT
  X%=a%*b%
  Y%=c%*d%
  Z%=Y%-X%
  FOR I%=X% TO Y% STEP X%
    e%+=num%(Z%)
  NEXT
```

Where you have a choice of loop constructions, write a small program to time these loops with various actions inside them. Make sure you've balanced the most frequently repeated actions against the most complex ones when you do your timings.

3.5.2 Subroutines

Generally procedures operate faster than functions, as the latter always return a result. Also, passing parameters in procedures and functions although desirable for clear, bug free programming, is highly time consuming. This is particularly true when using the RETURN keyword in procedures so that they can pass values back. Therefore, particularly inside fast repeating loops, it is better tô plan your data structures so that you use the minimum of parameter passing, and as few local variables as possible. A sensible range of global variables will prove to be far more efficient. It is most important though that you document these properly so that you don't try to use the same variable twice.

It is very rare these days for GOTO or GOSUB to be practical. If the program is of any size, and the lines identified are well into the program, then these commands will operate relatively slowly. There is only a speed advantage if the lines are very close to the beginning of the program, as few lines will need to be stepped through.

3.5.3 Faster printing

Experience shows that it is often worthwhile experimenting with small sections of a program as a matter of course, to see how the efficiency can be tightened up. The results can be quite surprising. Of the two lines below, it is the second that runs faster - about twice as fast actually.

```
  IF A%=4 THEN PRINT TAB(0,5)"Done"
  IF A%=4 PRINT TAB(0,5)"Done";
```

There are two reasons for this. The first is that, in the second line, the Basic interpreter will assume the keyword THEN, faster than it can actually decode it in the first line. The second factor, which is even more significant, is that the semicolon suppresses the newline normally produced by PRINT. This newline requires the sending of two characters through the VDU drivers, so the shorter the print string, the more significant this becomes. Where *A%* has any value other than 4 there is much less difference in speed.

If you have a commonly repeated group of colour change, tab and string printing, combine the commands in a single print string using the direct VDU equivalents for the commands. This should be done in your initialisation, then when wanted the string is printed with a single short statement. The two extremes are shown below.

```
VDU4
COLOUR5
PRINT TAB(15,0)"Score = ";
VDU5

a$=CHR$4+CHR$17+CHR$5+CHR$31+CHR$15+CHR$0+"Score = "+CHR$5
PRINTa$;
```

3.5.4 Arithmetic variations

Mathematical calculations can produce quite a few surprises. Of the two lines below, the second will execute 20% to 30% faster while producing exactly the same result. This is because it only has to use a simply multiply instruction, while the first line has to be handled by a complex power calculating algorithm.

```
A%=B%^2
A%=B%*B%
```

3.5.5 Decision ordering

Try to arrange IF-THEN-ELSE constructions so the first choice is the most frequently selected and, where practical, use the single line version rather than the block structured one that spreads over several lines. Also where you have rarely realised IF conditions with ANDs it is much better to use the combination of stacked IFs. The two arrangements are shown below.

```
IF seldom% AND sometimes% AND often% PROCsomething
IF seldom% : IF sometimes% : IF often% PROCsomething
```

With the latter example, if *seldom%* is FALSE, the rest of the line won't be evaluated at all. This can make a dramatic improvement in speed.

Where memory is more important than speed, such as for strings of instructions, use a simple loop that reads lines of data. On the other hand, if speed is more important, then, in your initialising, read the data into an array. Each item will now be instantly accessible by its array index. You can sometimes get the best of both worlds by storing the data on file and reading it in at the start of the game, re-reading it on any restarts if necessary.

3.5.6 Data

Where you are reading data, place the data immediately after the routine that reads it and use RESTORE+n, where n is the number of lines between the RESTORE line and the first line of data to be read. In a large program this is very much faster than restoring to an absolute line number. You should also put the data immediately after the ENDPROC of the reading procedure, not before it. In this way no processor time is lost stepping through data lines.

3.5.7 Look-up tables

A common method of increasing speed at the cost of memory is the use of look-up tables. This is particularly useful for things like vector calculations. For example, you can easily build up a table of sine values, keeping in mind the fact that, in most cases you only need to plot to an accuracy of four graphic units. Taking values from this table will be dramatically faster than using any other method. Don't forget that you only need to cover the first quadrant, and that the sine table, looked at from the other end, is of course a cosine table.

Almost any series of calculations can be put into look-up tables. This could be screen addresses, sprite locations and sizes, bounce directions, and even, in a simulator, equivalent prices for goods exchange. The list is limited only by your imagination, and the memory available. The deciding factors are, whether the data can be ordered and indexed, and whether there will be sufficient speed improvement to justify the memory usage.

3.5.8 Screen handling

Having high resolution 256 colours graphic modes, is inclined to tempt you to use them regardless. There is a considerable processor time overhead with these modes, as well as the problem of swallowing up large chunks of valuable memory. You should work out just how many colours you really need, remembering that in say, Mode 12, all 16 colours can be redefined in fractional amounts of red, green and blue. Also, a game often looks better if the screen pixels are square. The higher resolution modes have very squashed pixels.

Changing screen modes can produce other useful benefits. When working in lower screen modes, sprites take up less memory and execute faster. Sprite plotting can often present speed problems so two other ideas worth considering are reducing the number of sprites plotted at any time, and reducing the size of sprites, as smaller sprites can be plotted faster.

If you are using a lot of saved screens, you will almost certainly have to consider using screen compression techniques. Simple run-length encoding is often adequate, particularly on cartoon style graphics, and may give you three to four times the number of screens on the disk. The screen loading is also likely to be considerably faster too. On the other hand, there are a number of quite different screen compressors in PD libraries. It may pay you to try out several to see which is most efficient for your needs. You may even be able to benefit by combining two different techniques.

3.5.9 A last twiddle

A neat way of making quite a significant speed increase is with a simple call to the memory controller as below. This will make up to a 20% speed improvement in Basic programs, although it seems to have little effect with an ARM 3 processor, which is already several times faster anyway.

```
SYS "OS_UpdateMEMC",64,64
```

I strongly advise you to use the corresponding call to restore things to normal afterwards, as playing with the operating system like this can do strange things to applications or modules that don't expect it. For the same reason it it as well not to use it in multi-tasking games. The restoring call is:

```
SYS "OS_UpdateMEMC",0,64
```

3.6 Alternative Strategies

Unfortunately there are times when your game idea simply can't fit into the limitations imposed by the hardware. By far the commonest problem is, again, that of speed. Apart from abandoning the game altogether, the only action you can take is to look at different solutions to the same problem. For example, in the chapter on graphics there are two main methods of collision detection. If you read through, you will see that each method has its strengths and weaknesses. You may find that your first choice was the wrong one, in which case a change to the other might solve your problems.

3.6.1 Flags

One way that you can often make improvements is by keeping flags to identify changes in data that may require re-calculations to be performed. This is particularly relevant with three dimensional drawn objects. If only one object is moving, but you re-calculate all points of all objects before plotting, you will waste a great deal of time performing complex and unnecessary trigonometrical calculations. If, on the other hand, you keep a set of flags for each object identifying any changes, only the objects that have moved or rotated need to be adjusted. In real terms this will mean that for the same running speed you may be able to double or triple the number of drawn objects in a scene.

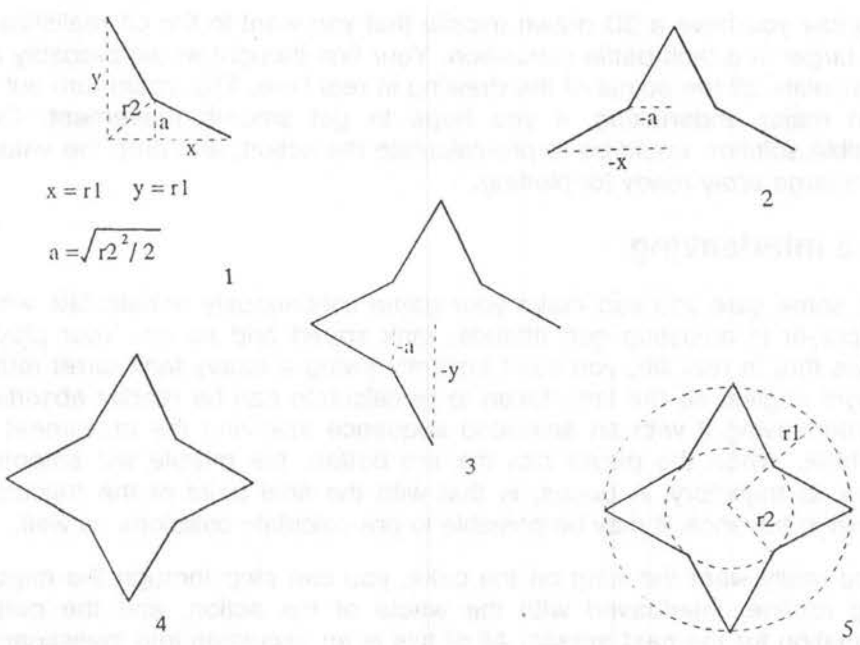

Figure 3.2: Building a star

3.6.2 Simplifying

Any change that simplifies calculations is beneficial, particularly if a complex expression can be replaced with an algorithm that uses simple addition and subtraction. A trivial example of this is shown in Figure 3.2. The obvious first choice for drawing a star is to calculate on the basis of two concentric circles, as in the first drawing. However, in this special case of a four pointed star, it is much simpler to use the steps shown in drawings 1 to 4. Only one point has to be calculated, and even that is a simple piece of Pythagoras. From then on, every new point can be found simply by reversing the sign of x, y or a.

There are many other shapes that can be synthesised without the need for time consuming trigonometrical calculations. However, lateral thinking can be applied like this to many other areas to give you a new, possibly better approach. Hit-and-go situations are particularly amenable to alternative treatments.

3.6.3 Advance calculations

Let's say you have a 3D drawn missile that you want to fire off realistically at a target in a tank battle simulation. Your first thought would probably be to calculate all the points of the drawing in real time. This could turn out to be a major undertaking, if you hope to get smooth movement. One possible solution would be to pre-calculate the action, and drop the values into a large array ready for plotting.

3.6.3 Interleaving

With some care you can make your game continuously re-calculate while the player is adjusting gun attitude, tank speed and so on. Your player knows that in real life, you can't instantly swing a heavy tank turret round at right angles, so the time taken to re-calculate can be readily absorbed by interleaving it with an animated sequence involving the movement of the tank. When the player hits the fire button, the missile will smoothly follow its trajectory. A bonus, is that with the final point of the trajectory known in advance, it may be possible to pre-calculate collisions as well.

If you really want the icing on the cake, you can step through the missile firing routine, interleaved with the whole of the action, and the partial calculation for the next missile. All of this is an excursion into investigating processor redundancy. This can often be put to good use.

3.7 Time Sharing

The greatest waste of time in any game is when the computer is waiting for player input. Often this is done with a simple G%=GET or the like. The computer will wait forever, doing nothing until a key is pressed. This is a waste, as your program could be building up the next level on a shadow screen, running a simulator of some kind in real time, or possibly just animating a few beasties round the screen. One way of doing this is shown below:

```
mark%=0
REPEAT
   G%=INKEY 1
   MOUSE X%,Y%,B%
   IF mark%<count% PROCbuild
   PROCanimate
UNTIL G%=32 OR B%>0:REM spacebar or mouse button
REM print acceptance message
IF mark%<count% THEN
   FOR I%=mark% TO count%
   PROCbuild
NEXT
ENDIF
REM rest of program
END
DEF PROCbuild
   REM build one object
   mark%+=1
ENDPROC
DEF PROCanimate
   REM move one beastie
ENDPROC
```

You will have to ensure that any routines you use for building your next screen can work in very short time bursts, in the same way that WIMP programs do, otherwise there may be an unacceptable delay in key response. Also, notice that I've included a loop after the input routine that finishes any of the building that wasn't completed in time. Again, this could be interleaved with other activities.

Some other aspects of time sharing and interleaving of animation with object movements are covered later in the book.

4

Static Graphics

As was discussed earlier, most modern games are of a highly graphical nature so, to compete effectively with commercial games, you need to have a clear understanding of what the Archimedes is capable of. In this chapter we will look at most of the general aspects of manipulating the screen, and a variety of different ways in which you can apply the graphic commands available. I strongly recommend that you regularly look at the odd bits and pieces pages that most of the Archimedes and BBC B magazines carry. There are often excellent little routines in them, and although the exact program segment is copyright to the magazine, publication has effectively put the concept itself into the public domain, so your can quite safely use your own special version.

4.1 Drawn Pictures

It's probable that you have already experimented with !Draw on the Apps 1 disk supplied with your machine, and therefore have an idea of the kind of drawing possible and the drawing speed. It is, in fact, possible to use draw files directly if you understand their format, but that is rather beyond the scope of this book. Instead we will restrict ourselves to handling the drawing primitives directly from Basic. The actual commands available are well described in the BBC Basic guide. What isn't so clear though, is the way you can combine the different drawing modes to the best effect.

In the first place you should plan any drawing you intend to do in such a way that you have groups of clearly defined objects, each consisting of a number of lines, circles, or whatever. It is often practical to use absolute coordinates for only the first point of each such object, then relative coordinates for all the parts within that object. If you include a scaling variable, you can then have multiple copies of any drawn object, anywhere on the screen. This is the basis for some of the most impressive cartoon style pictures. Listing 4.1 is just an outline of how this can be done.

Listing 4.1: Drawing program

```
 10 REM > Draw
 20 :
 30 MODE 13
 40 OFF
 50 GCOL %111110+128
 60 CLG
 70 GCOL %011101
 80 RECTANGLE FILL 0,320,1280,320
 90 GCOL %001000
100 RECTANGLE FILL 0,0,1280,320
110 PROCbird(300,900,10)
120 PROCbird(500,850,8)
130 PROCbird(650,800,6)
140 PROCbird(740,780,4)
150 PROCtree(200,600,3)
160 PROCtree(150,570,4)
170 PROCtree(250,570,4)
180 PROCtree(200,500,6)
190 PROCtree(100,420,10)
200 PROCtree(300,420,10)
210 X%=RND(-43)
220 FOR X%=4 TO 1280 STEP 28
230   PROCrock(X%,320+RND(8),RND(3)+1)
240 NEXT
250 FOR X%=4 TO 1280 STEP 58
260   PROCrock(X%,280+RND(32),RND(4)+2)
270 NEXT
280 FOR X%=4 TO 1280 STEP 90
290   PROCrock(X%,200+RND(64),RND(5)+4)
300 NEXT
310 FOR X%=4 TO 1280 STEP 220
320   PROCrock(X%,50+RND(128),RND(10)+16)
330 NEXT
340 PROCtree(1200,0,60)
350 END
360 :
370 DEF PROCbird(x%,y%,s%)
380 GCOL %111111 TINT&C0
```

```
390 IF s%>7 THEN
400   CIRCLE FILL x%,y%,s%
410   MOVE BY s%*3,0
420   ELSE
430   MOVE x%,y%
440 ENDIF
450 MOVE BY s%*4,s%*2
460 PLOT&A1,-s%*8,-s%*2
470 MOVE BY-s%*4,0
480 MOVE BY s%*4,0
490 PLOT&A1,-s%*8,s%*2
500 ENDPROC
510 :
520 DEF PROCtree(x%,y%,s%)
530 GCOL %000110 TINT 0
540 RECTANGLE FILL x%,y%,s%*4,s%*4
550 GCOL %000100
560 MOVE BY -s%*10,0
570 MOVE BY s%*16,0
580 PLOT&51,-s%*8,s%*8
590 GCOL %000100 TINT &40
600 MOVE BY -s%*7,-s%*5
610 MOVE BY s%*14,0
620 PLOT&51,-s%*7,s%*7
630 GCOL %000100 TINT &80
640 MOVE BY -s%*6,-s%*4.5
650 MOVE BY s%*12,0
660 PLOT&51,-s%*6,s%*6
670 GCOL %000100 TINT &C0
680 MOVE BY -s%*5,-s%*4
690 MOVE BY s%*10,0
700 PLOT&51,-s%*5,s%*5
710 ENDPROC
720 :
730 DEF PROCrock(x%,y%,s%)
740 GCOL %101010 TINT &C0
750 MOVE x%,y%
760 MOVE BY s%*4,-s%*2
770 PLOT&71,s%*3,s%
780 GCOL %101010 TINT &40
790 PLOT&51,s%*2,-s%*2
800 GCOL %010101 TINT &C0
810 MOVE BY -s%*7,-s%
820 PLOT&51,s%*2,s%*2
830 GCOL %10101 TINT &40
840 PLOT&71,-s%*4,s%*2
850 ENDPROC
```

As the drawing is being done in a 256 colour mode, colours have been selected using binary notation. This has only been done to make the red, green and blue components easier to see. The form in bits is *bbggrr*, giving you just four levels of the three primary colours. Once you are sure that you have everything as you want it, you can easily convert these figures to ordinary decimal notation if memory or speed is at all critical.

Similarly tints have been shown in hexadecimal for clarity. The only tint values currently valid are 0, &40, &80, &C0. Later versions of Risc Os may allow more steps. A small point here that is easily overlooked, is that tints retain the value they were last set at when you change colours, so never assume any tint level. If you use tints at all, always set the level explicitly at the start of each drawing module.

The plot commands in the drawing modules have been shown in hexadecimal, again for clarity. The first digit is the plot type, such as triangle, circle and so on, and the second digit is the plotting mode, such as absolute, relative and inverse.

The objects I've chosen are rather crude, but with ingenuity you can produce a wide range of drawn screens with very little data required for each. Just put all the objects you want drawn into a case statement and then set up lists of object numbers, positions and sizes, in arrays or lines of data.

You will notice, particularly with the rock drawing, that I have taken pains to make use of the fact that the VDU drivers retain the positions of the previous two points visited by the graphics cursor. If you plan your drawing out on paper first, you can use this technique to reduce considerably the number of MOVE statements needed and so speed up the drawing process.

When building up filled objects that can be constructed in several different ways, remember that the speed of drawing the basic shapes varies quite a lot. In order, fastest to slowest they are:

❑ Rectangle

❑ Triangle;

❑ Parallelogram

❑ Circle

❑ Ellipse.

4.2 Sprites

The drawing methods covered so far are fine for many applications but where real speed is necessary, as in the active part of an arcade game, they are still hopelessly slow. The answer then is to use sprite graphics. Sprites can be taken from pre-defined files, constructed from data, or built up by using drawn graphics and grabbing an area of screen enclosing the drawing. By far the best method is to a load pre-defined sprite file. This can be created initially in !Paint, or from one of the many art packages. Or, as an alternative, you can write a small sprite creation program that sends the sprites to file once they have all been created using the drawing primitives.

There is virtually no limit to the number of user sprites you can define. Obviously you can't handle too many at one time, and the larger the sprites, the more memory they take up and the more slowly they will be handled. So when creating moving scenes, you should be concentrating on giving the impression of a lot of activity, rather than actual activity.

4.2.1 User sprites

Although it may seem easiest to use so-called system sprites, there is little space for them. They generally run more slowly, and you actually have less control over them. With this in mind, and Acorn's own advice, all my examples are with user sprites, where the sprite storage area is in normal Ram, available to the program. User sprites are not only more versatile, but enable you to assess correctly the memory space required without the need for desktop sprite area testing. If you handle them by their memory address rather than their names they are even faster in execution.

4.2.2 Sprite masking

One of the features of Acorn's sprite system is that you have a choice as to whether you want a mask or not. Non-masked sprites are significantly faster than their masked counterparts, but suffer a disadvantage in that they always set the rectangular area enclosing the sprite to the background colour that was effective when the sprite was made. Masked sprites appear to sit on top of whatever the current background is, as the sprite's own background is rendered transparent.

4.2.3 Sprite control

Acorn's sprite handling may seem poor compared with some other systems, in that there are no dedicated routines for animation or collision detection. You will have to build these up yourself. On the other hand you do have considerable independence of screen mode, and the ability to modify sprites quite quickly, pixel by pixel, if need be. You can also plot them in a different size and proportion to that at their creation. Scaled sprites are very much slower though, so you shouldn't use them unless your really need to.

4.2.4 A sprite example

In Listing 4.2 you can see how best to set up sprite handling from scratch. Notice how no assumptions are made and OS calls are used liberally to declare areas and find sizes and addresses. The result is that, colours permitting, the routines will work in any screen mode without any change to the scale or proportions of the sprites. There is quite a lot you'll have to take 'as is' unfortunately, as there simply isn't the space to go into all the details. You'd need to examine the Programmers' Reference Manual if you want to learn just how the whole of the sprite system works.

Listing 4.2: Sprite handling

```
   10 REM > Rocks
   20 :
   30 ON ERROR PROCerror:END
   40 PROCinitialise
   50 PROCsprites
   60 OFF
   70 GCOL %100000+128
   80 REPEAT
   90   CLG
  100   FOR I%=0 TO 20
  110    PROCdisplay(rock%,RND(1100),RND(900),RND(3),RND(3),RND(5),RN
D(5))
  120   NEXT
  130   PRINT TAB(25,11) "Press a key - Return to stop";
  140 UNTIL INKEY500=13
  150 END
  160 :
  170 DEF PROCerror
  180 MODE 12
  190 IF ERR<>17 PRINT REPORT$ " @ ";ERL
  200 ENDPROC
  210 :
  220 DEF PROCinitialise
  230 MODE 15
  240 PRINT TAB(30,11)"Please Wait"
  250 SYS "OS_SWINumberFromString",,"OS_SpriteOp" TO sprite%
  260 DIM block% 19
  270 block%!0=4
```

```
280 block%!4=5
290 block%!8=-1
300 SYS "OS_ReadVduVariables",block%,block%+12
310 xeig%=block%!12
320 yeig%=block%!16
330 size%=&2000
340 DIM area% size%
350 area%!0=size%
360 area%!4=0
370 area%!8=16
380 DIM scale% 15
390 scale%!0=1
400 scale%!4=1
410 scale%!8=1
420 scale%!12=1
430 init%=256+9
440 def%=256+15
450 select%=256+24
460 mask%=512+29
470 getpix%=512+41
480 putpix%=512+44         :REM to put pixels in sprite itself use 512+42
490                        :REM then add ,colour%,tint% to end of call
500 plot%=512+52           :REM for non-scaled use 512+34 - much faster
510                        :REM then omit all reference to scale%
520 writeto%=256+60
530 style%=8               :REM use 0 for non-masked & omit PROCmasksprite
540 SYS sprite%,init%,area%
550 ENDPROC
560 :
570 DEF PROCsprites
580 LOCAL x%,y%
590 x%=202                            :REM repeated
600 y%=112                            :REM for
610 rock%=FNdefsprite("rock",x%,y%)   :REM each  probably better to use
620 PROCrock(x%,y%)                   :REM sprite  an array for multiple
630 PROCmasksprite(rock%,x%,y%)       :REM defined     sprite addresses
640 SYS sprite%,writeto%,area%,0
650 ENDPROC
660 :
670 DEF FNdefsprite(a$,x%,y%)
680 LOCAL add%
690 x%=x%>>xeig%
700 y%=y%>>yeig%
710 SYS sprite%,def%,area%,a$,0,x%,y%,MODE
720 SYS sprite%,writeto%,area%,a$
730 SYS sprite%,select%,area%,a$ TO ,,add%
740 =add%
750 :
760 DEF PROCrock(x%,y%)
770 GCOL %101010 TINT &C0
780 x%=x% DIV 11
790 y%=y% DIV 5
```

```
 800 MOVE x%*2,y%*4
 810 MOVE BY x%*4,-y%*2
 820 PLOT&71,x%*3,y%
 830 GCOL %101010 TINT &40
 840 PLOT&51,x%*2,-y%*2
 850 GCOL %010101 TINT &C0
 860 MOVE BY -x%*7,-y%
 870 PLOT&51,x%*2,y%*2
 880 GCOL %10101 TINT &40
 890 PLOT&71,-x%*4,y%*2
 900 ENDPROC
 910 :
 920 DEF PROCmasksprite(add%,x%,y%)
 930 LOCAL I%,J%,c%
 940 x%=x%>>xeig%
 950 y%=y%>>yeig%
 960 SYS sprite%,mask%,area%,add%
 970 FOR J%=0 TO y%-1
 980   FOR I%=0 TO x%-1
 990     SYS sprite%,getpix%,area%,add%,I%,J% TO,,,,,c%
1000     IF c%=0 SYS sprite%,putpix%,area%,add%,I%,J%
1010   NEXT
1020 NEXT
1030 ENDPROC
1040 :
1050 DEF PROCdisplay(add%,x%,y%,scale%!0,scale%!4,scale%!8,scale%!12)
1060 SYS sprite%,plot%,area%,add%,x%,y%,style%,scale%
1070 ENDPROC
```

The method used to produce the sprite, is to define a blank one to the correct size, then make the VDU drivers draw inside this instead of the screen, and finally restore normal VDU action. This is much neater, albeit marginally more complicated, than grabbing a sprite directly from the screen. Your players would think the game decidedly tatty if they had to watch a procession of objects being drawn while the sprites were being created.

The first OS call in the initialising procedure is just a very useful method of finding the internal number of the OS_SpriteOp call. As a variable, *sprite%*, it then operates much faster than the string form. The next call is used to find the relationship between actual pixels and normal graphic units. This ensures mode independent, consistent and relatively easy positioning of drawn objects within sprites. The last call initialises the area of memory set aside, as a valid sprite area.

In PROCdefsprite, the first call creates an empty sprite, and the second redirects VDU output to it. The last call finds the address of the sprite, so that all subsequent calls can use this instead of the rather slower name string.

In PROCmasksprite the first call sets up a blank mask for the sprite, with all bits set. Next, pairs of calls test each pixel of the sprite, and if it is the background colour, the corresponding mask bit is cleared. You can use a variant of this to actually build up sprites one pixel at a time, instead of drawing to them. This can be very useful for modifying a sprite after it has been created. You may want to do this half-way through a game to show damage to a sprite object that is being fired at.

Finally, after all sprites have been defined, only one in our example, another redirection call is made in PROCsprites to restore VDU writing to the screen. If you had several sprites to define you would repeatedly create empty sprites, redirect output to them, draw in them and create their masks.

Unfortunately Risc Os turns the cursor back on after VDU output has been re-directed, so it is necessary to use a second OFF command after all the sprites have been defined.

The actual sprite drawing is done by the call in PROCdisplay. For scaling the sprite, the multiplication and division factors are dropped directly into the small data block scale% as the procedure is entered.

To see how useful it is to lock all the dimensions and scaling together, you can change the screen mode very easily to Mode 13. All you need to do is change one line - the one that sets the screen mode itself. Everything else will be correctly adjusted, and although the resolution won't be as good, the overall sizes will be identical.

4.3 User Defined Characters

Risc Os still supports the older 8 x 8 grid, user-defined graphics found in almost all 8-bit computers. Although limited in scope they are still useful for building up highly repetitive patterned backgrounds. Listing 4.3 produces a variety of striking backgrounds with very little code, even though this has been expanded to make it as clear as possible.

Listing 4.3: User defined characters

```
10 REM > UDCs
20 :
30 ON ERROR PROCerror:END
40 PROCinitialise
50 :
60 COLOUR %101011+128
70 COLOUR %001111
```

```
  80 FOR I%=1 TO 8
  90   PRINT TAB(1,I%) STRING$(18,CHR$128)
 100 NEXT
 110 :
 120 COLOUR %111111
 130 COLOUR %001100+128
 140 FOR I%=1 TO 8
 150   PRINT TAB(21,I%) STRING$(18,CHR$129)
 160 NEXT
 170 :
 180 COLOUR %000000+128
 190 COLOUR %111111
 200 FOR I%=10 TO 16
 210   PRINT TAB(1,I%) STRING$(18,CHR$132)
 220 NEXT
 230 :
 240 COLOUR %111100
 250 COLOUR %000011+128
 260 FOR I%=10 TO 16
 270   PRINT TAB(21,I%) STRING$(18,CHR$133)
 280 NEXT
 290 :
 300 COLOUR %000010
 310 COLOUR %101111+128
 320 FOR I%=9 TO 11
 330    PRINT TAB(0,I%*2) STRING$(20,CHR$134+CHR$135) STRING$(20,CHR$1
35+CHR$134)
 340 NEXT
 350 :
 360 COLOUR %100000
 370 COLOUR %001011+128
 380 FOR I%=12 TO 14
 390   PRINT TAB(0,I%*2+1) STRING$(20,CHR$136+CHR$137) STRING$(20,CHR
$138+CHR$139);
 400 NEXT
 410 PROCtidy
 420 END
 430 :
 440 :
 450 DEF PROCerror
 460 PROCtidy
 470 PRINT REPORT$ " @ ";ERL;
 480 ENDPROC
 490 :
 500 DEF PROCtidy
 510 COLOUR %111111
 520 COLOUR %000000+128
 530 PRINT TAB(0,29);
 540 ENDPROC
 550 :
 560 DEF PROCinitialise
 570 MODE 13
 580 OFF
 590 VDU 23,128
 600 VDU%01111000
 610 VDU%01110111
 620 VDU%10110111
 630 VDU%11001111
 640 VDU%11001111
 650 VDU%10110111
 660 VDU%01110111
 670 VDU%01111000
 680 :
 690 VDU 23,129
 700 VDU%00000000
 710 VDU%01111000
 720 VDU%00110000
 730 VDU%01111000
 740 VDU%00000000
 750 VDU%11000011
 760 VDU%10000001
 770 VDU%11000011
 780 :
 790 VDU 23,132
 800 VDU%00000001
 810 VDU%00000010
 820 VDU%00000100
 830 VDU%00001000
 840 VDU%00011000
 850 VDU%00100100
 860 VDU%01000010
 870 VDU%10000001
 880 :
 890 VDU 23,133
 900 VDU%01000000
 910 VDU%01000000
 920 VDU%01000000
 930 VDU%01111100
 940 VDU%00000100
 950 VDU%00000100
 960 VDU%00000100
 970 VDU%00000100
 980 :
 990 VDU 23,134
1000 VDU%00000000
1010 VDU%01111111
1020 VDU%01111111
1030 VDU%01111111
1040 VDU%01111111
1050 VDU%01111111
1060 VDU%01111111
1070 VDU%00000000
1080 VDU 23,135
1090 VDU%00000000
```

```
1100 VDU%11111110
1110 VDU%11111110
1120 VDU%11111110
1130 VDU%11111110
1140 VDU%11111110
1150 VDU%11111110
1160 VDU%00000000
1170 :
1180 VDU 23,136
1190 VDU%10000000
1200 VDU%10000000
1210 VDU%10000000
1220 VDU%10000000
1230 VDU%01000000
1240 VDU%00100000
1250 VDU%00011111
1260 VDU%00000010
1270 VDU 23,137
1280 VDU%00001111
1290 VDU%00010000
1300 VDU%00100000
1310 VDU%01000000
1320 VDU%01000000
1330 VDU%01000000
1340 VDU%11000000
1350 VDU%01000000
1360 VDU 23,138
1370 VDU%00000010
1380 VDU%00000011
1390 VDU%00000010
1400 VDU%00000010
1410 VDU%00000010
1420 VDU%00000100
1430 VDU%00001000
1440 VDU%11110000
1450 VDU 23,139
1460 VDU%01000000
1470 VDU%11111000
1480 VDU%00000100
1490 VDU%00000010
1500 VDU%00000001
1510 VDU%00000001
1520 VDU%00000001
1530 VDU%00000001
1540 :
1550 COLOUR %100000+128
1560 CLS
1570 ENDPROC
```

You will see that all the VDU 23 character definitions have been broken down into their binary components. The pattern of 1s in the definitions clearly shows the shape of the characters that have been designed. Once you are happy with the definitions, you can compress them into the more normal single line of parameters separated by comas.

4.4 ECF Patterns

A more flexible approach to producing pattern effects is to use the extended colour fill system. There are four of these pseudo colours, that can be used with their pre-set patterns or re-defined to a pattern of your choice and then used. There are two ways of defining these fills. One, the default, is a backwards compatible method for the BBC Master. However, there is a simpler native mode, which we will use. This is enabled with a simple:

```
VDU 17,4,1|
```

after the screen mode is set. However, in the 256 colour modes you always work in native mode. This is logical, as there is no backward compatibility to be considered in these modes.

A giant ECF pattern can be made by using all four ECF patterns together in similar layout to that of four UDCs. The principal advantage of this system over that of blocks of UDCs is that they are used with the graphic drawing commands so that you can produce complex shapes containing these patterns. They will also draw faster, but you can only have four ordinary, or one giant ECF pattern at a time. UDCs will give you up to 224 single character patterns and, by using a greater number of characters together, much larger multiple character patterns.

Like UDCs, the pattern size is dependent on the screen mode when using the ECF colours. However it is also dependent on the colours in the mode so that where characters will be printed the same size in modes 9 and 13, the ECF pattern will be smaller in the latter. In Listing 4.4 there is an example of a giant ECF pattern used to get a wall effect. We are again splitting up the necessary VDU calls to make the patterning clearer. This isn't quite as straightforward as UDC patterns though, as the relationship of bits to colours is dependent on the screen mode.

Listing 4.4: Creating a wall effect

```
10 REM > Pattern
 20 :
 30 MODE 9
 40 OFF
 50 VDU 23,17,4,1|
 60 VDU 23,2
 70 VDU%01110111
 80 VDU%01110001
 90 VDU%01110001
100 VDU%01110001
110 VDU%01110111
120 VDU%00010001
130 VDU%00010001
140 VDU%00010001
150 :
160 VDU 23,3
170 VDU%01110111
180 VDU%00010001
190 VDU%00010001
200 VDU%00010001
210 VDU%01110111
220 VDU%00010001
230 VDU%00010001
240 VDU%00010001
250 :
260 VDU 23,4
270 VDU%01110111
280 VDU%00010001
290 VDU%00010001
300 VDU%00010001
310 VDU%01110111
320 VDU%01110001
330 VDU%01110001
340 VDU%01110001
350 :
360 VDU 23,5
370 VDU%01110111
380 VDU%00010001
390 VDU%00010001
400 VDU%00010001
410 VDU%01110111
420 VDU%00010001
430 VDU%00010001
440 VDU%00010001
450 :
460 GCOL 80,1
470 RECTANGLE FILL 320,256,640,512
480 CIRCLE FILL 128,128,64
490 MOVE 1000,800
500 MOVE 1180,920
510 PLOT &55,1200,720
520 END
```

4.5 Backgrounds

If you look closely at any platform games or graphic adventures you will notice that they actually have highly repetitive backgrounds, in spite of all the apparent detail. These backgrounds are surprisingly easy to create. All you need to do is to think of the play area as a grid of squares. If you choose a size of, say, 64 graphic units square the whole screen can be regarded as having 20 columns and 16 rows of cells.

What you can now do is define a set of sprites that will exactly fit the cells. These can be sections of wall, doors, arches, trees or whatever takes your fancy. You will have to make sure that all the elements lock together like a jigsaw. Walls in particular must have brick or stonework that can be matched on all four sides, and you will need to have coping sections and end walls. This is outlined in Figure 4.1. Notice how I've used more than one interlocking top middle section, to further break up any obvious line structures. You should use this duplication with most of the other sections, particularly large areas of the middle sections, where I recommend three or four interchangeable sprites. Your game players will be more impressed if the construction method is well hidden.

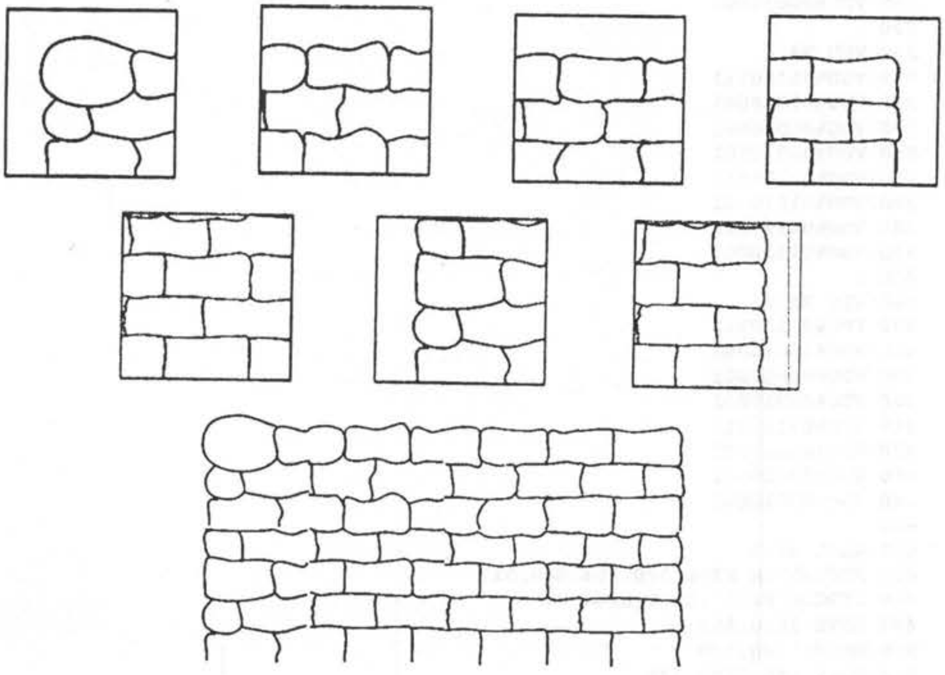

Figure 4.1: Building a wall from sprites

All your screens can now be defined as lists of sprite names - or addresses if you employ user sprites to the full. With as few as 30 sprites, you can create enormous maps of seemingly totally different backgrounds. It is even possible to arrange for cells to be replaced with others as the game progresses. This is useful for opening doors, collecting objects and the like.

As these sprites are for background construction it makes sense to use fixed size, non-masked sprites, for optimum speed. Added to this, there is the possibility of replacing only the background cells that have any moving sprites on top of them, instead of re-drawing the whole screen. If there is very little movement, this can make massive improvements to your overall efficiency.

While on the subject of backgrounds, it is worth mentioning a common technique for increasing the number of apparent colours, as well as giving them a textured appearance. This is known as dithering and involves a number of colours being used either in a fixed pattern, or randomly distributed over an area. Both of these arrangements are shown in Listing 4.5

Listing 4.5: Dither

```
 10 REM > Dither
 20 :
 30 MODE 9
 40 OFF
 50 left%=128
 60 width%=960
 70 bottom%=512
 80 height%=320
 90 GCOL 2
100 RECTANGLE FILL left%,bottom%,width%,height%
110 FOR J%=bottom% TO bottom%+height% STEP 12
120   FOR I%=left% TO left%+width% STEP 12
130     GCOL 3
140     POINT I%+4,J%+4
150     POINT I%,J%
160     GCOL 7
170     POINT I%,J%+4
180     GCOL 1
190     POINT I%+4,J%
200   NEXT
210 NEXT
220 :
230 bottom%=128
```

```
240 GCOL 2
250 RECTANGLE FILL left%,bottom%,width%,height%
260 FOR I%=0 TO 5000
270   R%=RND(3)
280   CASE R% OF
290     WHEN 1:GCOL 1
300     WHEN 2:GCOL 3
310     WHEN 3:GCOL 7
320   ENDCASE
330   POINT left%+RND(width%),bottom%+RND(height%)
340 NEXT
350 ON
360 END
```

4.6 Banked Screens

The screen layout of the Archimedes is such that an area of memory, usually of a size set by the user from the desktop, is reserved for the display. Risc Os allows as many screens as can be fitted into this area to be used. The visible screen is simply a window on this area. Such an arrangement lets you have several different screens set up and available for instant display.

Listing 4.6 shows an outline of how this can be done for MODE 0. Normally I wouldn't expect anyone seriously to use this mode for games, but it is easier to demonstrate the effect when you can be sure of several banks free.

Listing 4.6: Banked screens

```
 10 REM > Banked screens
 20 :
 30 ON ERROR PROCerror:END
 40 PROCinitialise
 50 PROCsetscreens
 60 :
 70 REPEAT
 80   bank%=GET-48
 90   IF bank%>0 AND bank%<max%+1 SYS "OS_Byte",113,bank%
100 UNTIL bank%=0
110 :
120 PROCtidy
130 END
140 :
150 DEF PROCerror
160 PROCtidy
170 PRINT REPORT$ " @ ";ERL
180 ENDPROC
190 :
  200 DEF PROCinitialise
  210 MODE 8
  220 MODE 0
  230 OFF
  240 DIM block% 12
  250 block%!0=150
  260 block%!4=-1
  270 SYS"OS_ReadVduVariables",block%,block%+8
  280 max%=block%!8 DIV &5000
  290 IF max%>9 max%=9
  300 ENDPROC
  310 :
  320 DEF PROCsetscreens
  330 RESTORE+14
  340 FOR I%=1 TO max%
  350   SYS "OS_Byte",112,I%
  360   COLOUR I%+128
  370   COLOUR I%+1
  380   CLS
  390   PRINT TAB(25,7) "Banked Screens Example"
  400   PRINT TAB(20,13) "Press numbers 1 - ";max% " to select bank"
  410   PRINT TAB(20,15) "Use 0 to finish"
  420   PRINT TAB(20,17) "This is screen bank ";I%
  430   READ text$
  440   PRINT"SPC 5 text$
  450 NEXT
  460 ENDPROC
  470 DATA This is rubbish,This is also rubbish,How much rubbish do yo
u want to read?,Just to prove it's a different screen bank,Yet more no
nsense to read,The sixth lot of garbage,Yep! It's me again,Boring isn'
t it?,The last lot!
  480 :
  490 DEF PROCtidy
  500 SYS "OS_Byte",112,1
  510 SYS "OS_Byte",113,1
  520 CLS
  530 ENDPROC
```

The SYS call in PROCinitialise is used to establish just how much screen memory is available. You should always do this rather than just assume that people will have sufficient screen memory configured. If the figure returned is too small then your game should tell the player, and advise on how to get more memory allocated to the screen.

The active commands are the OS_Byte calls 112 and 113. OS_Byte 112 determines which screen is to be written to by the VDU drivers, while OS_Byte 113 sets the screen bank to be displayed. The idea of writing to an invisible screen, while displaying another, can be a little confusing at first, but once familiar with the concept, you will see that it gives you a great deal of scope for avoiding unnecessary re-draws.

Multiple screen banks are also put to good effect in moving displays. In most cases, dual screen operation is the most practical. The idea here is that you draw on one screen while displaying the other. Once the re-draw is complete you swap screens and repeat the process for the other screen. In this way none of the actual drawing process is visible no matter how long it takes. This results in a dramatic reduction in flicker. The switchover is instantaneous, but you need a WAIT delay to ensure it takes place at the beginning of a display cycle, otherwise you could end up briefly displaying half of each screen. Most of the later examples use this technique.

4.7 Clearing Screen Areas

It isn't often realised that there are three quite different methods of clearing an area of screen, all of which have their strengths and weaknesses. If you want the entire screen cleared, by far the most efficient method is by using CLS. This will clear to the current text background colour, as set with COLOUR n+128, taking only about two thirds of the time of a CLG command. However, if you want a patterned ECF colour you can't use CLS but will have to use CLG. This will clear to the current graphics background colour, defined with GCOL n+128. Surprisingly, the RECTANGLE FILL command works slightly faster than CLG, and this will clear to the graphics foreground.

You will see from this that you can have three different colours set as clearing colours, and provided you aren't too concerned about speed, you can just pre-set your colours and select the one you want. However, there's far more to it than that.

Both text and graphic areas can have what are known as viewports defined. For a text viewport, this is calculated in character printing rows and columns, whereas a graphic viewport will be normal graphic units. CLS and CLG will then operate only within the designated viewports. If the area you wish to clear can be defined exactly in text rows and columns, CLS is still very much faster than the other two methods, and becomes a useful fast rectangle fill.

4.8 Smartening Up

All the examples so far have been rather bland flat-looking drawings, where most games these days use a variety of tricks to give depth to an otherwise ordinary display. A number of special effects have been put together in Listing 4.7 to give you some idea of what can be achieved with some quite simple ideas.

Listing 4.7: Some special effects

```
   10 REM > Frills
   20 :
   30 MODE 13
   40 VDU 5
   50 GCOL %100000+128
   60 CLG
   70 PROCroll(0,0,1280,1024,2,1,1,FALSE)
   80 PROCroll(128,640,1024,256,2,2,2,TRUE)
   90 PROCroll(896,160,256,416,0,2,0,TRUE)
  100 :
  110 PROCplinth(160,256,256,128,16,2,2,2,FALSE)
  120 PROCplinth(512,128,320,480,16,0,2,2,FALSE)
  130 PROCplinth(544,448,256,128,8,2,1,2,TRUE)
  140 :
  150 FOR I%=64 TO 1152 STEP 64
  160   PROCstud(I%+32,928,12,1,2,0)
  170   PROCstud(I%+32,96,12,2,2,1)
  180 NEXT
  190 :
  200 GCOL 0 TINT &C0
  210 PROCshadow("This is a test",400,800-16,%11,0,4)
  220 PROCshadow("This is a",140,544,%1111,%10,4)
  230 PROCshadow("test too!",140,496,%1111,%10,4)
  240 PROCshadow("Hello",208,336,%010101,%111111,4)
  250 PROCsize(15,24)
  260 PROCshadow("Boo!",560,556,%1110,0,8)
  270 PROCsize(15,64)
  280 PROCshadow("2",996+8,476-8,%1111,%1,4)
  290 PROCshadow("2",996,476,%11,%1111,4)
  300 :
  310 VDU 4
  320 PRINT TAB(3,24);
  330 END
  340 :
  350 DEF PROCroll(left%,bottom%,width%,height%,red%,green%,blue%,fill
%)
  360 lowcol%=red%+(green%<<2)+(blue%<<4)
  370 red%-=(red%>>0)
  380 green%-=(green%>>0)
  390 blue%-=(blue%>0)
  400 highcol%=red%+(green%<<2)+(blue%<<4)
  410 FOR I%=0 TO 3
  420   GCOL lowcol% TINT I%<<6
  430   PROCbox(I%<<2)
  440 NEXT
```

```
 450 FOR I%=0 TO 3
 460   GCOL highcol% TINT I%<<6
 470   PROCbox((I%<<2)+16)
 480 NEXT
 490 IF fill% THEN
 500   GCOL highcol%
 510   RECTANGLE FILL left%+64,bottom%+64,width%-128,height%-128
 520 ENDIF
 530 ENDPROC
 540 :
 550 DEF PROCbox(i%)
 560 RECTANGLE left%+i%,bottom%+i%,width%-i%*2,height%-i%*2
 570 RECTANGLE left%-i%+60,bottom%-i%+60,width%+i%*2-120,height%+i%*2
-120
 580 ENDPROC
 590 :
 600 DEF PROCplinth(left%,bottom%,width%,height%,edge%,red%,green%,bl
ue%,invert%)
 610 LOCAL right%,top%,lowtint%,hightint%
 620 right%=left%+width%
 630 top%=bottom%+height%
 640 lowcol%=red%+(green%<<2)+(blue%<<4)
 650 red%-=(red%>0)
 660 green%-=(green%>0)
 670 blue%-=(blue%>0)
 680 highcol%=red%+(green%<<2)+(blue%<<4)
 690 hightint%=&C0
 700 IF invert% THEN
 710   SWAP lowcol%,highcol%
 720   SWAP lowtint%,hightint%
 730 ENDIF
 740 GCOL lowcol% TINT lowtint%
 750 MOVE right%,top%
 760 MOVE right%-edge%,top%-edge%
 770 PLOT &55,right%,bottom%
 780 PLOT &55,right%-edge%,bottom%+edge%
 790 PLOT &55,left%,bottom%
 800 PLOT &55,left%+edge%,bottom%+edge%
 810 GCOL highcol% TINT hightint%
 820 PLOT &55,left%,top%
 830 PLOT &55,left%+edge%,top%-edge%
 840 PLOT &55,right%,top%
 850 PLOT &55,right%-edge%,top%-edge%
 860 GCOL lowcol% TINT hightint%
 870 LINE left%,top%,left%+edge%,top%-edge%
 880 LINE right%-edge%,bottom%+edge%,right%,bottom%
 890 RECTANGLE FILL left%+edge%,bottom%+edge%,width%-edge%*2,height%-
edge%*2
 900 ENDPROC
 910 :
 920 DEF PROCstud(x%,y%,s%,red%,green%,blue%)
 930 lowcol%=red%+(green%<<2)+(blue%<<4)
 940 red%-=(red%>0)
 950 green%-=(green%>0)
 960 blue%-=(blue%>0)
 970 highcol%=red%+(green%<<2)+(blue%<<4)
 980 GCOL lowcol% TINT &C0
 990 MOVE x%,y%+s%
1000 MOVE x%+s%,y%
1010 PLOT &55,x%,y%
1020 MOVE x%-s%,y%
1030 PLOT &55,x%,y%-s%
1040 GCOL lowcol% TINT 0
1050 MOVE x%+s%,y%
1060 PLOT &55,x%,y%
1070 GCOL highcol% TINT &C0
1080 MOVE x%-s%,y%
1090 PLOT &55,x%,y%+s%
1100 ENDPROC
1110 :
1120 DEF PROCshadow(t$,x%,y%,col%,shade%,gap%)
1130 GCOL shade%
1140 MOVE x%+gap%,y%-gap%
1150 PRINT t$
1160 GCOL col%
1170 MOVE x%,y%
1180 PRINT t$
1190 ENDPROC
1200 :
1210 DEF PROCsize(x%,y%)
1220 VDU 23,17,7,6,x%;y%|
1230 ENDPROC
```

The first, and probably the oldest idea, is to use a shadow style of printing for any text. This gives the impression that the text is slightly above the background, rather than on it. There is nothing particularly complicated about this. All you do is print the text twice in contrasting colours. The shadow colour is printed first, usually below and to the right of the top colour. When this latter is printed it partly obscures the shadow coloured text. Changing the offset between the two colours can be used to alter the apparent height of the text above the background. PROCshadow is the routine that does the work, and you can see that it is very short.

Another idea that has been around for a while, is to use a plinth effect. In our example, a 256 colour mode has been used which allows us to set up the colours for the various parts of the plinth automatically. The parameters for colours that are passed determine the amount of red, green and blue components present. Two shades and a tint change are calculated from this information, so only values 0,1 and 2 can be accepted as colour levels. Although this gives you a rather restricted range of colours, it is easy to implement. If you want greater control, you can change the procedure header so that you pass in the actual colours that you want for all sections of the plinth. If you wanted to use plinths in the 16 colour modes then you would have to do this in any case.

Although there looks to be rather a lot of code in PROCplinth, the construction method is in fact quite straightforward. If you look at Figure 4.2 you will see how the structure is broken up into a single rectangle, two lines and eight triangles - two to each side. Comparing this with the program listing itself you will see how I've managed to pack the triangles in such a way as to keep the number of plotting points to the absolute minimum. The points visited while the triangles are being drawn are numbered 1 to 10. Only the first two are simple move commands, all the rest are PLOT &55. After point 6 has been visited the colour is changed. Colour changes have no effect on plotting positions. It is easy to forget this, and to try to make each shape self-contained, which simply wastes time.

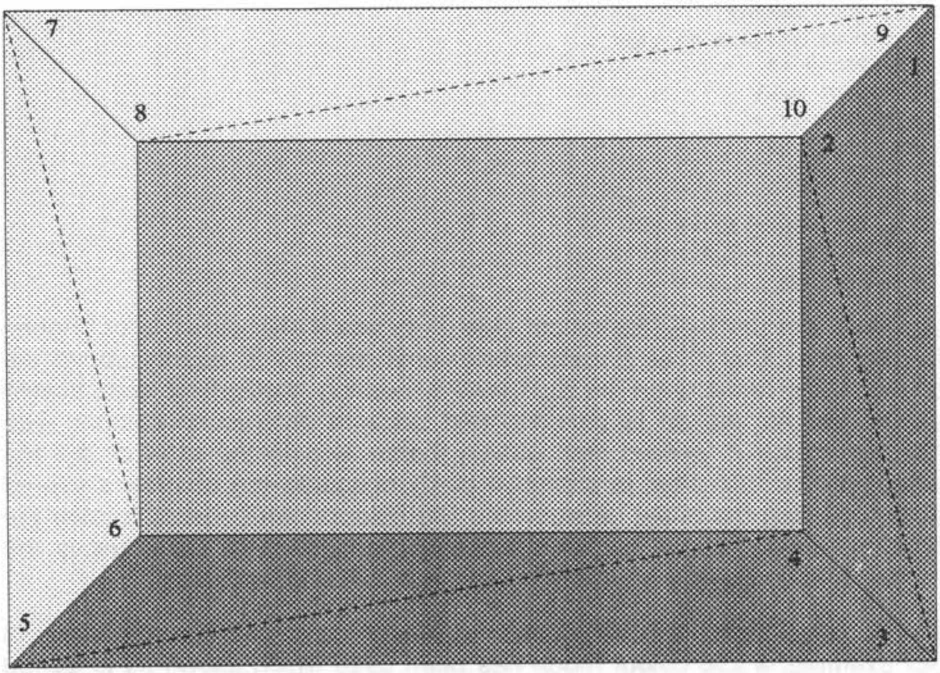

Figure 4.2: Plinth construction

An extension of using plinths is to use rolled edges instead of chamfer edges. This is actually much easier to implement than plinths are. PROCroll uses a similar colour interpolation to PROCplinth, but in this case two shading levels are used and all four tint levels are used to get a smooth change of hue from the darkest part at the edges to the highlight in the middle of the edging. Unlike plinths, there is no control over the width of the edging. This could be done but would require the added complication of calculating just how many shades and tints to use, in order to maintain the smooth shading effect, considerably detracting from the simplicity of the procedure.

Instead of using triangles, the routine just draws pairs of squares on common axis. The largest and smallest are drawn first in the darkest colour, then the tint lightened and another pair drawn, touching the first pair inside the largest and outside the smallest. This is repeated for all for tints, then the whole process repeated for the lighter shade. The spacing of the squares has been set to exactly the pixel width, for optimum speed. If you want to use higher resolution modes you will need to plot the rectangles closer together, and plot more of them. This slows things down, and in modes with rectangular rather than square pixels the effect isn't so good.

PROCstud, uses simple triangle drawing to produce an attractive nail stud effect. Only four triangles are needed, and again, the colours are controlled by separate RGB components.

A useful feature of Risc Os is the ability to scale the size of the system font when printing to the graphic cursor. This is handled by PROCsize, and as you can see has a single VDU call. This scaling technique allows you to change the size of the text to fit the space available, rather than having to squeeze other things up to make space.

4.9 Fancy Fonts

Although useful in its own right, the real relevance of scaling the system font is in conjunction with the font manager. If you set up a game to make use of the font manager, and the font you want is not available, then instead of disrupting the player with a request for fonts that he or she may not have, all you need to do is swap to using the system font. You then scale this to the same proportions that you would have used with the font manager. Listing 4.7 is a short program that does just this. The program works in any 16 or 256 colour mode, maintaining accurate scaling, but due to the way it is designed, only one colour at a time can be used in the 16 colour modes.

Listing 4.8: Scaling fonts

```
  10 REM > Fonts
  20 :
  30 PROCinitialise
  40 PROCcolour(15,12,8,0,0,0)
  50 PROCwrite("A rather good test",128,512)
  60 PROCcolour(8,15,15,0,0,0)
  70 PROCwrite("And another colour",128,384)
  80 PROCtidy
  90 END
 100 :
 110 DEF PROCinitialise
 120 MODE 13
 130 width%=28
 140 height%=44
 150 font$="Trinity.Medium.Italic"
 160 char%=width%*2
 170 yoffset%=height%*1.5
 180 hmatch=height%/1.8
 190 SYS "Font_CacheAddr" TO version%
 200 IF version%<200 font%=0 ELSE PROCcallfont
 210 IF font%=0 PROCsysfont
 220 ENDPROC
 230 :
 240 DEF PROCcallfont
 250 SYS "Font_ReadScaleFactor" TO ,fontx%,fonty%
 260 SYS "Font_ReadFontMax" TO f0%,f1%,f2%,f3%,f4%,f5%
 270 SYS "Font_SetFontMax",f0%,f1%,&8700,f3%,f4%,f5%
 280 DIM buffer% 40
 290 more%=0
 300 REPEAT
 310   SYS "Font_ListFonts",,buffer%,more%,-1 TO ,,more%
 320 UNTIL $buffer%=font$ OR more%=-1
 330 IF $buffer%=font$ THEN
 340   SYS "Font_FindFont",,font$,width%*fontx%>>4,hmatch*fonty% .>4,0
,0 TO font%
 350   ELSE
 360   font%=FALSE
 370 ENDIF
 380 ENDPROC
 390 :
 400 DEF PROCsysfont
 410 VDU 5
 420 DIM block% 19
 430 block%!0=4
 440 block%!4=5
 450 block%!8=-1
 460 SYS "OS_ReadVduVariables",block%,block%+12
 470 xeig%=block%!12
 480 yeig%=block%!16
```

```
  490 VDU 23,17,7,6,width% DIV xeig%;height% DIV yeig%|
  500 ENDPROC
  510 :
  520  DEFPROCcolour(redfore%,greenfore%,bluefore%,redback%,greenback%
,blueback%)
  530 IF font% THEN
  540  SYS"Font_SetPalette",,0,1,14,(blueback%<<28)+(blueback%<<24)+
(greenback%<<20)+(greenback%<<16)+(redback%<<12)+(redback%<<8),(bluefo
re%<<28)+(bluefore%<<24)+(greenfore%<<20)+(greenfore%<<16)+(redfore%<<
12)+(redfore%<<8)
  550   ELSE
  560   GCOL ((redfore%>>2)+((greenfore%>>2)<<2)+((bluefore%>>2)<<4))
  570 ENDIF
  580 ENDPROC
  590 :
  600 DEF PROCwrite(text$,x%,y%)
  610 IF font% THEN
  620   MOVE x%+char%*LEN text$,y%
  630   SYS "Font_Paint",,text$,%10001,x%,y%-yoffset%
  640   ELSE
  650   MOVE x%,y%
  660   PRINT text$
  670 ENDIF
  680 ENDPROC
  690 :
  700 DEF PROCtidy
  710 IF version%>=200 SYS "Font_SetFontMax",f0%,f1%,f2%,f3%,f4%,f5%
  720 IF font% SYS "Font_LoseFont",font% ELSE VDU 4
  730 ENDPROC
```

As usual, no assumptions are made in the program, and all the information needed for selecting and scaling is drawn from Risc Os via SYS calls. The first of these, in PROCinitialise, discovers whether the outline font manager has been installed. If it hasn't then scaled system fonts are used. The older bit-mapped manager returns results in the 100 range.

If the font manager is active, then, in PROCcallfont, the scaling factor is read. The scale factor is 400 by default, but it pays not to assume it hasn't been altered. It represents the relationship between the manager's internal units and the normal graphic units.

Next the fontmax figures are read. The one that really interests us is fontmax 2. This is the font size above which anti-aliasing is no longer applied. The next call actually increases this from the default, to ensure that anti-aliasing is available at all reasonable screen printing sizes.

The following call is inside a loop so that it can try to find the font we want to use, in all those available. If none are available, or the one we want is missing, then as before scaled character printing is used instead. If you like, you can check against two or more fonts that are reasonably similar. All you need to do is have another string test at the exit of the REPEAT-UNTIL loop. Finally, the required font is set using the last call in PROCcallfont.

Although fonts are supposed to be the same overall dimensions when at the same scaling factor, there are significant differences, so you will need to select the font that best matches the screen layout you intend to use. The constants at the beginning of PROCinitialise are used to make fine adjustments so that font proportions accurately match character printing.

If any of the font checks failed, PROCsysfont finds the vertical and horizontal pixel scaling factors, and the VDU command adjusts the proportions of the system font accordingly. Bear in mind, that the system font can be one you have designed yourself, and loaded from a file.

PROCcolour uses a variant of our bit manipulation for the red, green and blue foreground and background colours. Four bit numbers are used here instead of two bit, and the colour control attempts to get an approximate match between outline and system font colours. The background values need to be set to that of the actual background colour on which the font is to be painted. This is so that the anti-aliasing can correctly blend colours. Background colour is irrelevant when using the system font as, with graphic printing being used, the background is transparent.

5

Making it Move

5.1 Objects

You are probably familiar with the idea of using sprites to represent moving characters, but it may not have occurred to you that it is often practical to define all objects as sprites and store them in a sprite file. When they are loaded by your game, they will be instantly available in the form required.

It's often useful to fool your player into thinking there is far more movement in the game than there really is. The player will usually be concentrating mainly on a small area of the screen, and therefore won't really notice how much is going on over the whole screen. This is particularly true of graphic adventures where the player is moving only a single character. If you plan you game so that most of the action is in the vicinity of the player's character, with just the occasional creature or object appearing at the edges, you'll get the desired result.

Another method of increasing the apparent activity is to have groups of, say, four low priority objects that, instead of being moved with every loop of the game, only move on every fourth loop. This is particularly useful with slow moving objects, and provided the general action is reasonably fast, your player won't notice the subterfuge. You have gained by reducing the re-calculating time by four.

5.2 Movement

Making objects appear to move can, in its simplest form, consist of a loop of program statements as in the examples below.

Example 5.1

```
clear screen
draw background
Loop start
   plot objects
   calculate new positions
   check for collisions (including screen edges)
   update loop conditions
   wait for a while
   rub out objects
Loop end
```

Example 5.2

```
Loop start
   clear screen
   draw background
   plot objects
   calculate new positions
   check for collisions
   update loop conditions
   wait for a while
Loop end
```

Which example you use really depends on how many moving objects you expect to have. The former is generally to be advised where there are only one or two objects to be moved, and most of the screen is to remain unaltered. But if there are several objects you should bear in mind that plotting and rubbing out effectively doubles the number of object operations performed. In this case you are better off using the second method. Example 5.3 is a variant of the first method where, instead of rubbing out the object, which can be difficult over a complex background, the background itself is first stored, then the object plotted and later, the background restored. Similarly, Example 5.4 is an improved variant of the second method that would use a small piece of ARM code for whole screen storage and recovery.

Example 5.3

```
clear screen
draw background
Loop start
   store backgrounds where objects will be plotted
   plot objects
   calculate new positions
   check for collisions
   update loop conditions
   wait for a while
   recover backgrounds in reverse order
Loop end
```

Example 5.4

```
clear screen
draw background
store screen
Loop start
   recover screen
   plot objects
   calculate new positions
   check for collisions
   update loop conditions
   wait for a while
Loop end
```

Background recovery in Example 5.3 has to be done in reverse order for two or more moving objects in case they overlap. Where this occurs, the background of the second sprite will include part of the first, so you need to replace each background piece under exactly the same conditions as existed when it was stored, otherwise you'll see a strange progression of sprite debris, whenever two of them overlap.

In order to get really smooth movement it is vital that you keep the overall loop time to a minimum, and in Example 5.2 particularly, the time between the clearing of the screen and the plotting of the last object. So why the deliberate wait? The answer is that all the calculating will take a highly variable amount of time, and this would result in very erratic re-draw time and hence jerky movement. The time delay routine in Example 5.5 is designed to ensure that all the variation in your calculations is absorbed during the display time rather than the re-draw time. It also helps to ensure that the display time is longer than the re-draw time, reducing flicker. Zeroing the timer at the end of the main loop ensures that it is the total loop time that remains constant, not just the time wasting loop.

Example 5.5

```
zero time variable
Loop start
   perform redraws
   perform calculations
   Repeat
      {do nothing}
   Until time variable is greater than constant.
   zero time variable
Loop end
```

One problem you are likely to encounter if you use particularly slow loops, is that of jittering objects. This is not to be confused with flicker, which is mainly due to bad re-draw methods. At its most objectionable, jittering will make objects appear doubled and overlapped. This is caused by displaying objects in the same place on two or more successive screen refreshes - as opposed to two or more program loops - and then displaying them at the next position and holding them there for several screen refreshes, and so on.

Your brain assesses the distance between the two plot locations and therefore expects a steady progression from one to the next. Added to this, your eye's persistence of vision begins to break down when the time is much greater than one screen refresh. This results in the double image effect, and is particularly noticeable where there is supposed to be smooth movement of regular shaped objects, and the distance between movement steps is large. Listing 5.1 demonstrates the problem. The loop is so simple that it can easily be executed in one screen refresh, but if the spacebar is pressed the loop time becomes exactly twice the screen refresh.

Listing 5.1: The double image effect

```
 10 REM > Jitter
 20 :
 30 MODE 9
 40 OFF
 50 REPEAT
 60   FOR I%=0 TO 1280 STEP 8
 70     CLS
 80     PRINT TAB(5,10)"Hold spacebar to see jitter";
 90     PRINT TAB(12,12)"Escape to stop";
100     CIRCLE FILL I%,512,32
110     WAIT
120     IF INKEY-99 I%+=8:WAIT
130   NEXT
140 UNTIL FALSE
150 END
```

The ideal solution then, is to make your game loop work fast enough to be within a single screen refresh and then use the WAIT command to provide both your time delay and screen refresh synchronisation. This isn't always possible however, so you must employ a combination of the following subterfuges.

❑ Keep the movement step size as small as possible

❑ Make the objects move irregularly

❑ Change the shape, size and colours of the objects on successive plots

❑ Keep the objects as small and as irregularly shaped as possible.

In Listing 5.2 there is a fairly straightforward example using the mouse to move a star around the screen. This is based on the third method for movement of a single sprite. A trick is used to make the background storage easy. As only one small sprite is being plotted, and its movement is restricted to the middle of the screen, it is possible to use an inconspicuous corner of the screen as storage using the plot command for rectangle copy.

Listing 5.2: Moving a single sprite

```
 10 REM > Star
 20 :
 30 ON ERROR PROCerror:END
 40 PROCinitialise
 50 PROCsprites
 60 PROCbackground
 70 MOUSE RECTANGLE -scale% DIV 2,scale%*2,1280,1024-scale%*3
 80 REPEAT
 90   MOUSE x%,y%,b%
100   PROCmove
110 UNTIL b%
120 PROCtidy
130 END
140 :
150 :
160 DEF PROCerror
170 MODE 12
180 PRINT REPORT$ " @ ";ERL;
190 ENDPROC
200 :
210 DEF PROCtidy
220 COLOUR %111111
230 COLOUR %000000+128
240 PRINT TAB(0,29);
250 ENDPROC
260 :
270 DEF PROCinitialise
280 *Pointer 1
290 MODE 13
300 OFF
310 PRINT TAB(10,10)"Please Wait"
320 SYS "OS_SWINumberFromString",,"OS_SpriteOp" TO sprite%
330 DIM block% 19
340 block%!0=4
350 block%!4=5
360 block%!8=-1
370 SYS "OS_ReadVduVariables",block%,block%+12
380 xeig%=block%!12
390 yeig%=block%!16
400 size%=&2000
```

```
410 DIM area% size%
420 area%!0=size%
430 area%!4=0
440 area%!8=16
450 init%=256+9
460 def%=256+15
470 select%=256+24
480 mask%=512+29
490 getpix%=512+41
500 putpix%=512+44
510 plot%=512+34
520 writeto%=256+60
530 style%=8
540 SYS sprite%,init%,area%
550 scale%=64
560 VDU 23,132
570 VDU%00000001
580 VDU%00000010
590 VDU%00000100
600 VDU%00001000
610 VDU%00011000
620 VDU%00100100
630 VDU%01000010
640 VDU%10000001
650 VDU 23,133
660 VDU%00000000
670 VDU%01000000
680 VDU%01000000
690 VDU%01000000
700 VDU%01111100
710 VDU%00000100
720 VDU%00000100
730 VDU%00000100
740 RESTORE+9
750 READ numcols%
760 DIM cola%(numcols%)
770 DIM colb%(numcols%)
780 FOR I%=0 TO numcols%
790   READ cola%(I%)
800   READ colb%(I%)
810 NEXT
820 ENDPROC
830 DATA 4
840 DATA %111111,%111111
850 DATA %010101,%101010
860 DATA %000000,%010101
870 DATA %010101,%000000
880 DATA %101010,%010101
890 :
900 DEF PROCsprites
910 LOCAL s%,x%,y%
920 s%=scale% DIV 2
```

```
 930 x%=scale%
 940 y%=scale%
 950 star%=FNdefsprite("star",x%,y%)
 960 PROCstar(s%,s%,s%)
 970 PROCmasksprite(star%,x%,y%)
 980 SYS sprite%,writeto%,area%,0
 990 ENDPROC
1000 :
1010 DEF FNdefsprite(a$,x%,y%)
1020 LOCAL add%
1030 x%=x%>>xeig%
1040 y%=y%>>yeig%
1050 SYS sprite%,def%,area%,a$,0,x%,y%,MODE
1060 SYS sprite%,writeto%,area%,a$
1070 SYS sprite%,select%,area%,a$ TO ,,add%
1080 =add%
1090 :
1100 DEF PROCstar(x%,y%,s%)
1110 LOCAL i,p,q,r,a%,b%,c%,d%,e%,f%,i%,t%,u%
1120 p=PI/2.5
1130 q=p/2
1140 r=p/4+p
1150 t%=s% DIV 2
1160 u%=s%*3
1170 GCOL %100000+128
1180 CLG
1190 FOR i%=0 TO numcols%
1200   i=i%*p+r
1210   a%=x%+COS(i)*s%
1220   b%=y%+SIN(i)*s%
1230   c%=x%+COS(i+q)*t%
1240   d%=y%+SIN(i+q)*t%
1250   e%=x%+COS(i+p)*s%
1260   f%=y%+SIN(i+p)*s%
1270   g%=x%+COS(i+q)*u%
1280   h%=y%+SIN(i+q)*u%
1290   GCOL cola%(i%) TINT &C0
1300   MOVE a%,b%
1310   MOVE x%,y%
1320   PLOT&55,c%,d%
1330   GCOL colb%(i%) TINT &80
1340   PLOT&55,e%,f%
1350 NEXT
1360 ENDPROC
1370 :
1380 DEF PROCmasksprite(add%,x%,y%)
1390 LOCAL I%,J%,c%
1400 x%=x%>>xeig%
1410 y%=y%>>yeig%
1420 SYS sprite%,mask%,area%,add%
1430 FOR J%=0 TO y%-1
1440   FOR I%=0 TO x%-1
```

```
1450      SYS sprite%,getpix%,area%,add%,I%,J% TO,,,,c%
1460      IF c%=%100000 SYS sprite%,putpix%,area%,add%,I%,J%
1470    NEXT
1480 NEXT
1490 ENDPROC
1500 :
1510 DEF PROCbackground
1520 OFF
1530 COLOUR %100000+128
1540 CLS
1550 COLOUR %001100+128
1560 COLOUR %110011
1570 FOR I%=0 TO 14
1580   PRINT TAB(0,I%) STRING$(40,CHR$132)
1590 NEXT
1600 COLOUR %111100
1610 COLOUR %000011+128
1620 FOR I%=15 TO 27
1630   PRINT TAB(0,I%) STRING$(40,CHR$133)
1640 NEXT
1650 COLOUR 128
1660 COLOUR %111111
1670 PRINT TAB(5,10) "Use the mouse to move the star" TAB(5,12) "Obse
rve the bottom left corner" TAB(5,14) "Press a mouse button to stop"
1680 ENDPROC
1690 :
1700 DEF PROCmove
1710 RECTANGLE x%,y%,scale%,scale% TO 0,0
1720 SYS sprite%,plot%,area%,star%,x%,y%,style%
1730 WAIT
1740 RECTANGLE 0,0,scale%,scale% TO x%,y%
1750 ENDPROC
```

Apart from PROCmove, the movement routine itself, there is nothing really new in this program. In this routine the first RECTANGLE command moves a patch of screen to the bottom left corner, and the second, after sprite plotting and a suitable wait period, moves it back again.

5.3 Animation

Animation is often confused with general movement. However, while most games involve considerable movement, it has only been with the advent of high resolution graphics and fast processors that real animation has been practical. This is where characters move arms and legs to give an impression of realistic walking or running. Objects such as cars, have wheels that appear to go round, and monsters can slowly materialise instead of just flashing onto the screen.

5.3.1 Colour changing

If you are using a 16 colour mode, you can get an interesting pseudo animation using the flashing colours. These can be redefined in fractional amounts independently for each of the two colours that make up the flash. There are also two FX commands that allow you to change the flash rates. The best use of this technique is for small detail, such as silly eye movements or flapping ears.

Another pseudo animation technique involves palette swapping. Again this really only suited to 16 colour modes. In modes with fewer colours it isn't practical, and the 256 colour modes are too complicated to be worth manipulating in this way for such a simple effect. In brief, the idea is to break up the movement you wish to animate into, say, eight of the possible positions of the moving object. The completely overlapping sections are then drawn in the object's colour. All the other parts are drawn in different colours, which have been temporarily re-defined to the current background colour using the extended form of the COLOUR keyword. From then on all you need to do is alternately *switch on* each colour in turn by re-defining it to the desired object colour. This will produce the animation.

An example of this, Listing 5.3, also demonstrates the use of flashing colours, giving two forms of seemingly asynchronous animation.

Listing 5.3: Colour switching

```
 10 REM > AnimVDU19
 20 :
 30 MODE 9
 40 OFF
 50 ON ERROR PROCerror:END
 60 PROCinitialise
 70 PROCdraw
 80 :
 90 VDU19,14,18,240,240,192
100 col%=1
110 REPEAT
120   WAIT
130   COLOUR col%,0
140   col%+=1
150   IF col%=colmax% col%=1
160   COLOUR col%,7
170 UNTIL FALSE
180 END
190 :
200 DEF PROCerror
210 MODE 12
220 IF ERR <>17 PRINT REPORT$ " @ ";ERL
230 *FX 9 25
```

```
240 *FX 10 25
250 ENDPROC
260 :
270 DEF PROCinitialise
280 *FX9 100
290 *FX10 12
300 FOR I%=0 TO 14
310   COLOUR I%,0,0,0
320 NEXT
330 COLOUR 15,240,240,240
340 COLOUR 15
350 PRINTTAB(10,0) "Press Escape to stop"
360 num%=11
370 ymax%=64
380 colmax%=14: REM reduce for different effects
390 col%=0
400 DIM x%(num%),y%(num%),dx%(num%),dy%(num%)
410 x%()=640
420 y%()=ymax%
430 IF RND(-1)
440 FOR I%=0 TO num%
450   dx%(I%)=(num%>1)-I%
460   dy%(I%)=RND(5)+(num%-ABS dx%(I%))*2+ymax%
470 NEXT
480 ENDPROC
490 :
500 DEF PROCdraw
510 GCOL 14
520 FOR i=0 TO PI STEP .05
530   POINT 640+RND(180)*COS i,RND(96)*SIN i
540 NEXT
550 REPEAT
560   col%+=1
570   IF col%=colmax% col%=1
580   GCOL col%
590   flag%=0
600   x%()=x%()+dx%()
610   y%()=y%()+dy%()
620   dy%()=dy%()-1
630   FOR I%=0 TO num%
640     POINT x%(I%),y%(I%)>>2
650     IF y%(I%)<=ymax% THEN
660       flag%+=1
670       dy%(I%)=0
680       dx%(I%)=RND(9)-5
690       y%(I%)=RND(ymax%)
700     ENDIF
710   NEXT
720 UNTIL flag%>=num%-1
730 COLOUR 14
740 PRINT TAB(16,31)"YIPPIE!!";
750 ENDPROC
```

The Roman candle effect produced depends on having most of the plotted points invisible for most of the time. You can see what really was plotted more easily if you temporarily change the line that calls PROCdraw to:

```
PROCdraw:VDU20:STOP
```

The more colours that are used, the greater the distance between visible spots, and therefore the better the effect. You can prove this by changing the value of *colmax%* in the initialisation. In many instances it will be possible to restrict the range to say around six colours, and allowing another two for flash effects, you still have eight colours for all your genuine animated sprites.

5.3.2 Animating sprites

As was noted earlier, the Archimedes doesn't have a dedicated system of commands for sprite animation. However considerable flexibility is provided by the sprite commands available, and it is relatively easy to produce film-type animation. As well as giving more realism to moving objects this technique can be used as yet another method of fooling people into thinking there is more action than there really is.

In Figure 5.1 there are two sprite film-type animations. The first consists of a single moving object within a sprite. For clarity, this is just an x with the fine line showing the track that the object will seem to follow. Only nine frames have been used where in reality you'll probably need more, depending on the overall size of the sprite.

If the whole frame is now moved around the screen reasonably slowly while using each sprite in turn, the object will appear to weave erratically. Selecting each sprite in turn is simply a matter of putting all their addresses in an array when they are created, then picking them out of the array as follows:

```
index%=0
REPEAT
   PROCsprite(address%(index%))
   index%+=1
   IF index%=9 index%=0
   REM any other bits
UNTIL end%
```

The second film is a modification of the first. Instead of a single object we have three identical ones, a, b and c, all at different points on the same track. This is rather like getting something for nothing, as we now seem to have more objects but need fewer sprites to make the film. You could, of course, keep the same number of sprites and use a longer, more complicated track.

When using this form of animation, the film sequence would be interleaved with any other similar films and also with any single sprite object movements.

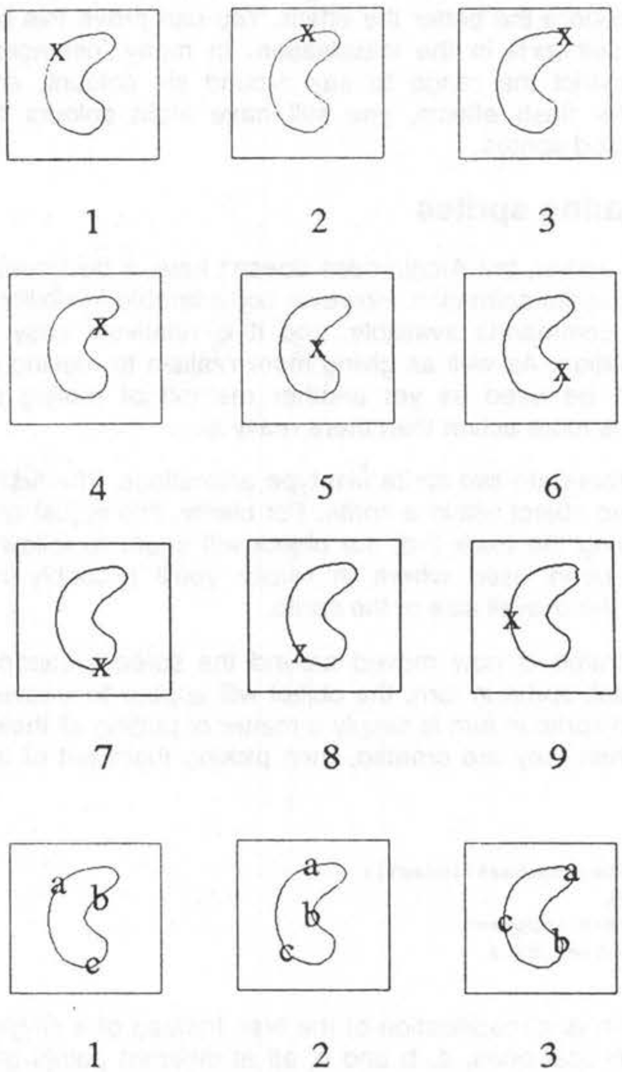

Figure 5.1: Sprite film animation

5.4 Collision Detection

Most games use variants of two basic methods of handling collisions:

❑ By comparing coordinates between objects

❑ By looking at the pixel colours where an object is about to move to.

We will look at these first, then go on to look at other techniques and refinements.

5.4.1 Coordinate collisions

This type of collision is calculated from the X,Y positions and movement vectors of each object. In its simplest form you compare the X and Y coordinates of one object with the X and Y coordinates of the other.

Quite often programmers waste a lot of processing time making redundant collision tests. As there is a need to test every moving object, they therefore loop through testing each object against every other object. If you think about it, you will realise that there is no need to check a stationary object. Things will bump against it, but it will never bump against them. As well as that, you don't need to check against an object that hasn't yet moved. All objects that are going to move will normally do so in the same pass, so an object may actually move out of the way. It would therefore be a cheat to assume a collision.

All that is necessary is to create a set of arrays representing a stack of objects, with the stationary ones at the bottom of the stack. The arrays should contain the object's X,Y coordinates and also their sizes. Starting with the first moving object you compare for collisions with all those lower in the stack. If you are restricted to very small, fast moving objects, you could just compare the coordinates and assume a collision when they are within one pixel size of each other. This is seldom practical though. More usually you need to compare the distance between the objects with the sum of their sizes. This comes down to a simple piece of Pythagoras as shown in Figure 5.2 where the size is defined as the radius of a circle that contains the object.

Collision detection could also include screen edges which should then be mapped as four very large stationary objects well outside the actual screen limits. These objects would be so big that their perimeter is almost a straight line as far as the screen is concerned.

If you study Listing 5.4 you will see how collisions can be handled in practice. There is unfortunately, a rapid deterioration in speed as the number of objects increases, so in the collision example, I've allowed for no stationary objects, and used fairly ordinary screen edge testing. Even so you will see considerable jitter, unless you are using an ARM 3 machine. This is mainly because, for simplicity, filled circle drawing is used rather than sprite plotting.

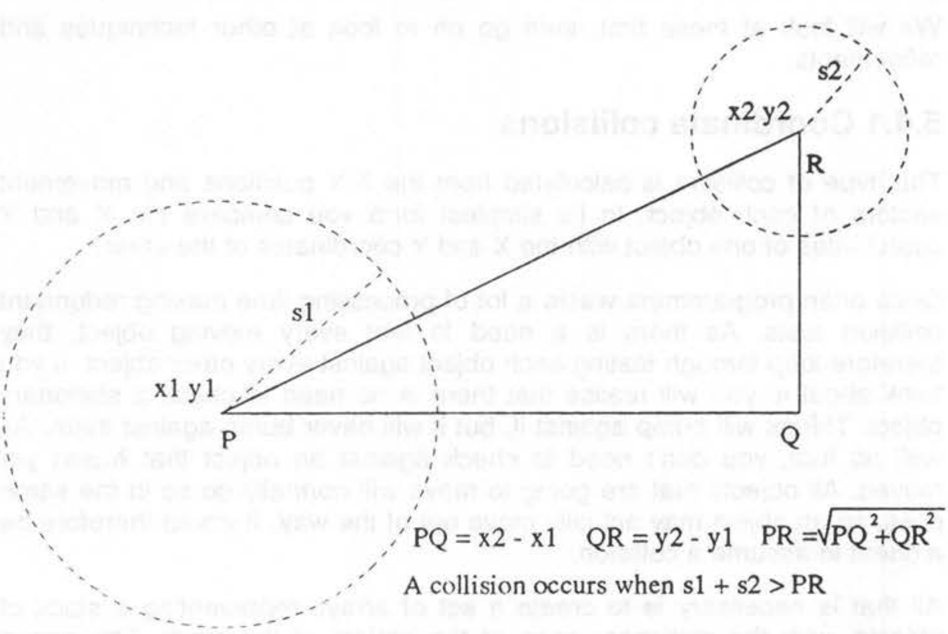
s2
x2 y2
R
s1
x1 y1
P
Q
$PQ = x2 - x1 \quad QR = y2 - y1 \quad PR = \sqrt{PQ^2 + QR^2}$
A collision occurs when $s1 + s2 > PR$

Figure 5.2: Coordinate collisions

Listing 5.4: Handling coordinate collisions

```
   10 REM > CoOrdinate
   20 :
   30 ON ERROR PROCerror:END
   40 PROCinitialise
   50 REPEAT
   60   PROCplot
   70   PROCupdate
   80 UNTIL FALSE
   90 END
  100 :
  110 DEF PROCerror
```

```
120 MODE 12
130 IF ERR<>17 PRINT REPORT$ " @ ";ERL
140 ENDPROC
150 :
160 DEF PROCinitialise
170 MODE 12
180 MODE 9
190 OFF
200 SYS "OS_SWINumberFromString",,"OS_Byte" TO byte%
210 max%=3
220 xmax%=1280
230 ymax%=1024
240 DIM x%(max%)
250 DIM y%(max%)
260 DIM s%(max%)
270 DIM dx%(max%)
280 DIM dy%(max%)
290 FOR I%=0 TO max%
300   s%(I%)=8+RND(10)*4
310   x%(I%)=s%(I%)+RND(xmax% DIV 4-s%(I%)DIV 2)*4
320   y%(I%)=s%(I%)+RND(ymax% DIV 4-s%(I%)DIV 2)*4
330   dx%(I%)=RND(6)*4-12
340   dy%(I%)=RND(6)*4-12
350 NEXT
360 PRINT TAB(9,7) "Co-Ordinate Collisions"
370 PRINT TAB(10,11) "Press Escape to stop"
380 IF INKEY 100
390 sc%=1
400 ENDPROC
410 :
420 DEF PROCplot
430 WAIT
440 SYS byte%,113,sc%
450 sc%=sc% EOR3
460 SYS byte%,112,sc%
470 CLS
480 FOR I%=0 TO max%
490   GCOL I%+1
500   CIRCLE FILL x%(I%),y%(I%),s%(I%)
510 NEXT
520 ENDPROC
530 :
540 DEF PROCupdate
550 FOR J%=1 TO max%
560   FOR I%=0 TO J%-1
570     a%=x%(J%)-x%(I%)
580     b%=y%(J%)-y%(I%)
590     c%=s%(I%)+s%(J%)
600     IF a%*a%+b%*b%<c%*c% PROCbounce
610   NEXT
620 NEXT
630 FOR I%=0 TO max%
```

```
640    IF x%(I%)<s%(I%) OR x%(I%)>xmax%-s%(I%) PROCdx
650    IF y%(I%)<s%(I%) OR y%(I%)>ymax%-s%(I%) PROCdy
660 NEXT
670 x%()=x%()+dx%()
680 y%()=y%()+dy%()
690 ENDPROC
700 :
710 DEF PROCbounce
720 a%=dx%(J%)-dx%(I%)
730 dx%(J%)=dx%(J%)-a%
740 dx%(I%)=dx%(I%)+a%
750 b%=dy%(J%)-dy%(I%)
760 dy%(I%)=dy%(I%)+b%
770 dy%(J%)=dy%(J%)-b%
780 ENDPROC
790 :
800 DEF PROCdx
810 a%=ABS dx%(I%)
820 IF x%(I%)<s%(I%) dx%(I%)=a% ELSE dx%(I%)=-a%
830 ENDPROC
840 :
850 DEF PROCdy
860 b%=ABS dy%(I%)
870 IF y%(I%)<s%(I%) dy%(I%)=b% ELSE dy%(I%)=-b%
880 ENDPROC
```

A small point that might cause confusion is the use of two mode changes. The only function that the first one serves is to ensure that there is enough screen memory available for bank switching. It takes up exactly twice the memory of the wanted mode, so will give a *Bad mode* error message if there isn't enough space for two screen banks in the wanted mode.

If you are using ARM code for collision calculations, considerable speed increase can be made by, instead of taking the square root for the hypotenuse, simply comparing the squared distance with the squared value of the sum of the object sizes. Only simple MUL instructions would be needed in this case, whereas square rooting would require your own routine in ARM code.

If all your objects are about the same size you can cheat by using pre-calculated values for the object sizes that approximate to the squares of the sum of the sizes. You will find that larger objects overlap a little, while smaller ones don't quite meet, but you can actually use that to your advantage, and if the game is moving reasonably fast, your player won't notice anyway.

This method works well for fairly regular objects that can fit reasonably inside a circle, but will fail for objects that are very long and thin, or have long appendages. There are two solutions to this problem. The first is simply to regard the object as a cluster of objects - indeed, it may pay you to plot it as a cluster rather than a single sprite. Alternatively, if the object can be fitted approximately into an ellipse, you can modify the size parameters of each object by a factor dependent on its orientation and collision angle. This is really pushing the method to the limit, so I'll leave that to you to work out if you want to follow it up.

There's nothing more annoying than a game that indicates a collision when you can see that there is clear background between the objects which have supposedly collided. On the other hand, most people will consider that they were clever and got away with it, if there is a slight overlap without a collision being indicated. The rule therefore, is always to give the player the benefit of the doubt. This is most easily done by assuming player objects to be slightly smaller than they really are when calculating for collisions. The exception is where the player is firing at the enemy. In this case, it pays to calculate on the basis that the player's missiles are slightly bigger than they are.

5.4.2 Pixel collisions

Pixel collisions are where screen points under an object are examined to see if they match any know collision colours. Unlike coordinate collisions, speed is unaffected by the number of objects that any single object may collide with, and these collisions can be made more tolerant of the shape of the objects that have collided.

If you are using 256 colour modes, there is a very easy way of arranging intelligent collision detection, by sacrificing a little colour accuracy. Instead of using colour as such, you can use the four tint levels. Where a value 0 is returned regard this as non collision objects in the background. 64 can be solid but benign objects and borders, 128 can be enemy objects and 192 can be player objects or missiles. Listing 5.5 is a rather crude demonstration of the basic idea behind this.

A simple CASE statement is used to determine what action should be taken for any given collision. The TINT command is being used to find the tint of the point that the object would next be visiting rather than the point where it already is. This should prevent any overlapping. However, you will see that I've used Exclusive Or plotting of the moving object, which helps to show the overlaps that still can occur.

Listing 5.5: Pixel collision detection

```
 10 REM > Pixel
 20 :
 30 PROCinitialise
 40 GCOL 3,%111111 TINT &C0
 50 x%=640
 60 y%=512
 70 MOUSE TO x%,y%
 80 CIRCLE FILL x%,y%,s%
 90 REPEAT
100   MOUSE nx%,ny%,b%
110   dx%=SGN(nx%-x%>>2)
120   dy%=SGN(ny%-y%>>2)
130   WAIT
140   CIRCLE FILL x%,y%,s%
150   tint%=TINT(x%+dx%*s%,y%+dy%*s%)
160   CASE tint% OF
170     WHEN 0:PRINT TAB(15,30) SPC 9;
180     WHEN &40:dx%=0:dy%=0:PRINT TAB(17,30) "Boing!";
190     WHEN &80:dx%=0:dy%=0:PRINT TAB(18,30) "Ouch";
200     WHEN &C0:PRINT TAB(15,30) "Hi Friend";
210   ENDCASE
220   x%+=dx%*4
230   y%+=dy%*4
240   CIRCLE FILL x%,y%,s%
250 UNTIL b%>0
260 PRINT TAB(0,28)
270 END
280 :
290 DEF PROCinitialise
300 *Pointer 1
310 MODE 13
320 OFF
330 GCOL%000010 TINT 0
340 MOVE 640,800
350 MOVE 600,760
360 PLOT&55,680,760
370 GCOL%000010 TINT &40
380 RECTANGLE FILL 128,896,1024,128
390 GCOL%001000 TINT &40
400 RECTANGLE FILL 0,0,128,1024
410 RECTANGLE FILL 1152,0,128,1024
420 RECTANGLE FILL 128,0,1024,128
430 GCOL%000010 TINT &80
440 CIRCLE FILL 320,512,32
450 GCOL%000010 TINT &C0
460 CIRCLE FILL 960,512,32
470 GCOL%101010 TINT 0
480 RECTANGLE FILL 600,256,80,80
490 COLOUR%001111 TINT 0
```

```
  500 PRINT TAB(11,10) "Use mouse to move" TAB(9,11) "Press a button t
o stop";
  510 COLOUR%001000+128 TINT &40
  520 s%=16
  530 MOUSE ON
  540 ENDPROC
```

When you run this program you will see that the different types of collision are very positively identified, even though the colours of the obstructions may be very similar.

5.4.3 Cell collisions

You've probably not considered the idea of collisions in board games, but nevertheless a specialised form does take place. It's normal to split the board of, say, a draughts game into an 8 x 8 two dimensional array. As pieces are put on the board the appropriate element in the array is set to indicate that a piece is there, and which player it belongs to. In games like chess, not only is the existence of the piece identified, but its type as well. When testing player moves, the array element corresponding with the destination screen position is checked to see if there is a piece already in that cell - in other words, a collision.

This suggests a third form of collision detection in arcade style games, an extension of coordinate collisions. You keep all sprites to a given size, or a multiple of that size, and split up the screen into a sprite sized grid. You then have to ensure that sprites are always positioned exactly in one of these cells. From then on you can maintain a two dimensional array of the play area and can perform simple quick array tests for collisions.

Probably the best method of ensuring sprite positioning is to set up an animation sequence for all moving objects, where, in the case of the player's object, movement directions are initiated by the player, but the actual movement itself is taken over by the animator for a fixed number of steps. You can include film type animation at the same time, so that the animation sequence gives the illusion of smooth, continuous movement. This kind of movement and collision system is probably best suited to the simpler platform games and graphic adventures, where you know the characters will always be at certain levels and the restricted amount of angular movement is acceptable.

In Listing 5.6 there is a more complete program than usual. This not only demonstrates cell collisions but also brings together a number of points we've looked at.

Listing 5.6: Cell collisions

```
 10 REM > Cells
 20 :
 30 ON ERROR PROCerror:END
 40 PROCinitialise
 50 PROCassemble
 60 PROCsprites
 70 PROCbackground
 80 PROCstart
 90 :
100 E%=1
110 T%=TIME
120 REPEAT
130   IF t%(0)=0 PROChuman
140   FOR I%=1 TO obs%
150     IF t%(I%)=0 AND RND(50)=1 t%(I%)=RND(4)
160   NEXT
170   FOR I%=0 TO obs%
180     IF t%(I%) PROCcheck
190   NEXT
200   REPEAT UNTIL TIME-T%>3
210   WAIT
220   T%=TIME
230   SYS byte%,113,E%
240   E%=E% EOR3
250   SYS byte%,112,E%
260   CALL copy
270   FOR I%=0 TO obs%
280     PROCmove
290   NEXT
300 UNTIL FALSE
310 END
320 :
330 DEF PROCerror
340 MODE 12
350 *FX 15 1
360 IF ERR<>17 PRINT REPORT$" @ ";ERL
370 ENDPROC
380 :
390 DEF PROCinitialise
400 MODE 15
410 MODE 13
420 OFF
430 smax%=3           : REM number of sprites (-1)
440 DIM list%(smax%): REM sprite addresses
450 size%=&3000
460 DIM area% size%
470 SYS "OS_SWINumberFromString",,"OS_Byte" TO byte%
480 SYS "OS_SWINumberFromString",,"OS_SpriteOp" TO sprite%
490 !area%=size%
```

```
 500 area%!4=0
 510 area%!8=16
 520 SYS sprite%,&209,area%
 530 norm%=&222
 540 spec%=&234
 550 DIM scale% 15
 560 scale%!0=1
 570 scale%!4=4
 580 scale%!8=1
 590 scale%!12=1
 600 DIM cells%(9,7)  : REM position cells
 610 obs%=4           : REM number of moving objects
 620 DIM n%(obs%)     : REM film slide number
 630 DIM t%(obs%)     : REM movement direction
 640 DIM x%(obs%)     : REM obvious!
 650 DIM y%(obs%)
 660 ENDPROC
 670 :
 680 DEFPROCassemble
 690 DIM block% &80
 700 block%!0=148
 710 block%!4=7
 720 block%!8=-1
 730 SYS "OS_ReadVduVariables",block%,block%+12
 740 C%=block%!16     : REM screen size
 750 DIM A% C%        : REM stored screen start
 760 B%=block%!12     : REM screen base
 770 D%=A%+C%         : REM stored screen end
 780 memory=0
 790 screen=1
 800 size=2
 810 memoryend=3
 820 bank=4
 830 lowreg=4
 840 highreg=11
 850 link=14
 860 FOR I%=0 TO 2 STEP 2
 870   P%=block%
 880   [ OPT I%
 890   .copy
 900   CMP bank,#2
 910   ADDEQ screen,screen,size
 920   .copyloop
 930   LDMIA (memory)!,{lowreg-highreg}
 940   STMIA (screen)!,{lowreg-highreg}
 950   CMP memory,memoryend
 960   BLT copyloop
 970   MOV PC,link
 980   ;
 990   .store
1000   CMP bank,#2
1010   ADDEQ screen,screen,size
```

```
 1020   .storeloop
 1030   LDMIA (screen)!,{lowreg-highreg}
 1040   STMIA (memory)!,{lowreg-highreg}
 1050   CMP memory,memoryend
 1060   BLT storeloop
 1070   MOV PC,link
 1080   ]
 1090 NEXT
 1100 ENDPROC
 1110 :
 1120 DEF PROCsprites
 1130 J=0
 1140 FOR I%=0 TO 3
 1150   GCOL%001111
 1160   CLS
 1170   J+=PI/32
 1180   FOR I=0 TO PI*2 STEP PI/8
 1190     MOVE (COS(I+J))*64+64,(SIN(I+J))*64+64
 1200     DRAW 64,64
 1210   NEXT
 1220   GCOL%000011
 1230   CIRCLE FILL 64,64,14
 1240   PROCdefsprite(0,0,128,128,I%)
 1250 NEXT
 1260 ENDPROC
 1270 :
 1280 DEF PROCdefsprite(x%,y%,a%,b%,n%)
 1290 SYS "OS_SpriteOp",&110,area%,STR$ n%,,x%,y%,x%+a%,y%+b% TO,,list
%(n%)
 1300 PROCspritemask(list%(n%),area%,sprite%)
 1310 ENDPROC
 1320 :
 1330 DEF PROCspritemask(N%,R%,S%)
 1340 LOCAL A%,B%,F%,J%,I%
 1350 SYS S%,&21D,R%,N%
 1360 SYS S%,&228,R%,N% TO ,,,A%,B%
 1370 FOR J%=0 TO A%-1
 1380   FOR I%=0 TO B%-1
 1390     SYS S%,&229,R%,N%,J%,I% TO,,,,,F%
 1400     IF F%=0 SYS S%,&22C,R%,N%,J%,I%,0
 1410   NEXT
 1420 NEXT
 1430 ENDPROC
 1440 :
 1450 DEF PROCbackground
 1460 CLS
 1470 GCOL%111111
 1480 FOR I%=128 TO 1023 STEP 128
 1490   LINE 0,I%,1279,I%
 1500 NEXT
 1510 FOR I%=128 TO 1279 STEP 128
 1520   LINE I%,0,I%,1023
```

```
1530 NEXT
1540 FOR I%=0 TO 7
1550   cells%(0,I%)=1
1560   cells%(9,I%)=1
1570   SYS sprite%,norm%,area%,list%(0),0,I%<<7,8
1580   SYS sprite%,norm%,area%,list%(0),1152,I%<<7,8
1590 NEXT
1600 FOR I%=1 TO 8
1610   cells%(I%,0)=1
1620   cells%(I%,7)=1
1630   SYS sprite%,norm%,area%,list%(0),I%<<7,0,8
1640   SYS sprite%,norm%,area%,list%(0),I%<<7,896,8
1650 NEXT
1660 COLOUR%110000+128
1670 GCOL%110000 TINT 0
1680 RECTANGLE FILL 96,0,1088,80
1690 PRINT TAB(4,30) "Z left   X right   ' up  / down"
1700 PRINT TAB(13,31) "Escape to Exit";
1710 E%=1
1720 CALL store
1730 ENDPROC
1740 :
1750 DEF PROCstart
1760 FOR I%=0 TO obs%
1770   x%(I%)=I%+2<<7
1780   y%(I%)=RND(4)+1<<7
1790   cells%(I%+2,y%(I%)>>7)=1
1800 NEXT
1810 ENDPROC
1820 :
1830 DEF PROChuman
1840 IF INKEY-98 t%(0)=1
1850 IF INKEY-67 t%(0)=2
1860 IF INKEY-80 t%(0)=3
1870 IF INKEY-105 t%(0)=4
1880 ENDPROC
1890 :
1900 DEF PROCcheck
1910 IF n%(I%) ENDPROC
1920 a%=x%(I%)>>7
1930 b%=y%(I%)>>7
1940 CASE t%(I%) OF
1950   WHEN 1:IF cells%(a%-1,b%) t%(I%)=0 ELSE cells%(a%-1,b%)=1
1960   WHEN 2:IF cells%(a%+1,b%) t%(I%)=0 ELSE cells%(a%+1,b%)=1
1970   WHEN 3:IF cells%(a%,b%+1) t%(I%)=0 ELSE cells%(a%,b%+1)=1
1980   WHEN 4:IF cells%(a%,b%-1) t%(I%)=0 ELSE cells%(a%,b%-1)=1
1990 ENDCASE
2000 IF t%(I%) n%(I%)=8:cells%(a%,b%)=0
2010 ENDPROC
2020 :
2030 DEF PROCmove
2040 CASE t%(I%) OF
```

```
2050   WHEN 0:PROCstill
2060   WHEN 1:PROCleft
2070   WHEN 2:PROCright
2080   WHEN 3:PROCup
2090   WHEN 4:PROCdown
2100 ENDCASE
2110 ENDPROC
2120 :
2130 DEF PROCstill
2140 SYS sprite%,norm%,area%,list%(0),x%(I%),y%(I%),8
2150 ENDPROC
2160 :
2170 DEF PROCleft
2180 x%(I%)-=16
2190 SYS sprite%,norm%,area%,list%(3-n%(I%)AND3),x%(I%),y%(I%),8
2200 n%(I%)-=1
2210 IF n%(I%)=0 t%(I%)=0
2220 ENDPROC
2230 :
2240 DEF PROCright
2250 x%(I%)+=16
2260 SYS sprite%,norm%,area%,list%(n%(I%)AND3),x%(I%),y%(I%),8
2270 n%(I%)-=1
2280 IF n%(I%)=0 t%(I%)=0
2290 ENDPROC
2300 :
2310 DEF PROCup
2320 y%(I%)+=16
2330 IF n%(I%)>2 AND n%(I%)<6 scale%!12=3 ELSE scale%!12=4
2340 SYS sprite%,spec%,area%,list%(0),x%(I%),y%(I%),8,scale%
2350 n%(I%)-=1
2360 IF n%(I%)=0 t%(I%)=0
2370 ENDPROC
2380 :
2390 DEF PROCdown
2400 y%(I%)-=16
2410 IF n%(I%)<4 scale%!12=n%(I%)+4 ELSE scale%!12=12-n%(I%)
2420 SYS sprite%,spec%,area%,list%(0),x%(I%),y%(I%),8,scale%
2430 n%(I%)-=1
2440 IF n%(I%)=0 t%(I%)=0
2450 ENDPROC
```

In the first place you will notice there is some ARM code, in PROCassemble. This is the screen storing and recovery system, discussed earlier, for complex backgrounds. An indication of its efficiency is given by the fact that it takes about twice as long to execute as the CLS command. This is very much faster than a CLS followed by only a couple of plotting commands, so is dramatically faster than re-plotting all the sprites round the edge of the screen.

When moving objects horizontally, film animation is used to give the impression of rolling wheels, whereas sprite scaling is used on the y axis for stretching and squashing the wheels as they move vertically.

Instead of drawing directly into the sprites, as normal, I've drawn to the screen so that you can see the animation sequence as it is built up. You will notice that only four sprites need to be defined as the spokes then overlap.

The sprites used are rather large and slow to plot, so due to the relatively long loop time of about two screen refreshes, there should be a considerable amount of jitter associated with the movement routine, but as the sprites are so irregular and change with every re-plotting, the action looks reasonably smooth.

To avoid the need for screen edge testing and array subscript range problems, the screen is surrounded by sprites, all of which are flagged in the array. By making a small alteration to the array and X and Y division factors, you could easily move the limiting array elements right off the screen. This gives lots of possibilities for special off screen collisions.

As a final point, you can easily identify the type of collision, and therefore the most appropriate action, by using different values for different objects in the array, as you would for the chess game mentioned before.

5.4.4 Pointer collisions

One last form of collision system is given to you by Risc Os itself. It is the mouse pointer system. The Wimp is designed to be able to identify which window, or icon within a window, the pointer is currently over. Therefore all you need to do is make up a scene that will be a window's sprite background, then overlay selected areas with sprite icons that blend in with the picture.

You now have a true desktop game. You can go further by re-defining the pointer to say, cross hairs, and use the mouse to control a form of alien zapping game. Unfortunately, speed will be poor, and a little variable, but it certainly gives you scope for really pushing the computer and your programming to the limit. I've given no program example of this, as I don't want to get embroiled in managing the Wimp, but Figure 5.3 shows how the sprites would be positioned and recognised. You will see that I've used two icons to enclose the area of the stream in the picture. This avoids the possibility of selecting the stream over a large area of land, as could occur with a single larger icon.

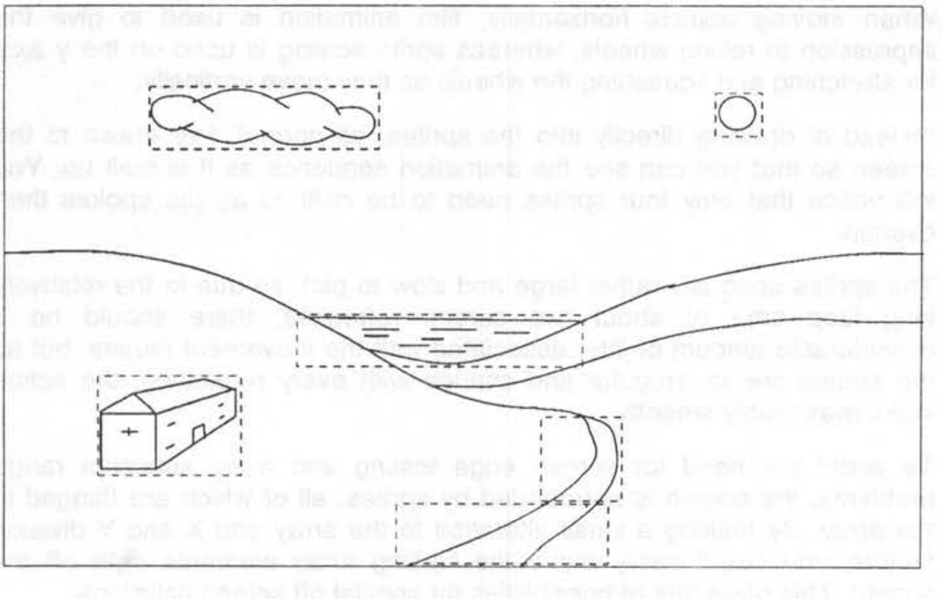

Figure 5.3: Hidden icons in pictures

If you want to investigate this area more fully you will have to study the Programmers' Reference Manual. I suggest that initially, you look at the idea of separating the pointer from the mouse (page 301), then redefining the pointer (page 331), and moving the pointer independently (page 338).

To understand how to make best use of the pointer/icon system under the Wimp you will need to pay particular attention to pages 1138 - 1140, 1146, 1180 and 1189.

Finally, it is sometimes practical to use a combination of two methods. You may, for example, decide that it is most efficient to use pixel tests to establish that some form of collision has taken place, backed up by coordinate examination to establish precisely what has been met.

5.4.5 Look ahead

One problem with collision tests is that of a skip-over taking place. This is most likely to happen with pixel testing, and is due to an object moving a significantly greater distance than the distance between it and some small obstruction. The effect is that the object seems to pass through the obstruction. The way to resolve this problem is to compare the object's movement vector with the size of the smallest obstruction in the game. If the vector is larger you need to make one or more intermediate tests, along the line of movement and in steps that are smaller than the size of the obstruction.

A similar problem arises where you have glancing hit situations. In this case, the vectors of the moving objects don't intersect but are close enough for a collision to take place, allowing for the object sizes. If this is a problem you should consider additional off-axis tests. This all tends to slow things down and add to the complexity. As usual, some compromise will probably be needed. Figure 5.4 shows both the look-ahead and off-axis situations to make the points clearer.

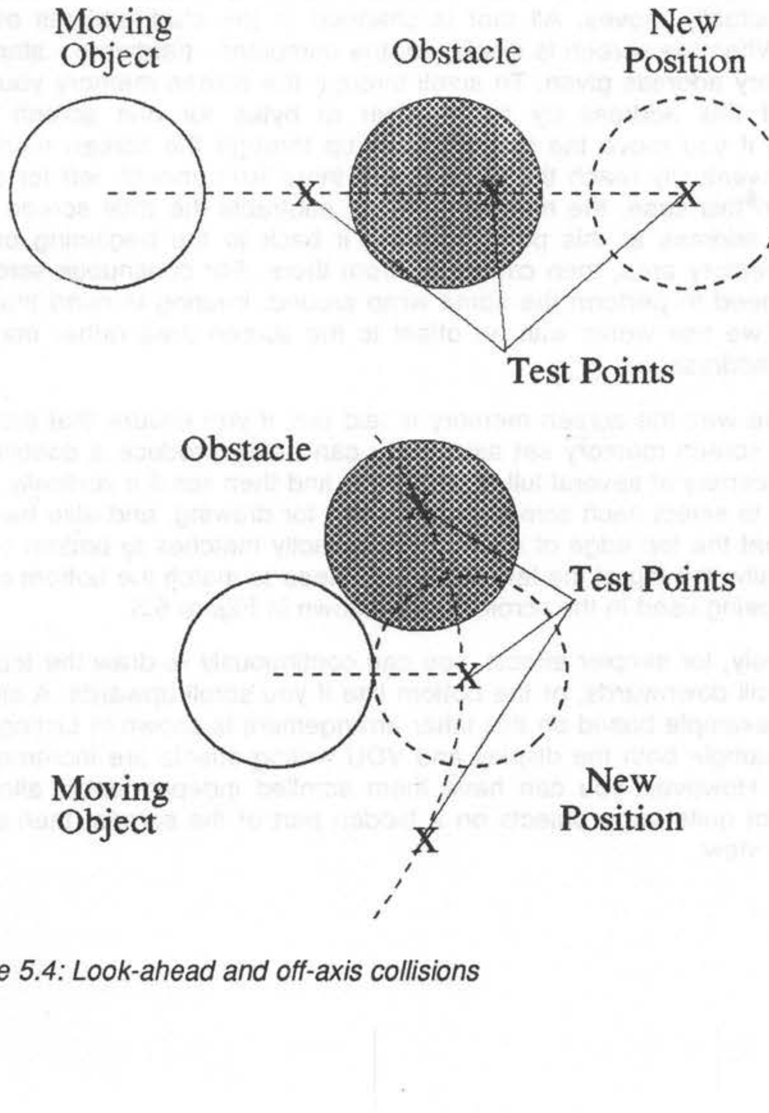

Figure 5.4: Look-ahead and off-axis collisions

5.5 Scrolling

A familiar feature of wordprocessors, spreadsheets and the like, taken quite for granted, is that of scrolling. There are many games that also use scrolling to good effect. On most machines, vertical scrolling is the easiest, and the Archimedes is no exception. There are two basic methods you can use on the Archimedes: hardware and software scrolling.

5.5.1 Hardware scrolling

This is very much faster than software scrolling because in esse nothing actually moves. All that is changed is the start address of screen. When the screen is displayed, the computer - hardware - starts at the memory address given. To scroll through the screen memory you just increment this address by the number of bytes for one screen line. Logically, if you move the start address up through the screen memory, you will eventually reach the point where there isn't enough left for a full screen. In this case, the hardware simply subtracts the total screen size from the address at this point, to bring it back to the beginning of the screen memory area, then carries on from there. For continuous scrolling you will need to perform the same wrap around, bearing in mind that the SYS call we use works with an offset to the screen area rather than an absolute address.

Due to the way the screen memory is laid out, if you ensure that there is sufficient screen memory set aside, you can easily produce a continuous band of scenery of several full screen sizes and then scroll it vertically. You will need to select each screen bank in turn for drawing, and also have to ensure that the top edge of each screen exactly matches to bottom of the next. Finally, the top of the last screen will need to match the bottom of the first one being used in the scroll. This is shown in Figure 5.5.

Alternatively, for simpler effects, you can continuously re-draw the top line if you scroll downwards, or the bottom line if you scroll upwards. A simple scrolling example based on this latter arrangement is shown in Listing 5.7. In this example both the display and VDU writing offsets are incremented together. However, you can have them scrolled independently, allowing you to plot quite large objects on a hidden part of the screen, then scroll them into view.

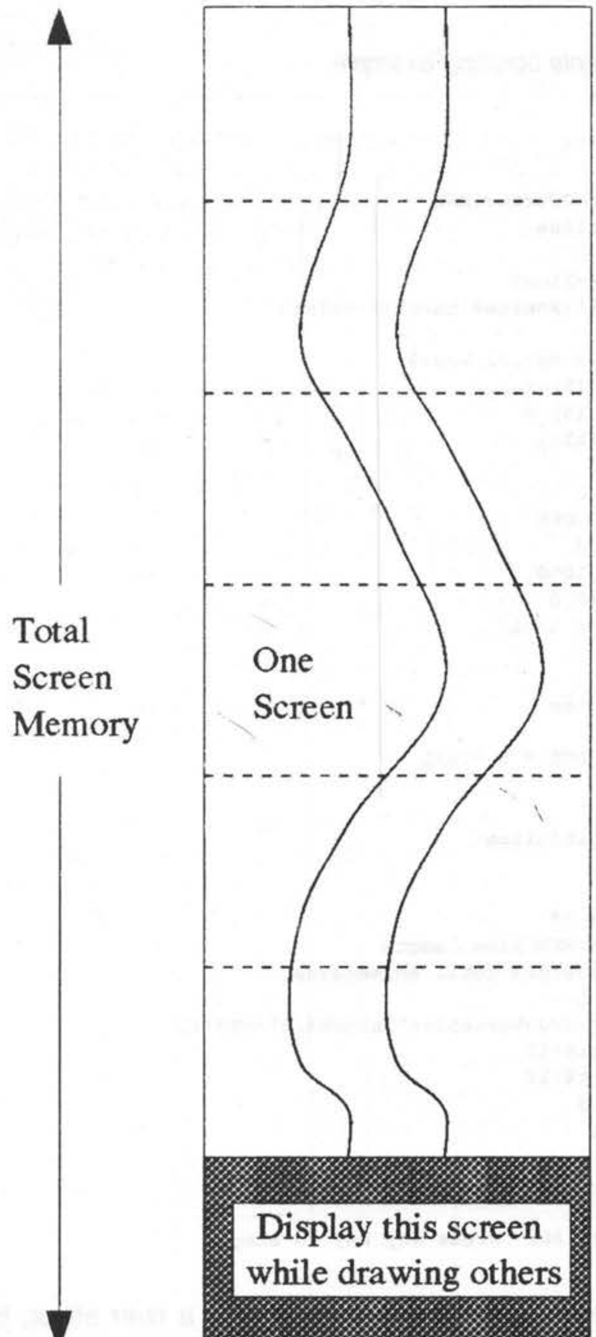

Figure 5.5: Vertical scrolling

Listing 5.7: A simple scrolling example

```
 10 REM > Scroll
 20 :
 30 ON ERROR PROCerror:END
 40 PROCinitialise
 50 REPEAT
 60   base%!1+=line%
 70   IF base%!1>=size% base%!1-=size%
 80   WAIT
 90   SYS "OS_Word",22,base%
100   a%+=RND(15)-8
110   b%+=RND(15)-8
120   GCOL %011101
130   MOVE 0,0
140   DRAW a%,0
150   GCOL %100000
160   DRAW b%,0
170   GCOL %001000
180   DRAW 1279,0
190 UNTIL INKEY 1>-1
200 END
210 :
220 DEF PROCerror
230 MODE 12
240 PRINT REPORT$ " @ ";ERL
250 ENDPROC
260 :
270 DEF PROCinitialise
280 MODE 13
290 OFF
300 DIM block% 19
310 block%!0=6:REM line length
320 block%!4=150:REM total screensize
330 block%!8=-1
340 SYS "OS_ReadVduVariables",block%,block%+12
350 line%=block%!12
360 size%=block%!16
370 DIM base% 5
380 ?base%=%11
390 base%!1=0
400 a%=512
410 b%=768
420 PRINT TAB(9,30) "Press any key to stop"
430 ENDPROC
```

I've only used simple line drawing to produce a river effect, but you can easily add sprites on top of the scroll action. If you do, you must remember to add the scrolling offset to their screen position when you rub them out, ready for the next screen refresh.

Although it is possible to get sideways scrolling using this method, it is difficult to get a smooth effect, and the results are not really worth the effort. This is because the screen start address can't be offset by only one byte, but has to be a number of words depending on the current screen mode. The result is, that you need to re-draw quite a wide block before scrolling. For the same reason, diagonal scrolling is even more difficult using this technique, although, it would be very interesting to see someone come up with a practical way of doing it.

5.5.2 Scrolling in software

This method involves re-plotting every point of the screen, offset by the amount of movement required. As before, you will then need to re-draw the newly exposed areas. One simple, elegant solution for this re-plotting, is to define a sprite to be the entire screen area, then re-plot the sprite offset by the degree of scroll movement you want. This makes for extremely easy, albeit rather slow, re-plotting. Provided you take care of the necessary edge filling, you have the basis for a simple, all directions, scroll. However, it is really best suited for the lower resolution, 16 colour modes. This is demonstrated in Listing 5.8.

Listing 5.8: Software scrolling

```
 10 REM > SpriteScrl
 20 :
 30 ON ERROR PROCerror:END
 40 PROCinitialise
 50 REPEAT
 60   SYS sprite%,get%,area%,"S",0,xl%,yl%,xh%,yh% TO ,,add%
 70   x%=(INKEY-98)-(INKEY-67)<<2
 80   y%=(INKEY-105)-(INKEY-80)<<2
 90   WAIT
100   SYS sprite%,put%,area%,add%,xl%+x%,yl%+y%
110   IF x% PROCvert
120   IF y% PROChoriz
130   IF RND(20)=1 PROCblot
140 UNTIL FALSE
150 END
160 :
170 DEF PROCerror
180 *FX 21
190 MODE 12
200 IF ERR<>17 PRINT REPORT$ " @ ";ERL
210 ENDPROC
220 :
230 DEF PROCinitialise
240 MODE 9
```

```
250 OFF
260 SYS "OS_SWINumberFromString",,"OS_SpriteOp" TO sprite%
270 size%=&14000
280 DIM area% size%
290 area%!0=size%
300 area%!4=0
310 area%!8=16
320 init%=256+9
330 get%=512+16
340 put%=512+34
350 SYS sprite%,init%,area%
360 COLOUR 8,128,128,128
370 PRINT TAB(10,2) "Eight direction scroll"
380 PRINT TAB(2,5) "Z left   X right   ' up   / down"
390 PRINT TAB(10,26) "Press Escape to exit"
400 xl%=160
410 xh%=1120
420 yl%=256
430 yh%=768
440 back%=8
450 GCOL 128+back%
460 VDU 24,xl%;yl%;xh%;yh%;
470 CLG
480 ENDPROC
490 :
500 DEF PROCvert
510 IF x%>0 PROCleft ELSE PROCright
520 ENDPROC
530 :
540 DEF PROChoriz
550 IF y%>0 PROCbottom ELSE PROCtop
560 ENDPROC
570 :
580 DEF PROCleft
590 GCOL back%
600 LINE xl%,yl%,xl%,yh%
610 ENDPROC
620 :
630 DEF PROCright
640 GCOL back%
650 LINE xh%,yl%,xh%,yh%
660 ENDPROC
670 :
680 DEF PROCtop
690 GCOL back%
700 LINE xl%,yh%,xh%,yh%
710 ENDPROC
720 :
730 DEF PROCbottom
740 GCOL back%
750 LINE xl%,yl%,xh%,yl%
760 ENDPROC
```

```
770 :
780 DEF PROCblot
790 GCOL RND(7)
800 r%=RND(63)
810 CIRCLE FILL xl%+r%+RND(960-r%*2),yl%+r%+RND(512-r%*2),r%
820 ENDPROC
```

You will see that I've cheated and used only the central portion of the screen. If you change the constants *xl%, xh%, yl%* and *yh%*, you will soon see the very real need for this. Also, the edge filling only consists of simple background line re-draws. You can improve on this by using an algorithm that can break down drawn objects into vertical and horizontal lines. I've split the re-drawing routine up very thoroughly so that you can see where the line re-drawing could be replaced by your improved filling algorithm.

The overprinted circles should, ideally, be sprites. As before you will need to make allowances for the movement of these sprites when you rub them out.

5.5.3 ARM code scrolling

This is by far the best scrolling arrangement if you want really complicated scrolling action. It is covered more fully in the ARM code chapter, so the only comment I'll make here, is that such routines will enable you to scroll any part of the screen in any direction, and even two parts in different directions.

6

More Dimensions

6.1 3D

Over the last 20 years or so, there has been an enormous amount of interest in three dimensional representation using computers. With the event of relatively cheap and powerful domestic machines, it was only to be expected that this would result in a rash of 3D games. There are now reams of mathematical papers and discussion documents on 3D projection and rotation techniques. Therefore I'll give you just enough to get started here, with the minimum of mathematics. If you want to go further then I suggest you read up on the subject.

6.1.1 Cartoon styles

One of the benefits of having scalable sprites is that you can construct pseudo 3D effects in exactly the same manner as is used in cartoon animation. Although not true 3D, the result is quite acceptable, and widely used in many games. The principle simply revolves around the fact that the further away an object is, the smaller it seems. Typically, in a real cartoon and in most games, there will only be four or five distinct layers, usually referred to as parallax layers. These are usually set out as horizontal strips, the layer furthest away enclosing the horizon line. In Figure 6.1 you can see how this all fits together.

For simplicity I've only shown three layers and offset them to one side for clarity. You will see that I've also outlined the area enclosing each with dotted lines. If you define sprites for all these sections they can be plotted quickly, along with the objects in your game.

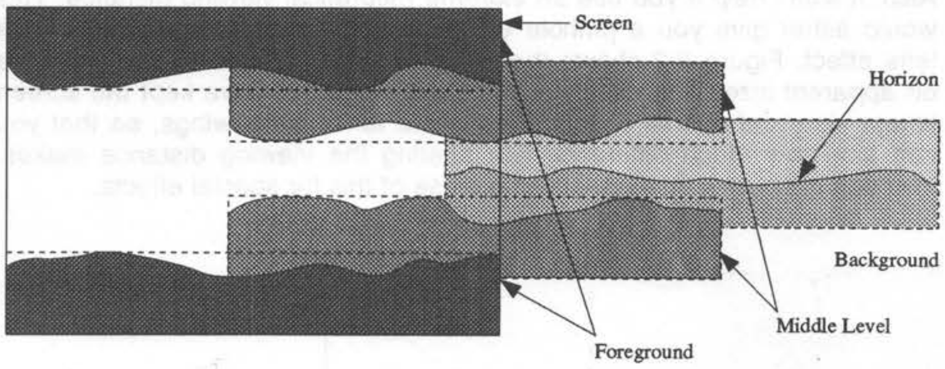

Figure 6.1: Cartoon layers

To get your objects to appear on the right layers, you will need to plot them in the right order. The background scene should be plotted first, then any objects intended to be immediately in front of that layer. Next you plot the middle scenes, followed by the sprites on these layers. Finally you plot the foreground scene, and any very prominent objects. As you can see, you don't need to plot the parts of the scenes covered by layers that are to be nearer the front, so your sprites only need to be just big enough to ensure that no gaps appear between the levels. Also, you don't need to plot any object sprites wholly behind one of the intermediate layers, thus saving processor time. Even so, you may find that you either have to keep all other game activities to the absolute minimum, or use your own ARM code sprite routine.

6.1.2 Scaling

The apparent distance from the viewer is determined by a scaling factor derived from the theoretical distance of the viewer from the screen, as well as the size of the objects that are to seem nearest. As you can't control the actual viewing distance, all you can do is make sure that the foreground objects are of a size and type that gives the illusion of pushing the foreground back to where you want it. For this to be effective you have to give the viewer some idea of the size of at least one object. In car racing games, for example, a pair of hands on a steering wheel are often shown as foreground objects. Everyone knows how big their hands are so they make a subconscious adjustment to bring the observed view into scale.

Also, it won't help if you use an extreme theoretical viewing distance. This would either give you a pinhole effect or at the other extreme a fisheye lens effect. Figure 6.2 shows the effect of changing the viewing distance on apparent sizes and distances. You will see that I have kept the screen image size and overall distance the same for both drawings, so that you can see how much difference just altering the viewing distance makes. You can of course, intentionally make use of this for special effects.

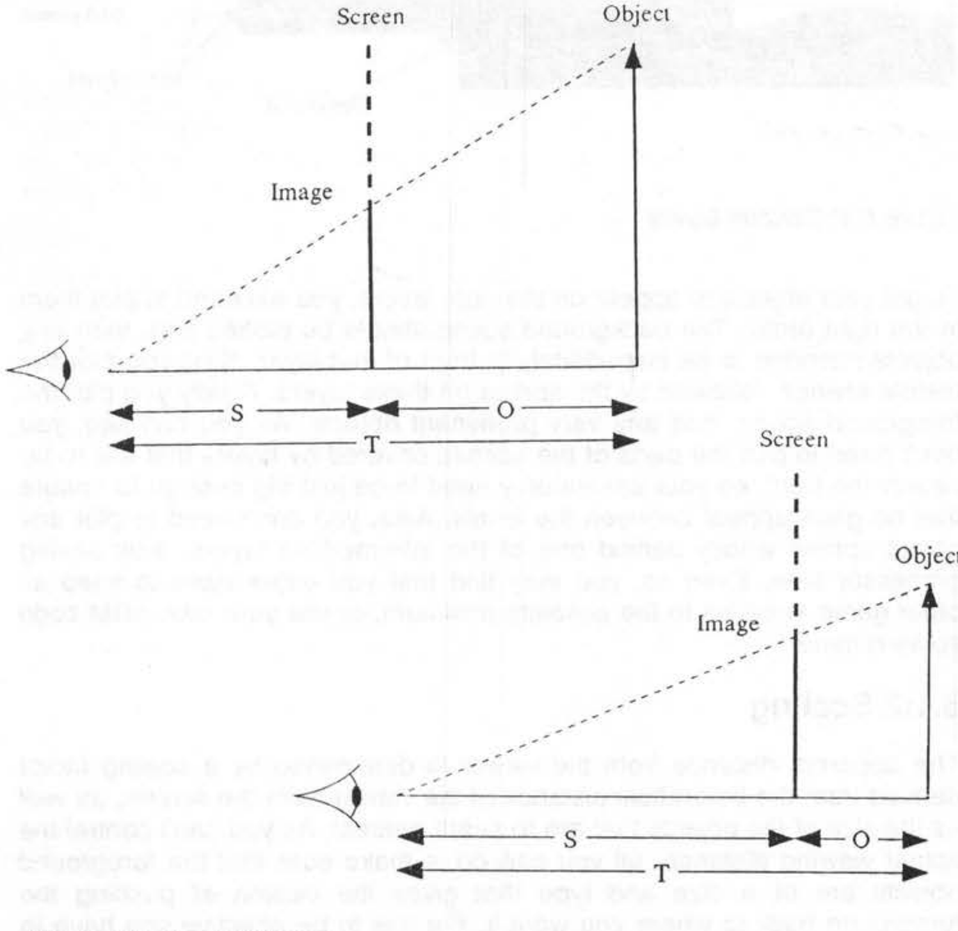

Figure 6.2: Scaling factors

The values S and T are the viewer-to-screen and total distance respectively. To scale any object at any distance correctly, you only need a simple piece of maths.

apparent size = real size * S / T

For various reasons it is often more practical to use the object to screen distance, O, rather than the total distance. One reason for this is that any object that has moved beyond the screen towards the viewer develops a negative value, and for plotting purposes can therefore be ignored.

Our formula then becomes:

apparent size = real size * S / (S + O)

6.1.3 Perspective

So all you need to do is scale the sprite to the distance from the screen to the horizon line. Well, actually no, it's a bit more complicated than that. If your sprite is to one side of the centre line of the display, steadily reducing its size will make it seem to veer even further to the side as it moves backwards. This is where perspective comes in.

Without going into the mathematics of light and lenses, what normally happens is that all objects moving away from you seem to converge on a single point, known as the vanishing point, directly at the centre of your line of vision. Therefore, to get realistic movement backwards and forwards you need to scale not only the size of the sprite but also its coordinates relative to this centre line. From the distance formula above, we can derive a perspective factor for all our calculations on any given object.

P = S / (S + O)

If you assume the centre line, or Z axis has a value of zero at the screen surface, with positive values towards the vanishing point, and that all X and Y coordinates are also relative to the centre line, you can develop the following scaling formula:

X plot	= X position * P
Y plot	= Y position * P
sprite scale	= object size * P

Listing 6.1 shows these basic principles, and the practical application of the mathematics. This gives you typical flat object, cartoon 3D movement, but with rather more freedom of depth, as movement is not restricted to

only four or five horizontal planes. As sprites are plotted from their bottom left-hand corner, it is necessary to add a half size offset to both the X and Y coordinates for correct positioning.

Listing 6.1: Principles of scaling

```
 10 REM > Cartoon
 20 :
 30 ON ERROR PROCerror:END
 40 PROCinitialise
 50 PROCsprite(rockx%,rocky%)
 60 PROCstart
 70 REPEAT
 80   MOUSE nx%,ny%,b%
 90   dx%=SGN((nx%-x%)>>4)
100   dy%=SGN((ny%-y%)>>4)
110   x%+=dx%<<4
120   y%+=dy%<<4
130   IF b%=4 z%-=64 ELSE IF b%=1 z%+=64
140   IF z%<64 z%=64
150   WAIT
160   SYS byte%,113,sc%
170   sc%=sc% EOR 3
180   SYS byte%,112,sc%
190   PROCback
200   IF z%<256 PROCmiddle
210   IF z%<100 PROCfore
220   PROCdisplay(rock%,(x%-rockx%*2)*s%/(s%+z%),(y%-rocky%*2)*s%/(s
%+z%),s%<<2,s%<<2,s%+z%,s%+z%)
230   IF z%>255 PROCmiddle
240   IF z%>99 PROCfore
250   PROCprint
260 UNTIL FALSE
270 END
280 :
290 DEF PROCerror
300 MODE 12
310 IF ERR<>17 PRINT REPORT$ " @ ";ERL
320 ENDPROC
330 :
340 DEF PROCinitialise
350 *Pointer 1
360 MODE 15
370 MODE 13
380 PRINT TAB(10,10)"Please Wait"
390 SYS "OS_SWINumberFromString",,"OS_SpriteOp" TO sprite%
400 SYS "OS_SWINumberFromString",,"OS_Byte" TO byte%
410 DIM block% 19
420 block%!0=4
430 block%!4=5
```

```
440 block%!8=-1
450 SYS "OS_ReadVduVariables",block%,block%+12
460 xeig%=block%!12
470 yeig%=block%!16
480 size%=&2000
490 DIM area% size%
500 area%!0=size%
510 area%!4=0
520 area%!8=16
530 DIM scale% 15
540 scale%!0=1
550 scale%!4=1
560 scale%!8=1
570 scale%!12=1
580 init%=256+9
590 def%=256+15
600 select%=256+24
610 mask%=512+29
620 getpix%=512+41
630 putpix%=512+44
640 plot%=512+52
650 writeto%=256+60
660 style%=8
670 SYS sprite%,init%,area%
680 rockx%=202
690 rocky%=112
700 ENDPROC
710 :
720 DEF PROCsprite(x%,y%)
730 rock%=FNdefsprite("rock",x%,y%)
740 PROCrock(x%,y%)
750 PROCmasksprite(rock%,x%,y%)
760 SYS sprite%,writeto%,area%,0
770 ENDPROC
780 :
790 DEF FNdefsprite(a$,x%,y%)
800 LOCAL add%
810 x%=x%>>xeig%
820 y%=y%>>yeig%
830 SYS sprite%,def%,area%,a$,0,x%,y%,MODE
840 SYS sprite%,writeto%,area%,a$
850 SYS sprite%,select%,area%,a$ TO ,,add%
860 =add%
870 :
880 DEF PROCrock(x%,y%)
890 GCOL %101010 TINT &C0
900 x%=x% DIV 11
910 y%=y% DIV 5
920 MOVE x%*2,y%*4
930 MOVE BY x%*4,-y%*2
940 PLOT&71,x%*3,y%
950 GCOL %101010 TINT &40
```

```
 960 PLOT&51,x%*2,-y%*2
 970 GCOL %010101 TINT &C0
 980 MOVE BY -x%*7,-y%
 990 PLOT&51,x%*2,y%*2
1000 GCOL %10101 TINT &40
1010 PLOT&71,-x%*4,y%*2
1020 ENDPROC
1030 :
1040 DEF PROCmasksprite(add%,x%,y%)
1050 LOCAL I%,J%,c%
1060 x%=x%>>xeig%
1070 y%=y%>>yeig%
1080 SYS sprite%,mask%,area%,add%
1090 FOR J%=0 TO y%-1
1100   FOR I%=0 TO x%-1
1110     SYS sprite%,getpix%,area%,add%,I%,J% TO,,,,,c%
1120     IF c%=0 SYS sprite%,putpix%,area%,add%,I%,J%
1130   NEXT
1140 NEXT
1150 ENDPROC
1160 :
1170 DEF PROCstart
1180 VDU 5
1190 x%=640
1200 y%=512
1210 s%=128
1220 z%=256
1230 sc%=1
1240 ORIGIN 640,512
1250 MOUSE ON
1260 ENDPROC
1270 :
1280 DEF PROCdisplay(add%,x%,y%,scale%!0,scale%!4,scale%!8,scale%!12)
1290 SYS sprite%,plot%,area%,add%,x%,y%,style%,scale%
1300 ENDPROC
1310 :
1320 DEF PROCback
1330 GCOL %111010
1340 RECTANGLE FILL -640,0,1280,124
1350 GCOL %101111
1360 RECTANGLE FILL -640,-128,1280,124
1370 ENDPROC
1380 :
1390 DEF PROCmiddle
1400 GCOL %100101
1410 RECTANGLE FILL -640,128,1280,156
1420 GCOL %1110
1430 RECTANGLE FILL -640,-288,1280,156
1440 ENDPROC
1450 :
1460 DEF PROCfore
1470 GCOL %010000
```

```
1480 RECTANGLE FILL -640,288,1280,220
1490 GCOL %1001
1500 RECTANGLE FILL -640,-512,1280,220
1510 ENDPROC
1520 :
1530 DEF PROCprint
1540 GCOL %111111
1550 MOVE -400,440
1560 PRINT "The mouse moves the rock"
1570 MOVE -480,360
1580 PRINT "Use select to bring it nearer"
1590 MOVE -480,280
1600 PRINT "Use adjust to to move it away"
1610 MOVE -320,200
1620 PRINT "Press Escape to stop"
1630 ENDPROC
```

In the demonstration you will see the effect of the background layering as the rock drifts downwards, apparently from behind the middle level. Moving the rock backwards and forwards, while keeping it either high or low on the screen, will make it jump from level to level.

For the sake of simplicity, instead of sprites I've just used filled rectangles to represent the different background levels, and a simple distance check to set the plotting order. However you should easily be able to see the possibilities. With full sprite plotting I suggest you maintain an array with the sprite pointers in it, and as objects move forward and back, you simply swap adjacent pointers to ensure they are always plotted in the right order.

There are occasions where you can have more than one vanishing point. These will be at the edges of the screen, or could even be right off the screen altogether. For our purposes we'll ignore that situation though, as it gets far too complicated.

Finally, so far we've only considered the horizon as being half way up the screen. Depending on the type of game, it may be better to have it somewhat below half way. Setting the sky to ground ratio at about 1.6 to 1 usually looks quite good. As a general rule, the higher the horizon, the more you seem to look down at the scene, whereas a low horizon gives the impression of being very close to the ground, looking up. Changing the position of the horizon is often used in race games, and is essential for flight simulators.

6.1.4 Wire-frame drawings

This is the most familiar form of 3D projection, and very effective if used thoughtfully. Perspective affects any face of an object that isn't exactly at right angles to the viewer. This is clearly illustrated by the buildings either side of the road in Figure 6.3. If you think of the drawing in terms of a collection of points connected by straight lines you will see that the perspective rules we already know can be immediately applied to individual objects as well as the overall scene. The lines representing the depth of an object are scaled to the perspective factor in exactly the same way as the overall distance.

Figure 6.3: Perspective drawing

It is usual to consider the corner points of an object from a central reference point rather than one of the actual corners. This makes scaling and positioning much simpler, as well as assisting collision detection. You can maintain an array of the positions of the corner points relative to this centre, then another array that simply lists these points, to give the actual lines that are to be drawn.

It sometimes takes a bit of thought to get a clear picture of this. What we have is one table looking into another. So you could have an entry in the line list table of 2,3. These are not the actual coordinates but the index numbers for the points table where the two sets of actual X,Y,Z coordinates can be found. Unfortunately this is further complicated by the fact that you need to maintain yet another array, giving the absolute position of each object in a given scene. Combining the relative corner positions with the absolute object position and adding the perspective factor, will then give you the actual screen plotting points for every line.

Listing 6.2 shows this in action. A wire-frame cube can be moved and rotated in all planes. However, you should bear in mind that, as it stands, the system works only for a fixed viewpoint relative to the absolute centre of the game scene, this being a point, apparently at the surface of the screen. The program has been kept as simple as possible, so movement is a little jerky, but as you will see later, this can be improved.

Listing 6.2: Wire frame drawing and rotation

```
 10 REM > WireFrame
 20 :
 30 ON ERROR PROCerror
 40 PROCinitialise
 50 REPEAT
 60   PROCmove
 70   WAIT
 80   SYS byte%,113,sc%
 90   sc%=sc% EOR 3
100   SYS byte%,112,sc%
110   CLS
120   PROCprint
130   PROCdraw
140 UNTIL FALSE
150 END
160 :
170 DEF PROCerror
180 MODE 12
190 *FX 21
200 IF ERR<> 17 PRINT REPORT$ " @ ";ERL
210 END
220 :
230 DEF PROCinitialise
240 MODE 12
250 MODE 9
260 OFF
270 SYS "OS_SWINumberFromString",,"OS_Byte" TO byte%
280 sc%=1
290 ORIGIN 640,512
300 scale%=800: REM overall scaling
310 E%=FALSE:   REM zero/negative Z axis flag
320 spos=SIN.1
330 sneg=SIN-.1
340 cpos=COS.1
350 cneg=COS-.1
360 RESTORE+29
```

```
370 READ numpoints%
380 READ numfaces%
390 READ maxlines%
400 :
410 DIM px%(numpoints%): REM \ final
420 DIM py%(numpoints%): REM / plotting co-ordinates
430 DIM vx(numpoints%):  REM \  object
440 DIM vy(numpoints%):  REM  > face
450 DIM vz(numpoints%):  REM /  vertices
460 DIM points%(numfaces%,maxlines%)
470 DIM lines%(numfaces%)
480 :
490 FOR I%=0 TO numpoints%
500   READ vx(I%)
510   READ vy(I%)
520   READ vz(I%)
530 NEXT
540 READ X%
550 READ Y%
560 READ Z%
570 READ S%
580 FOR J%=0 TO numfaces%
590   READ lines%(J%)
600   FOR I%=0 TO lines%(J%)
610     READ points%(J%,I%)
620   NEXT
630 NEXT
640 ENDPROC
650 DATA 7:        REM number of points
660 DATA 5:        REM number of faces
670 DATA 3:        REM maximum number of lines
680 :
690 DATA -1,-1,-1:  REM relative points
700 DATA 1,-1,-1
710 DATA 1,1,-1
720 DATA -1,1,-1
730 DATA -1,-1,1
740 DATA 1,-1,1
750 DATA 1,1,1
760 DATA -1,1,1
770 :
780 DATA 256,-128,0: REM game co-ordinates
790 DATA 128:        REM size
800 DATA 3,0,1,2,3:  REM lines this face / point numbers
810 DATA 3,1,5,6,2
820 DATA 3,5,4,7,6
830 DATA 3,4,0,3,7
840 DATA 3,3,2,6,7
850 DATA 3,4,5,1,0
860 :
870 DEF PROCmove
880 IF INKEY-98 X%-=8 ELSE IF INKEY-67 X%+=8
```

```
  890 IF INKEY-80 Y%+=8 ELSE IF INKEY-105 Y%-=8
  900 IF INKEY-99 Z%+=8:E%=1 ELSE IF INKEY-74 AND E%>0 Z%-=8
  910 IF INKEY-58 PROCrot(vy(),vz(),spos,cpos) ELSE IF INKEY-42 PROCro
t(vy(),vz(),sneg,cneg)
  920 IF INKEY-26 PROCrot(vx(),vz(),sneg,cneg) ELSE IF INKEY-122 PROCr
ot(vx(),vz(),spos,cpos)
  930 IF INKEY-57 PROCrot(vx(),vy(),spos,cpos) ELSE IF INKEY-89 PROCro
t(vx(),vy(),sneg,cneg)
  940 ENDPROC
  950 :
  960 DEF PROCrot(RETURN a(),RETURN b(),s,c)
  970 FOR I%=0 TO numpoints%
  980   u=a(I%)
  990   v=b(I%)
 1000   a(I%)=u*c-v*s
 1010   b(I%)=v*c+u*s
 1020 NEXT
 1030 ENDPROC
 1040 :
 1050 DEF PROCprint
 1060 PRINT TAB(14,1) CHR$138 " " CHR$139 " - Rotate X"
 1070 PRINT TAB(14,3) CHR$136 " " CHR$137 " - Rotate Y"
 1080 PRINT TAB(14,5) "{ } - Rotate Z"
 1090 PRINT TAB(10,7) "Z - left" SPC8 "X - right"
 1100 PRINT TAB(10,9) "' - up" SPC9 "/ - down"
 1110 PRINT TAB(3,11) "Spacebar - back  Return - forward"
 1120 PRINT TAB(5,13) "Escape - stop"
 1130 ENDPROC
 1140 :
 1150 DEF PROCdraw
 1160 FOR J%=0 TO numfaces%
 1170   IF FNcansee PROCface(lines%(J%))
 1180 NEXT
 1190 ENDPROC
 1200 :
 1210 DEF FNcansee
 1220 IF FNset(0,2):=FALSE
 1230 =((px%(0)-px%(1))*(py%(2)-py%(1))-(py%(0)-py%(1))*(px%(2)-px%(1)
))<0
 1240 :
 1250 DEF FNset(a%,b%)
 1260 FOR I%=a% TO b%
 1270   E%=(vz(points%(J%,I%))*S%+Z%+scale%)
 1280   IF E%>0 pers=scale%/E% ELSE I%=b%
 1290   px%(I%)=(vx(points%(J%,I%))*S%+X%)*pers
 1300   py%(I%)=(vy(points%(J%,I%))*S%+Y%)*pers
 1310 NEXT
 1320 =E%<1
 1330 :
 1340 DEF PROCface(n%)
 1350 IF FNset(3,n%) ENDPROC
 1360 GCOL J%+1
```

```
1370 MOVE px%(n%),py%(n%)
1380 FOR I%=0 TO n%
1390   DRAW px%(I%),py%(I%)
1400 NEXT
1410 ENDPROC
```

PROCinitialise does quite a lot of work. As well as some important constants, such as sine and cosine values, there are two others of particular interest. The scaling factor, *scale%* is more or less at its optimum value, but can be altered to see the effect. As it is, the drawn cube looks about right. A smaller scaling factor would make it look stretched, and a larger factor would give it a stubby appearance.

The ORIGIN command allows you to move the vanishing point either horizontally or vertically to any part of the screen for special effects. As shown, it is centralised for simplicity.

Also in the initialisation, there is quite a bit of data. There are figures defined for the number of corners or points that makes up the cube, and similar figures for faces and lines. The rest of the data is then dropped into the appropriate arrays.

First you have a set of x,y,z offsets from the centre of the cube. These values are dropped into the arrays *vx(), vy(),* and *vz().* Next you have the x,y and z absolute coordinates of the whole object in the game world, followed by its size scaling factor.

The figures following the number of faces are put in the *faces%()* array. These are not actual point coordinates, but are lists giving the order in which points are visited. The real coordinates are those found in the first list. This, as I pointed out earlier, takes quite a bit of thinking about to grasp properly. It is very easy to get muddled up, so when looking at the program I suggest you keep referring back to this.

PROCmove is a fairly straightforward negative INKEY system, to give movement for the demonstration.

I haven't bothered much with limiting the range of movement and have allowed the absolute object Z coordinate – not its final drawing value – to become negative. If you use the Return key to bring the cube a long way forward, you will see that sideways movement then gives the uncanny impression that the object is floating just in front of the screen.

6.1.5 Hidden lines

If you want to make your drawing look more realistic, or if you want to draw solid faces, the first problem that becomes apparent is that of removing the parts hidden by the bulk of the object. The solution is to consider the face that each group of lines bounds. If the face, is pointing away from you, towards the vanishing point, it is invisible and therefore doesn't need to be drawn.

To determine which way the face is pointing you need to use a little vector mathematics. In the first place you must ensure that when you construct each face, the corner points would always be visited in the same direction, assuming you were looking directly at the face. This is conventionally anti-clockwise. Once you have performed all the adjustments to these coordinates to represent the face in its correct orientation, the direction of the final plotting points will have become clockwise if the face is pointing away from you.

For each face you only need to use the first three pairs of coordinates and the expression below. X1,Y1 X2,Y2 X3,Y3 are of course the relevant coordinate pairs.

```
visible = ((X1 – X2) * (Y3 – Y2) – (Y1 – Y2) * (X3 – X2))<0
```

Logically, you would consider that for wire frame drawings, none of the bounding lines need to be drawn round an invisible face, but the situation is complicated by the fact that some of these lines are shared with other, visible faces. Therefore you either have to find some method of discovering which edges are visible, or you need to accept the time loss of drawing some lines twice, as we have done in the example.

In our example program FNcansee returns the visibility result for each face, calling PROCface if all is well. The call to FNset performs a dual operation. The call from FNcansee calculates only the first three coordinate pairs. The second call from PROCface completes the calculation of all other coordinate pairs of the face for the drawing routine. This avoids wasting a lot of time calculating points on an invisible face.

FNset performs a second function. If *E%* is zero, a *Division by zero* error could result. If it's negative, that object face is, apparently, the wrong side of the viewer and, therefore, can't be drawn. It is always preferable to trap possible errors in this way, before they have actually occurred, rather than rely on error correction later.

If you want to be more rigorous with your line drawing, it is probably simplest to arrange a system of flags so that once a line is drawn, or has been declared invisible, it is marked as not being needed again. If the calculations are separated from the actual drawing, the lines can be drawn simply and quickly, irrespective of which face they are connected with.

6.1.6 Rotation

If you want your object to rotate, I'm afraid you have to delve into yet more mathematics. It looks horrible but is in fact quite easy to implement. Assumimg you want to rotate around the Z axis, only the X and Y coordinates need to be altered:

```
newX = oldX * COS(angle) – oldY * SIN(angle)
newY = oldY * COS(angle) + oldX * SIN(angle)
```

If it's the Y axis you want to rotate around you simply exchange all the Y terms for Z terms in the two expressions. Similarly you can swap in Z for X if you want rotation on the X axis.

One of the features of Basic V is that you can pass whole arrays as parameters in procedures. This is particularly useful here, because you can produce a generalised rotation procedure for all objects and rotation directions. You can see this in PROCrot in Listing 6.2.

A point to bear in mind is whether accuracy or speed is most important. By defining a rotation increment and repeatedly adding it to the object coordinates, I've chosen speed. For accuracy, you'd define your total rotation angle for each step, then use this to produce a rotated copy of your master array, thus eliminating cumulative errors.

6.1.7 Matrices

Basic V, as well as supporting whole array arithmetic, also provides true matrix multiplication. As with all the array operations, this is considerably faster than using nested FOR-NEXT loops and picking out individual values. This is particularly useful when you want to perform a rotation of an object with a large number of points, or a group of objects round a common centre.

In the fragment below, I've shown the significant points of the rotation method. I won't go into the details of matrix manipulation. You should have no difficulty using the transformation routines without having to understand them.

```
DIM obs(points-1,2)   :REM object points/x,y,z
DIM rotate(2,2)       :REM 3d rotation matrix
DIM r(2,2)            :REM construction matrix
  X = 0.1
  Y = 0.02            :REM specimen values
  Z = 0.03
  PROCtemplate

  REM main loop
  obs()=obs().rotate()  :REM rotates the whole array
  PROCdraw
  REM end of loop
  REM end of everything

  DEFPROCtemplate
    sz=SIN(Z):cz=COS(Z)
    sy=SIN(Y):cy=COS(Y)
    sx=SIN(X):cx=COS(X)

    r(0,0)= cz:r(1,0)= sz:r(2,0)=  0
    r(0,1)=-sz:r(1,1)= cz:r(2,1)=  0
    r(0,2)=  0:r(1,2)=  0:r(2,2)=  1

    rotate()=r()

    r(0,0)= cy:r(1,0)=  0:r(2,0)= sy
    r(0,1)=  0:r(1,1)=  1:r(2,1)=  0
    r(0,2)=-sy:r(1,2)=  0:r(2,2)= cy

    rotate()=rotate().r()

    r(0,0)=  1:r(1,0)=  0:r(2,0)=  0
    r(0,1)=  0:r(1,1)= cx:r(2,1)= sx
    r(0,2)=  0:r(1,2)=-sx:r(2,2)= cx

    rotate()=rotate().r()
  ENDPROC
```

The array *obs* contains the X,Y,Z coordinates of the whole scene to be rotated. This could be a complex many cornered object, or a group of simpler objects. Whereas, in the earlier example, we gained speed by keeping the X,Y,Z object coordinates in separate arrays, you will see that we have to use a common two dimensional array for matrix manipulation. The payoff is that the actual rotation is performed by a single statement that executes remarkably fast.

The rotation matrix itself is built up in PROCtemplate from three separate matrices, one for each rotation axis. While it would have been possible to perform the necessary mathematics to combine these matrices by hand, it seems pointless to do so when the computer can do it for you. It is rather ironic that the rotation matrix can be built up with exactly the same command used to perform the actual rotation. If you need to alter the rotation matrix within the loop, it might well pay to use a single hand calculated combination, as it will execute faster.

The same note about speed and accuracy applies with matrix rotation as with the form in the earlier example.

6.1.8 Universal movement and rotation

So far we have looked at our game world, regarding the viewer as the centre. With a game of any size, a space scenario for example, this isn't a good idea. What you should do is maintain a map – a three dimensional array – of all objects with their coordinates relative to some fixed central point. By doing this, instead of moving objects relative to the viewer, you can move them in this absolute map, and more importantly, you can move the viewer as well in say, a space ship.

Rotating the direction of view also becomes a practical possibility. I haven't given an example program for this, but if you want to pursue this line, I suggest you start by regarding the viewer as another object. A good choice for this would be object 0. This object's coordinates can be moved and rotated in exactly the same way as any other. This has the added benefit of giving you the capability of swapping object coordinates, and therefore hopping from one object to another. A further benefit is that the viewer's object can easily be incorporated into your main collision system.

The only problem then becomes that of translating the view of the whole game world, to that of the viewing object. In the first instance, you need to find out which objects are actually visible.

As well as being in the viewer's line of sight, objects need to be close enough to see. With the scale factor used in the previous example, you would can consider any object with a distance from the viewer of around 9,000 to 10,000 as being too far away to be seen.

To avoid wasting time drawing objects that are right off the side of the screen, you can make a rough check in the face visibility procedure. If the first pair of coordinates is well outside the normal graphic coordinate range and the object isn't enormous, the routine can safely mark it as invisible.

6.2 Sound

With a little imagination you can easily regard sound as another dimension to your game, and probably a fairly essential one these days. It is certain that your game will have a distinctly *flat* feel to it without any sound at all.

6.2.1 Music

Many games have background music running through them. As mentioned earlier, if you decide to incorporate music in your game it's vital that you also have an option to turn it off. A number of sound tracker utilities are now available to make this quite a painless process. Having created a music file, this is run, quite transparently, within the playing module. Usually all you need to do is make simple SYS calls to start and stop the music. While I strongly recommend that you use a sound tracker utility for this reason, you can produce quite acceptable results from a home grown Basic routine interleaved with your main game.

The Archimedes sound system trades off amplitude against number of channels, so you don't really want to enable any more channels than you actually require. Having said that, it often pays to use different channels for your music and sound effects. That way you won't have them fighting each other for control, and producing some rather strange sounds in the process.

The Acorn method of timing the beats of music is quite sophisticated, and although fine for most purposes, especially in music editing programs, it is not so easy to use inside a game loop. It is simpler to fall back on the older method used in the 8-bit machines. This is shown in Listing 6.3.

Listing 6.3: Creating music

```
 10 REM > Music
 20 :
 30 ON ERROR PROCerror:END
 40 PROCinitialise
 50 PROCsetchans
 60 PROCtitle
 70 mark%=TIME
 80 count%=mark%
 90 REPEAT
100   IF INKEY-17 count%=TIME+&FFFFF
110   IF INKEY-102 mark%=TIME:count%=mark%
120   IF INKEY-38 SOUND 4,-15,0,20
130   IF TIME-count%>0 PROCsound
140   WAIT
150   SYS byte%,113,sc%
```

```
160   sc%=sc% EOR 3
170   SYS byte%,112,sc%
180   CLS
190   PRINT text$;
200   PROCdraw
210 UNTIL FALSE
220 END
230 :
240 DEF PROCerror
250 MODE 12
260 IF ERR<>17 PRINT REPORT$ " @ ";ERL
270 VOICES 1
280 *ChannelVoice 1 WaveSynth-Beep
290 ENDPROC
300 :
310 DEF PROCinitialise
320 MODE 12
330 MODE 9
340 OFF
350 COLOUR 0,0,0,128
360 GCOL 2
370 SYS "OS_SWINumberFromString",,"OS_Byte" TO byte%
380 sc%=1
390 blobs%=3
400 DIM x%(blobs%)
410 DIM y%(blobs%)
420 DIM dx%(blobs%)
430 DIM dy%(blobs%)
440 FOR I%=0 TO blobs%
450   x%(I%)=4+RND(1277)
460   y%(I%)=4+RND(1019)
470   dx%(I%)=RND(9)-5
480   dy%(I%)=RND(9)-5
490   size%=24
500   xmin%=5+size%
510   ymin%=5+size%
520   xmax%=1274-size%
530   ymax%=1018-size%
540 NEXT
550 index%=0
560 RESTORE+10
570 READ notes%
580 DIM chan%(notes%)
590 DIM vol%(notes%)
600 DIM pitch%(notes%)
610 DIM time%(notes%)
620 FOR I%=0 TO notes%
630   READ chan%(I%),vol%(I%),pitch%(I%),time%(I%)
640 NEXT
650 ENDPROC
660 DATA 78
670 DATA 3,-15,69,60
```

```
 680 DATA 3,-13,61,60
 690 DATA 3,-13,53,120
 700 DATA 2,-8,5,0
 710 DATA 3,-15,69,60
 720 DATA 2,-8,33,0
 730 DATA 3,-13,61,60
 740 DATA 2,-8,5,0
 750 DATA 3,-13,53,120
 760 :
 770 DATA 2,-8,21,0
 780 DATA 1,-10,0,0
 790 DATA 3,-15,81,60
 800 DATA 2,-8,25,0
 810 DATA 3,-15,73,40
 820 DATA 3,-12,73,20
 830 DATA 2,-8,33,0
 840 DATA 1,-10,0,0
 850 DATA 3,-12,69,120
 860 DATA 2,-8,21,0
 870 DATA 1,-10,0,0
 880 DATA 3,-15,81,60
 890 DATA 2,-8,25,0
 900 DATA 3,-15,73,40
 910 DATA 3,-12,73,20
 920 DATA 2,-8,33,0
 930 DATA 1,-10,0,0
 940 DATA 3,-12,69,100
 950 :
 960 DATA 3,-12,81,20
 970 :
 980 DATA 2,-8,33,0
 990 DATA 1,-10,0,0
1000 DATA 3,-15,101,40
1010 DATA 3,-12,101,20
1020 DATA 2,-8,25,0
1030 DATA 3,-15,97,20
1040 DATA 3,-12,89,20
1050 DATA 3,-12,97,20
1060 DATA 2,-8,21,0
1070 DATA 1,-10,0,0
1080 DATA 3,-15,101,40
1090 DATA 3,-12,81,20
1100 DATA 3,-12,81,40
1110 :
1120 DATA 3,-12,81,20
1130 :
1140 DATA 2,-8,33,0
1150 DATA 1,-10,0,0
1160 DATA 3,-15,101,40
1170 DATA 3,-12,101,20
1180 DATA 2,-8,25,0
1190 DATA 3,-15,97,20
```

```
1200 DATA 3,-12,89,20
1210 DATA 3,-12,97,20
1220 DATA 2,-8,21,0
1230 DATA 1,-10,0,0
1240 DATA 3,-15,101,40
1250 DATA 3,-12,81,20
1260 DATA 3,-12,81,40
1270 :
1280 DATA 3,-12,81,20
1290 :
1300 DATA 2,-8,33,0
1310 DATA 1,-10,0,0
1320 DATA 3,-15,101,20
1330 DATA 3,-12,101,20
1340 DATA 3,-12,101,20
1350 DATA 2,-8,25,0
1360 DATA 3,-15,97,20
1370 DATA 3,-12,89,20
1380 DATA 3,-12,97,20
1390 DATA 2,-8,21,0
1400 DATA 1,-10,0,0
1410 DATA 3,-15,101,20
1420 DATA 3,-12,81,20
1430 DATA 3,-12,81,20
1440 DATA 3,-12,81,40
1450 :
1460 DATA 3,-12,73,20
1470 :
1480 DATA 2,-8,5,0
1490 DATA 1,-10,0,0
1500 DATA 3,-15,69,60
1510 DATA 2,-8,33,0
1520 DATA 3,-13,61,60
1530 DATA 2,-8,5,0
1540 DATA 3,-13,53,120
1550 :
1560 DEF PROCsetchans
1570 VOICES 4
1580 *ChannelVoice 1 Percussion-Soft
1590 *ChannelVoice 2 StringLib-Soft
1600 *ChannelVoice 3 WaveSynth-Beep
1610 *ChannelVoice 4 Percussion-Snare
1620 ENDPROC
1630 :
1640 DEF PROCtitle
1650 RESTORE+0
1660 READ num%
1670 FOR I%=0 TO num%
1680   READ x%,y%,t$
1690   text$=text$+CHR$31+CHR$x%+CHR$y%+t$
1700 NEXT
1710 ENDPROC
```

```
1720 DATA 4
1730 DATA 5,12,Interleaved sound and graphics
1740 DATA 8,14,Q  -  Quiet
1750 DATA 8,16,M  -  Music
1760 DATA 8,18,I  -  Immediate
1770 DATA 5,20,Escape to stop
1780 :
1790 DEF PROCsound
1800 SOUND chan%(index%),vol%(index%),pitch%(index%),10
1810 count%=time%(index%)+mark%
1820 index%+=1
1830 IF index%>notes% index%=0
1840 IF count%>mark% mark%=count% ELSE PROCsound
1850 ENDPROC
1860 :
1870 DEF PROCdraw
1880 x%()=x%()+dx%()
1890 y%()=y%()+dy%()
1900 FOR I%=0 TO blobs%
1910   CIRCLE FILL x%(I%),y%(I%),size%
1920   IF RND(50)=1 dx%(I%)=RND(9)-5:dy%(I%)=RND(9)-5
1930   IF x%(I%)<xmin% dx%(I%)=1 ELSE IF x%(I%)>xmax% dx%(I%)=-1
1940   IF y%(I%)<ymin% dy%(I%)=1 ELSE IF y%(I%)>ymax% dy%(I%)=-1
1950 NEXT
1960 ENDPROC
```

In this example four of the possible eight channels are used. Three are dedicated to the music, and one is used for a gunfire type sound effect. Tune data is stored as four items per note, in the form: channel, amplitude pitch, time to next note. The whole tune is stored in an array for easy, fast access, the pointer to the next note, *index%*, being incremented after each call to PROCsound. When the whole tune has been played the pointer is zeroed so that the tune continuously repeats.

Having a choice of channels, not only gives you the option of different instrument sounds, as in the example, but can also be used for producing chords.

The pitch is in quarter tones with middle C having a value of 53, provided you haven't altered the tuning.

Time is in centi-seconds. You will notice that I talk in terms of time to next note, rather than note length. This latter is fixed at 10 in our example, but if you use your own sound modules, and have a relatively fast tune, the final note length may be long enough for the sound to appear continuous. Also, you will see that in PROCsound I use a recursive call where a note length is zero. This gives almost perfect synchronisation. Only when all eight channels are is use, is there any significant ripple.

The variables *count%* and *mark%* are used together to give a queueing system that is immune to quite large time fluctuations in the game loop. To prove the point, try changing the constant *blobs%* in PROCinitialise to around 20. The animation will slow dramatically, but the tune will hardly be affected. Actual time accuracy of each individual note is the game loop time. In a game that runs at 50 frames per second, this accuracy is two centi-seconds. However, because the time to next note is added on to *mark%* and not *TIME* itself, these errors are not cumulative, and will sound like natural note variations in normal playing.

Silencing the music is a bit of a cheat. What I've done is to force *count%* to such a high value, that *TIME-count%* is never likely to reach zero, let alone pass it, while the game is running. To restore the music, both *count%* and *mark%* are brought back into range.

6.2.2 More voices

The range of voices available by default is distinctly limited, and it is almost certain that you will need to use extra voice modules to get useful results. At the end of this chapter is a utility that enables you to create your own voices for music tracks, as well as giving you an interesting keyboard player.

Once you have your voice module, whether it comes from a source like Music Maker or from a sampler or other commercial source, you need to integrate it into the sound system. To do this you need a procedure like the one below. There are two distinct steps that are taken: first, the module is loaded and the voices initialised, then each channel is assigned a voice but in a slightly different way to our previous example.

```
   DEF PROCattach
   *RMLoad SoundMod
   RESTORE+12
   READ voices%
   SYS "Sound_Configure",voices%,208,48,0,0 TO
oldvc%,oldsm%,oldhd%,oldsh%
   SYS "Sound_Volume",100 TO oldvl%
   SYS "Sound_Tuning",&7000 TO oldtn%
   DIM oldch%(voices%)
   FOR I%=1 TO voices%
      READ voice$
      SYS "Sound_AttachVoice",I%,0 TO ,oldch%(I%)
      SYS "Sound_AttachNamedVoice",I%,voice$
   NEXT
   ENDPROC
   DATA 3
   DATA UserLib-Bell,UserLib-Saw,UserLib-BlockSynth
```

This procedure, as well as attaching the new voices, stores a list of them and a lot of other information about the sound system. This is very important. You must detach your voices when your game exits, then kill your module to release RMA to Risc OS. If you fail to do this, the machine is liable to crash on the next RMtidy, as your, now unused, module area could move or be overwritten, leaving the sound controller up in the air, with pointers aimed at gibberish. The procedure below should take care of this problem.

```
DEF PROCdetach
FOR I%=1 TO voices%
   SYS "Sound_AttachVoice",I%,oldch%(I%)
NEXT
SYS "Sound_Tuning",oldtn%
SYS "Sound_Volume",oldvl%
SYS "Sound_Configure",oldvc%,oldsm%,oldhd%,oldsh%
*RMKill UserVoiceLib
ENDPROC
```

Finally, you will see that all the most significant sound configuration information is also read, and restored afterwards. If you use these two procedures together, you can be assured that you will always receive the sound system in a comprehensible form, and will always return it as you found it.

6.2.3 Voice generator utilitiy

Listing 6.4: MusicMaker

```
 10 REM > MusicMaker
 20 :
 30 ON ERROR PROCerror
 40 PROCinit
 50 PROCassemble
 60 PROCchannels
 70 MODE 12
 80 PROChour(FALSE)
 90 ON ERROR PROCerror
100 PROCdisplay
110 PROCmenu
120 PROCtidy
130 END
140 :
150 :
160 DEF PROCerror
170 VDU 4
180 PRINT REPORT$ " - Press a key"
190 IF GET
```

```
 200 IF NOT INKEY TRUE ENDPROC
 210 ON ERROR OFF
 220 *RMREINIT SoundDMA
 230 *RMREINIT SoundChannels
 240 *RMREINIT SoundScheduler
 250 *RMREINIT WaveSynth
 260 *RMREINIT StringLib
 270 *RMREINIT Percussion
 280 *FX 4
 290 *FX 12
 300 *FX 229
 310 END
 320 :
 330 DEF PROCinit
 340 I%=PAGE+4
 350 REPEAT
 360   I%+=1
 370 UNTIL I%?-1=ASC">"
 380 PRINT $I%
 390 VoiceSize%=&500
 400 VoiceMax%=12
 410 DIM Module% &200,Code% VoiceSize%*VoiceMax%,Work% &40
 420 DIM VoiceName$(VoiceMax%),VoiceEnable%(VoiceMax%)
 430 DIM AmpEnv%(VoiceMax%),FreEnv%(VoiceMax%),WaveTable%(VoiceMax%)
 440 DIM OldVoice$(VoiceMax%),OldChannel%(8)
 450 DIM w%(VoiceMax%,255),par%(VoiceMax%,25),Sine%(255,9)
 460 *Pointer 1
 470 MOUSE OFF
 480 PROChour(TRUE)
 490 FOR I%=0 TO 255
 500   PROChour(I% DIV 25+1)
 510   Sine%(I%,0)=(RND(1)-.5)*16777216
 520   FOR J%=1 TO 9
 530     Sine%(I%,J%)=SIN(PI*I%*J%/128)*16777216
 540   NEXT
 550 NEXT
 560 *KEY 0 |!
 570 *FX 4 1
 580 *FX 229 1
 590 *KEY 13 |!!
 600 ENDPROC
 610 :
 620 DEF PROCassemble
 630 FOR v%=1 TO VoiceMax%
 640   PROChour(10+v%)
 650   VoiceEnable%(v%)=TRUE
 660   FOR I%=0 TO 2 STEP 2
 670     P%=Code%+VoiceSize%*v%-VoiceSize%
 680     [OPT I%
 690     .VoiceBase
 700     B Fill
 710     B Fill
```

```
720      B GateOn
730      B GateOff
740      B Instance
750      LDMFD R13!,{PC}
760      LDMFD R13!,{PC}
770      EQUD VoiceName%(v%)-VoiceBase
780      ;
790      .VoiceName%(v%)
800      =FNbytes(19)
810      ;
820      .LogAmpPtr
830      EQUD 0
840      .WaveBase
850      EQUD 0
860      .AmpEnvPtr
870      EQUD 0
880      .FreEnvPtr
890      EQUD 0
900      .AmpEnv%(v%)
910      =FNbytes(255)
920      .FreEnv%(v%)
930      =FNbytes(255)
940      .WaveTable%(v%)
950      =FNbytes(255)
960      ;
970      .Instance
980      STMFD R13!,{R0-R4}
990      ADR R1,VoiceBase
1000     MOV R0,#AmpEnv%(v%)-VoiceBase
1010     ADD R0,R0,R1
1020     STR R0,AmpEnvPtr
1030     MOV R0,#FreEnv%(v%)-VoiceBase
1040     ADD R0,R0,R1
1050     STR R0,FreEnvPtr
1060     MOV R0,#WaveTable%(v%)-VoiceBase
1070     ADD R0,R0,R1
1080     STR R0,WaveBase
1090     MOV R0,#0
1100     MOV R1,#0
1110     MOV R2,#0
1120     MOV R3,#0
1130     MOV R4,#0
1140     SWI "Sound_Configure"
1150     LDR R0,[R3,#12]
1160     STR R0,LogAmpPtr
1170     LDMFD R13!,{R0-R4,PC}
1180     ;
1190     .GateOn
1200     LDMIA R9,{R1-R8}
1210     LDR R3,AmpEnvPtr
1220     LDR R5,WaveBase
1230     LDR R6,LogAmpPtr
```

```
1240      LDR R7,FreEnvPtr
1250      B FillGate
1260      ;
1270      .Fill
1280      LDMIA R9,{R1-R8}
1290      .FillGate
1300      LDRB R0,[R7],#1
1310      MOV R0,R0,ASL #24
1320      ADD R2,R2,R0,ASR #24
1330      AND R1,R1,#127
1340      LDRB R0,[R3],#1
1350      CMP R0,#255
1360      MOVEQ R4,#0
1370      CMPNE R4,#0
1380      MOVEQ R0,#2
1390      LDMEQFD R13!,{PC}
1400      ;
1410      SUBS R1,R1,R0
1420      MOVMI R1,#0
1430      LDRB R1,[R6,R1,LSL #1]
1440      MOV R1,R1,LSR #1
1450      RSB R1,R1,#127
1460      .FillLoop
1470      ADD R2,R2,R2,LSL #16
1480      LDRB R0,[R5,R2,LSR #24]
1490      SUBS R0,R0,R1,LSL #1
1500      MOVMI R0,#0
1510      STRB R0,[R12],R11
1520      ADD R2,R2,R2,LSL #16
1530      LDRB R0,[R5,R2,LSR #24]
1540      SUBS R0,R0,R1,LSL #1
1550      MOVMI R0,#0
1560      STRB R0,[R12],R11
1570      ADD R2,R2,R2,LSL #16
1580      LDRB R0,[R5,R2,LSR #24]
1590      SUBS R0,R0,R1,LSL #1
1600      MOVMI R0,#0
1610      STRB R0,[R12],R11
1620      ADD R2,R2,R2,LSL #16
1630      LDRB R0,[R5,R2,LSR #24]
1640      SUBS R0,R0,R1,LSL #1
1650      MOVMI R0,#0
1660      STRB R0,[R12],R11
1670      CMP R12,R10
1680      BLT FillLoop
1690      .FillExit
1700      SUB R4,R4,#1
1710      STMIB R9,{R2-R8}
1720      MOV R0,#8
1730      LDMFD R13!,{PC}
1740      ;
1750      .GateOff
```

```
 1760     MOV R0,#0
 1770     .FlushLoop
 1780     STRB R0,[R12],R11
 1790     STRB R0,[R12],R11
 1800     STRB R0,[R12],R11
 1810     STRB R0,[R12],R11
 1820     CMP R12,R10
 1830 BLT FlushLoop
 1840     MOV R0,#1
 1850 LDMFD R13!,{PC}
 1860     ]
 1870   NEXT
 1880 NEXT
 1890 :
 1900 FOR I%=0 TO 2 STEP 2
 1910   P%=Work%
 1920   [ OPT I%
 1930   .Log
 1940   SWI "Sound_SoundLog"
 1950   SWI "Sound_LogScale"
 1960   STRB R0,[R2,R1]
 1970   MOV PC,R14
 1980   .Type
 1990   =FNbytes(7)
 2000   ]
 2010 NEXT
 2020 ENDPROC
 2030 :
 2040 DEF FNbytes(n%)
 2050 P%+=n%
 2060 =I%
 2070 :
 2080 DEF PROCchannels
 2090 RESTORE +0
 2100 FOR v%=1 TO VoiceMax%
 2110   PROChour(22+v%*6)
 2120   OSCLI"KEY "+STR$ v%+CHR$(v%+199)+"|M"
 2130   READ a$,b$
 2140   $VoiceName%(v%)="UserLib-"+LEFT$(a$+STRING$(11,CHR$0),11)
 2150   FOR I%=0 TO 25
 2160     par%(v%,I%)=EVAL("&"+LEFT$(b$,2))
 2170     b$=MID$(b$,3)
 2180   NEXT
 2190   PROCfillwave(v%)
 2200   PROChour(25+v%*6)
 2210   PROCenvelope(v%)
 2220   SYS "Sound_InstallVoice",Code%+v%*VoiceSize%-VoiceSize%,0 TO a
%,OldVoice%(v%)
 2230 NEXT
 2240 Voice%=1
 2250 FOR v%=1 TO 8
 2260   SYS "Sound_AttachVoice",v%,0 TO z%,OldChannel%(v%)
 2270   VOICE v%,$VoiceName%(Voice%)
 2280 NEXT
 2290 VOICES 8
 2300 ENDPROC
 2310 DATA Bell,003A0000230028002D00007F01700804FB0005000000800080835
 2320 DATA HammondOrg,002F120D1D001E0023000F77047F1015FE070500000C0011
0F05
 2330 DATA Ethereal,016112110608080609OA0F6F077F1926FF0002030002003208
35
 2340 DATA Saw,00512E1C0000000100012690B7F2808FF000500000200140865
 2350 DATA Vibraphone,004701000F00010027000C7E016F041AF017060202020200
0435
 2360 DATA ChurchOrg,0771071B000000006000F6F077F1932F800020302020000O
805
 2370 DATA Harpsicord,0139100E150E12100F11007C0170080AC2000203000200000
0805
 2380 DATA Flute,1356152903000100000106E077F160A9E0504000202000000835
 2390 DATA Horn,062E2719151308080200007C036E1E103B000202000200000705
 2400 DATA PipeOrg,083C0014012C011A0100126003683F37FA050300020200000070
5
 2410 DATA BlockSynth,001700170016002801290087A0170092AFF2303000236O211
0B05
 2420 DATA Fantasy,0033001418000800162З0A7C0454332CF5220500401340110053
0
 2430 :
 2440 DEF PROCmenu
 2450 Sel%=14
 2460 REPEAT
 2470   IF Sel%>12 THEN
 2480     OFF
 2490     PROCmenu_bars
 2500     MOUSE TO 560,48
 2510   ENDIF
 2520   MOUSE ON
 2530   *FX 21 9
 2540   REPEAT
 2550     MOUSE x%,y%,b%
 2560     IF y%>288 PROCedit
 2570     x%=x% DIV 208
 2580     IF x%>5 x%=5
 2590     y%=3-(y%+16) DIV 64
 2600     Sel%=x%+y%*6+1
 2610   UNTIL b%=1 OR b%=4
 2620   REPEAT
 2630     MOUSE x%,x%,z%
 2640   UNTIL z%=0
 2650   *FX 21
 2660   MOUSE OFF
 2670   VDU 28,0,31,79,24
 2680   IF Sel%>12 CLS
 2690   CASE Sel% OF
 2700     WHEN 1,2,3,4,5,6,7,8,9,10,11,12:IF b%=1 THEN
 2710        IF Sel%<>Voice% VoiceEnable%(Sel%)=VoiceEnable%(Sel%) EOR
TRUE
 2720        PROCmenu_bars
 2730        ELSE
 2740        PROCvoice(Sel%)
 2750      ENDIF
 2760      WHEN 13:PROCplay
 2770      WHEN 14:PROCload
 2780      WHEN 15:PROCsave
 2790      WHEN 16:PROCrename
 2800      WHEN 17:PROCmodule
 2810      WHEN 18:PROCstar
 2820    ENDCASE
 2830 UNTIL Sel%=19
 2840 ENDPROC
 2850 :
 2860 DEF PROCtidy
 2870 FOR v%=1 TO VoiceMax%
 2880    SYS "Sound_RemoveVoice",0,OldVoice%(v%)
 2890 NEXT
 2900 FOR v%=1 TO 8
 2910    SYS "Sound_AttachVoice",v%,OldChannel%(v%)
 2920 NEXT
 2930 VOICES 1
 2940 *FX 4
 2950 *FX 12
 2960 *FX 229
 2970 MOUSE OFF
 2980 ON
 2990 PRINT
 3000 ENDPROC
 3010 :
 3020 DEF PROCmenu_bars
 3030 VDU 28,0,31,79,24,12
 3040 RESTORE+0
 3050 FOR I%=0 TO VoiceMax%-1
 3060    IF VoiceEnable%(I%+1) THEN
 3070      COLOUR 0
 3080      COLOUR 134
 3090      ELSE
 3100      COLOUR 7
 3110      COLOUR 132
 3120    ENDIF
 3130    PRINT TAB(I% MOD 6*13,I% DIV 6*2+1) SPC 12 STRING$(11,CHR$ 8)
MID$($VoiceName%(I%+1),9)
 3140 NEXT
 3150 COLOUR 0
 3160 COLOUR 134
 3170 FOR I%=0 TO 6
 3180    READ a$
 3190    PRINT TAB(I% MOD 6*13,I% DIV 6*2+5) SPC 12 STRING$(11,CHR$ 8)
a$;
```

```
3200 NEXT
3210 COLOUR 7
3220 COLOUR 128
3230 ENDPROC
3240 DATA Play,Load,Save,Rename,Module,MOS (*),Quit
3250 :
3260 DEF PROCplay
3270 V%=1
3280 OSCLI"FX 11 "+STR$ par%(Voice%,10)
3290 IF par%(Voice%,10) OSCLI"FX 12 "+STR$ par%(Voice%,10)
3300 VDU 26
3310 RESTORE+0
3320 PROCdata_print(7)
3330 PROCprompt
3340 REPEAT
3350   *FX 21
3360   G%=GET
3370   IF INKEY-98:PROCkey(0)
3380   IF INKEY-82:PROCkey(4)
3390   IF INKEY-67:PROCkey(8)
3400   IF INKEY-51:PROCkey(12)
3410   IF INKEY-83:PROCkey(16)
3420   IF INKEY-100:PROCkey(20)
3430   IF INKEY-84:PROCkey(24)
3440   IF INKEY-101:PROCkey(28)
3450   IF INKEY-85:PROCkey(32)
3460   IF INKEY-86:PROCkey(36)
3470   IF INKEY-70:PROCkey(40)
3480   IF INKEY-102:PROCkey(44)
3490   IF INKEY-103:PROCkey(48)
3500   IF INKEY-87:PROCkey(52)
3510   IF INKEY-104:PROCkey(56)
3520   IF INKEY-88:PROCkey(60)
3530   IF INKEY-105:PROCkey(64)
3540   IF INKEY-97:PROCkey(20)
3550   IF INKEY-49:PROCkey(24)
3560   IF INKEY-17:PROCkey(28)
3570   IF INKEY-50:PROCkey(32)
3580   IF INKEY-34:PROCkey(36)
3590   IF INKEY-18:PROCkey(40)
3600   IF INKEY-35:PROCkey(44)
3610   IF INKEY-52:PROCkey(48)
3620   IF INKEY-20:PROCkey(52)
3630   IF INKEY-36:PROCkey(56)
3640   IF INKEY-53:PROCkey(60)
3650   IF INKEY-69:PROCkey(64)
3660   IF INKEY-54:PROCkey(68)
3670   IF INKEY-22:PROCkey(72)
3680   IF INKEY-38:PROCkey(76)
3690   IF INKEY-39:PROCkey(80)
3700   IF INKEY-55:PROCkey(84)
3710   IF INKEY-40:PROCkey(88)
```

```
3720   IF INKEY-56:PROCkey(92)
3730   IF INKEY-57:PROCkey(96)
3740   IF INKEY-94:PROCkey(100)
3750   IF INKEY-89:PROCkey(104)
3760   IF INKEY-47:PROCkey(108)
3770   IF INKEY-121:PROCkey(112)
3780   IF INKEY-42:PROCpitch(-1)
3790   IF INKEY-58:PROCpitch(1)
3800   IF G%>199 AND G%<213 PROCvoice(G%-199):PROCprompt
3810 UNTIL G%=27
3820 PRINT TAB(44,0) SPC20 TAB(45,1) SPC18
3830 ENDPROC
3840 DATA 3,1
3850 DATA 64,26,Upper Keyboard,64,29,Lower Keyboard
3860 DATA 1,1
3870 DATA 4,25,1   2   3       5   6       8   9   0       =   FALSE
3880 DATA 12,29,S   D       G   H   J       L   ;
3890 DATA 7,1
3900 DATA 1,26,tab   Q   W   E   R   T   Y   U   I   O   P   [   ]   \
3910 DATA 10,30,"Z   X   C   V   B   N   M   ,  .   /"
3920 :
3930 DEF PROCedit
3940 LOCAL Env%,Wave%,Flag%
3950 VDU 26
3960 REPEAT
3970   MOUSE X%,Y%,B%
3980   G%=INKEY(0)
3990   IF B%=4 THEN
4000     IF Flag%=0 PROCflag
4010     CASE Flag% OF
4020       WHEN 10:Wave%=FNsetwave
4030       WHEN 9:PROCsetrep
4040       WHEN 1,2,3:PROCsetamp
4050       WHEN 7:Env%=FNsettrem
4060       WHEN 4,5,6:PROCsetfre
4070       WHEN 8:Env%=FNsetvib
4080     ENDCASE
4090     ELSE
4100     Flag%=0
4110   ENDIF
4120 UNTIL Y%<256
4130 IF Wave% PROCfillwave(Voice%)
4140 IF Env% PROCenvelope(Voice%)
4150 IF Env% OR Wave% PROCdisplay
4160 ENDPROC
4170 :
4180 DEF PROCload
4190 LOCAL File%,File$
4200 File$=MID$($VoiceName%(Voice%),9)
4210 PROCname("voice to load",File$)
4220 File%=OPENIN File$
4230 IF File%=0 THEN
```

```
4240    PRINT "No such file - Press a key"
4250    IF GET
4260    ELSE
4270    FOR I%=0 TO 5
4280      Type?I%=BGET#File%
4290    NEXT
4300    Type?5=13
4310    IF $Type<>"Synth" THEN
4320      PRINT "Invalid file type - Press a key"
4330      IF GET
4340      ELSE
4350      FOR I%=0 TO 25
4360        INPUT#File%,par%(Voice%,I%)
4370      NEXT
4380      PROCfillwave(Voice%)
4390      PROCenvelope(Voice%)
4400      $VoiceName%(Voice%)="UserLib-"+LEFT$(File$+STRING$(11,CHR$0)
11)
4410      PROCdisplay
4420    ENDIF
4430    CLOSE#File%
4440 ENDIF
4450 ENDPROC
4460 :
4470 DEF PROCsave
4480 LOCAL File%,File$
4490 File$=MID$($VoiceName%(Voice%),9)
4500 PROCname("voice to save",File$)
4510 File%=OPENOUT File$
4520 IF File% THEN
4530    BPUT#File%,"Synth"
4540    FOR I%=0 TO 25
4550      PRINT#File%,par%(Voice%,I%)
4560    NEXT
4570    CLOSE#File%
4580    $VoiceName%(Voice%)="UserLib-"+LEFT$(File$+STRING$(11,CHR$0),1
1)
4590    PROCdisplay
4600 ENDIF
4610 ENDPROC
4620 :
4630 DEF PROCrename
4640 LOCAL Name$
4650 Name$=MID$($VoiceName%(Voice%),9)
4660 PROCname("new voice",Name$)
4670 $VoiceName%(Voice%)="UserLib-"+LEFT$(Name$+STRING$(11,CHR$0),11)
4680 PROCdisplay
4690 ENDPROC
4700 :
4710 DEF PROCmodule
4720 LOCAL VoiceTotal%,File%,File$
4730 File$="SMOD"
```

```
4740 PROCname("module to save",File$)
4750 FOR v%=1 TO VoiceMax%
4760   IF VoiceEnable%(v%) VoiceTotal%+=1
4770 NEXT
4780 FOR I%=0 TO 2 STEP 2
4790   P%=Module%
4800   [ OPT I%
4810   EQUD 0
4820   EQUD Initialise-Module%
4830   EQUD Finalise-Module%
4840   EQUD 0
4850   EQUD Title-Module%
4860   EQUD Help-Module%
4870   EQUD 0
4880   ;
4890   .Title
4900   EQUS File$+"-UserVoiceLib"
4910   EQUB 0
4920   ALIGN
4930   ;
4940   .Help
4950   EQUS "User Defined Sound Voices"+MID$(TIME$,7,9)
4960   EQUB 0
4970   ALIGN
4980   ;
4990   .VoiceNumbers
5000   =FNbytes(VoiceTotal%-1)
5010   ALIGN
5020   ;
5030   .Initialise
5040   STMFD R13!,{R14}
5050   ADR R2,VoiceCode
5060   ADR R3,VoiceNumbers
5070   ]
5080   FOR v%=1 TO VoiceMax%
5090     IF VoiceEnable%(v%) THEN
5100       [OPT I%
5110       MOV R0,R2
5120       MOV R1,#0
5130       SWI "Sound_InstallVoice"
5140       STRB R1,[R3],#1
5150       ADD R2,R2,#VoiceSize%
5160       ]
5170     ENDIF
5180   NEXT
5190   [OPT I%
5200   LDMFD R13!,{R15}
5210   ;
5220   .Finalise
5230   STMFD R13!,{R14}
5240   ADR R2,VoiceNumbers
5250   ADD R3,R2,#VoiceTotal%
5260   .FinLoop
5270   LDRB R1,[R2],#1
5280   SWI "Sound_RemoveVoice"
5290   CMP R3,R2
5300   BNE FinLoop
5310   LDMFD R13!,{R15}
5320   .VoiceCode
5330   ]
5340 NEXT
5350 File%=OPENOUT File$
5360 FOR I%=Module% TO VoiceCode-1
5370   BPUT# File%,?I%
5380 NEXT
5390 FOR v%=1 TO VoiceMax%
5400   IF VoiceEnable%(v%) THEN
5410     FOR I%=0 TO VoiceSize%-1
5420       BPUT# File%,Code%?(I%+v%*VoiceSize%-VoiceSize%)
5430     NEXT
5440   ENDIF
5450 NEXT
5460 CLOSE# File%
5470 OSCLI "SETTYPE "+File$+" Module"
5480 ENDPROC
5490 :
5500 DEF PROCstar
5510 LOCAL Input$
5520 ON
5530 REPEAT
5540   INPUT"'*" Input$
5550   OSCLI Input$
5560   PRINT "Press a key";
5570   Input$=GET$
5580 UNTIL Input$<>"*"
5590 ENDPROC
5600 :
5610 DEF PROCdisplay
5620 VDU 28,0,23,79,0,12,5
5630 GCOL0,2
5640 RECTANGLE 0,680,512,256
5650 RECTANGLE 0,488,1024,128
5660 RECTANGLE 0,298,1024,128
5670 MOVE 0,808:PLOT 49,512,0
5680 MOVE 0,362:PLOT 49,1024,0
5690 FOR I%=0 TO 9
5700   GCOL0,3:MOVE I%*64+560,796:PRINT"f";I%
5710   GCOL0,6:MOVE I%*64+568,764:PRINT"+";
5720   PRINT CHR$8 CHR$10"-"
5730 NEXT
5740 PROCharmonic(7)
5750 MOVE 0,808
5760 FOR I%=0 TO 255
5770   DRAW I%*2,w%(Voice%,I%)+808
```

```
5780 NEXT
5790 PROCamplitude(7)
5800 PROCfrequency(7)
5810 VDU 4:OFF
5820 RESTORE+0
5830 PROCdata_print(6)
5840 COLOUR 7
5850 PROCpitch(0)
5860 PRINT TAB(8,0)"f";Voice% " - " MID$($VoiceName%(Voice%),9) TAB(8
1)CHR$139" " CHR$138 TAB(57,11);par%(Voice%,10)
5870 OSCLI"FX 11 "+STR$ par%(Voice%,10)
5880 IF par%(Voice%,10) OSCLI"FX 12 "+STR$ par%(Voice%,10)
5890 ENDPROC
5900 DATA 3,9
5910 DATA 0,0,Playing
5920 DATA 0,1,Octave,10,11,Waveshape,50,11,Repeat,68,13,Tremelo
5930 DATA 66,14,Depth  Speed,10,17,Amplitude Envelope,68,19,Vibrato
5940 DATA 66,20,Depth  Speed,10,23,Pitch Envelope
5950 DATA 6,5
5960 DATA 62,11,+,68,15,+      +,68,21,+      +
5970 DATA 60,11,-,68,16,-      -,68,22,-      -
5980 :
5990 DEF PROCkey(T%)
6000 SOUND V%,-15,T%+par%(Voice%,25),40:V%=V%MOD8+1
6010 ENDPROC
6020 :
6030 DEF PROCvoice(n%)
6040 IF VoiceEnable%(n%) THEN
6050   IF Voice%<>n%
6060   Voice%=n%
6070   FOR v%=1 TO 8
6080     VOICE v%,$VoiceName%(Voice%)
6090   NEXT
6100   PROCdisplay
6110 ENDIF
6120 ENDIF
6130 ENDPROC
6140 :
6150 DEF PROCpitch(n%)
6160 IF n%>0 THEN
6170   IF par%(Voice%,25)<101 par%(Voice%,25)+=48
6180   ELSE
6190   IF n%<0 THEN
6200     IF par%(Voice%,25)>5 par%(Voice%,25)-=48
6210   ENDIF
6220 ENDIF
6230 PRINT TAB(12,1);par%(Voice%,25) DIV6-8" "
6240 REPEAT
6250 UNTIL NOT INKEY-42 AND NOT INKEY-58
6260 ENDPROC
6270 :
6280 DEF PROCflag
```

```
 6290 LOCAL a%
 6300 a%=X% DIV4
 6310 IF Y%>636 THEN
 6320   IF X%>512 AND Y%>696 AND Y%<768 Flag%=10 ELSE IF X%>928 AND Y%
<672 Flag%=9
 6330   ELSE
 6340   IF X%<1024 THEN
 6350     IF Y%>488 AND Y%<620 THEN
 6360       IF a%-par%(Voice%,12)<par%(Voice%,14)-a% Flag%=1 ELSE IF a
%-par%(Voice%,14)>par%(Voice%,16)-a% Flag%=3 ELSE Flag%=2
 6370       ELSE
 6380       IF Y%>296 AND Y%<424 THEN
 6390         IF a%<par%(Voice%,21)-a% Flag%=4 ELSE IF a%-par%(Voice%,
21)>par%(Voice%,16)-a% Flag%=6 ELSE Flag%=5
 6400       ENDIF
 6410     ENDIF
 6420   ELSE
 6430     IF X%>1036 THEN
 6440       IF Y%>482 AND Y%<544 Flag%=7 ELSE IF Y%>290 AND Y%<352 Fla
g%=8
 6450     ENDIF
 6460   ENDIF
 6470 ENDIF
 6480 ENDPROC
 6490 :
 6500 DEF FNsetwave
 6510 PROCharmonic(0)
 6520 N%=(X%-544)DIV64
 6530 IF N%>9 N%=9
 6540 IF Y%<732 AND par%(Voice%,N%)>0 par%(Voice%,N%)=par%(Voice%,N%)-
1 ELSE IF par%(Voice%,N%)<127 par%(Voice%,N%)=par%(Voice%,N%)+1
 6550 *FX 21
 6560 PROCharmonic(7)
 6570 =TRUE
 6580 :
 6590 DEF PROCsetrep
 6600 COLOUR 7
 6610 IF X%<992 THEN
 6620   IF par%(Voice%,10)>6 par%(Voice%,10)-=1 ELSE par%(Voice%,10)=0
 6630   ELSE
 6640   IF par%(Voice%,10)<98 AND par%(Voice%,10) par%(Voice%,10)+=1 E
LSE par%(Voice%,10)=6
 6650 ENDIF
 6660 PRINT TAB(57,11);par%(Voice%,10)" "
 6670 PROCwait
 6680 ENDPROC
 6690 :
 6700 DEF PROCsetamp
 6710 PROCamplitude(0)
 6720 PROCfrequency(0)
 6730 IF Flag%=1 PROCsetbars(488,par%(Voice%,12),par%(Voice%,11)) ELSE
 IF Flag%=2 PROCsetbars(488,par%(Voice%,14),par%(Voice%,13)) ELSE
```

```
setbars(488,par%(Voice%,16),par%(Voice%,15))
 6740 IF par%(Voice%,16)<12 par%(Voice%,16)=12
 6750 IF par%(Voice%,21)>par%(Voice%,16) par%(Voice%,21)=par%(Voice%,1
6)-1
 6760 IF par%(Voice%,12)=0 par%(Voice%,12)=1
 6770 IF par%(Voice%,14)<=par%(Voice%,12) par%(Voice%,14)=par%(Voice%,
12)+1 ELSE IF par%(Voice%,14)>=par%(Voice%,16) par%(Voice%,14)=par%(Vo
ice%,16)-1
 6780 PROCfrequency(7)
 6790 PROCamplitude(7)
 6800 ENDPROC
 6810 :
 6820 DEF FNsettrem
 6830 PROCamplitude(0)
 6840 IF X%<1152 THEN
 6850   IF Y%<508 THEN
 6860     IF par%(Voice%,17)>0 par%(Voice%,17)-=1
 6870     ELSE
 6880     IF par%(Voice%,17)<100 par%(Voice%,17)+=1
 6890   ENDIF
 6900   ELSE
 6910   IF Y%<508 THEN
 6920     IF par%(Voice%,18)<40 par%(Voice%,18)+=1
 6930     ELSE
 6940     IF par%(Voice%,18)>2 par%(Voice%,18)-=1
 6950   ENDIF
 6960 ENDIF
 6970 PROCamplitude(7)
 6980 PROCwait
 6990 =TRUE
 7000 :
 7010 DEF PROCsetfre
 7020 LOCAL Null%
 7030 IF X%<12 Null%=par%(Voice%,16)
 7040 PROCfrequency(0)
 7050 IF Flag%=4 PROCsetbars(360,Null%,par%(Voice%,19)) ELSE IF Flag%=
5 PROCsetbars(360,par%(Voice%,21),par%(Voice%,20)) ELSE PROCsetbars(36
0,Null%,par%(Voice%,22))
 7060 IF par%(Voice%,21)<2 par%(Voice%,21)=2 ELSE IF par%(Voice%,21)>=
par%(Voice%,16) par%(Voice%,21)=par%(Voice%,16)-1
 7070 PROCfrequency(7)
 7080 ENDPROC
 7090 :
 7100 DEF FNsetvib
 7110 PROCfrequency(0)
 7120 IF X%<1152 THEN
 7130   IF Y%<316 THEN
 7140     IF par%(Voice%,23)>0 par%(Voice%,23)-=1
 7150     ELSE
 7160     IF par%(Voice%,23)<100 par%(Voice%,23)+=1
 7170   ENDIF
 7180   ELSE
 7190   IF Y%<316 THEN
 7200     IF par%(Voice%,24)<36 par%(Voice%,24)+=1
 7210     ELSE
 7220     IF par%(Voice%,24)>2 par%(Voice%,24)-=1
 7230   ENDIF
 7240 ENDIF
 7250 PROCfrequency(7)
 7260 PROCwait
 7270 =TRUE
 7280 :
 7290 DEF PROCsetbars(a%,RETURN H%,RETURN V%)
 7300 Env%=TRUE
 7310 H%=X% DIV 4
 7320 V%=Y%-a%
 7330 IF a%=488 THEN
 7340   IF V%>127 V%=127 ELSE IF V%<0 V%=0
 7350   ELSE
 7360   IF V%>64 V%=64 ELSE IF V%<-64 V%=-64
 7370 ENDIF
 7380 IF H%>255 H%=255
 7390 ENDPROC
 7400 :
 7410 DEF PROCfillwave(Voice%)
 7420 FOR B%=0 TO 255
 7430   A%=0
 7440   FOR J%=0 TO 9
 7450     IF par%(Voice%,J%) A%+=par%(Voice%,J%)*Sine%(B%,J%)
 7460   NEXT
 7470   w%(Voice%,B%)=A%>24
 7480   C%=WaveTable%(Voice%)
 7490   CALL Log
 7500 NEXT
 7510 ENDPROC
 7520 :
 7530 DEF PROCenvelope(Voice%)
 7540 M%=AmpEnv%(Voice%)
 7550 level=0
 7560 PROCamp_env(par%(Voice%,11)/par%(Voice%,12),par%(Voice%,12))
 7570 PROCamp_env((par%(Voice%,13)-par%(Voice%,11))/(par%(Voice%,14)-p
ar%(Voice%,12)),par%(Voice%,14)-par%(Voice%,12))
 7580 S%=par%(Voice%,14)+par%(Voice%,18)
 7590 sus=(par%(Voice%,13)-par%(Voice%,15))/(par%(Voice%,16)-par%(Voic
e%,14))*4
 7600 WHILE S%<par%(Voice%,16)
 7610   PROCamp_env(-sus-par%(Voice%,17)/par%(Voice%,18),par%(Voice%,18))
 7620   S%+=par%(Voice%,18)
 7630   IF S%<par%(Voice%,16) PROCamp_env(0,par%(Voice%,18)):S%+=par%(
Voice%,18)
 7640   IF S%<par%(Voice%,16) PROCamp_env(par%(Voice%,17)/par%(Voice%,
18),par%(Voice%,18)):S%+=par%(Voice%,18)
 7650   IF S%<par%(Voice%,16) PROCamp_env(0,par%(Voice%,18)):S%+=par%(
Voice%,18)
```

```
 7660 ENDWHILE
 7670 IF M%-AmpEnv%(Voice%)<256 ?M%=255 ELSE AmpEnv%(Voice%)?255=255
 7680 M%=FreEnv%(Voice%)
 7690 PROCfre_env(par%(Voice%,19),1)
 7700 PROCfre_env((par%(Voice%,20)-par%(Voice%,19))/(par%(Voice%,21)-1
),par%(Voice%,21))
 7710 S%=par%(Voice%,21)+par%(Voice%,24)
 7720 sus=(par%(Voice%,20)-par%(Voice%,22))/(par%(Voice%,16)-par%(Voic
e%,21))*2
 7730 WHILE S%<par%(Voice%,16)
 7740   PROCfre_env(-sus-par%(Voice%,23)/par%(Voice%,24),par%(Voice%,24))
 7750   S%+=par%(Voice%,24)
 7760   IF S%<par%(Voice%,16) PROCfre_env(par%(Voice%,23)/par%(Voice%,
24),par%(Voice%,24)):S%+=par%(Voice%,24)
 7770 ENDWHILE
 7780 IF M%-FreEnv%(Voice%)>255 THEN
 7790   M%?255=0
 7800   ELSE
 7810   WHILE M%-FreEnv%(Voice%)<256
 7820     ?M%=0
 7830     M%+=1
 7840   ENDWHILE
 7850 ENDIF
 7860 ENDPROC
 7870 :
 7880 DEF PROCamp_env(s,n%)
 7890 FOR I%=1 TO n%
 7900   level+=s
 7910   IF level>127 ?M%=0 ELSE IF level<0 ?M%=127 ELSE ?M%=127-level
 7920   M%+=1
 7930 NEXT
 7940 ENDPROC
 7950 :
 7960 DEF PROCfre_env(s%,n%)
 7970 FOR I%=1 TO n%
 7980   ?M%=s%
 7990   M%+=1
 8000 NEXT
 8010 ENDPROC
 8020 :
 8030 DEF PROCharmonic(Col%)
 8040 GCOL 0,Col%
 8050 FOR I%=0 TO 9
 8060   MOVE I%*64+576,822:PLOT 1,0,par%(Voice%,I%)
 8070 NEXT
 8080 ENDPROC
 8090 :
 8100 DEF PROCamplitude(Col%)
 8110 GCOL0,Col%:MOVE 0,488:PLOT 1,par%(Voice%,12)*4,par%(Voice%,11)
 8120 PROCpeg
 8130 PLOT 1,(par%(Voice%,14)-par%(Voice%,12))*4,(par%(Voice%,13)-par%
(Voice%,11))
```

```
 8140 PROCpeg
 8150 S%=par%(Voice%,14)+par%(Voice%,18)
 8160 sus=(par%(Voice%,13)-par%(Voice%,15))/(par%(Voice%,16)-par%(Voic
e%,14))*4
 8170 WHILE S%<par%(Voice%,16)
 8180    PLOT 1,par%(Voice%,18)*4,-sus*par%(Voice%,18)-par%(Voice%,17)
 8190    S%+=par%(Voice%,18)
 8200    IF S%<par%(Voice%,16) PLOT 1,par%(Voice%,18)*4,0:S%+=par%(Voic
e%,18)
 8210    IF S%<par%(Voice%,16) PLOT 1,par%(Voice%,18)*4,par%(Voice%,17)
:S%+=par%(Voice%,18)
 8220    IF S%<par%(Voice%,16) PLOT 1,par%(Voice%,18)*4,0:S%+=par%(Voic
e%,18)
 8230 ENDWHILE
 8240 PLOT 1,(par%(Voice%,16)-S%+par%(Voice%,18))*4,0
 8250 PROCpeg
 8260 ENDPROC
 8270 :
 8280 DEF PROCfrequency(Col%)
 8290 GCOL0,Col%:MOVE 0,360+par%(Voice%,19)
 8300 PROCpeg
 8310 PLOT 1,par%(Voice%,21)*4,par%(Voice%,20)-par%(Voice%,19)
 8320 PROCpeg
 8330 S%=par%(Voice%,21)+par%(Voice%,24)
 8340 sus=(par%(Voice%,20)-par%(Voice%,22))/(par%(Voice%,16)-par%(Voic
e%,21))*2
 8350 WHILE S%<par%(Voice%,16)
 8360    PLOT 1,par%(Voice%,24)*4,-sus*par%(Voice%,24)-par%(Voice%,23)
 8370    S%+=par%(Voice%,24)
 8380    IF S%<par%(Voice%,16) PLOT 1,par%(Voice%,24)*4,par%(Voice%,23)
:S%+=par%(Voice%,24)
 8390 ENDWHILE
 8400 PLOT 1,(par%(Voice%,16)-S%+par%(Voice%,24))*4,0
 8410 PROCpeg
 8420 ENDPROC
 8430 :
 8440 DEF PROCpeg
 8450 IF Col% GCOL0,6 ELSE GCOL0,0
 8460 PLOT 0,0,16:PLOT 1,0,-32:PLOT 1,2,0:PLOT 1,0,32:PLOT 1,-2,0:PLOT
0,0,-16
 8470 GCOL0,Col%
 8480 ENDPROC
 8490 :
 8500 DEF PROCprompt
 8510 RESTORE+0
 8520 PROCdata_print(7)
 8530 ENDPROC
 8540 DATA 7,1
 8550 DATA 44,0,F1 - F12  For Voices,45,1,Escape For Options
 8560 :
 8570 DEF PROCname(Text$,RETURN Old$)
 8580 LOCAL x%,Input$
 8590 x%=17+LEN Text$
 8600 ON
 8610 REPEAT
 8620   PRINT"'Enter name of " Text$ " > " Old$;
 8630   INPUT TAB(x%,VPOS) "" Input$
 8640 UNTIL LEN Input$<11
 8650 IF Input$>"" Old$=Input$
 8660 ENDPROC
 8670 :
 8680 DEF PROCdata_print(Col%)
 8690 REPEAT
 8700   READ C%,J%
 8710   COLOUR C%
 8720   FOR I%=0 TO J%
 8730     READ X%,Y%,a$
 8740     PRINT TAB(X%,Y%)a$;
 8750   NEXT
 8760 UNTIL C%=Col%
 8770 ENDPROC
 8780 :
 8790 DEF PROCwait
 8800 LOCAL t%
 8810 t%=TIME
 8820 REPEAT UNTIL TIME-t%>10
 8830 ENDPROC
 8840 :
 8850 DEF PROChour(p%)
 8860 IF p%=FALSE SYS "Hourglass_Smash" ELSE IF p%=TRUE SYS "Hourglass
_On" ELSE SYS "Hourglass_Percentage",p%
 8870 ENDPROC
```

Operation of MusicMaker is fairly straightforward. The voices it creates fully implement extended pitch control, but only partially implement extended amplitude. Duration is implemented up to the envelope length only, infinite sound is not possible.

❑ Clicking on Play will clear the other options and display your synthesiser keyboard. Two strips of letter keys are used, laid out piano style. On this keyboard white is for white notes and red for black notes.

❑ The line ZXCVB has the letter Z as the note C and the line QWERTY with letter R as the note C one octave above Z. Up to eight notes will sound at any time. If a ninth note is pressed the oldest one is lost. Pressing several keys at the same time won't always realise this number though as the negative inkey system can get confused.

❑ You can step up and down octaves with the up and down cursor keys. Octave settings are –8, 0, 8. The change is instantaneous so you don't need to interrupt your playing when changing octaves. With –8 and no frequency envelope in use the Z key is middle C.

❑ Function keys 1 to 12 will change voices reasonably quickly, usually fast enough for you to do this between notes while playing.

❑ Pressing Escape returns to the edit mode. All cyan parts of the display are responsive to the mouse by dragging or clicking on Select. Generally yellow is used for information, green for outline limits, and white indicates values that can be altered.

❑ Clicking Select over any of the 12 listed voice names duplicates the function key action. This not only selects voices for playing but also for loading, saving and renaming.

❑ Clicking Adjust on the voices will toggle them as enabled or disabled. Disabled voices are not selectable in play and will not be saved in the sound module. You can't disable the currently selected voice.

❑ You can save the current voice by clicking on Save, or load a previously saved one using Load. Prompts are given and *filename* is the displayed voice name. Saving and loading is to the currently selected function key number.

❑ Saving a voice stores all its parameters, including the current octave. When loading a voice a check is made for file validity. Simply pressing Return when prompted for a filename will save or load to the currently displayed name.

❑ Clicking on Rename changes the name of the current voice without altering any of its parameters.

❑ Clicking on Module prompts for a filename in the same way as Save and Load, and then creates a re-locatable module of all the currently enabled voices. Therefore a library of from 1 to 12 different voices can be assembled.

❑ Octave and Repeat are not stored in the module as these are a feature of the keyboard not the voices.

❑ Editing the current voice is done simply by moving the mouse to the appropriate part of the display and clicking or dragging on the cyan parts.

❑ When dragging envelope shape markers some of the white envelope outlines may go outside their frames. This does no harm but may give unpredictable results. Similarly the harmonic content can be set too high. This is easily identified by sharp re-entrant spikes in the waveshape and a sudden harshening of the sound.

❑ The timbre is altered by summing sinewaves at the fundamental frequency and its main harmonics f1-f9. Random noise is added with f0 to give wind type sounds.

❑ The amplitude envelope has three variable nodes, all of which can be changed vertically in amplitude and horizontally in time. These are attack, decay and sustain. Tremolo depth and speed modify the sustain phase only. It isn't usually possible to get zero tremolo amplitude, due to the way the sustain is created in chunks. However, setting a very fast tremolo time will tend to smooth out the steps.

❑ Careful selection of repeat time in conjunction with attack amplitude can be used to get almost continuous smooth sound.

❑ The frequency envelope has a further three nodes. The attack and sustain nodes can only be altered vertically by frequency deviation, whereas decay can be altered horizontally in time as well. Vibrato depth and speed only modify the sustain phase.

7

Arcade Games

7.1 Alien Zapping

Until the advent of reasonably cheap processors and VDUs, this type of game simply couldn't exist. Unlike all other types, they have no direct parallel in the real world being, in their simplest form, a very fast reaction test. No real world shoot-'em-up can possibly compare with the sheer speed and volume of targets these games present.

7.1.1 Movement tables

The most notable feature of this class of game these days is the smooth movement of the attacking aliens as they loop, dip and dive. At first sight, programming this looks a daunting prospect, but is in fact quite easy. On close examination, you will find that in most games, only a few fixed movement patterns are in fact used. These are most easily programmed using look-up tables. These will simply consist of lists of X,Y coordinates, to be used within the main game loop interleaved with any other movement or animation. Bear in mind that these movement tables can be used in both directions, and can also be used as an offset rather than absolute plotting positions. This latter arrangement gives far more flexibility as it allows complex manoeuvres to be initiated from any point of the display.

The only problem that remains now, is that of producing the movement table itself. There are several ways this can be achieved. The obvious one is to draw the movement out onto a sheet of graph paper set out as one division per graphic unit, then pick off evenly spaced coordinate pairs. These can be put in lines of data for reading into the array when the program runs. This can be extremely tedious, but is potentially very accurate, and you can immediately see exactly what you will get.

Figure 7.1 shows how this would look. For clarity I've not drawn in the background grid. As the objects following the path will probably be moving quite quickly, you can afford to space out the plotting points out quite a lot. Changing this spacing can be used to good effect. In the example, movement will seem slower going upwards.

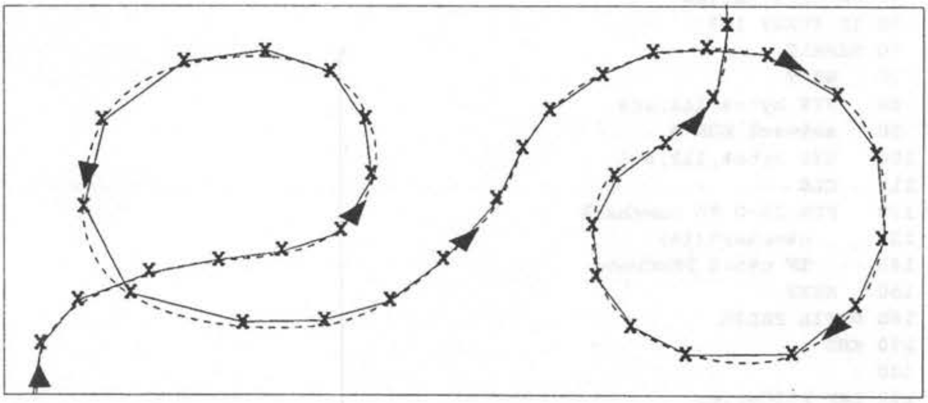
Figure 7.1: Tracking movement

Another way of handing the problem is to write a small program that uses the mouse to draw the movement shape on-screen, while mouse clicks store the coordinates. This can be a bit hit-and-miss, and, in reality, is only a little faster than hand drawing. Apart from other considerations you will still need to turn the coordinates into lines of data.

For the most elegant solution, you will almost always find that you can break the movement down into groups of circles, arcs and straight lines. Coordinates from these, taken during game initialisation can then be dropped straight into the movement tables, rather than being used for screen plotting. As well as being a neat, accurate solution, this method doesn't require any lines of data, so will end up more efficient from all points of view.

You can see a practical implementation of this in Listing 7.1. Character printing has been used so that you can easily see which object is moving where. As it stands only nine objects can be moved this way, without screen jitter developing. You would, of course, be able to have more if you use sprite plotting. However, if you were to use system sprites, you can't actually have many objects without jitter, which just underlines the need to stick to directly addressed user sprites.

Listing 7.1: Movement using look-up tables

```
 10 REM > Movement
 20 :
 30 ON ERROR PROCerror:END
 40 PROCinitialise
 50 IF INKEY 100
 60 REPEAT
 70   WAIT
 80   SYS byte%,113,sc%
 90   sc%=sc% EOR 3
100   SYS byte%,112,sc%
110   CLS
120   FOR I%=0 TO numchar%
130     n%=char%(I%)
140     IF n%>=0 PROCmove
150   NEXT
160 UNTIL FALSE
170 END
180 :
190 DEF PROCerror
200 MODE 12
210 IF ERR<>17 PRINT REPORT$ " @ ";ERL
220 ENDPROC
230 :
240 DEF PROCinitialise
250 MODE 12
260 MODE 9
270 COLOUR 0,128,0,0
280 PRINT TAB(8,10) "Movement Table Example"
290 PRINT TAB(9,13) "Press Escape to stop"
300 VDU 5
310 SYS "OS_SWINumberFromString",,"OS_Byte" TO byte%
320 sc%=1
330 maxpoints%=200
340 DIM x%(maxpoints%)
350 DIM y%(maxpoints%)
360 numchar%=8
370 DIM char%(numchar%)
380 char%()=-1
390 char%(0)=0
400 A%=ASC"A"
410 x%=0
420 y%=512
430 mark%=-1
440 PROCline(40)
```

```
450 PROCcircle(128,1)
460 PROCline(10)
470 PROCcircle(128,-1)
480 PROCline(55)
490 ENDPROC
500 :
510 DEF PROCline(n%)
520 FOR I%=0 TO n%
530   mark%+=1
540   x%+=12
550   x%(mark%)=x%
560   y%(mark%)=y%
570 NEXT
580 ENDPROC
590 :
600 DEF PROCcircle(rad%,dir%)
610 start=-PI/2
620 step=PI/20
630 end=start+PI*2+step
640 FOR I=start TO end STEP step
650   mark%+=1
660   x%(mark%)=x%+COS(I*dir%)*rad%
670   y%(mark%)=y%+SIN(I*dir%)*rad%+rad%*dir%
680 NEXT
690 ENDPROC
700 :
710 DEF PROCmove
720 MOVE x%(n%),y%(n%)
730 VDU A%+I%
740 IF n%<mark% char%(I%)+=1 ELSE char%(I%)=0
750 IF n%=4:IF I%<numchar% char%(I%+1)=0
760 ENDPROC
```

A point that needs a little explanation with this example, is that there are, in fact, two tables. The first consists of the parallel arrays *x%()* and *y%()*. These contain the actual plotting coordinates, and is the true movement table. The other table is *char%()*. This is a list of pointers into the movement table, so that each individual character can progress along the movement table independently. In a more sophisticated version objects could then traverse the table at different speeds and even in opposite directions.

Finally, whichever method you use to generate the tables, you can always create them in a separate program and save them as data files. These can then be quickly and efficiently loaded into the final game.

7.1.2 Formations

Usually, along with one or two independent attackers, there is a large group of aliens in a holding pattern with very little movement. This presents a very real problem, as apart from using your own ARM code sprite system, you'll have great difficulty moving 40 to 50 sprites round the screen. The simplest solution is to use a special pre-defined sprite that consists of a whole block of aliens. If you detect a collision with one of these aliens you rub it out of the sprite, and initiate an explosion film animation sequence with a free moving sprite, plotted on top of that alien's position in the main sprite.

The rubbing out process can be performed by clearing individual bits from the sprite definition. As it will be masked by the explosion sequence, it can take several screen refreshes. This will prevent the erasure routine slowing things down significantly. Another way to erase the alien is to temporarily redirect output to the sprite then use the rectangle fill with the graphic colour set to the background to overprint the offending creature.

Listing 7.2 shows all these points put together. The main sprite uses a film animation of eight sprites. Erasure is done by rectangle filling, and the erasure procedure is masked by an explosion sprite. For best effects this latter should also be film animated.

Listing 7.2: Formation

```
   10 REM > Formation
   20 :
   30 ON ERROR PROCerror:END
   40 PROCinitialise
   50 PROCbuild
   60 PROCstart
   70 REPEAT
   80   FOR J%=0 TO mark%
   90     WAIT
  100     SYS byte%,113,sc%
  110     sc%=sc% EOR 3
  120     SYS byte%,112,sc%
  130     CLS
  140     SYS sprite%,put%,area%,frame%(L%),x%(J%),y%(J%)
  150     IF INKEY-74 AND (J% AND 7)=0 PROCshoot
  160     IF INKEY-98 PROCgun(-4)
  170     IF INKEY-67 PROCgun(4)
  180     SYS sprite%,put%,area%,gun%,gx%,gy%
  190     IF wipef%(L%) PROCwipe(wipex%(L%),wipey%(L%),wipef%(L%))
  200     FOR I%=0 TO pins%
  210        IF wipef%(I%) SYS sprite%,put%,area%,blot%,wipex%(I%)+x%(J
%),wipey%(I%)+y%(J%)
  220        IF piny%(I%) PROCpin(pinx%(I%),piny%(I%),wipex%(I%),wipey%
(I%),wipef%(I%))
  230      NEXT
  240      IF L%=0 OR L%=pins% D%=-D%
  250      L%+=D%
  260    NEXT
  270 UNTIL FALSE
  280 END
  290 :
  300 DEF PROCerror
  310 VOICE 1,"WaveSynth-Beep"
  320 MODE 12
  330 *FX 9 25
  340 *FX 10 25
  350 IF ERR<>17 PRINT REPORT$ " @ ";ERL
  360 ENDPROC
  370 :
  380 DEF PROCinitialise
  390 MODE 12
  400 MODE 9
  410 mode%=MODE
  420 OFF
  430 COLOUR 0,128,128,128
  440 *FX 9 2
  450 *FX 10 2
  460 PRINT TAB(12,6) "Alien formation"
  470 PRINT TAB(2,10) "Z - left   X - right   Return - fire"
  480 PRINT TAB(14,14) "Please wait"
  490 PRINT TAB(10,18) "Press Escape to exit"
  500 SYS "OS_SWINumberFromString",,"OS_Byte" TO byte%
  510 sc%=1
  520 SYS "OS_SWINumberFromString",,"OS_SpriteOp" TO sprite%
  530 DIM block% 19
  540 block%!0=4
  550 block%!4=5
  560 block%!8=-1
  570 SYS "OS_ReadVduVariables",block%,block%+12
  580 xeig%=block%!12
  590 yeig%=block%!16
  600 size%=&1F000
  610 DIM area% size%
  620 area%!0=size%
  630 area%!4=0
  640 area%!8=16
  650 init%=256+9
  660 def%=256+15
  670 select%=256+24
  680 put%=512+34
  690 swap%=512+60
  700 SYS sprite%,init%,area%
  710 DIM frame%(7)
  720 :
  730 xmin%=8
  740 xmax%=1244
  750 ymin%=0
  760 ymax%=800
  770 asize%=24
  780 ah%=12
  790 av%=4
  800 ax2%=asize%*2
  810 ax3%=asize%*3
  820 xsize%=ah%*ax3%-asize% DIV 2
  830 ysize%=av%*ax3%-asize%
  840 xpos%=(1280-xsize%)>>1
  850 ypos%=(1024-ysize%)>>1
  860 :
  870 maxpoints%=200
  880 DIM x%(maxpoints%)
  890 DIM y%(maxpoints%)
  900 mark%=-1
  910 PROCcircle(64,1)
  920 PROCcircle(64,-1)
  930 :
  940 gwide%=24
  950 ghigh%=32
  960 gx%=(xmax%-xmin%-gwide%)>1
  970 gy%=ymin%+16
  980 pins%=7
  990 DIM pinx%(pins%)
 1000 DIM piny%(pins%)
 1010 DIM wipex%(pins%)
 1020 DIM wipey%(pins%)
 1030 DIM wipef%(pins%)
 1040 ENDPROC
 1050 :
 1060 DEF PROCcircle(rad%,dir%)
 1070 step=PI/50
 1080 start=step*2
 1090 end=start+PI*2-step*2
 1100 FOR I=start TO end STEP step
 1110   mark%+=1
 1120   x%(mark%)=xpos%+(COS(I)*rad%-rad%)*dir%
 1130   y%(mark%)=ypos%+SIN(I)*rad%/3
 1140 NEXT
 1150 ENDPROC
 1160 :
 1170 DEF PROCbuild
 1180 a=PI/2
 1190 p=PI*2
 1200 FOR L%=0 TO pins%
 1210   SYS sprite%,def%,area%,"S"+STR$L%,0,xsize%>>xeig%,ysize%>>yeig
%,mode%
```

```
 1220    SYS sprite%,select%,area%,"S"+STR$L% TO ,,frame%(L%)
 1230    SYS sprite%,swap%,area%,frame%(L%)
 1240    PROCdraw(asize%+4,asize%+4,L%+6)
 1250 NEXT
 1260 :
 1270 SYS sprite%,def%,area%,"SG",0,(gwide%+4)>>xeig%,(ghigh%+4)>>yeig%,
mode%
 1280 SYS sprite%,select%,area%,"SG" TO ,,gun%
 1290 SYS sprite%,swap%,area%,gun%
 1300 GCOL 7
 1310 MOVE 0,0
 1320 MOVE gwide%,0
 1330 PLOT 85,gwide% DIV 2,ghigh%
 1340 :
 1350 SYS sprite%,def%,area%,"SB",0,(ax2%+12)>xeig%,(ax2%+12)>yeig%,mode%
 1360 SYS sprite%,select%,area%,"SB" TO ,,blot%
 1370 SYS sprite%,swap%,area%,blot%
 1380 GCOL 8
 1390 CIRCLE FILL asize%+4,asize%+4,asize%+4
 1400 ENDPROC
 1410 :
 1420 DEF PROCdraw(x%,y%,l%)
 1430 FOR K%=0 TO av%-1
 1440    FOR J%=0 TO ah%-1
 1450       FOR I%=3 TO 1 STEP-1
 1460          dx%=x%+J%*ax3%
 1470          dy%=y%+K%*ax3%+I%
 1480          GCOL I%
 1490          MOVE dx%,dy%
 1500          MOVE dx%+COS(l%/p+I%/a)*asize%,dy%-SIN(l%/p+I%/a)*asize%
 1510          PLOT &B5,dx%-COS(l%/p+I%/a)*asize%,dy%-SIN(l%/p+I%/a)*asiz
e%
 1520       NEXT
 1530    NEXT
 1540 NEXT
 1550 ENDPROC
 1560 :
 1570 DEF PROCstart
 1580 SYS sprite%,swap%,area%
 1590 OFF
 1600 next%=0
 1610 D%=1
 1620 L%=1
 1630 GCOL 7
 1640 PRINT TAB(10,14) "Press a key to start"
 1650 VDU 7
 1660 IF GET
 1670 VOICE 1,"Percussion-Medium"
 1680 ENDPROC
 1690 :
 1700 DEF PROCshoot
 1710 IF piny%(next%) ENDPROC
```

```
1720 pinx%(next%)=gx%+gwide% DIV 2
1730 piny%(next%)=gy%+ghigh%+4
1740 POINT pinx%(next%),piny%(next%)
1750 next%+=1
1760 IF next%=pins% next%=0
1770 ENDPROC
1780 :
1790 DEF PROCgun(x%)
1800 IF gx%+x%<xmin% OR gx%+x%>xmax% ENDPROC
1810 gx%+=x%
1820 ENDPROC
1830 :
1840 DEF PROCpin(x%,RETURN y%,RETURN wx%,RETURN wy%,RETURN wf%)
1850 POINT x%,y%
1860 y%+=4
1870 IF y%>ymax% y%=0:ENDPROC
1880 IF POINT(x%,y%) AND 7 PROCcollide
1890 ENDPROC
1900 :
1910 DEF PROCcollide
1920 wx%=x%-x%(J%)
1930 wx%-=(wx% MOD ax3%)
1940 wy%=y%-y%(J%)
1950 wy%-=(wy% MOD ax3%)
1960 wf%=1
1970 y%=0
1980 SYS sprite%,put%,area%,blot%,wx%+x%(J%),wy%+y%(J%)
1990 VDU 7
2000 ENDPROC
2010 :
2020 DEF PROCwipe(x%,y%,RETURN f%)
2030 SYS sprite%,swap%,area%,frame%(f%-1)
2040 GCOL 0
2050 RECTANGLE FILL x%,y%,ax2%+8,ax2%
2060 SYS sprite%,swap%,area%
2070 OFF
2080 f%+=1
2090 IF f%>8 f%=0
2100 ENDPROC
```

By now you should be quite familiar with the techniques used for building up the sprites and animating them. The only complication is the interleaving of the main sprite movement, with its animation and the rest of the game. I could have cheated and made the movement count an exact multiple of the frames of animation, but I decided to retain the flexibility of keeping a separate counter, *L%* for the animation, with *D%* as a switch for running up and down through the sprite list, effectively doubling the number of frames.

Movement of the gun is quite straightforward and could have been animated. In the same way, you could animate the actual firing sequence, giving say, a little puff of smoke and a shower of sparks as it fires.

Arranging the firing of the bullets seems a bit complicated at first. Arrays are maintained for up to eight bullets. When the firing key is pressed, the Y coordinate of the currently selected bullet is examined to see if it is in motion, and therefore can't be fired again. If the bullet isn't active its X coordinate is set to the centre of the gun, and the Y coordinate to the tip. It is then plotted for the first time and the counter for the bullets incremented for the next free one.

Once started, each bullet will move up the screen for every pass of the main loop, until it has either reached top of the screen or collided with an alien. In either case its Y coordinate is set to zero, marking it as not active any more.

In PROCcollide, the alien hit is identified by calculating its coordinates in the overall sprite, and the absolute screen position of the sprite in its movement track. The coordinates are marked and a flag set for the wiping routine.

Actual erasing of the alien has to be spread through the animation sequence as every frame has to have the alien at that position erased. You will see that the alien wiping routine is also interleaved with the bullet movement. This is necessary to ensure that two bullets don't claim the same alien. Each time PROCwipe is entered screen writing is re-directed to the main sprite, the rectangle filled and screen writing restored. The counter is then updated. When all aliens have been erased, the counter is zeroed to prevent further entries into the PROCwipe for that alien.

This complexity is necessary, as you can never be sure on which frame a collision takes place. All you know is that you need to count through all of them. There is in fact a bug in this routine. What I failed to allow for in the original specification was the fact that if you are half way through the animation sequence, you will still fail to erase some sprites as the routine just counts sprites to erase. It doesn't allow for the fact that the animation is counting up then down again. A sledgehammer cure would be simply to double the count, and hence the explosion time. A more elegant solution involves a flag table. I'll leave it to you to work this out if you want to.

There is considerable room for improvement in other ways, but the essential ideas are there. It is a little jerky when everything is happening at once, but bear in mind that this program is running in interpreted Basic, yet is still superior to many 8-bit machines running their native machine code.

These often used a 25 frames per second film speed, where in this example, most of the time we manage 50 frames – the screen refresh rate. With the type of movement of the aliens, the game could easily be run at 25 frames, doubling all the movement distances, with little loss of apparent smoothness.

7.1.3 Flocking

A number of games in recent years have developed the idea of groups of enemy objects homing in on the player's object. This flocking performance is quite easy to implement. The objects that are to flock, have their positions and movement vectors stored in common arrays. The position elements for each flock object are compared with that of the victim, and the movement elements altered accordingly, so that the flock is always moving towards the victim.

Like this, the flocking is probably far too savage for most games, and the player has little chance of escape. The usual remedy is to add a flocking deviation, in the form of a random fluctuation in the flock object's movement vectors. Many games then adopt an approach of reducing the deviation with each attack wave or game level, making it progressively harder to escape.

A far more subtle approach is to modify the deviation by an amount dependent on each flock object's distance from the victim, so that as it *sees* the victim it becomes steadily more determined to reach it.

You can turn the whole idea upside down, and make the flocking objects be repelled by the player's object. This becomes more like herd action, as with sheep and a sheepdog.

In Listing 7.3 you can see this all put together, with seven different coloured stars as the flock, and the mouse pointer as the victim.

Listing 7.3: Flocks

```
   10 REM > Flocks
   20 :
   30 ON ERROR PROCerror:END
   40 PROCinitialise
   50 :
   60 REPEAT
   70   WAIT
   80   SYS byte%,113,sc%
   90   sc%=sc% EOR 3
  100   SYS byte%,112,sc%
```

```
 110    CLS
 120    MOUSE x%,y%,b%
 130    IF b%=4 REPEAT UNTIL NOT INKEY-10:repel%=1-repel%
 140    vx()=x%-x()
 150    vy()=y%-y()
 160    distx()=vx()*vx()
 170    disty()=vy()*vy()
 180    distx()=distx()+disty()
 190    FOR I%=0 TO num%
 200      IF x(I%)<xmin% OR x(I%)>xmax% dx(I%)=-dx(I%):x(I%)+=dx(I%)
 210      IF y(I%)<ymin% OR y(I%)>ymax% dy(I%)=-dy(I%):y(I%)+=dy(I%)
 220      GCOL I%+1
 230      MOVE x(I%),y(I%)
 240      PRINT"*";
 250      IF distx(I%)<scatter% distx(I%)=scatter%
 260      vx(I%)=force%*SGN vx(I%)
 270      vy(I%)=force%*SGN vy(I%)
 280    NEXT
 290    vx()=vx()/distx()
 300    vy()=vy()/distx()
 310    dx()=dx()-vx()
 320    dy()=dy()-vy()
 330    dx()=dx()*damp
 340    dy()=dy()*damp
 350     IF repel% x()=x()+dx():y()=y()+dy() ELSE x()=x()-dx():y()=y()-
dy()
 360 UNTIL b%=2
 370 *FX 112 1
 380 *FX 113 1
 390 END
 400 :
 410 DEF PROCerror
 420 MODE 12
 430 PRINT REPORT$ " @ ";ERL
 440 ENDPROC
 450 :
 460 DEF PROCinitialise
 470 MODE 13
 480 MODE 9
 490 *Pointer 1
 500 SYS "OS_SWINumberFromString",,"OS_Byte" TO byte%
 510 sc%=1
 520 num%=6
 530 DIM x(num%),y(num%)
 540 DIM dx(num%),dy(num%)
 550 DIM vx(num%),vy(num%)
 560 DIM distx(num%),disty(num%)
 570 xmin%=-630
 580 xmax%=610
 590 ymin%=-490
 600 ymax%=510
 610 xs%=xmax%-xmin%
```

```
  620 ys%=ymax%-ymin%
  630 force%=6400
  640 scatter%=900
  650 damp=0.9
  660 repel%=0
  670 FOR I%=0 TO num%
  680   x(I%)=(RND(xs%)-xs% DIV 2)DIV 2
  690   y(I%)=(RND(ys%)-ys% DIV 2)DIV 2
  700 NEXT
  710 ORIGIN 640,512
  720 PRINT TAB(10,7) "Flocks and Herds" TAB(5,12) "Move mouse to trac
k" TAB(5,14) "Select - toggle attract/repel" TAB(5,16) "Menu   - quit"
TAB(5,20) "Any button to start";
  730 REPEAT
  740   MOUSE x%,y%,b%
  750 UNTIL b%
  760 REPEAT
  770   MOUSE x%,y%,b%
  780 UNTIL b%=0
  790 VDU 5
  800 ENDPROC
```

I've made very heavy use of the whole array calculations in this example. Although making it rather long-winded and harder to follow, the program runs about 30% faster than it would if all the calculations were done inside the main FOR-NEXT loop. Some operations can't be performed on whole arrays, so these are inside the loop along with the printing of the stars.

In spite of the apparent complexity, there are in fact, no new programming ideas in this example. For each object an attractive force is calculated, based purely on its distance from the pointer and put in *vx(), vy()*. This is used to modify the actual movement vectors *dx(),dy()*, which in turn modify the X,Y coordinates of the object. Finally, a damping factor is used on *dx(),dy()* to prevent oscillations building up to ridiculous amplitudes.

The *repel%* flag determines whether the movement vector is subtracted or added, thus giving attraction or repulsion.

In the FOR-NEXT loop a check is made against *scatter%* to see if an object has got too close to the pointer. It stops the acceleration becoming too fierce and guards against any possible zero division. You will see that I've cheated by using the square of the distance to speed things up.

The screen limit testing is only really necessary to stop repelled objects going right off the screen.

In a game using this technique, a degree of randomness can still be retained, and you can use the various game levels and attack waves to set the number of flock objects and the distance at which they first seem to become aware of the victim.

A further enhancement to flocking can be provided by making all the flock objects tend to repel each other. This will result in a very realistic jostling action. This is most easily done with pixel collision detection in the main collision detection routine.

7.2 Rebounds

Probably the best known rebound game is Breakout, in all its many guises. One of the attractions of the earlier forms, both for players and programmers, is the limited freedom of the movement of the ball. This is usually restricted to one of eight possible directions, as shown in Figure 7.2. With the usual rectangular play area, and rectangular objects, you will

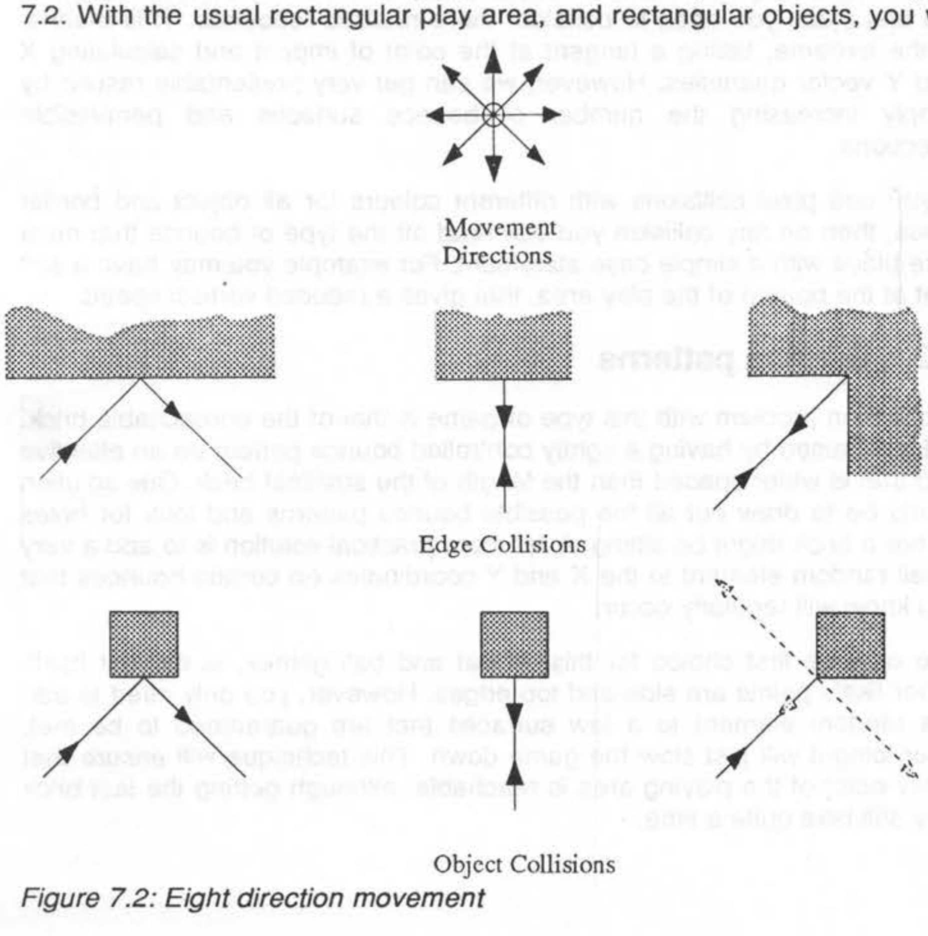

Figure 7.2: Eight direction movement

see that there are very few bounce conditions that have to be met. Usually, there is no in-flight change of speed or direction, so straight line interpolations are adequate.

In the first two examples shown, you only need to reverse the sign of the Y vector for your bounce. Were it a side wall that was being met, then it would be the X vector that needed to be reversed, the Y vector changing for a vertical bounce. The third example requires the sign of both X and Y vectors to be reversed. Object collisions are largely the same. The only one at all complicated is that of a collision with the corner of an object. Three possible bounces are simply chosen at random. This gives a bit of variety and makes the game more interesting.

It is unlikely that you'd be able to get away with such a simple structure these days, so you have to consider more realistic bounce conditions. In the first place you need to consider more realistic rebounds. This means, in the extreme, taking a tangent at the point of impact and calculating X and Y vector quantities. However, we can get very presentable results by simply increasing the number of bounce surfaces and permissible directions.

If you use pixel collisions with different colours for all object and border types, then on any collision you can read off the type of bounce that must take place with a simple case statement. For example you may have a soft mat at the bottom of the play area, that gives a reduced vertical speed.

7.2.1 Bounce patterns

A common problem with this type of game is that of the unreachable brick. This is caused by having a tightly controlled bounce pattern on an effective grid that is wider spaced than the length of the smallest brick. One solution would be to draw out all the possible bounce patterns and look for holes where a brick might be sitting. A far more practical solution is to add a very small random element to the X and Y coordinates on certain bounces that you know will regularly occur.

The obvious first choice for this, in bat and ball games, is the bat itself. Other likely points are side and top edges. However, you only need to add this random element to a few surfaces that are guaranteed to be met. Overdoing it will just slow the game down. This technique will ensure that every point of the playing area is reachable, although getting the last brick may still take quite a time.

7.2.2 Baby Breakout

By the time you get this far, you've actually got rid of the fixed bounce patterns completely. You will see this more clearly in Listing 7.4, a cut-down Rebound game.

Listing 7.4: Rebound

```
   10 REM > Rebound
   20 :
   30 ON ERROR PROCerror:END
   40 PROCinitialise
   50 PROCscreen
   60 REPEAT
   70   PROCstart
   80   REPEAT
   90     WAIT
  100     CIRCLE FILL x%,y%,8
  110     x%+=dx%
  120     y%+=dy%
  130     bounce%=POINT(x%,y%)
  140     IF bounce% PROCbounce ELSE IF glue% x%=rx%+rw% DIV 2
  150     RECTANGLE FILL rx%,ry%,rw%,rh%
  160     IF INKEY-98 PROCleft ELSE IF INKEY-67 PROCright
  170     RECTANGLE FILL rx%,ry%,rw%,rh%
  180     CIRCLE FILL x%,y%,8
  190     IF INKEY-74:IF glue% glue%=FALSE:dy%=RND(6)+3
  200   UNTIL end%
  210   CIRCLE FILL x%,y%,8
  220   RECTANGLE FILL rx%,ry%,rw%,rh%
  230   *FX 21
  240   t%=TIME
  250   REPEAT UNTIL TIME-t%>100
  260 UNTIL bnum%=0
  270 END
  280 :
  290 DEF PROCerror
  300 MODE 12
  310 IF ERR<>17 PRINT REPORT$ " @ ";ERL
  320 ENDPROC
  330 :
  340 DEF PROCinitialise
  350 MODE 13
  360 MODE 9
  370 OFF
  380 COLOUR 8,192,192,192
  390 COLOUR 9,96,96,96
  400 PRINT TAB(11,7) "Basic Rebound Game" TAB(5,11) "Z       - Left" T
AB(5,13) "X       - Right" TAB(5,15) "Return - Release ball" TAB(12,19)
"Escape to exit" TAB(11,21) "Any key to start"
  410 IF GET
```

```
420 CLS
430 min%=30          :REM no objects to be smaller
440 rs%=12           :REM bat speed
450 rw%=196          :REM bat width
460 rh%=32           :REM bat height
470 ry%=min%+4       :REM bat Y axis
480 bw%=64           :REM brick width
490 bh%=32           :REM brick height
500 hb%=bw% DIV 2    :REM half brick
510 wy%=bh%*16       :REM wall position
520 bn%=15           :REM bricks per course
530 wh%=8            :REM wall height
540 z%=rh%+ry%+12    :REM ball on bat Y axis
550 s%=min%-8        :REM ball speed
560 ENDPROC
570 :
580 DEF PROCscreen
590 GCOL 1
600 RECTANGLE FILL 0,0,min%,1023
610 RECTANGLE FILL 1280-min%,0,min%,1023
620 GCOL 2
630 RECTANGLE FILL 0,0,1279,min%
640 RECTANGLE FILL 0,1024-min%,1279,min%
650 off%=hb%
660 FOR J%=0 TO wh%
670   off%=off% EOR hb%
680   IF off% PROCbrick(bw%*2,wy%+bh%*J%,bw% DIV 2,bh%)
690   FOR I%=0 TO bn%+(off%>0)
700     PROCbrick(bw%*2+off%+bw%*I%,wy%+bh%*J%,bw%,bh%)
710   NEXT
720   IF off% PROCbrick(bw%*2+off%+bw%*bn%,wy%+bh%*J%,bw% DIV 2,bh%)
730 NEXT
740 GCOL 4
750 FOR I%=0 TO 8
760   GCOL 8+(I% MOD 2)
770   RECTANGLE FILL 96+I%*128,0,64,min%
780 NEXT
790 GCOL 5
800 MOVE 0,1023
810 MOVE 0,895
820 PLOT &55,128,1023
830 GCOL 6
840 MOVE 1279,1023
850 MOVE 1279-128,1023
860 PLOT &55,1279,895
870 bnum%=(bn%+1)*(wh%+1)+(wh%+1)DIV 2
880 ENDPROC
890 :
900 DEF PROCbrick(x%,y%,w%,h%)
910 GCOL 3
920 RECTANGLE FILL x%,y%,w%-4,h%-4
930 GCOL 4
```

```
 940 RECTANGLE x%,y%,w%-4,h%-4
 950 ENDPROC
 960 :
 970 DEF PROCstart
 980 end%=FALSE
 990 glue%=TRUE
1000 rx%=480
1010 x%=rx%+rw% DIV 2
1020 y%=z%
1030 dx%=0
1040 dy%=0
1050 GCOL 3,7
1060 CIRCLE FILL x%,y%,8
1070 RECTANGLE FILL rx%,ry%,rw%,rh%
1080 ENDPROC
1090 :
1100 DEF PROCbounce
1110 a%=ABS dx%
1120 b%=ABS dy%
1130 c%=SGN dx%
1140 d%=SGN dy%
1150 CASE bounce% OF
1160   WHEN 1:x%-=dx%:dx%=-dx%
1170   WHEN 2:y%-=dy%:dy%=-dy%:c%+=RND(3)-2
1180   WHEN 3,4:PROChit
1190   WHEN 5:x%-=dx%:y%-=dy%:SWAP dx%,dy%
1200   WHEN 6:x%-=dx%:y%-=dy%:SWAP dx%,dy%:dy%=-dy%:dx%=-dx%
1210   WHEN 7:PROCbat
1220   WHEN 8:y%-=dy%:dy%=-dy% DIV 2
1230   WHEN 9:end%=TRUE
1240 ENDCASE
1250 IF a%<4 dx%=4*c% ELSE IF a%>s% dx%=s%*c%
1260 IF b%<4 dy%=4*d% ELSE IF b%>s% dy%=s%*d%
1270 VDU 7
1280 ENDPROC
1290 :
1300 DEF PROCleft
1310 IF rx%>min%+rs% rx%-=rs%
1320 ENDPROC
1330 :
1340 DEF PROCright
1350 IF rx%+rw%<1280-min%-rs% rx%+=rs%
1360 ENDPROC
1370 :
1380 DEF PROChit
1390 by%=y% DIV bh%*bh%
1400 y%-=dy%
1410 IF by% DIV bh% MOD 2 bx%=(x%+hb%)DIV bw%*bw%:bx%-=hb% ELSE bx%=x
% DIV bw%*bw%
1420 x%-=dx%
1430 GCOL 0
1440 RECTANGLE FILL bx%,by%,bw%,bh%
```

```
1450 GCOL 3,7
1460 dy%=-dy%
1470 IF a%<3 c%=RND(5)-3
1480 bnum%-=1
1490 IF bnum%=0 end%=TRUE
1500 ENDPROC
1510 :
1520 DEF PROCbat
1530 y%=z%
1540 dy%=b%+4
1550 IF RND(9)>1 dx%+=RND(3)-2 ELSE glue%=TRUE:dx%=0:dy%=0
1560 ENDPROC
```

In this example, I've deliberately kept the logical colours used looking visibly different so that you can see how the surfaces are built up. In practice you would make all the edges look the same colour. If you are using more attractive sprites in a 256 colour mode, you'd probably be better off using a combination of tints and coordinates for collisions.

By limiting the ball speed to less than the thickness of any object, I've avoided the need for a complex look-ahead system. Also, by using Exclusive Or plotting, any slight overlaps will go unnoticed.

Working out which brick has to be removed is done in PROChit by turning the ball coordinates into exact brick multiples, with a half brick horizontal offset being added for the even numbered courses. Simple rectangle filling then erases the brick.

I've fiddled the bounce of the ball on the bat, so that the ball always sits on top of it. This saves having to do anything special about edge hits.

A point of interest is the way the ball follows the bat when you start. This is controlled by the flag *glue%*. If it is set then the ball's X coordinate is continually adjusted to that of the centre of the bat. You could easily make this flag a countdown timer, so that if the player doesn't release the ball after a certain number of game loops, then it releases itself.

7.2.3 Spin

It is well worth while considering some form of allowance for forward and reverse spin in bat and ball type rebounds. This is fairly simple to implement, but only really practical if you allow all direction movement instead of the eight direction forms. All you need to do is perform a rotation of the X,Y vectors in the same way as was used for 3D rotation. The amount of spin determines the rotation angle. The direction, of course, relates directly. The program fragment bellow assumes *dx%* and *dy%* are the vectors, and *ang%* is the amount of spin that you want to emulate.

```
tempx%=dx%
dx%=dx%*COS(ang%)-dy%*SIN(ang%)
dy%=dy%*COS(ang%)+tempx%*SIN(ang%)
```

Normally this spin only takes effect when objects come into contact with a solid surface. However, you can simulate drag effects of spinning objects through air or water. To do this, continually add a tiny fraction of the spin angle every pass of the game loop. This won't be terribly accurate, but will look convincing enough.

7.2.4 Gravity

Another useful extension is the ability for objects to attract or repel each other in flight. To be realistic you have to consider how gravitation and magnetic forces work although, again, you don't have to be too precise. Also, I'll only look at the two body situation. If you want to go further than that I'm afraid you'll have to delve into your physics book.

In the first place the total attractive force between two objects will be inversely proportional to the square of the distance between them. It will also be proportional to the product of their masses. In our game world it is generally easiest to equate mass with size, as we already need this figure for collision calculations. The actual deflection of each object will be inversely proportional to the mass ratio. Armed with that information we can produce the example of Listing 7.5

Listing 7.5: Attraction

```
 10 REM > Attract
 20 :
 30 ON ERROR PROCerror:END
 40 PROCinitialise
 50 PRINT TAB(30,12) "Press Escape to stop"
 60 IF INKEY 100
 70 FOR I%=0 TO loops%
 80   flag%=FALSE
 90   IF INKEY 50
100   CLG
110   READ mass1%,mass2%
120   READ x1,y1,x2,y2
130   READ dx1,dy1,dx2,dy2
140   ratio=mass1%/mass2%
150   PROCplot
160   REPEAT
170     WAIT
180     SYS byte%,113,sc%
190     sc%=sc% EOR 3
```

```
200      SYS byte%,112,sc%
210      CLS
220      dist=(x2-x1)^2+(y2-y1)^2
230      IF dist<1 dist=1:REM prevents division by zero
240      attract=(mass1%*mass2%)/dist
250      xs%=SGN(x2-x1)
260      ys%=SGN(y2-y1)
270      dx1+=(attract/ratio*xs%)
280      dx2-=(attract*ratio*xs%)
290      dy1+=(attract/ratio*ys%)
300      dy2-=(attract*ratio*ys%)
310      x1+=dx1
320      x2+=dx2
330      y1+=dy1
340      y2+=dy2
350      PROCplot
360      IF x1<xmin% OR x1>xmax% flag%=TRUE
370      IF x2<xmin% OR x2>xmax% flag%=TRUE
380      IF y2<ymin% OR y2>ymax% flag%=TRUE
390      IF y1<ymin% OR y1>ymax% flag%=TRUE
400      IF dist<(mass1%+mass2%)^2 flag%=TRUE
410    UNTIL flag%
420 NEXT
430 PRINT"'End of program"
440 END
450 :
460 DEF PROCerror
470 MODE 12
480 IF ERR<>17 PRINT REPORT$ " @ ";ERL
490 ENDPROC
500 :
510 DEF PROCinitialise
520 MODE 15
530 MODE 12
540 OFF
550 COLOUR 0,0,0,128
560 SYS "OS_SWINumberFromString",,"OS_Byte" TO byte%
570 sc%=1
580 xmin%=0
590 xmax%=1279
600 ymin%=0
610 ymax%=1023
620 RESTORE+3
630 READ loops%
640 ENDPROC
650 DATA 3
660 DATA 20,40
670 DATA 320,1000,800,1000
680 DATA 0,-1,0,-3
690 DATA 17,34
700 DATA 768,704,384,448
710 DATA -1,0,.4,0
```

```
720 DATA 32,32
730 DATA 256,512,1023,512
740 DATA .4,-.75,-.4,.75
750 DATA 8,128
760 DATA 256,512,640,512
770 DATA 0,8.5,0,-.03
780 :
790 DEF PROCplot
800 GCOL 3
810 CIRCLE FILL x1,y1,mass1%
820 GCOL 1
830 CIRCLE FILL x2,y2,mass2%
840 ENDPROC
```

The simulation isn't perfect, but then again neither is the mathematics used, nor the accuracy of Basic V. However the results are quite good enough for all but the most stringent cases.

You will see that the distance calculation is used not only for the deflection calculations but also for collision detection. It's always rather nice when you can make one piece of arithmetic do two jobs.

For repelling objects simply reverse the signs of the *dx* and *dy* additions and subtractions.

7.3 Platforms

The basis for most platform games is the old table-top Snakes and Ladders. However, instead of relying on chance dice throws, you now have direct control of the character. Jumps and lifts take the place of the ladders while holes, monsters and *soft* platforms play the part of the snakes.

One of the features most platform games have in common with the older rebounds is that there is very restricted movement.

By the very nature of the games, platforms assume the player's character is always on a platform of some sort. This presents a small problem when looking at collisions. Not only do you have to look ahead in the direction you are moving, but also down to check that the platform is still there, and what type of platform it is. Then again, you may choose to write a game where the player character can stick to a wall or ceiling, in which case up, left or right may be the second direction that needs to be checked. To complicate matters a little more, if you are handling a jump of some sort, you may need to modify the second direction part until the completion of the jump action.

As usual there's more than one solution to the problem. You could keep a list of all the coordinates where a change in direction is required, but this would be rather tedious and inflexible. A better idea would be to use cell collisions, and instead of marking the row of cells below the player's object, mark the actual row that the player is on with, say, a 1 value. Now, whichever way the object moves it will see the track of 1s. Any other value would be invalid for normal movement. Zero would be acceptable for jumps, and other values would represent collision situations.

The zero value would be particularly important, as in the event of no other movement information, either from the player or some special movement twiddle, you would assume that gravity takes over and the object moves gracefully down, until it is sitting over some other value. If you keep a count of the number of cells dropped through in this way, you can decide whether or not it was a survivable drop, and take the appropriate action. This is shown in Listing 7.6.

Listing 7.6: Platform

```
   10 REM > Platform
   20 :
   30 ON ERROR PROCerror:END
   40 PROCinitialise
   50 PROCassemble
   60 REPEAT
   70   PROCbuild
   80   PROCdraw
   90   PROCstart
  100   REPEAT
  110     FOR mark%=0 TO 7
  120       CALL code%
  130       monx%()=monx%()+mondx%()
  140       mony%()=mony%()+mondy%()
  150       FOR I%=0 TO mons%
  160         IF mark%=0 PROCmonmove
  170         CIRCLE FILL monx%(I%)+half%,mony%(I%)+half%,10
  180       NEXT
  190       CASE objects%(nx%>>5,ny%>>5) OF
  200         WHEN 0,15:PROCdrop(0,-32)
  210         WHEN 7:mark%=7:end%=TRUE
  220         WHEN 9,11,13:IF count%=0 PROCsink
  230       ENDCASE
  240       IF count% PROCanimate ELSE PROCkey
  250       CIRCLE FILL X%+half%,Y%+half%,circ%
  260       FOR I%=0 TO mons%
  270         IF mark%=0 objects%(monx%(I%)>>5,mony%(I%)>>5)=oldmon%(I
%)
  280       NEXT
  290       NEXT
  300    UNTIL end%
  310    VDU 7
  320 UNTIL FALSE
  330 END
  340 :
  350 DEF PROCerror
  360 *FX 4
  370 *FX 21
  380 MODE 12
  390 IF ERR<>17 PRINT REPORT$ " @ ";ERL
  400 ENDPROC
  410 :
  420 DEF PROCinitialise
  430 *FX 4 1
  440 MODE 15
  450 MODE 9
  460 PRINT TAB(12,6) "Platform Demo" TAB(5,9) "Z - Left" SPC 9 "X - R
ight" TAB(5,11) "' - Up" SPC 11 "/ - Down" TAB(5,13) "Return - Jump" S
PC 4 "Escape - Stop"
  470
  480 COLOUR 8,128,128,128
  490 COLOUR 9,128,128,128
  500 COLOUR 10,128,128,128
  510 COLOUR 11,128,128,128
  520 VDU 5
  530 DIM objects%(39,31)
  540 DIM plot%(39,31)
  550 circ%=12
  560 vert%=32-4
  570 half%=16
  580 maxmonsters%=9
  590 DIM monx%(maxmonsters%)
  600 DIM mondx%(maxmonsters%)
  610 DIM mony%(maxmonsters%)
  620 DIM mondy%(maxmonsters%)
  630 DIM oldmon%(maxmonsters%)
  640 VDU23,129,&81,&81,&FF,&81,&81,&81,&FF,&81
  650 VDU23,130,&FF,&00,&FF,&00,&FF,&00,&FF,&00
  660 VDU23,131,&AA,&55,&AA,&55,&AA,&55,&AA,&55
  670 VDU23,132,&FF,&FF,&C3,&C3,&C3,&C3,&FF,&FF
  680 VDU23,136,&AA,&55,&AA,&55,&AA,&55,&AA,&55
  690 VDU23,137,&00,&00,&00,&00,&AA,&55,&AA,&55
  700 VDU23,138,&00,&00,&AA,&55,&00,&00,&00,&00
  710 VDU23,139,&AA,&55,&00,&00,&00,&00,&00,&00
  720 X%=-1
  730 ENDPROC
  740 :
  750 DEFPROCassemble
  760 DIM block% &100
  770 block%!0=148
  780 block%!4=7
```

```
 790 block%!8=-1
 800 SYS "OS_ReadVduVariables",block%,block%+12
 810 !block%=1                  : REM bank number
 820 block%!4=block%!16         : REM screen size
 830 block%!8=block%!12         : REM screen start
 840 block%!12+=(block%!16)*2 : REM stored screen start
 850 block%!16+=block%!12       : REM stored screen end
 860
 870 lowreg=0
 880 highreg=7
 890 bank=7
 900 size=8
 910 screen=9
 920 memory=10
 930 end=11
 940 store=12
 950 link=14
 960 code%=block%+20
 970 FOR I%=0 TO 2 STEP 2
 980   P%=code%
 990   [ OPT I%
1000   ADR store,block%
1010   LDMIA store,{bank-end}
1020   MOV R0,#19
1030   SWI "OS_Byte"
1040   MOV R0,#113
1050   MOV R1,bank
1060   SWI "OS_Byte"
1070   EOR bank,bank,#3
1080   MOV R0,#112
1090   MOV R1,bank
1100   SWI "OS_Byte"
1110   CMP bank,#2
1120   ADDEQ screen,screen,size
1130   STR bank,[store]
1140   .copy
1150   LDMIA (memory)!,{lowreg-highreg}
1160   STMIA (screen)!,{lowreg-highreg}
1170   CMP memory,end
1180   BLT copy
1190   MOV PC,link
1200   ]
1210 NEXT
1220 ENDPROC
1230 :
1240 DEF PROCbuild
1250 RESTORE+21
1260 READ num%
1270 FOR J%=0 TO num%
1280   READ x%,y%,horiz%,count%,value%
1290   FOR I%=0 TO count%
1300     IF horiz% THEN
```

```
1310        objects%(x%+I%,y%)=value%
1320        ELSE
1330        objects%(x%,y%+I%)=value%
1340      ENDIF
1350    NEXT
1360 NEXT
1370 READ single%
1380 FOR I%=0 TO single%
1390   READ x%,y%,value%
1400   objects%(x%,y%)=value%
1410 NEXT
1420 :
1430 READ num%
1440 FOR J%=0 TO num%
1450   READ x%,y%,horiz%,count%,char%
1460   FOR I%=0 TO count%
1470     IF horiz% THEN
1480       plot%(x%+I%,y%)=char%
1490       ELSE
1500       plot%(x%,y%+I%)=char%
1510     ENDIF
1520   NEXT
1530 NEXT
1540 READ single%
1550 FOR I%=0 TO single%
1560   READ x%,y%,char%
1570   plot%(x%,y%)=char%
1580 NEXT
1590 REM monsters must be defined after all other objects
1600 READ mons%
1610 FOR I%=0 TO mons%
1620   READ monx%(I%),mony%(I%),mondx%(I%),mondy%(I%)
1630   monx%(I%)=monx%(I%)<<5
1640   mony%(I%)=mony%(I%)<<5
1650 NEXT
1660 ENDPROC
1670 DATA 11: REM objects
1680 DATA 20,3,1,15,1
1690 DATA 1,15,1,37,1
1700 DATA 1,24,1,28,1
1710 DATA 32,26,1,6,1
1720 DATA 19,3,0,12,3
1730 DATA 2,16,0,8,3
1740 DATA 29,16,0,8,3
1750 DATA 0,0,1,39,7
1760 DATA 0,1,0,29,7
1770 DATA 39,1,0,29,7
1780 DATA 0,31,1,39,7
1790 DATA 30,24,1,5,9
1800 DATA 7
1810 DATA 12,24,15
1820 DATA 19,2,15
```

```
1830 DATA 1,15,17
1840 DATA 20,15,17
1850 DATA 1,3,17
1860 DATA 32,3,17
1870 DATA 34,3,17
1880 DATA 34,25,17
1890 :
1900 DATA 11: REM visible bits
1910 DATA 20,2,1,15,130
1920 DATA 1,14,1,37,130
1930 DATA 1,23,1,28,130
1940 DATA 32,25,1,6,130
1950 DATA 19,2,0,12,129
1960 DATA 2,15,0,8,129
1970 DATA 29,15,0,8,129
1980 DATA 0,0,1,39,132
1990 DATA 0,1,0,29,132
2000 DATA 39,1,0,29,132
2010 DATA 0,31,1,39,132
2020 DATA 30,23,1,5,136
2030 DATA 0
2040 DATA 12,23,131
2050 :
2060 DATA 2: REM monsters
2070 DATA 2,15,4,0
2080 DATA 28,3,-4,0
2090 DATA 34,4,0,4
2100 :
2110 DEF PROCdraw
2120 *FX 112 3
2130 CLS
2140 FOR J%=0 TO 39
2150   FOR I%=0 TO 31
2160     GCOL plot%(J%,I%) AND 127
2170     IF plot%(J%,I%) MOVE J%<<5,(I%<<5)+vert%:VDU plot%(J%,I%)
2180   NEXT
2190 NEXT
2200 E%=1
2210 *FX 112 1
2220 GCOL 7
2230 ENDPROC
2240 :
2250 DEF PROCstart
2260 IF X%<0 THEN
2270   PRINT TAB(5,17) "Any key to start"
2280   IF GET
2290 ENDIF
2300 X%=28<5
2310 Y%=15<5
2320 nx%=X%
2330 ny%=Y%
2340 count%=0
```

```
 2350 end%=FALSE
 2360 ENDPROC
 2370 :
 2380 DEF PROCmonmove
 2390 oldmon%(I%)=objects%(monx%(I%)>>5,mony%(I%)>>5)
 2400 IF objects%(monx%(I%)>>5,mony%(I%)>>5)=17 PROCmonrev ELSE object
s%(monx%(I%)>>5,mony%(I%)>>5)=7
 2410 ENDPROC
 2420 :
 2430 DEF PROCmonrev
 2440 mondx%(I%)=-mondx%(I%)
 2450 mondy%(I%)=-mondy%(I%)
 2460 ENDPROC
 2470 :
 2480 DEF PROCsink
 2490 objects%(X%>>5,Y%>>5)+=2
 2500 y%=Y%-32
 2510 *FX 112 3
 2520 plot%(X%>>5,y%>>5)+=1
 2530 GCOL 3,8
 2540 MOVE X%,y%+vert%
 2550 VDU plot%(X%>>5,y%>>5)
 2560 *FX112 1
 2570 GCOL 7
 2580 ENDPROC
 2590 :
 2600 DEF PROCanimate
 2610 X%+=ix%
 2620 Y%+=iy%
 2630 count%-=1
 2640 ENDPROC
 2650 :
 2660 DEF PROCkey
 2670 CASE TRUE OF
 2680   WHEN INKEY-98:PROCmove(-32,0)
 2690   WHEN INKEY-67:PROCmove(32,0)
 2700   WHEN INKEY-80:PROCmove(0,32)
 2710   WHEN INKEY-105:PROCmove(0,-32)
 2720   WHEN INKEY-74:PROCmove(0,0)
 2730 ENDCASE
 2740 ENDPROC
 2750 :
 2760 DEF PROCmove(h%,v%)
 2770 IF INKEY-74 v%=64
 2780 IF v%=64 OR objects%((X%+h%)>>5,(Y%+v%)>>5)AND 1 THEN
 2790   nx%=X%+h%
 2800   ny%=Y%+v%
 2810   ix%=h%>>3
 2820   iy%=v%>>3
 2830   X%+=ix%
 2840   Y%+=iy%
 2850   count%=7
```

```
2860 ENDIF
2870 ENDPROC
2880 :
2890 DEF PROCdrop(h%,v%)
2900 IF count%=0 THEN
2910   nx%=X%+h%
2920   ny%=Y%+v%
2930   ix%=h%>>3
2940   iy%=v%>>3
2950   count%=8
2960 ENDIF
2970 ENDPROC
```

Instead of restricting the player's character to move over cells marked with the value 1, to allow for a range of additional action, the game recognises any odd number value as being valid for the player to move on to. In the example, this allows ladders, platforms, and soft platforms all to be uniquely recognised.

You will see that cell collisions allow for the automatic movement of the monsters, as well as the player. All that is needed is a marker value in the object array at the end of each monster's track. Value 17 in used in this instance. This allows the player to move over it unimpeded, while causing a reversal of monster movement direction.

While using cells for the collision system, the actual coordinates have all been kept in graphic units. This makes plotting and animation easier and faster. Where array accessing has to be done, a simple barrel shift is all that's necessary to find the array indices. In this example the animation simply consists of a straight line plot. In fact you would normally use a film animation, and for jumps you would plot out a curved trajectory taken from a suitable movement table.

To give further flexibility, I've separated the actual platforms from their visible representations by using two different arrays. If you go on to using a table of sprites for creating the visible platforms, you can add all sorts of different types of edging and platform, without having to disturb the main sensing array. This is particularly relevant for things like brittle stalactites, or soft and collapsing platforms. These are an essential feature in the best games, and the crude character swapping I've done in the example just doesn't do the idea justice.

As an added point of interest, you will see that I've modified the screen store and copy routine so that it uses a spare screen bank. This allows the screen to be drawn up invisibly, at a leisurely pace with ordinary drawing commands, and also, by bank switching, allows the stored screen to be modified. In this example we only use the feature for soft platforms, but it is equally applicable to picking up treasures and the like.

In the example, all the monsters have the same effect. They kill you. However, if you keep another parallel array of monster types, you can have different actions by bedding a different number in the main object array. This can include collectable items.

Finally, there is a bug in the game. If you move towards one of the monsters while, at the same time, it is moving towards you, then under some circumstances you will miss each other. This is because, if the two counters *count%* and *mark%* have become synchronised, your object and the monster will simply swap places in the same loop.

7.4 Map Compression

The two dimensional grid used by the previous example can be considered as a basic map, or more correctly a pair of maps. As shown, these are very wasteful of memory, bearing in mind that you will need a similar pair for every game level.

The first thing to realise is that you don't need to use full integers for the information in the arrays. It is inconceivable that anyone would want more than 255 different cell types or sprite blocks, so you can immediately think about splitting up integers into four byte numbers consisting of consecutive cells. This works particularly well if you intend to put some of the program into ARM code.

Unfortunately the integer splitting and, where necessary, re-combining is rather time consuming, so a better idea is to go the whole hog and use a byte array. The program fragment below shows how a two dimensional array can be synthesised from this in Basic.

```
wide%=39 : REM number of cells across - 1
high%=31 : REM number of cells down - 1
size%=(high%+1)*(wide%+1)
DIM objects% size%
REM some code
IF FNread(X%,Y%)>23 REM do something
REM more code
PROCwrite(X%,Y%,monstertype%)
DEF FNread(xpos%,ypos%)
=objects%?(xpos%+ypos%*wide%)
DEFPROCwrite(xpos%,ypos%,byte%)
   objects%?(xpos%+ypos%*wide%)=byte%
ENDPROC
```

If you can limit your object values and sprite list to just 16 each, you can combine the two arrays as below.

```
DEF FNreadobject(xpos%,ypos%)
=(objects%?(xpos%+ypos%*wide%))AND 15
DEF FNreadsprite(xpos%,ypos%)
=(objects%?(xpos%+ypos%*wide%))AND 240
```

Object numbers are now from 0 to 15 and sprites are 16 to 240 in steps of 16. Barrel shifting the sprite numbers to get 0 to 15 numbering, while possible, serves no useful purpose and wastes processor time.

For the size of screen we've been discussing, our map data is now only a mere 1,280 bytes per level. These levels are best constructed in an editor program of some sort, then saved as binary files. These can then be loaded directly . If your game turns out to be really successful, you can offer the editor program as an extra. This has been done with many commercial releases.

As a point of interest, by using a byte array, you have the capability of instantly switching to another level. Simply store all the levels as one continuous line of bytes, then instead of using the array base itself, use a pointer that can be stepped up and down in whole level blocks, as follows:

```
levels%=30
step%=1280
DIM objects% step%*levels%
point%=objects%
top%=point%+step%*(levels%-1)
REM some code
IF donelev% AND point%<top% point%+=step%:PROCdrawit
REM yet more
DEF FNreadsprite(xpos%,ypos%)
=(point%?(xpos%+ypos%*wide%))AND 240
```

7.5 Score Tables

There is a temptation to forget about score tables until a game has been completely programmed. This is then hurriedly tacked on, and as a result can be pretty awful. You should make score tables an integral part of the game. Include bonus points achieved, completion times and the number and type of aliens dispatched.

Quite a lot of players don't bother to put their names in, leaving a blank entry by their score. Some programmers try to prevent this by making the scoring routine insist on three or more letters. This just tends to irritate players who simply want to get on. A much better approach is to set up an array of silly, mildly insulting, but not offensive names. If the player just hits the return key without an entry, you pick one of these at random.

You should provide the player with the option of saving a score table, re-loading it, and especially deleting the stored copy and starting from scratch. Otherwise continual play will eventually produce a situation where it is impossible for anyone to get their name on the table, which will considerably discourage further play. At the same time it is worth considering only saving half the table back onto the disk.

If you have a table of eight entries, which is fairly typical, only save the top four. When you next reload the score table you add four synthesised fairly mediocre scores at the bottom. This means that any player has a chance of getting onto the table. Persistent average players will always have their names saved, and the really good players can compete with each other for the top few places.

Time spent livening up the your score table routine adds a considerable polish, as was mentioned in the section on layouts. It is still common to see games using the system font for score tables, either because the programmer can't handle Acorn's fancy fonts, or through lack of disk space.

One possible solution is to develop your own set of drawn characters. You will have to plan them out on graph paper, then translate the lines into a combination of MOVE, DRAW, CIRCLE, and other plot commands. These can then be read off from lines of data. The result is very compact, albeit tedious to develop.

An example of a score table that covers most of the points above is given in Listing 7.7. This was originally developed on the old BBC Model B, and the ARM code section added for extra speed, along with a few other improvements. The routine is provided *as is*, but you can experiment with it to see how it works and what the possibilities are.

Listing 7.7: Score tables

```
10 REM > Scores
20 :
30 PROCinitialise
40 PROCassemble
50 PROCloadscores
60 :
70 PROCtable(8020)
80 PROCtable(6741)
90 PROCtable(4127)
```

```
 100 PROCtable(391)
 110 END
 120 :
 130 DEF PROCinitialise
 140 DIM score$(8),score%(8)
 150 DIM font$(94)
 160 DIM code% &100
 170 DIM D% 100,T% 80,U% 20 : REM don't use elswhere!!
 180 file$="ScoreTable"
 190 RESTORE+4
 200 FOR I%=13 TO 90
 210   READ font$(I%)
 220 NEXT
 230 ENDPROC
 240 DATA DZ0NH1
 250 DATA IH0@I1
 260 DATA
 270 DATA FH0FH1DJ1BL1@'1>L1<J1:H1D1@01BD1DF1
 280 DATA FH0FH1=H0@11?F1>G1
 290 DATA @f0BL1DJ1FH1DF1BD1@B1>D12:1>D1@F1RH1
 300 DATA @f0BL1DJ1FH1DF1BD1@B1>D1<F1DF1BD1@B1>D1<F1:H1<J1>L1
 310 DATA LH0@11401RH1
 320 DATA R10.H1@61BL1DJ1FH1DF1BD1@<1>D1<F1:H1<J1>L1
 330 DATA Rf0>L1<J1:H1D1@01BD1DF1FH1DJ1BL1@T1>L1<J1:H1D1
 340 DATA @10RH1.$1
 350 DATA FH0FH1DJ1BL1@N1>L1<J1:H1<J1>L1@N1BL1DJ1FH1DF1BD1@B1>D1<F1:H
0D1@B1BD1DF1
 360 DATA @N0BD1DF1FH1DJ1BL1@'1>L1<J1:H1D1@<1BD1DF1FH1DJ1BL1
 370 DATA,,,,,
 380 DATA @f1BL1DJ1FH1DF1BD1@*1@Z0.H1
 390 DATA LH1DJ1BL1@N1>L1<J14H1JH0DJ1BL1@N1>L1<J16H1@$1
 400 DATA RN0>D1<F1:H1<J1>L1@'1BL1DJ1FH1DF1BD1
 410 DATA LH1DJ1BL1@'1>L1<J14H1@$1
 420 DATA RH1.Z0LH1FZ0.H1@$1
 430 DATA @Z0LH1FZ0.H1@$1
 440 DATA RH0@Z1:H1F<0>D1<F1:H1<J1>L1@'1BL1DJ1FH1DF1BD1
 450 DATA @11@60RH1@Z0@$1
 460 DATA FH0FH1=H0@11CH0:H1
 470 DATA @N0BD1DF1FH1DJ1BL1@f1
 480 DATA @11@00R'1481L41
 490 DATA RH1.H0@11
 500 DATA @11I$1I11@$1
 510 DATA @11R$1@11
 520 DATA @N0BD1DF1FH1DJ1BL1@'1>L1<J1:H1D1@01
 530 DATA @11LH1DF1BD1@B1>D1<F14H1
 540 DATA @N0BD1DF1FH1DJ1BL1@'1>L1<J1:H1D1@01LH0FB1
 550 DATA @11LH1DF1BD1@B1>D1<F14H1IH0I61
 560 DATA @N0BD1DF1FH1DJ1BL1@N1>L1<J1:H1<J1>L1@N1BL1DJ1FH1DF1BD1
 570 DATA @10RH17H0@$1
 580 DATA @10@*1BD1DF1FH1DJ1BL1@f1
 590 DATA @10I$1I11
 600 DATA @10E$1D11E$1E11
```

```
  610 DATA R11.H0R$1
  620 DATA IH0@Z1IZ1.H0I61
  630 DATA @10RH1.$1RH1
  640 DATA,,,,,
  650 DATA @Z0BL1DJ1DH1DF1BD1@81BF1>N0>D1<F1<H1<J1>L1@L1BL1DJ1DH1DF1BD
1
  660 DATA @10@*1BD1DF1FH1DJ1BL1@T1>L1<J1:H1D1
  670 DATA RZ0>L1<J1:H1D1@<1BD1DF1FH1DJ1BL1
  680 DATA R10@$1@N0>D1<F1:H1<J1>L1@T1BL1DJ1FH1DF1BD1
  690 DATA @R0LH1DJ1BL1@J1>L1<J1:H1D1@<1BD1DF1FH1DJ1BL1
  700 DATA @H0@f1BL1DJ1FH1DF1BD1.<0LH1
  710 DATA FL0D1@D1BD1DF1FH1DJ1BL1@L1>L1<J1:H1>J1@J1BJ1<J1>L1@J1BL
1DJ1DH1DF1BD1@F1>D1<F1<H1JR0BJ1
  720 DATA @11@60BL1DJ1FH1DF1BD1@61
  730 DATA GH0DH1>H0@Z1>H1BM0@I1
  740 DATA F<0DJ1BL1@'1@M0@I1
  750 DATA @11@*0RZ17?0I91
  760 DATA LH0<J1>L1@f1
  770 DATA @]1AJ1BI1CF1BE1@D1@L0BK1CJ1BG1AF1@31
  780 DATA @'1@B0BL1DJ1FH1DF1BD1@61
  790 DATA @N0BD1DF1FH1DJ1BL1@T1>L1<J1:H1D1@<1
  800 DATA @<0@11@B0BL1DJ1FH1DF1BD1@<1>D1<F1:H1<J1>L1
  810 DATA T<0>H1@11@B0>L1<J1:H1D1@<1BD1DF1FH1DJ1BL1
  820 DATA @'1@B0BL1DJ1FH1DF1BD1
  830 DATA @N0BD1DF1FH1DJ1BL1>L1<J1:H1<J1>L1BL1DJ1FH1DF1BD1
  840 DATA F10@*1BD1DF1DJ1BL11Z0OH1
  850 DATA @'0@61BD1DF1FH1DJ1BL1@Z0@01
  860 DATA @'0I01I'1
  870 DATA @'0@31AE1CG1CJ1BK1@L1@D0BE1CF1CI1AK1@]1
  880 DATA R'1.H0R01
  890 DATA @'0@61BD1DF1FH1DJ1BL1@Z0@*1>D1<F1:H1<J1>L1
  900 DATA @'0RH1.01RH1
  910 :
  920 DEF PROCassemble
  930 link=14
  940 sp=13
  950 FOR pass=0 TO 2 STEP 2
  960   P%=code%
  970   [ OPT pass
  980   STMFD (sp)!,{link}
  990   MOV R10,R3
 1000   SWI &112
 1010   SWI &100
 1020   MOV R0,R5
 1030   SWI "OS_WriteC"
 1040   MOV R7,R1
 1050   MOV R8,R2
 1060   SUB R2,R2,#4
 1070   BL char
 1080   SUB R1,R7,#4
 1090   MOV R2,R8
 1100   MOV R3,R10
```

```
1110   BL char
1120   MOV R1,R7
1130   ADD R2,R8,#4
1140   MOV R3,R10
1150   BL char
1160   ADD R1,R7,#4
1170   MOV R2,R8
1180   MOV R3,R10
1190   BL char
1200   SWI &112
1210   SWI &100
1220   MOV R0,R6
1230   SWI "OS_WriteC"
1240   MOV R1,R7
1250   MOV R2,R8
1260   MOV R3,R10
1270   BL frontChar
1280   LDMFD (sp)!,{PC}
1290   :
1300   .frontChar
1310   STMFD (sp)!,{link}
1320   MOV R5,R1
1330   BL char
1340   MOV R0,#24
1350   MLA R0,R4,R0,R5
1360   LDMFD (sp)!,{PC}
1370   :
1380   .char
1390   MOV R0,#4
1400   SWI "OS_Plot"
1410   :
1420   .char_loop
1430   LDRB R1,[R3],#1
1440   CMP R1,#32
1450   MOVLT PC,link
1460   SUB R1,R1,#64
1470   MUL R1,R4,R1
1480   LDRB R2,[R3],#1
1490   SUB R2,R2,#72
1500   MUL R2,R4,R2
1510   LDRB R0,[R3],#1
1520   SUB R0,R0,#48
1530   SWI "OS_Plot"
1540   B char_loop
1550   ]
1560 NEXT
1570 ENDPROC
1580 :
1590 DEF PROCloadscores
1600 FOR i%=0 TO 7
1610   score$(i%)="Little Me"
1620   score%(i%)=8000-i%*1000
```

```
 1630 NEXT
 1640 f%=OPENIN file$
 1650 IF f% THEN
 1660   FOR i%=0 TO 3
 1670     INPUT# f%,score$(i%),score%(i%)
 1680   NEXT
 1690   CLOSE# f%
 1700 ENDIF
 1710 ENDPROC
 1720 :
 1730 DEF PROCtable(newscore%)
 1740 LOCAL I%,i%,col%
 1750 MODE 12
 1760 OFF
 1770 IF newscore%>=score%(7) PROCinsert
 1780 PROCoutline(256,944,"DEVILISH DEMOS",2,3,6)
 1790 FOR i%=0 TO 7
 1800   IF score%(i%)=newscore% THEN
 1810     col%=5
 1820     newscore%+=1
 1830     ELSE
 1840     col%=3
 1850   ENDIF
 1860   PROCoutline(64,832-i%*96,score$(i%)+STRING$(23-LENscore$(i%)-L
EN STR$ score%(i%),".")+STR$ score%(i%),2,1,col%)
 1870 NEXT
 1880 IF newscore%>=score%(3) THEN
 1890   f%=OPENOUT file$
 1900   FOR i%=0 TO 3
 1910     PRINT# f%,score$(i%),score%(i%)
 1920   NEXT
 1930   CLOSE# f%
 1940 ENDIF
 1950 PROCoutline(128,32,"Hit spacebar to play",2,6,3)
 1960 REPEAT
 1970   *FX 21
 1980 UNTIL NOT INKEY-99
 1990 REPEAT
 2000 UNTIL GET=32
 2010 ENDPROC
 2020 :
 2030 DEFPROCinsert
 2040 LOCAL char%,i%,mark%
 2050 PROCoutline(96,800,"CONGRATULATIONS",3,3,2)
 2060 IF newscore%>=score%(0) THEN
 2070   PROCoutline(224,560,"WOW",4,6,7)
 2080   PROCoutline(608,560,"Top Score",2,1,3)
 2090   ELSE
 2100   PROCoutline(0,656,"You have gained a place in",2,3,1)
 2110   PROCoutline(0,528,"the DEMO hall of fame.",2,3,1)
 2120 ENDIF
 2130 mark%=8
 2140 REPEAT
 2150   mark%-=1
 2160   score$(mark%+1)=score$(mark%)
 2170   score%(mark%+1)=score%(mark%)
 2180 UNTIL score%(mark%)>newscore% OR mark%=0
 2190 IF score%(mark%)>newscore% mark%+=1
 2200 score%(mark%)=newscore%
 2210 i%=0
 2220 char%=32
 2230 U%?i%=13
 2240 PROCoutline(0,336,"Please enter your name.",2,3,1)
 2250 PROCoutline(0,120,"-------------",2,4,6)
 2260 WHILE i%<14 AND char%<>13
 2270   REPEAT
 2280     char%=GET
 2290     IF char%=127 THEN
 2300       char%=0
 2310       IF i%>0 THEN
 2320         GCOL 0,0
 2330         RECTANGLE FILL i%*48-52,164,56,104
 2340         i%=i%-1
 2350       ENDIF
 2360     ENDIF
 2370     IF char%>31 AND i%>12 char%=0
 2380     IF char%>=ASC "a" char%-=32
 2390     IF char%=13 AND i%=0 PROCsilly ELSE IF char%<ASC"!" AND i%=0
char%=0
 2400   UNTIL char%=13 OR char%=32 OR (char%>=ASC"A" AND char%<=ASC "Z
")
 2410   IF char%>=ASC"A" THEN
 2420     IF i%>0 AND U%?(i%-1)>32 char%=char% OR 32
 2430     PROCoutline(i%*48,192,CHR$ char%,2,4,6)
 2440   ENDIF
 2450   U%?i%=char%
 2460   i%=i%+1
 2470 ENDWHILE
 2480 score$(mark%)=$U%
 2490 CLS
 2500 ENDPROC
 2510 :
 2520 DEF PROCsilly
 2530 LOCAL j%
 2540 RESTORE+5
 2550 FOR j%=1 TO RND(6)
 2560   READ $U%
 2570 NEXT
 2580 i%=LEN $U%
 2590 ENDPROC
 2600 DATA Willie Wibble,Mickey Mouse,Fred Flatfoot,Bossy Bess,Moaning
Megan,Fuzzy Bear
 2610 :
 2620 DEFPROCoutline(B%,C%,$T%,E%,F%,G%)
```

```
2630 FOR I%=0 TO LEN $T%-1
2640   $D%=font$(T%?I%-32)
2650   B%=USR code%
2660 NEXT
2670 ENDPROC
```

The data lines are in a special highly compacted form, and each line represents all the draw commands for a different character. Only the numbers, upper and lower case letters, full stop, and minus are provided. The other characters would just waste space.

The routine first fills in a dummy score table, then looks for a top half to load from disk. There are then four demo calls to the score table covering all the possibilities: top score, saveable score, non saveable score and below table score. You will notice that the routine only actually saves the score table if a new entry appears in the top four places.

Suitable messages are given for two scoring situations. A score too low is quietly ignored to save embarrassment, and no note is made of whether a score will be saved or not. You can easily change this if you want to.

Entering a player's name, not only thoroughly traps unwanted characters, but also intelligently sets upper and lower case letters in the name. If no name is entered then one of six silly names is chosen at random.

8

Role Play

Role play games are regarded here as both a generic term and a specific game type. As a generic term it covers all types of game where you are taking on a simulated task of some sort, whether it be that of a band of thieves in a forest, an airline pilot or simply a bank manager.

8.1 RPGs

Role play games as a specific game type, don't originate with computers at all, but with people acting out character parts within a set of rules and guidelines set out by the creator of the RPG. Generally, a *world* is described where the players live, fight and perform almost all the normal human activities. As the scenario is artificial, the game maker can make available whatever characteristics or new sciences he or she likes. The favourite addition is, of course, magic.

RPGs are generally open ended. There is often no specific task to perform, but general guidelines are given for improving your status in the game world. From there on, as in the real world, you learn from experience. For this reason you must be absolutely consistent in any rules you apply. Your players will rapidly lose interest if they find that the same monster is dispatched in total different ways simply because they meet it in a different area of the game.

RPGs fit very easily into computers. At a basic level you can write one using entirely text instructions and keyboard input. The game can be completely open ended, in that there need not be a specific set of tasks to perform. This is the typical scenario of the Cells and Serpents type games. Your players can simply wander around your artificial world picking up treasures and becoming steadily more adept at dealing with obstacles puzzles and monsters.

The first essential point is that you must have a map of the overall layout of your game. This can be based on a two dimensional grid or, if the scenario is a castle for example, you can draw out a three dimensional plan. All treasure, wizards and monsters can be identified simply by grid reference. If your player has the same reference, he is in the same area and you can provide a brief description of the scene and what options are available.

This structure actually lends itself very well to desktop working as you can use a set of icons for both objects and activity choices. These can be dropped on a simple sprite background within a window, with mouse clicks over the icons for various actions. In this way no text is needed at all.

Taking a two dimensional forest game world with textual and graphic components as an example, you can maintain a two dimensional location map of place description strings, and a parallel array with sprite lists for the major drawing. In this simple plan you could limit the area to between 0 and 8 locations in both the X and Y directions. Your player starts off with area coordinates of 4,4. In the absence of any specific alternative, the game can assume a standard trees and shrubs background.

Although the world is two dimensional the scene can be drawn as a full three dimensional picture. You now need a string array with a list of all the objects, along with a parallel array giving their X,Y map coordinates and object type. The arrays would also include the screen plotting position and sprite number of the graphic representation. These arrays are then scanned to see if any objects are located at 4,4, and those that are, plotted and described.

Anything picked up should be marked as being at location –1,0. This can be scanned when the object needs to be used, and as we have no negative elements in the map, only the X coordinate need to be checked. Location –2,0 can be used for objects or monsters that are destroyed or hidden. Your game can randomly re-generate these from time to time, in new locations. This gives you an apparently limitless supply of creatures to use. For items that are permanently destroyed, never to be used again, I suggest –3,0 as their location

Using ordinary X,Y movement controls your player can move to 4,5 or 5,4 or 3,4 or 4,3. Logically, these would correspond to East, North, West and South respectively. Obviously directions like 5,5 would give Northeast. Once at the new location, the arrays can be scanned again and the new scene drawn up.

In Listing 8.1 is a skeleton of just this idea, but with minimal text descriptions rather than a graphical display.

Listing 8.1: Role playing

```
   10 REM > RPG
   20 :
   30 PROCinitialise
   40 action%=1
   50 REPEAT
   60   IF action% PROCdescribe
   70   action%=GET
   80   CASE action% OF
   90     WHEN ASC"Z":IF X%>0 X%-=1
  100     WHEN ASC"X":IF X%<wide% X%+=1
  110     WHEN ASC"/":IF Y%>0 Y%-=1
  120     WHEN ASC"'":IF Y%<high% Y%+=1
  130     WHEN ASC"C":PROCcollect
  140     WHEN ASC"F":PROCfight
  150     OTHERWISE action%=0
  160   ENDCASE
  170 UNTIL FALSE
  180 END
  190 :
  200 DEF PROCinitialise
  210 MODE 12
  220 OFF
  230 PRINT TAB(31,5) "Demo RPG Game"
  240 X%=4:Y%=4
  250 xloc%=0:yloc%=1:type%=2
  260 collect%=FALSE
  270 fight%=FALSE
  280 RESTORE+0
  290 READ wide%,high%
  300 DIM place$(wide%,high%)
  310 FOR J%=0 TO wide%
  320   FOR I%=0 TO high%
  330     READ place$(I%,J%)
  340     IF place$(I%,J%)="" place$(I%,J%)="You are deep in the fores
t. Tall pines block your view."
  350   NEXT
  360 NEXT
  370 READ numobs%
  380 DIM object%(numobs%,2),object$(I%)
  390 FOR I%=0 TO numobs%
  400   READ object%(I%,xloc%),object%(I%,yloc%),object%(I%,type%),obj
ect$(I%)
  410 NEXT
  420 ENDPROC
  430 DATA 7,7
  440 DATA Loc.0/0,,,,,You are by a swift river,You are on open heath,
This is the wide road to Hanri
  450 DATA ,,,,,,You are near a babbling brook,You are on an old grave
l road
  460 DATA ,,,,,,You are by a crystal stream,You find you are on a nar
row track
  470 DATA ,,,,,,,
  480 DATA Loc.0/4,,,,,,,LOC.8/4
  490 DATA ,,,,,,,
  500 DATA ,,,,,,,
  510 DATA Loc.8/0,,,,LOC.8/4,,,Loc.8/8
  520
  530 DATA 2
  540 DATA 4,4,0,A stone seat
  550 DATA 3,4,1,A silver chalice
  560 DATA 2,4,2,An ugly Troll
  570 :
  580 DEF PROCdescribe
  590 collect%=FALSE
  600 fight%=FALSE
  610 PRINT"'place$(X%,Y%)
  620 FOR I%=0 TO numobs%
  630   IF object%(I%,xloc%)=X% AND object%(I%,yloc%)=Y% THEN
  640     PRINT object$(I%) " is here"
  650     CASE object%(I%,type%) OF
  660       WHEN 1:collect%=TRUE
  670       WHEN 2:fight%=TRUE
  680     ENDCASE
  690   ENDIF
  700 NEXT
  710 PRINT' "Options:""SPC 5 "Z - West"'SPC 5 "X - East"'SPC 5 "' -
North"'SPC 5 "/ - South"
  720 IF collect% PRINT SPC 5 "C - Collect"
  730 IF fight% PRINT SPC 5 "F - Fight"
  740 PRINT"'Escape - Stop"""Your choice :"
  750 ENDPROC
  760 :
  770 DEF PROCcollect
  780 action%=0
  790 IF NOT collect% ENDPROC
  800 FOR I%=0 TO numobs%
  810   IF object%(I%,xloc%)=X% AND object%(I%,yloc%)=Y% AND object%(I
%,type%)=1 THEN
  820     object%(I%,xloc%)=-1
  830     PRINT object$(I%) " - carried"
  840   ENDIF
  850 NEXT
  860 collect%=FALSE
  870 ENDPROC
  880 :
  890 DEF PROCfight
  900 action%=0
  910 IF NOT fight% ENDPROC
  920 FOR I%=0 TO numobs%
  930   IF object%(I%,xloc%)=X% AND object%(I%,yloc%)=Y% AND object%(I
%,type%)=2 THEN
  940     object%(I%,xloc%)=-2
  950     PRINT object$(I%) " - killed"
  960   ENDIF
  970 NEXT
  980 fight%=FALSE
  990 ENDPROC
```

Although there is primitive fight recognition, full combat sequences are a bit more complicated as you have to give a real air of urgency, and at the same time enable the player to respond quickly to the changing situation. It may well pay to expand the object types into parallel arrays of friends, foes, treasures, and utility objects.

Each player takes control of a character in the game, each typically with the following attributes:

Strength	combat
Psi	magical ability
Intelligence	general capability
Experience	learned abilities

A player's ability to win any combat depends on matching these attributes against those of the adversary, and then assessing which actions are attempted and the response time.

Combat is usually broken down to melee rounds, and the player attributes updated after each round. As well as giving players strike by strike control, this allows you to generate multi-character combat. Only one-to-one fights take place in each round, but by alternating the characters you can give the appearance of a real gang fight.

A simple way of achieving this is to set up a pair of arrays containing character attributes. One array is for goodies and the other for baddies. At random you can then match any goodie against any baddie. If any character dies, its position in the array is taken by another and the array compacted. The combat session finishes when one or other array contains no more characters. There is an outline of this technique in Listing 8.2.

Listing 8.2: Combat

```
   10 REM > Combat
   20 :
   30 PROCinitialise
   40 PROCplayer
   50 PROCfight
   60 PROCendgame
   70 END
   80 :
   90 DEF PROCinitialise
  100 MODE 12
  110 OFF
  120 RESTORE+35
  130 READ weapons%
  140 DIM weapon$(weapons%)
  150 FOR I%=1 TO weapons%
  160   READ weapon$(I%)
  170 NEXT
  180 sword%=1
  190 spell%=2
  200 feint%=3
  210 READ attribnum%
  220 line$="  Character       Strength     Psi   Intelligence  Experie
nce"
  230 strength%=1
  240 psi%=2
  250 intelligence%=3
  260 experience%=4
  270 READ goodies%,baddies%
  280 IF goodies%>baddies% size%=goodies% ELSE size%=baddies%
  290 DIM attributes%(1,size%,attribnum%):REM goodies/baddies, charact
ers,size%
  300 DIM gang%(1)
  310 DIM name$(1,size%)
  320 FOR I%=1 TO goodies%
  330   READ name$(0,I%)
  340   FOR J%=1 TO attribnum%:REM zero element used as player flag
  350     READ attributes%(0,I%,J%)
  360   NEXT
  370 NEXT
  380 FOR I%=1 TO baddies%
  390   READ name$(1,I%)
  400   FOR J%=1 TO attribnum%
  410     READ attributes%(1,I%,J%)
  420   NEXT
  430 NEXT
  440 PRINT""Test Combat sequence"'
  450 VDU 28,0,31,79,VPOS
  460 ENDPROC
  470 DATA 3
```

```
480 DATA Sword,Spell,Feint
490 DATA 4
500 DATA 3,3
510 DATA Dwarf,50,10,9,1
520 DATA Elf,10,50,15,1
530 DATA Wizard,5,60,20,1
540 DATA Troll,50,5,5,1
550 DATA Minataur,20,30,15,1
560 DATA Pixie,5,60,15,1
570 :
580 DEF PROCplayer
590 PRINT "Select your character by number"
600 gang%(0)=goodies%
610 gang%(1)=baddies%
620 PROClist(0)
630 REPEAT
640   num%=GET-48
650 UNTIL num%>0 AND num%<=goodies%
660 CLS
670 attributes%(0,num%,0)=TRUE:REM set player flag
680 REM could be repeated for several players & with baddies too
690 ENDPROC
700 :
710 DEF PROClist(flag%)
720 LOCAL I%,J%
730 PRINT'line$
740 FOR I%=1 TO gang%(flag%)
750   PRINT';I% TAB(2) name$(flag%,I%);
760   FOR J%=1 TO attribnum%
770     PRINT TAB(J%*11) attributes%(flag%,I%,J%);
780   NEXT
790 NEXT
800 ENDPROC
810 :
820 DEF PROCfight
830 LOCAL attack%,defend%
840 gang%(0)=goodies%:REM temp store for fight only
850 gang%(1)=baddies%
860 REPEAT
870   CLS
880   PRINT"'Goodies"
890   PROClist(0)
900   PRINT""'Baddies"
910   PROClist(1)
920   PRINT'
930   attack%=RND(2)-1
940   defend%=1-attack%
950   assail%=RND(gang%(attack%))
960   IF assail%=0 assail%=1
970   IF attributes%(attack%,assail%,0) PROCselect ELSE PROCmatch
980   PROCstrike
990 UNTIL gang%(attack%)=0 OR gang%(defend%)=0
```

```
 1000 ENDPROC
 1010 :
 1020 DEF PROCselect
 1030 PRINT name$(attack%,assail%) ", select your opponent by number"'
 1040 REPEAT
 1050   oppose%=GET-48
 1060 UNTIL oppose%>0 AND oppose%<=gang%(defend%)
 1070 PRINT"'Select your weapon by number"'
 1080 FOR I%=1 TO weapons%
 1090   PRINT"(";I%")   " weapon$(I%)
 1100 NEXT
 1110 REPEAT
 1120   type%=GET-48
 1130 UNTIL type%>0 AND type%<=weapons%
 1140 ENDPROC
 1150 :
 1160 DEF PROCmatch
 1170 oppose%=RND(gang%(defend%))
 1180 IF oppose%=0 oppose%=1
 1190 IF attributes%(attack%,assail%,strength%)>attributes%(defend%,op
pose%,strength%) type%=1 ELSE IF attributes%(attack%,assail%,psi%)>att
ributes%(defend%,oppose%,psi%) type%=2 ELSE type%=3
 1200 ENDPROC
 1210 :
 1220 DEF PROCstrike
 1230 PRINT "The " name$(attack%,assail%) " attacks the " name$(defend
%,oppose%) " ";
 1240 CASE type% OF
 1250   WHEN sword%:PROCsword
 1260   WHEN spell%:PROCspell
 1270   WHEN feint%:PROCfeint
 1280 ENDCASE
 1290 IF INKEY 150
 1300 ENDPROC
 1310 :
 1320 DEF PROCsword
 1330 LOCAL diff%,sum%
 1340 diff%=attributes%(attack%,assail%,intelligence%)*attributes%(att
ack%,assail%,experience%)-attributes%(defend%,oppose%,intelligence%)*a
ttributes%(defend%,oppose%,experience%)
 1350 IF diff%>90 diff%=90
 1360 sum%=(attributes%(attack%,assail%,strength%)+attributes%(defend%
,oppose%,strength%))DIV 4+1
 1370 PRINT "with a blade of true steel."
 1380 IF RND(9)=1 THEN
 1390   PRINT "Fumbled! The " name$(defend%,oppose%) " has a lucky esc
ape."
 1400   ELSE
 1410   IF RND(diff%)>30 THEN
 1420     PRINT "The " name$(defend%,oppose%) " is too clever to be ca
ught so easily."
 1430     ELSE
 1440      PRINT "A hit. ";
 1450      attributes%(defend%,oppose%,strength%)-=sum%
 1460      IF attributes%(defend%,oppose%,strength%)<1 THEN
 1470        PROCdead(defend%,oppose%)
 1480        attributes%(attack%,assail%,strength%)+=(sum% DIV 2)
 1490        attributes%(attack%,assail%,experience%)+=1
 1500        ELSE
 1510        PRINT "The " name$(defend%,oppose%) " is still strong."
 1520        attributes%(attack%,assail%,strength%)-=(sum% DIV 2)
 1530       IF attributes%(attack%,assail%,strength%)<1 PROCdead(attac
k%,assail%)
 1540      ENDIF
 1550    ENDIF
 1560 ENDIF
 1570 ENDPROC
 1580 :
 1590 DEF PROCspell
 1600 LOCAL diff%
 1610 diff%=attributes%(attack%,assail%,intelligence%)-attributes%(def
end%,oppose%,intelligence%)
 1620 IF diff%>20 diff%=20
 1630 PRINT "with a strange enchantment."
 1640 IF RND(9)=1 THEN
 1650    PRINT "The " name$(defend%,oppose%) " ducks the spell."
 1660    attributes%(defend%,oppose%,experience%)+=1
 1670    ELSE
 1680    IF attributes%(attack%,assail%,psi%)>attributes%(defend%,oppos
e%,psi%) AND diff%>3 THEN
 1690     PRINT "The " name$(defend%,oppose%) " is ensnared by the spell."
 1700      attributes%(attack%,assail%,psi%)+=1
 1710      attributes%(attack%,assail%,experience%)+=1
 1720      attributes%(defend%,oppose%,strength%)-=(attributes%(defend%
,oppose%,strength%) DIV 3)
 1730      IF attributes%(defend%,oppose%,strength%)<1 PROCdead(defend%
,oppose%)
 1740      ELSE
 1750      PRINT "The " name$(attack%,assail%) " hasn't the mental powe
r to overcome the " name$(defend%,oppose%)
 1760      attributes%(defend%,oppose%,experience%)+=1
 1770    ENDIF
 1780 ENDIF
 1790 ENDPROC
 1800 :
 1810 DEF PROCfeint
 1820 PRINT "by a cunning feint."
1830 IFattributes%(attack%,assail%,intelligence%)>attributes%(defend
%,oppose%,intelligence%) THEN
 1840    IF attributes%(attack%,assail%,experience%)>attributes%(defend
%,oppose%,experience%) DIV 2 THEN
 1850     PRINT "It fools the " name$(defend%,oppose%) ", but saps strength."
 1860      attributes%(attack%,assail%,str
ength%)-=2
```

```
1870    ENDIF
1880    ELSE
1890   PRINT "The " name$(defend%,oppose%) " laughs and continues the fight."
1900    attributes%(defend%,assail%,experience%)+=1
1910  ENDIF
1920 ENDPROC
1930 :
1940 DEF PROCdead(flag%,character%)
1950 PRINT "The " name$(flag%,character%) " dies."
1960 IF character%<gang%(flag%) PROCmovedown
1970 gang%(flag%)-=1
1980 ENDPROC
1990 :
2000 DEF PROCmovedown
2010 LOCAL I%,J%
2020 FOR I%=character% TO gang%(flag%)-1
2030    name$(flag%,I%)=name$(flag%,I%+1)
2040    FOR J%=0 TO attribnum%
2050      attributes%(flag%,I%,J%)=attributes%(flag%,I%+1,J%)
2060    NEXT
2070 NEXT
2080 ENDPROC
2090 :
2100 DEF PROCendgame
2110 LOCAL flag%
2120 CLS
2130 IF gang%(0)>0 THEN
2140    FOR I%=1 TO gang%(0)
2150      IF attributes%(0,I%,0) flag%=TRUE
2160    NEXT
2170    IF NOT flag% PRINT "Unfortunately your character died but t";
ELSE PRINT "T";
2180    PRINT "he good guys won the fight."
2190    ELSE
2200    IF gang%(1)>0 THEN
2210      PRINT "The baddies rule OK!"
2220      ELSE
2230      PRINT "Everyone died. There are no victors."
2240    ENDIF
2250 ENDIF
2260 VDU 26
2270 PRINT TAB(0,5)
2280 ON
2290 ENDPROC
```

A problem that can often arise, is where a character, apparently dying, suddenly unleashes a spell of enormous power that completely destroys the enemy, and yet still hasn't the strength to pick up a sword. To overcome this you should make character death occur on the basis of several attributes falling below a certain point, rather than any single one dropping to zero. To add further realism, instead of working directly with the stored figures for attributes, work from a continuously updated set of inter-related ones.

If you decide to write an RPG, you must take the trouble to ensure that any magic or pseudo-science you devise is consistent. There is nothing more irritating than finding that, say, a firemaking spell produces a roaring inferno when you have a few wet twigs but not so much as a sniff of smoke from a bundle of old newspapers.

Combining the RPG map routine with the combat program will give you a basic text only RPG. This is obviously too crude for today's players, but there is enough there to give you an idea as to how you can develop your own ideas into a fully fledged graphical game.

8.2 Adventures

Adventure games are often thought of as RPGs with the combat section removed, although that is a rather simplistic view. In the first place, adventures tend to be more focussed, in that there is a specific set of tasks to be performed, usually in a fixed order. Similarly there is usually a fixed number of objects, monsters and the like.

8.2.1 Rooms

The map structure is not so rigidly defined, but moving to different locations in the game world should be reasonably logical. Going East from a room that you travelled West to reach should take you back to your original location. The exception is in mazes, where experienced players will expect peculiar directions. Unlike RPGs, adventures don't usually have all possible locations set out in a grid. It's normally much more of a free-flowing map.

Because of this, the usual representation of the game map, is not a simple two dimensional array, but an array of pointers and links. The player's current location is simply a location number, this being an index to the main location array. This is often referred to as a room list. Typically there will be four different other rooms that can be reached from any one room. Testing an attempt to move North say, would involve looking at the first direction link, assuming that this is to represent North, and seeing which room number it points to. Zero would mean that this direction is closed off. This map representation is shown in Figure 8.1.

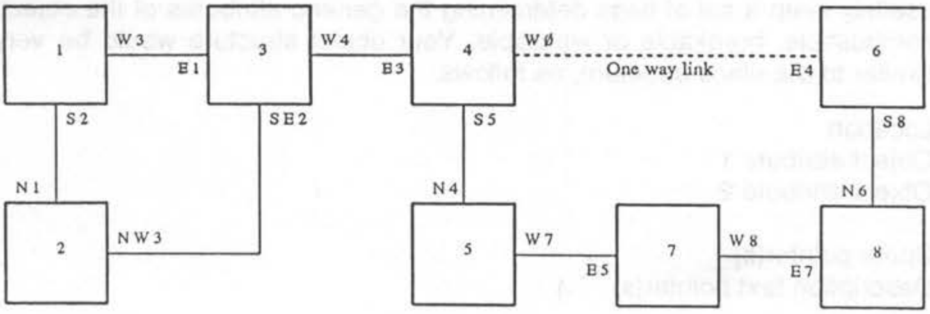

Figure 8.1: Adventure game room map

Notice how the room numbers don't need to follow any special pattern. This is particularly useful when you want to open or close links as the game progresses. Notice the one way link between rooms 6 and 4. This is a very common way of dealing with cliff tops, pits and the like, where the player can get in, but not back out again. Keeping the West option reserved in room 4 allows you to create a magic door back later in the game.

The room list will also have flags for any special characteristics that each room may have. Caves would need light for example. Below is a typical room list structure. This can conveniently be held in parallel arrays, with the text information a string array. The array index numbers would be the actual room numbers.

Direction link 1
Direction link 2
......
Room attribute 1
Room attribute 2
......
Sprite pointer(s)
Description text pointer(s).

8.2.2 Locating objects

Once you have your map sorted out, you need to consider how objects are to be placed. This is most easily done in a similar manner to that of RPGs. However, instead of storing X,Y coordinates you simply have an array of objects storing room numbers. The array index number itself is a unique identifier, so you don't need a type element in the array. However, you can usefully keep a set of flags determining the generic attributes of the object: combustible, breakable or wearable. Your object structure would be very similar to the place structure, as follows:

Location
Object attribute 1
Object attribute 2
......
Sprite pointer(s)
Description text pointer(s).

All you need to do now for descriptions, or for any other object handling, is scan the list, picking out any object marked as at the room in question. It is usual to mark room 0 as hidden objects, room –1 as carried, and room –2 as worn.

One possible source of confusion arises when an object can be carried by another object. Does location 5 refer to room 5 or object 5? The solution is very simple. All you do is add an offset of say, &1000 to the object numbers. You are never likely to want 4,096 rooms and it can easily be masked in and out of calculations and array indices as follows:

```
index%=object% AND &FFF
object%=index% OR &1000
```

8.2.3 Understanding instructions

The core of a text adventure game is the parser. This is the part of the program that reads in a player's input and works out exactly what is being requested. All words read from the text are given number representation. If no known words are found then the word identifiers are set to zero. Older adventures used to recognise simple verb/noun combinations such as *Get Brick* or *Drop Fish*. Later versions scanned the text for just these two combinations but were able to skip over and discard as rubbish, any unwanted words. This didn't improve the flexibility of the parser but it did allow more natural sentences to be decoded.

A useful addition to this simple parser is the handling of prepositions. These are words that indicate how a noun is used or where it may be found such as:

Put hat on peg
Sweep floor with broom
Look under rug
Hide key in pocket.

Two other useful additions are adjectives and adverbs. Adjectives would be noun modifiers giving results like:

Get the green bottle
Find a rough spot
Examine the open box.

Adverbs, as their name suggests are verb modifiers. Using all of these word types will give remarkably intelligent decoding results. All the following can be recognised by such a parser:

Quietly enter the open door
Carefully drop a silver coin into the metal bucket
Eat the big cake quickly.

It is doubtful whether there is much to be gained by taking your parser much further, even though it might be an interesting challenge to produce really sophisticated parser, of the kind that can handle a sentence like:

Use the string to tie all the keys except the red one to the tag and hang them up.

Most game players will quickly revert to quick-fire three or four word commands, with only the occasional attempt at something more exotic when all else fails. Therefore effort spent in developing your parser would, sadly, be wasted.

A somewhat ticklish area is that of handling obscenities. Most adventurers get frustrated sometimes, and then use words that no decent computer wants to read. If you decide to recognise such words, you must take great care to ensure that it is completely impossible for your game player to accidentally reveal them, otherwise you could cause very real offence to an innocent player. You are best off recognising the offending words as verbs and then giving simple responses like:

I don't like that kind of language
You can't do that sort of thing in this game
This is a family game.

You can keep an obscenity count if you like, and after a couple of warnings terminate the player with an act of the supreme being's displeasure.

A working verb, noun, adverb, adjective parser is shown in Listing 8.3

Listing 8.3: A working parser

```
10 REM > Parser
20 :
 30 PROCinitialise
  40 REPEAT
  50   REPEAT
  60     command$=FNinput
  70   UNTIL command$>""
  80   PROCparse(command$)
  90   PRINT "Verb=";ver%,"Noun1=";no1%,"Noun2=";no2%,"Prep.=";pre%,"
Adj.1=";ad1%,"Adj.2=";ad2%,"Adverb=";adv%'
 100 UNTIL command$="*"
 110 VDU 26
 120 PRINT TAB(0,30)
 130 END
 140 :
 150 DEF PROCinitialise
 160 MODE 12
 170 READ numverbs%
 180 DIM verb$(numverbs%),verb%(numverbs%)
 190 PRINT "Verbs"
 200 FOR I%=0 TO numverbs%
 210   READ verb$(I%),verb%(I%)
 220   PRINT TAB(I%*10) verb$(I%);
 230 NEXT
 240 READ numnouns%
 250 DIM noun$(numnouns%),noun%(numnouns%)
 260 PRINT''"Nouns"
 270 FOR I%=0 TO numnouns%
 280   READ noun$(I%),noun%(I%)
 290   PRINT TAB(I%*10) noun$(I%);
 300 NEXT
 310 READ numpreps%
 320 DIM prep$(numpreps%),prep%(numpreps%)
 330 PRINT''"Prepositions"
 340 FOR I%=0 TO numpreps%
 350   READ prep$(I%),prep%(I%)
 360   PRINT TAB(I%*10) prep$(I%);
 370 NEXT
 380 READ numadjes%
 390 DIM adje$(numadjes%),adje%(numadjes%)
 400 PRINT''"Adjectives"
 410 FOR I%=0 TO numadjes%
 420   READ adje$(I%),adje%(I%)
 430   PRINT TAB(I%*10) adje$(I%);
 440 NEXT
 450 READ numadves%
 460 DIM adve$(numadves%),adve%(numadves%)
 470 PRINT''"Adverbs"
 480 FOR I%=0 TO numadves%
 490   READ adve$(I%),adve%(I%)
 500   PRINT TAB(I%*10) adve$(I%);
 510 NEXT
 520 PRINT''"Enter sentence to parse (capitals only), or * to stop."
 530 VDU 28,0,31,79,VPOS+1
```

```
  540 ENDPROC
  550 DATA 7:REM verbs
  560 DATA GO,1,GET,2,CARRY,2,DROP,3,KILL,4,HIT,5,EXAMINE,6,FIND,7
  570 DATA 6:REM nouns
  580 DATA KEY,1,DOG,2,GLASS,3,BEAKER,3,BOX,4,STONE,5,ROCK,5
  590 DATA 11:REM prepositions
  600 DATA INSIDE,1,IN,1,OUTSIDE,2,UNDER,3,OVER,4,ON,5,ONTO,5,BESIDE,6
,BY,6,AGAINST,6,WITH,7,USING,7
  610 DATA 7:REM adjectives
  620 DATA RED,1,BLUE,2,GREEN,3,YELLOW,4,BIG,5,LITTLE,6,ROUGH,7,SMOOT
H,8
  630 DATA 4:REM adverbs
  640 DATA QUICKLY,1,SLOWLY,2,QUIETLY,3,GENTLY,4,HEAVILY,5
  650 :
  660 DEF FNinput
  670 LOCAL c$
  680 INPUT c$
  690 IF NOT FNmore THEN
  700   WHILE RIGHT$(c$,1)=" "
  710     c$=LEFT$(c$,LEN c$-1)
  720   ENDWHILE
  730 ENDIF
  740 =c$
  750 :
  760 DEF PROCparse(c$)
  770 ver%=0:no1%=0:no2%=0:pre%=0:ad1%=0:ad2%=0:adv%=0
  780 REPEAT
  790   flag%=FNmore
  800   PROCword(verb$(),verb%(),ver%,numverbs%)
  810   PROCword(noun$(),noun%(),no1%,numnouns%)
  820   PROCword(noun$(),noun%(),no2%,numnouns%)
  830   PROCword(prep$(),prep%(),pre%,numpreps%)
  840   IF no1%=0 PROCword(adje$(),adje%(),ad1%,numadjes%) ELSE PROCwo
rd(adje$(),adje%(),ad2%,numadjes%)
  850   PROCword(adve$(),adve%(),adv%,numadves%)
  860   IF flag%=FALSE PROCdiscard
  870 UNTIL c$=""
  880 ENDPROC
  890 :
  900 DEF FNmore
  910 WHILE LEFT$(c$,1)=" "
  920   c$=RIGHT$(c$,LEN c$-1)
  930 ENDWHILE
  940 =c$=""
  950 :
  960 DEF PROCword(t$(),t%(),RETURN w%,n%)
  970 LOCAL a%,w$
  980 IF w%>0 OR flag% ENDPROC
  990 a%=INSTR(c$," ")
 1000 IF a%=0 THEN
 1010   w$=c$
 1020   ELSE
```

```
1030   w$=LEFT$(c$,a%-1)
1040 ENDIF
1050 I%=-1
1060 REPEAT
1070   I%+=1
1080 UNTIL I%>=n% OR t$(I%)=w$
1090 IF t$(I%)=w$ THEN
1100   w%=t%(I%)
1110   IF a%=0 c$="" ELSE c$=MID$(c$,a%+1)
1120   flag%=TRUE
1130 ENDIF
1140 ENDPROC
1150 :
1160 DEF PROCdiscard
1170 LOCAL a%,w$
1180 a%=INSTR(c$," ")
1190 IF a%=0 THEN
1200   c$=""
1210   ELSE
1220   c$=MID$(c$,a%+1)
1230 ENDIF
1240 ENDPROC
```

To extend the vocabulary all you need to do is alter the data lines, increasing the word counter, and fit in the words. You will see that several words can have the same reference number which is essential to allow for players using variants of the same command.

Notice how, as it strips out words from the input text, the parser re-checks for word types it may have missed. This is to cope with the vagaries of English grammar that allow adverbs, in particular, to be put almost anywhere. Both of the following will be correctly read by the example parser:

Softly stroke the cat
Stroke the cat softly.

Any words that can't be matched at all are dumped by the discard procedure. This lets your player use all the common redundant conjunctions, giving the feel of real understanding from your game.

8.2.4 Finding nouns

While in a graphic adventure, you can isolate objects, and hence their nouns simply and unambiguously with a simple mouse click. With a text adventure things are much more difficult. There are two basic approaches to resolving this problem. The first and commonest, is to simply keep a list of words, as was done in the example parser. However, as with verbs you need to allow for the player using a similar but not identical word. Take the sentence:

You are on a rocky hill path. Sharp flinty stones are all around.

If you have a stone as an object, your player could quite easily describe it as a rock or a flint, and be most annoyed if the game refuses to understand. This means that you need to look at your text very carefully and make sure that you have covered all reasonable possibilities with duplicate noun names. Once you have a match, you need to make sure that the object referred to is actually there, or has been seen by the player. To simply say: *You don't have it yet* is a dead giveaway that the object actually exists somewhere in the game.

An alternative method of finding nouns, is to use the description text itself. You scan all the text for the current room, and all objects in that room, a word at a time. By doing this you can guarantee that the object or place exists, and also that the player can see it. All you need to do is keep a pointer to the text you are scanning at the time the match is made. This is slightly slower than the other method but far more reliable.

8.2.5 Puzzles

Having described and identified everything in the game you now need to do something with that information. Adventure generator programs use quite a complex pseudo language for building up sets of responses to player input. These puzzle systems can consist of many hundreds of lines that are tested until an exact match is found with the action attempted by the player. However, in a home grown game you can get equally good results by using a simple tree structure.

The first priority is to isolate your verbs. This is readily done with a case statement:

```
CASE verb% OF
   WHEN 1:PROCcarry
   WHEN 2:PROCdrop
   WHEN 3:PROCthrow
   OTHERWISE:PROCimpossible
ENDCASE
```

Taking the carry situation you can remove some of the tedious checks as below.

```
CASE TRUE OF
   WHEN carried%>maxc%:PROCtoomany
   WHEN weight%>maxw%:PROCtooheavy
```

```
    WHEN size%>maxs%:PROCtoobig
    WHEN objectloc%<1:PROCalreadycarried
    WHEN objectloc%<&1000:PROCcarryroom
    OTHERWISE:PROCcarryobject
  ENDCASE
```

Notice that there may be some valid situations where you appear to carry a room, but are really handling a hidden object, hence the extra procedure.

Finally, in the Carryobject procedure you can have the individual lines testing specific object characteristics and room locations. Anything not handled specifically, would be passed on to a general pickup routine at the end of the procedure.

8.3 Combat

There is a small group of games that rely on combat only. They still fit in with the general classification of role play however, but have no storyline or special attributes. These are based on martial arts and only really became attractive once there were machines with the ability to run high resolution animated graphics. They are usually single or two player games. The player is given a number of kicks, punches, jumps and rolls, and the character on the screen will perform these in response to a keypress or mouse click.

The graphical part is really very easy to implement. You simply build up a set of film animations for all the actions you require. These will all take place within a fixed character area. You have an identical set of actions, but with a different sprite set, for the opponent, whether it be another player or computer driven.

If you use a variant of coordinate collision testing you can produce a realistic hit system. You relate the damage to your opponent to the distance from the attacking player. Where the two sprites concerned are obviously not in contact at all, the strike just wastes the attacker's energy. As they begin to overlap during the action, you produce a graded energy loss to the character under attack. Hence the reason for keeping the animation within a fixed area.

Usually you only need to keep a single energy variable for each combatant, incrementing it with time and successful encounters, decreasing it with failed encounters or unnecessary action. When the energy falls below a certain level, the character dies. You can give greater realism to this by only allowing the character to perform the more energetic actions, like high kicks, if its energy level is above a certain figure.

Because these games tend to be particularly fast and furious, it is probably best to organise the function keys so that each is assigned a single action. This can be duplicated with a row of matching mouse sensitive icons. This not only allows your player to decide which is more important – speed or keyboard life – but is a useful on screen reminder.

8.4 Simulators

I suppose that simulators can only be thought of as role play in the very broadest sense but nevertheless they still fit the overall classification. The games that most quickly spring to mind when simulations are mentioned are aircraft flight simulators. However, almost any day-to-day activity can provide a basis for simulation. Many education centres use simulated shopping to help teach small children how to handle their money wisely. The logical extension to this is, of course, a trading simulation, where the player runs a large corporate business or even an entire country's economy.

8.4.1 Real world situation

With these real world simulations you need to make a distinction between real time and game time. Logically, if you maintain a ratio of 1 transaction : 1 move, then one year of trading will take you a year to play out on the game. Hardly practical! For many of these games you can use a ratio as coarse as 1 year : 1 move. On the other hand, if you are simulating the running of a power station or chemical works, you would probably work at nearer 1 hour : 1 move.

Unless you are developing a simulation for purely education use, you will need to fine tune the time scale to give a game that is slow enough to be playable without becoming boring. Also, with real world simulators you either need to know a fair bit about statistical analysis, or you will have to develop an idea a bit at a time, and make empirical adjustments to keep the simulation in balance.

As an example I'll outline a simulation for Bodgit, computer manufacturers. Mr Bodgit only makes cheap machines, with no monitor, disk drives, or other accessories. At its basic level, the simulation needs to handle three areas:

1 Purchase of components
2 Manufacture of computers
3 Sale of computers.

These can be expanded as follows:

1.1	Cost of components
1.2	Delivery charges
1.3	Working capital
1.4	Factory storage space
2.1	Labour costs
2.2	Throughput
2.3	Rejects
2.4	Warehouse facilities
3.1	Asking price
3.2	Dealer network/delivery costs
3.3	Market saturation.

We should also, at this point, consider general aspects that will affect all areas of production:

4.1	Services (gas, electricity)
4.2	Rent/rates
4.3	Breakages
4.4	Crime.

You could now produce a fair simulation with just this information. We'll look at section one in some detail, so you can see how the ideas develop.

The factory storage space will limit how many items you can hold in stock. This, along with your working capital will limit your purchasing. At the same time, a reasonable simulation should allow for lower price breaks on bulk orders, and lower delivery charges, even free delivery over a certain order size. All of this can be done with quite simple mathematics.

Scaling and balance

It pays to put everything in terms of anonymous units, rather than real figures. It's the ratios that are important, not the actual figures. These can be scaled later to give meaningful results to the game player as well as a balanced simulation. In our example, we can assume, for example, that our main stores has a storage volume of 500 units, and storage units required for the parts needed for one computer are as follows.

Units	Item
1	Plain PCB
3	PCB components
2	Keyboard
6	Computer case.

This gives a total requirement of 12 units, and our factory can store the materials for almost 42 computers. However, you also need storage for completed machines. These would logically require slightly more storage space than the empty cases, say seven units. So your player will have to balance the two.

The component cost can be looked at in exactly the same way. You can start with a working capital of 400 units, and cost the parts so:

Units	Item
2	Plain PCB
20	PCB components
5	Keyboard
1	Computer case.

Again, not all the working capital can be used, as you also have to pay wages and other costs. However you should allow your players to make this mistake. Let them find out the hard way exactly what happens when their workers don't get paid!

Component costs, wages and final unit price all have to be kept in proportion. You can usually assume that component costs only come to about a tenth of the asking price of the completed computer. As a rough guide, an average week's wages should be set at about half the asking price of one machine. But this is all information that you can readily find out by asking the right questions.

You will find all sorts of balances work out quite naturally as you develop your game. If, for example, your player allows too little storage for completed machines, the labour force will have to stop work until some computers have been sold off, but will still demand the same wages. All you will need to do is tweak the figures so that you don't get runaway situations.

External influences

The situation is slightly more complicated regarding things like market saturation. Here you have to relate the actual number of computers sold to the apparent reluctance for people to buy. For simplicity, we'll consider that all factors, like inflation, recession, and total competitive computer manufacturing are lumped together as a single negative factor.

A separate factor is the total number of computers you've already sold. As people buy your machine, they won't be likely to want another, unless it fails. Eventually your selling capability could stagnate completely.

Putting it simply you have the very approximate formula:

computers sold =	computers available/(machines sold/time)*asking price*saturation

Time and saturation are pseudo constants you should fiddle to get a reasonable balance. Time partially represents the ageing of Bodgit's cheap computers. With both this and asking price, I've simplified the situation. A very long ageing time will give poorer sales, but in reality too short a time would have the customers grumbling. Similarly, too low an asking price would look rather suspicious. Also, you obviously can't sell 0.132 of a computer, so you take only the integer value.

8.4.2 Community simulations

Of increasing popularity now, is a whole community simulation. Such a simulation has enormous scope for the programmer and player alike. Below is a relatively brief list of the sort of factors you can integrate into such a simulation. The secondary factors listed are just a taste of the sort of relationships you will need to follow up. In fact, almost everything will inter-relate, so the feel of your simulation for your players, will be a direct reflection on how thoroughly you understand your community.

Population	Birth rate, death rate
Housing	Building, civil engineering
Employment	Manufactured goods, agriculture, housing
Crime	Laws, poverty, population
Services	Politics, infrastructure
Disasters	Man made, natural.

Obviously these whole communities are highly complex living organisms in their own right, and at best, yours will be only a very limited simulation. A key point to remember when planning such a game is that trends are usually more significant than isolated incidents. If you can, you should pursue the following lines of enquiry for more information about how groups of people behave:

History
Politics
Local government
Sociology
Market research;
Statistics.

8.4.3 Graphical simulations

Unfortunately, many graphical simulations require a considerable amount of drawing, rather than sprite plotting, and in general drawing will be much slower than sprite plotting. This is particularly relevant with flight simulators, where you are drawing in real time, as the plane flies its course. However it is often possible to work out a compromise. If you look at Figure 8.2, an admittedly crude drawing, you will see that only the central shaded area has to be drawn, using the three dimensional techniques described earlier.

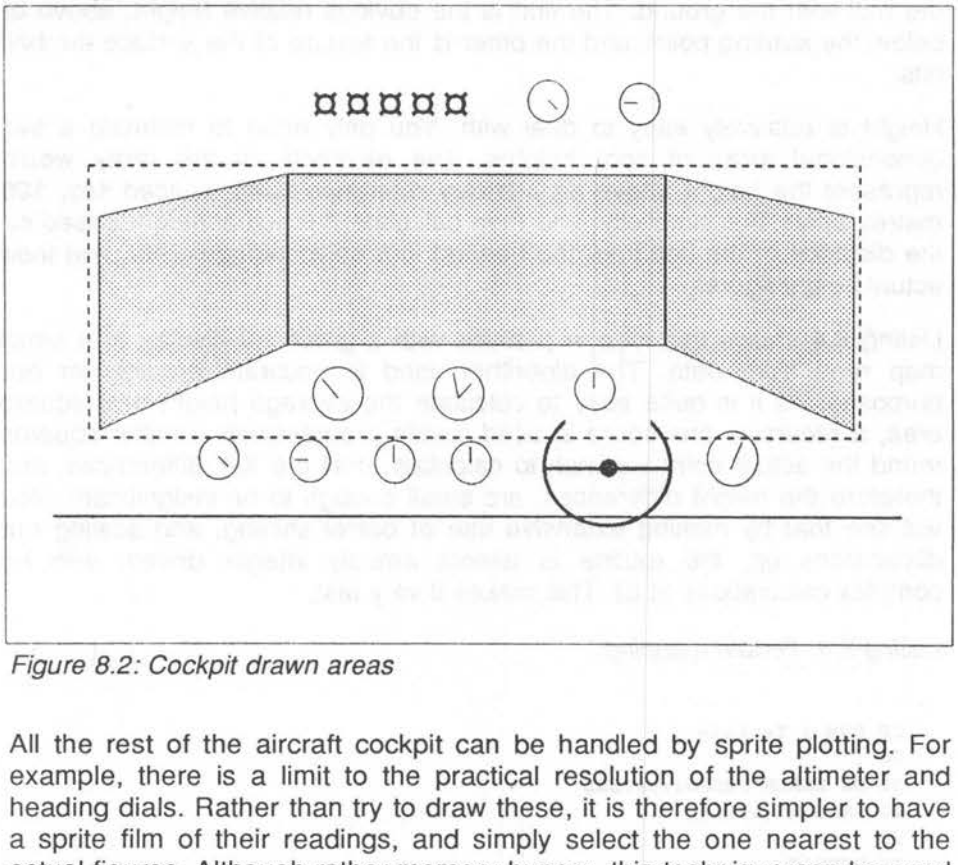
Figure 8.2: Cockpit drawn areas

All the rest of the aircraft cockpit can be handled by sprite plotting. For example, there is a limit to the practical resolution of the altimeter and heading dials. Rather than try to draw these, it is therefore simpler to have a sprite film of their readings, and simply select the one nearest to the actual figures. Although rather memory hungry, this technique can be used for numeric as well as metered displays, to considerably speed up response time.

Initially you would define a graphic viewport where the dotted rectangle is and perform your drawing in this area. Then you mask out the unwanted parts with sprites of the dashboard, window framing and overhead area. These sprites would also contain all the fine detail of switches, dials and lights that are not in fact active. Finally you can plot in the small dynamic detail.

8.4.4 Terrain mapping

All the discussion so far has assumed flat earth type scenarios. With many simulations this is far from reality. The game where this is of greatest significance is probably golf. There are two main variables with impact of the ball with the ground. The first is the obvious relative height, above or below the starting point, and the other is the texture of the surface the ball hits.

Height is relatively easy to deal with. You only need to maintain a two dimensional array of spot heights. The elements of the array would represent the height above an arbitrary reference point, spaced say, 100 metres apart. For simplicity, you then calculate the actual height based on the distance of the ball from the nearest four surrounding points, and their actual height figures.

Listing 8.4 shows this idea in practice with a graphical display of a small map read from data. The algorithm used is accurate enough for our purposes. As it is quite easy to calculate the average height of a square area, a recursive procedure is used create progressively smaller squares round the actual point we wish to calculate, until the X,Y differences, and therefore the height differences, are small enough to be insignificant. You will see that by making extensive use of barrel shifting, and scaling our dimensions up, the routine is almost entirely integer driven, with no complex calculations at all. This makes it very fast.

Listing 8.4: Terrain mapping

```
 10 REM > Terrain
 20 :
 30 ON ERROR PROCerror:END
 40 PROCinitialise
 50 :
 60 REPEAT
 70   INPUT "Start X (1151 max):" xpos
 80   INPUT "Start Y (511 max):" ypos
 90   INPUT "End X (1151 max):" xend%
100   INPUT "End Y (511 max):" yend%
110   CLG
120   xstep=(xend%-xpos)/width%
130   ystep=(yend%-ypos)/width%
```

```
140   FOR I%=0 TO width%
150     pX%=xpos
160     pY%=ypos
170     xpos+=xstep
180     ypos+=ystep
190     PROCset
200     POINT I%<<2,FNhigh(S%)>>3
210   NEXT
220 UNTIL FALSE
230 END
240 :
250 DEF PROCerror
260 MODE 12
270 IF ERR<>17 PRINT REPORT$ " @ ";ERL
280 ENDPROC
290 :
300 DEF PROCinitialise
310 MODE 12
320 COLOUR 0,128,128,128
330 PRINT TAB(30,5) "Terrain Map" TAB(30,7) "Escape to stop"
340 VDU 28,0,9,23,0
350 VDU 24,0;0;1279;511;
360 width%=320
370 size%=128
380 acc%=4
390 RESTORE+10
400 READ X%,Y%
410 DIM map%(X%,Y%)
420 FOR J%=0 TO Y%
430   FOR I%=0 TO X%
440     READ height%
450     map%(I%,J%)=height%<10
460   NEXT
470 NEXT
480 ENDPROC
490 DATA 9,5
500 DATA 1,2,3,4,5,6,6,5,4,3
510 DATA 2,3,5,5,6,6,7,6,5,3
520 DATA 4,4,6,6,7,8,7,6,4,2
530 DATA 5,5,7,6,6,8,6,5,4,3
540 DATA 3,4,6,5,5,7,6,5,4,3
550 DATA 2,2,5,3,3,4,3,3,3,4
560 :
570 DEF PROCset
580 X%=pX% DIV size%
590 Y%=pY% DIV size%
600 Ah%=map%(X%,Y%)
610 Bh%=map%(X%+1,Y%)
620 Ch%=map%(X%+1,Y%+1)
630 Dh%=map%(X%,Y%+1)
640 S%=size%
650 pX%=pX% MOD size%
```

```
660 pY%=pY% MOD size%
670 ENDPROC
680 :
690 DEF FNhigh(S%)
700 IF S%<acc% THEN
710   height%=(Ah%+Bh%+Ch%+Dh%)>>3
720   ELSE
730   half%=S%>>1
740   :
750   IF pX%>half% THEN
760     pX%-=half%
770     Ah%=(Ah%+Bh%)>>1
780     Dh%=(Ch%+Dh%)>>1
790     ELSE
800     Bh%=(Ah%+Bh%)>>1
810     Ch%=(Ch%+Dh%)>>1
820   ENDIF
830   :
840   IF pY%>half% THEN
850     pY%-=half%
860     Ah%=(Ah%+Dh%)>>1
870     Bh%=(Bh%+Ch%)>>1
880     ELSE
890     Dh%=(Ah%+Dh%)>>1
900     Ch%=(Bh%+Ch%)>>1
910   ENDIF
920   :
930   height%=FNhigh(half%)
940 ENDIF
950 =height%
```

Mapping surface texture is probably best done slightly differently. You can usually assume that the fairway is of reasonably consistent bounce and roll performance, with just some minor random deviations. The rough and bunkers will have virtually zero bounce, and water obstructions will lose the ball altogether. Therefore you need to plan out a finer set of coordinates for these obstructions. These are best stored as a list rather than a map, and the ball's position compared with the nominal centre of such obstructions. This also lends itself to defining specific, above ground obstructions, such as trees.

It should be possible to integrate these maps, with the game drawing routine so that you have consistent views and behaviour, regardless of the location of the ball.

8.5 Status Saving and Reloading

Most role play games are designed for many hours of play. Where arcade style games can just about survive without game saving facilities, it is quite unreasonable to expect a role-player to start from scratch each time. To facilitate saving and re-loading you should split your game data into distinct static and dynamic parts. Static data will be the overall map or plan of the game, including any text, sprites and the like. Dynamic data is everything that can possibly be modified by the game as it progresses.

It is remarkably easy to leave out dynamic data from a saving routine. The simplest mistake is where most of the data is in arrays, but just a few items are ordinary integer variables. What you have to look out for is saving all the arrays but forgetting, say, the number of monsters killed.

Unless your game is vast and takes up all available disk space, I recommend that your program saves dynamic data in a special directory within the game application. There is then no need for the player to hunt out separate disks, remembering which are needed. Your saving and loading system becomes simpler too, as you only need to scan the relevant directory and present a list of the player positions available.

As usual, there is an exception to this. If your game is implemented as a fully multi-tasking application, it makes sense to use a distinct file type, so that a player double clicking on this will load the application and game together in true desktop fashion.

9 Strategy

This chapter is a bit of a hotch-potch really. There is no easy classification in this group, as it covers everything from Ludo to Crib. Essentially, we are talking about the more static games, sometimes known as parlour games. The ability to think many moves ahead is far more important than speed, and this is reflected in a rather different way of using the computer. All the real work is now being done when nothing is happening to the screen. A side effect of this is that such games are likely to be far more amenable to multi-tasking. This is an extreme contrast to arcade games where everything has to happen very rapidly in parallel. Again, the fact that the player will be staring at the display for long periods, means you will have to devote a great deal of time to producing a polished, but uncluttered layout.

9.1 Algorithms and Rules

Before you can make any progress with this class of game, you need to develop a clear understanding of the rules, and turn these into a precise methodology of implementing them – the algorithm. In other words. attempting to build up this kind of game piecemeal, is a fairly certain recipe for disaster. I have to confess that this is a lesson that I learnt the hard way some years ago, on my first attempt at Othello.

9.1.1 Regional variations

It seems that the older, and better known a game is, the more variations you are likely to come across. You must therefore try to find out as much about the regional differences as possible and provide the facility for your players to be able to select which variant they wish to play. Ideally you should give them the option of saving their preference on the disk so that they don't need to set the options each time they play.

Two games that come to mind where this is particularly relevant are Patience and Draughts. There are so many variations in Patience that, unless you're going to write a particularly unusual form, you may decide to avoid altogether! However, in the best known seven column form there are a number of major differences that you can easily accommodate. A few of these are:

❑ Permit any picture card to be placed in any empty column

❑ Shuffle remainder stack each time it has been worked through

❑ Allow only one pass through the stack

❑ Allow ordered cards to be split and moved from one line to another.

The problem with Draughts is the huffing rule. You will find this is quite hotly contested as to whether it should be applied or not, so if you're programming Draughts do make sure you provide huffing as an option. The fragment below shows how easy this is to implement.

```
IF cantake% AND huffrule% PROCtest ELSE PROCmovepiece
```

There is a very obscure variant of this that changes the strategy quite significantly. One player sets up a double huff situation, so that the opponent has two choices where a piece has to be taken. Whichever choice is taken, a huff is called on the other option, resulting in a guaranteed loss of one piece each.

9.1.2 Symmetrical patterns

When working out algorithms for strategy games it is very easy to forget that you frequently have a symmetrical layout, and that many possible moves are simply mirror images of each other, particularly where opening strategies are concerned. A fairly obvious example of this is Naughts and Crosses, or Tic-Tac-Toe if you prefer. If you look at Figure 9.1 you will see that three squares are shaded. These are the only starting locations your program needs to consider. All the other positions are simply rotations or mirror reflections of these positions. As a result you can examine the opening move completely with only three sets of calculations instead of nine.

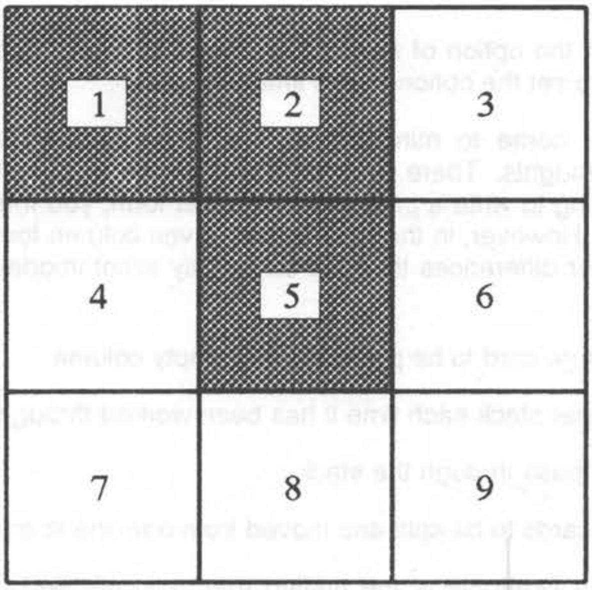

Figure 9.1: The Tic-Tac-Toe start positions

If the computer makes the first move, it is easy to arrange for apparent changes in opening strategy by randomly selecting the other images of these starting locations. The fragment below shows how this can be done using the box numbers of Figure 9.1, although I wouldn't normally recommend such an inefficient way of doing it.

```
IF select%=5 THEN
   {do nothing}
ELSE
   rotate% = RND(4)*2
   IF select%=1 THEN rotate%-=1
   IF rotate%=5 THEN select%=9 ELSE select%=rotate%
END IF
```

The position becomes only slightly more complex if the computer makes the second move. In this case, after rotating the player's move to the form we recognise, you take the diagonal through 1,5,9. The computer's response can then be reflected either side of this axis. Here you will find there are five possible positions to be examined rather than eight.

With a game as simple as this, with so few possible combinations, it is a practical proposition to specify directly a fixed set of rules defining how the computer should respond. However, this is not normally the case.

9.2 Recursive Computer Moves

Mention recursion to many people and they go weak at the knees. This is basically because the concept is rather alien to our normal straight-line style of thinking. Instead of a linear progression of tests and action, with recursion you perform a few tests, hold the results and perform another similar set of tests, and repeat this activity, maybe many times, piling tests on top of groups of tests, until you eventually have a possible best action. This kind of mental juggling, keeping so many balls in the air, is very difficult for most people. However the rules for recursive code are usually quite simple, and it can be very satisfying to watch the computer wind up a recursive problem then adroitly unwind with a solution.

One point to watch very carefully with recursion is that you always have an exit point. You will see this in the pseudo code example below, for a four in a line type game. The recursion entry is only made after all possible terminating conditions have been tested. The second test is remarkably easy to forget.

```
Recursion entry;
If this move completes a line, store move details and exit;
If no more moves possible, set flag and exit;
Swap opponent with computer and call recursion entry.
```

9.2.2 Minimaxing

The basic concept behind recursive algorithms is quite straightforward. The computer finds the first valid position for its next piece. It tests to see if it is a winning move, and if not it plays devil's advocate to see if that move could give the opponent the winning move. The computer then checks to see if any of the opponent's possible responses could provide a winning computer move at the next level. If the move looks dangerous the computer will try the next valid position, until either a winning position has been found, or all the positions have been investigated. In the latter case, depending on the precise algorithm, the computer will either choose the move that is likely to produce a win for itself in the least number of moves, or take the most number of moves to allow a win for the opponent. This latter option could produce a computer win if the human player makes a mistake.

This searching out best or computer maximums in parallel with worst or opponent minimums, often called minimaxing, can make huge demands on processor time. In almost all games, any move made will give quite a number of choices to the opponent, and if all of these are investigated, you will see that the number of tests made rapidly expands in a tree like structure until it becomes impractical to continue.

9.2.3 Limiting recursion

What is needed, therefore, are methods of limiting the number of tests that are made recursively. The first, and most obvious way of doing this is to control the number of moves the computer looks ahead. This simply involves the use of a counter to control the recursion depth. Putting this under player control can be used to give coarse difficulty settings.

A similar, but subtly different solution is to limit the time the computer can take following any one recursive path. Where the former method gives a fixed cut-off point regardless, using time as the limiting factor enables a promising line of moves to be investigated more thoroughly.

9.2.3 Pruning

A common technique for time saving is to do some tree pruning. If you are looking for maximums, say, and branch A has leaves that produce the values 2, 7, 8 and 6, but the first leaf of branch B produces 9, you don't need to bother to check any other leaf of branch B as you know that at least one route in the B branch is better than any in the A branch. You will still need to check branch C however, unless branch B produced the maximum value possible. If branch C produces a higher value than B then you will have to go back and check the remaining parts of B to see if it will again yield a better value. This is shown in Figure 9.2.

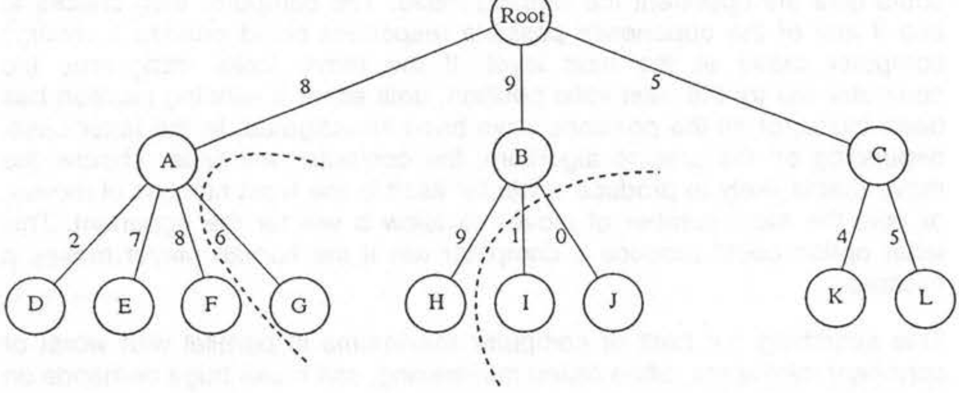

Figure 9.2: Pruning moves

Note that the pruning attempt of G is not valid. It is easy to become confused here, but as a general rule you can't prune any of the first branch, as there is no earlier one for it to better.

Had C given a result of 11 from its first leaf, you would go back to B and look at the I leaf. If this was greater, say 15, you wouldn't bother with J, but would go back to looking at the L leaf. In the worst case, with L producing 15 or more, you would then go back to J and examine this leaf. All this swapping backwards and forwards may seem wasteful, but I've given the worst case situation. Usually, there are considerable savings to be made. Also, keep in mind that although the computer looks ahead several moves, it is only actually going to make the move at the root of the tree.

9.2.4 Fuzzy thinking

Many games have more than just good or bad moves. With these, if you keep a count of the quality of the moves, you can develop a strategy of progressively ignoring the more unlikely situations the deeper you go recursively, only stopping at the point where no further conditions are tested. This will speed up calculations to some degree, and give a sort of fuzzy edge to the computer's thinking. This may not make a particularly smart machine game, but it will make it harder for an opponent to predict. This in itself can make a game far more attractive, and seeming human. Many people have the idea in their minds that the computer can't possible make mistakes. They will relax considerably if they see it make a move that they know can be bettered.

9.3 Weighting Schemes

Using recursion is not the only way you can produce intelligent machine gameplay, and indeed, not the first that most people think of. Instead of just totalling the number of pieces taken, and the potential gains from the later moves, you can maintain a board array with the weighting values for the various positions. Where you have more than one valid position, you should then perform your recursive scan to find the most advantageous moves. After this you apply the weightings so that they hold the greatest number of high value positions, possibly at the cost of actual pieces at this level. This can become very complicated so you need to strike a balance between improved machine intelligence and complexity. Mind you, with the simpler games it may be possible to produce a weighting scheme that is good enough to be able to do away with recursion altogether.

9.3.1 Key moves

A good example of the way you can improve machine intelligence can be shown with the game Reversi, or Othello as it is sometimes known. If you only use recursion to pick the best moves you may need to wait a very long time, going to the depth necessary to beat an experienced player. However, when you examine the game, knowing the rules, you will see that there are a number of key positions that can greatly enhance your likelihood of winning.

The most obvious ones are the four corner positions. Pieces placed here can't be taken so, unless the opponent uses some pretty fancy strategy, whole lines can be controlled from the corners. This is particularly true if you command two corners on the same side of the board.

In Figure 9.3 there is one possible set of weightings for the Othello board. The numbers are only intended as a guide as to the importance of the positions, not some absolute value. You will notice that I've given the squares adjacent to the outside squares the lowest values. Generally, if you put a piece in these squares you are letting your opponent get to the outer edge, and possibly the commanding position of a corner. Finally, you can make your weightings dynamic and adjust them as a game progresses to reflect the changing status of certain moves or positions.

7	2	5	4	4	5	2	7
2	1	3	3	3	3	1	2
5	3	6	5	5	6	3	5
4	3	5	6	6	5	3	4
4	3	5	6	6	5	3	4
5	3	6	5	5	6	3	5
2	1	3	3	3	3	1	2
7	2	5	4	4	5	2	7

Figure 9.3: Othello weightings

9.3.2 Randomising

Finally, keep your opponent guessing. Where there is little significant difference between two or more moves, don't make the mistake of always choosing the first. Your player will eventually be able to follow the pattern that the machine plays and therefore beat it every time. Instead, make a random selection, even allowing slightly less advantageous moves to be made in the earlier stages.

9.4 Introducing Othello

Many games can be reduced to a few simple rules. Following the Othello example in more detail, we first need to establish the starting conditions. It's easy to forget that the initial moves may be quite different from any others. It is a common mistake to try to patch the main game loop to include starting conditions. This is error prone, and often slows the program down. It's usually far simpler to have a separate routine.

❑ The game is started with the first four pieces already placed in the central four squares with the colours lined up on the diagonals.

Now you can specify the rules for valid moves. This should always be separate from the main move calculations for two reasons: in the first place, a quick scan for validity saves time if the move is invalid. Secondly, the same checking routine can be used for both computer and player moves.

❑ Every piece must be placed within the 8 * 8 grid

❑ Every piece must be placed adjacent to at least one opponent's piece

❑ There must be a players piece beyond and adjacent to the opponent's piece, or line of pieces, in at least one direction

❑ If a piece can't be placed the move is forfeit

❑ If a piece is wrongly placed the move is forfeit (optional).

Once a valid move has been found, the following rules can then be applied to develop the game strategy:

❑ All pieces have the same value

❑ Board positions have a weighting value

❑ A move that increases the player's piece count is a potential good move

❑ A move that gives the opponent a chance to increase his count may be a bad move

❑ A move that reduces the opponent's piece count to zero is a winning move

❑ A move that allows no more moves to be made closes the game.

The first rule is easy to overlook. In a game like Chess, for example, the pieces would have very different values in a weighting scheme, but board positions would become less important.

The last two rules inter-relate, in that the very fact of capturing the opponent's last piece automatically makes any further move invalid for either player. It is important to be clear about winning moves. The routine you use must always be able to spot these, and then ignore all other moves. At the same time you mustn't confuse a winning move, with one that leads to a win.

In this game, particularly in the early stages, a move that gives the player more pieces is not necessarily a good move. Similarly a move that allows the opponent to take pieces may not be a bad move. In a game where pieces could only be captured once, this would be true, but in Othello an individual piece can swap colours many times. The unfortunate result of this is that tree pruning is most unwise. Recursion limiting should be done with a combination of recursion depth and key move testing, using the board weighting to evaluate key moves.

9.5 Card Games

Unfortunately, most multi-player card games are impractical, as there is no way of preventing your opponent from seeing your cards. It would be possible to develop a game using two machines, linked via their serial ports, but there would be very little demand. Not many people can justify two Archimedes! However, games played against the computer are still quite practical, as most people will readily accept the idea of a computer player not being able to see your cards, while the computer referee sees them all.

9.5.1 Displaying a hand

One of the main difficulties that arises when programming card games, is the matter of displaying a large number of playing cards in such a way that they are clearly visible, and yet all fit on the limited screen area in a reasonable pattern. While you can gain space by overlapping, in exactly the same way as a player normally does with a hand, this can still leave you short of space.

If you are programming your game to work within the desktop you can circumvent this to some extent by using a scrollable window, and only display about half the cards. If you do this, you must give your player the option of re-ordering the cards, exactly as he or she would with a real hand. Actually, I recommend using the desktop, and then defining your cards as sprite icons. This makes selection and dragging remarkably easy, as the WIMP does most of the work for you.

If you're not working within the desktop, you will either have to reduce the size of the playing cards, which will reduce the detail and attractiveness, or display the cards in blocks. These can then be flicked through with say, Select and Adjust mouse clicks. I would again recommend using simple sprites rather than drawn cards. If you want to add a bit of style, you can have an animated film of the cards being bent as they are placed, synchronised with a suitable sampled thwack.

9.5.2 Patience layout

Figure 9.4 is a specimen layout for seven card patience. It looks a bit sparse as it is, but would in reality quickly fill out as the game progresses. Also, for simplicity, I haven't bothered with the detail of the card faces. The background should ideally be a dithered green, to give a card table appearance. The plinths for the stack, and the piles, could either be tinted for a metallic effect, or better still, given a wood grain appearance.

You will see that card edges are shown to give an indication of the number of cards in the piles. This is particularly relevant for the laying out columns, as there is unlikely to be enough room to show the reversed cards spaced down as they normally are, along with the visible cards.

Assuming a mouse driven game, everything that is needed is visible. Cards would be selected by dragging. The game can be re-started by clicking on the New icon, and the program abandoned, returning to the desktop, by clicking on the Quit icon. If you wanted to, you could easily add another two icons for load and save game options.

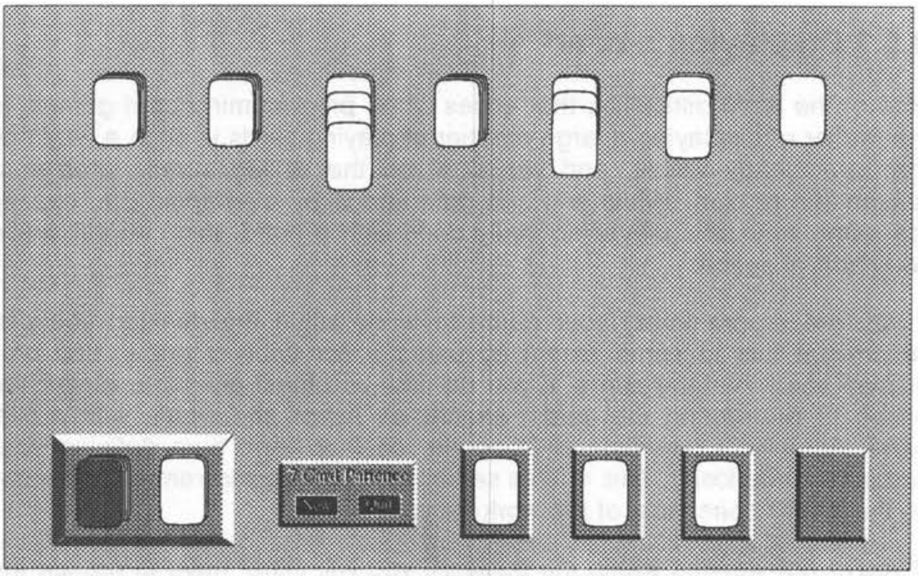

Figure 9.4: Patience layout

9.5.3 Implementing Patience

It is useful to have a brief look at the game itself from data structure point of view. You need to know how many cards are in each of the columns, whether they are visible or not, and what the cards are. In the stack, you need to know how many cards are in the unused section and how many in the used section, again with their values. With the piles you only need to know the number of cards in each pile. Logically you know the top card of each pile as they are in numerical order.

The stack can easily be held as a pair of arrays. As 28 cards are already placed in the columns, the array sizes only need to be 24. The zero element can be used to indicate the number of cards in the stack, or more specifically, the pointer to the next card to be handled, while all the other elements are actual card values. As the stack is handled, three card values are moved from the unused stack to the used stack, and the pointers updated. As cards are lifted from the used stack, its pointer is simply decremented. When the entire stack has been seen, a simple swap of the contents, remembering to reverse the order, is all that's needed.

Columns are also best implemented as arrays. This time the array size has to be 17. This is to allow for the, admittedly unlikely, possibility that the first column, while still with seven cards, will have the whole of an ordered line on top of it. This time you need to maintain two pointers per array. The zero element can again be the pointer to the last card. However, you also need a pointer to the first visible card, or last hidden card, whichever is most convenient.

Finally, the ordered piles can be simple integers with a card count. When the sum of these integers is 52, the game has been successfully completed.

9.5.4 Shuffling

One of the commonest mistakes made with card games is with regard to shuffling the pack. I've seen some of the most horrendous and convoluted programs that attempt to find random numbers between 1 and 52, then check that they haven't already been selected, place them in an array and increment a counter. If you stop and think for a moment you will realise that it is far simpler to shuffle your array in exactly the same manner as real cards.

First fill it linearly with the numbers 1 to 52 using a FOR NEXT loop. This takes care of the problem of ensuring that there are no repeats. All you need to do now is randomly select array positions and swap their contents. To get a good shuffle you need to perform about twice as many swaps as there are items in the array, 104 in this case. You don't need to prevent your random selections repeating the same swaps. As with real shuffling moving a card out of a position and then back again is quite valid.

You will see that almost an identical approach can be used for similar counted choices, such as shaking the bag for a Bingo session, or mixing up dominoes before laying them out.

9.6 Tile-based Games

This sub class in itself, covers quite a wide range, and includes Dominoes, Mah Jong and Scrabble. The special problems here are not so much that of display, but orientation and matching. Having said that, Mah Jong tiles can take up considerable space.

Taking Dominoes as an example, you need to establish, not only which dominoe is being handled, but also its orientation, as well as the orientation of those already laid. This is of particular importance with the fives and threes game, where both you and the computer will want to maximise your fives and threes count. In this case you not only need to know that a match has been made, but also the total spot count. This has to be a multiple of five or three to score.

Probably the simplest way of handling this sort of problem is by using a two dimensional array for the dominoe stacks of both players. For the dominoes already laid, you only need to keep a record of the two end points. For convenience these could be marked with two variables that would be set to spot count the ends of the first dominoe placed. From then on, they would be set to the free end of each dominoe placed at that position. Below is a list of the sort of tests that you would need to make for this.

❑ Does dominoe left end match stack left end?

❑ Does dominoe right end match stack left end?

❑ Does dominoe right end match stack right end?

❑ Does dominoe right end match stack right end?

❑ Does left placing give 5 or 3?

❑ Does right placing give 5 or 3?

❑ Does left placing give higher 5 or 3 than right placing?

❑ Is this the highest scoring dominoe?

These tests would be in addition to the normal strategy assessment for obstructing the opponent, and maximising the options for placing all dominoes.

9.7 Word Games

Although many word games fit in the sub-class of tile games, by virtue of the orientation and matching necessary for individual letters, and indeed, whole words, the core of this type of game is the dictionary. There are public domain dictionary utilities available, but you can generate your own without too much difficulty.

Initially you can use a system similar to that suggested for adventure games, where you have a simple array holding a list of words. You then scan this for a match with the word being tested. However, this becomes impractical with a dictionary of any size. The solution, in part, is to use a binary search.

9.7.1 Binary searching

For this, it is essential that the word list is in alphabetical order. As you don't want to waste time with sorting algorithms, the most practical solution is to ensure that the dictionary is alphabetically ordered in the first place.

For the actual search, you start by dividing the word list in two, then compare your word with the one in the middle of the list. If it matches, you flag it accordingly, otherwise, if it is alphabetically lower you repeat the operation with the bottom half of the list. If it is higher, you work on the top half. This is repeated until either the word is matched, or the list can't be split any further.

9.7.2 Text compression

Another way you can slightly improve the response time of your dictionary is by using text compression, making the strings that have to be compared shorter, and also reducing you memory requirements. Incidentally, this is again applicable to adventure games.

The byte-by-byte representation of characters in strings is very wasteful, out of a possible 256 values, you only use 26. These can be represented in only five bits instead of eight. This fact is used in Listing 9.1 which times the difference between compressed and non-compressed binary searching.

Listing 9.1: Dictionary

```
 10 REM > Dictionary
 20 :
 30 PROCinitialise
 40 :
 50 REM   test routines
 60 :
 70 PRINT""Sorting normal  Time = ";
 80 TIME=0
 90 PROCsort(word$())
100 PRINT;TIME
110 :
120 PRINT "Sorting packed  Time = ";
130 TIME=0
140 PROCsort(pack$())
150 PRINT;TIME
```

```
160 :
170 a$=word$(103)
180 p$=pack$(103)
190 PRINT""Performing searches ";N%+1 " times"
200 :
210 PRINT"'Searching normal  Time = ";
220 TIME=0
230 FOR I%=0 TO N%
240   f%=FNmatch(a$,word$())
250 NEXT
260 PRINT;TIME
270 :
280 PRINT "Searching packed  Time = ";
290 TIME=0
300 FOR I%=0 TO N%
310   f%=FNmatch(p$,pack$())
320 NEXT
330 PRINT;TIME
340 END
350 :
360 DEF PROCinitialise
370 wordnum%=999
380 DIM word$(wordnum%),pack$(wordnum%)
390 N%=499
400 a$=STRING$(16,".") : REM fixing string lengths
410 b$=a$              : REM speeds up swaps
420 MODE 12
430 PRINT""Generating ";wordnum%+1 " dummy words - Please wait";
440 FOR I%=0 TO wordnum%
450   word$(I%)=a$
460   word$(I%)=FNline
470   pack$(I%)=a$
480   pack$(I%)=FNpack(word$(I%))
490 NEXT
500 ENDPROC
510 :
520 DEF FNline
530 LOCAL I%,a$
540 FOR I%=0 TO 4+RND(5)
550   a$+=CHR$(64+RND(26))
560 NEXT
570 =a$
580 :
590 DEF FNpack(a$)
600 LOCAL s%,d%,b$
610 REPEAT
620   PROCget
630   d%=s%
640   PROCget
650   d%=d% OR s%<<5
660   b$+=CHR$ d%
670   d%=s%>>3
```

```
 680   PROCget
 690   d%=d% OR s%<<2
 700   PROCget
 710   d%=d% OR s%<<7
 720   b$+=CHR$ d%
 730   d%=s%>>1
 740   PROCget
 750   d%=d% OR s%<<4
 760   b$+=CHR$ d%
 770   d%=s%>>4
 780   PROCget
 790   d%=d% OR s%<<1
 800   PROCget
 810   d%=d% OR s%<<6
 820   b$+=CHR$ d%
 830   d%=s%>>2
 840   PROCget
 850   d%=d% OR s%<<3
 860   b$+=CHR$ d%
 870 UNTIL a$=""
 880 =b$
 890 :
 900 DEF PROCget
 910 s%=ASC a$-64
 920 a$=RIGHT$(a$,LENa$-1)
 930 ENDPROC
 940 :
 950 DEF PROCsort(word$())
 960 LOCAL a$,a%,b%,n%
 970 n%=DIM(word$(),1)
 980 PROCqsort(0,n%)
 990 ENDPROC
1000 :
1010 DEF PROCqsort(s%,e%)
1020 IF s%>=e% ENDPROC
1030 a$=word$((s%+e%)>>1)
1040 a%=s%-1
1050 b%=e%+1
1060 REPEAT
1070   REPEAT
1080     a%+=1
1090   UNTIL word$(a%)>=a$
1100   REPEAT
1110     b%-=1
1120   UNTIL word$(b%)<=a$
1130   IF a%<b% SWAP word$(a%),word$(b%)
1140 UNTIL a%>=b%
1150 PROCqsort(s%,a%-1)
1160 PROCqsort(b%+1,e%)
1170 ENDPROC
1180 :
1190 DEF FNmatch(a$,word$())
```

```
1200 LOCAL s%,e%,h%,f%
1210 e%=DIM(word$(),1)
1220 REPEAT
1230   PROCbin(s%,e%)
1240 UNTIL f%
1250 =f%
1260 :
1270 DEF PROCbin(s%,e%)
1280 IF e%>=s% THEN
1290   h%=(e%-s%)>>1
1300   IF a$>word$(s%+h%) THEN
1310     PROCbin(s%+h%+1,e%)
1320     ELSE
1330     IF a$<word$(s%+h%) THEN
1340       PROCbin(s%,e%-h%-1)
1350       ELSE
1360       f%=s%+h%+1
1370     ENDIF
1380   ENDIF
1390   ELSE
1400   f%=TRUE
1410 ENDIF
1420 ENDPROC
```

A list of random words is generated in the initialisation, of eight characters average length. At the same time a compressed copy of each word is produced by FNpack. As it stands, the routine assumes that all characters are upper case letters. The two lists are then sorted with a quicksort routine, and for interest this is timed. Finally the two lists are searched for a string known to be at position 103. This value is chosen to reduce the possibility of unnaturally fast binary divisions occurring.

You will see that the search is in fact extremely fast. It needs 500 iterations to get a meaningful timing. What is even more impressive, is that if you increase the word list to 10,000, the generating time is several minutes, and the sorting time slows considerably too, but the search time is hardly affected. Also the larger the number of words, the more reliable the time differences become. With 10,000 words, both sort and search times are about 10% faster using packed words, and the memory saving is around 25%

9.7.3 User dictionaries

One problem with word games is the fact that your dictionary will be incomplete. Therefore you should make some kind of allowance for additions to be made. The simplest solution is to maintain a text file of additional words. This is built up by the game itself in response to unmatched words and, the next time the game is played, is loaded into a separate array at the start. As your players are likely to generate a much smaller list than the main dictionary, you can probably get away with a simple linear search of this list. If any new words are added as the game progresses, a flag should be set, and when the game closes the player should be offered the choice of saving the new additions. To avoid the interruptions caused by continual requests for confirmation that a new word has been entered, your game could have a switchable learn mode. For normal play, this can be disabled so that the computer can respond quickly to wrong spellings.

9.8 Strategy in 3D

A few games, such as Naughts and Crosses, have been revamped with three dimensional implementation. This can make an otherwise dull game considerably more exciting. Theoretically, almost any two dimensional board game can be made three dimensional. However, personally, I'd hate to try to play 3D Chess.

In principle, all you need to do is add an extra dimension to your board array, and another counting loop in your move validation and computer. On the downside is the fact that all calculations will now take very much longer. So much so, that a computer-human game may not be practical, and you may have to satisfy yourself with human-human implementation.

Unfortunately, many programmers, shrink from a true 3D layout, and just place a group of boards alongside each other. This not only detracts from the game play, but also significantly changes the difficulty of the game. In some cases it makes it easier, in others its harder. Using the perspective ideas in Chapter 6, you can draw out a board relatively simply, and if you like, use scaled sprites for the pieces. To give the player more control, you can also use the 3D rotation formulae, so that he or she can view the game from any angle.

The easiest way to produce an unambiguous display, is to use different tints for the pieces on different levels. Logically you would use the darkest tint for the lowest board. Taking 3D Naughts and Crosses as an example, all four boards can be uniquely identified. If you have more than four layers, you will need to use combinations of colours and tints that show an obvious progression. This is quite easy in the 256 colour modes, if you refer back to my earlier suggestion of selecting colours bit-wise.

If you are using the mouse to pick up pieces, which is really just about essential, you should use inverse video or switch to flashing colours to highlight the piece that the mouse is looking at. As you move the mouse, the piece should follow it in jumps, always staying clearly placed within the playing grid.

10

ARM Code

10.1 Why Use It?

The obvious reason that tempts programmers to use ARM code in a game is, of course, that of speed chasing. However, there are other equally relevant reasons. It may be that you want to run background activities, such as music. Risc Os simply might not have a routine that you want, or only in a form that is impractical for your particular need. With practice, you can develop highly efficient, complex routines that are completely unimaginable on many processors, and yet seem to employ remarkably little code.

10.2 Fast Object Tables

In Chapter 7 I described the use of movement tables for sprite plotting. These are extremely easy to implement as byte arrays, putting them in a form readable within ARM code. This is particularly beneficial where, as in this case, you have one table providing pointers into another table. The sophisticated stack and indirect addressing features of the ARM processor really come into their own here. In principle, Listing 10.1 is only slightly changed from the original Listing 7.1, but you will see from the number of characters printed, it runs much faster.

Listing 10.1: Movement tables

```
   10 REM > ARMtable
   20 :
   30 ON ERROR PROCerror:END
   40 PROCinitialise
   50 PROCassemble
   60 IF INKEY 100
   70 CALL code%
   80 END
   90 :
  100 DEF PROCerror
  110 MODE 12
  120 IF ERR<>17 PRINT REPORT$ " @ ";ERL
  130 ENDPROC
  140 :
  150 DEF PROCinitialise
  160 MODE 12
  170 MODE 9
  180 COLOUR 0,128,0,0
  190 PRINT TAB(11,10) "ARM code movement"
  200 PRINT TAB(9,13) "Press Escape to stop"
  210 VDU 5
  220 maxpoints%=200
  230 DIM table% maxpoints%*4
  240 numchars%=21
  250 DIM char% numchars%*4
  260 FOR I%=0 TO numchars%*4-4 STEP4
  270   char%!I%=-1
  280 NEXT
  290 char%!(numchars%*4)=0
  300 x%=0
  310 y%=512
  320 mark%=0
  330 PROCline(40)
  340 PROCcircle(128,1)
  350 PROCline(10)
  360 PROCcircle(128,-1)
  370 PROCline(55)
  380 ENDPROC
  390 :
  400 DEF PROCassemble
  410 DIM code% &100
  420 work=0
  430 charbase=3
  440 charcount=4
  450 tablebase=5
  460 tableindex=6
  470 mainpointer=7
  480 bank=8
  490 sp=13
```

```
 500 link=14
 510 FOR I%=0 TO 2 STEP 2
 520   P%=code%
 530   [ OPT I%
 540   STMFD(sp)!,{link}
 550   ADR mainpointer,charadd
 560   LDMIA (mainpointer)!,{charbase,tablebase}
 570   MOV bank,#1
 580   .outerloop
 590   MOV R0,#113
 600   MOV R1,bank
 610   SWI "OS_Byte"               ; set display bank
 620   EOR bank,bank,#3            ; swap bank register
 630   MOV R0,#112
 640   MOV R1,bank
 650   SWI "OS_Byte"               ; set write bank
 660   SWI &10C
 670   MOV charcount,#numchars%
 680   .charloop
 690   LDR tableindex,[charbase,charcount,LSL#2]
 700   CMP tableindex,#0
 710   BLGE move
 720   SUBS charcount,charcount,#1
 730   BGE charloop
 740   MOV R0,#19
 750   SWI "OS_Byte"               ; wait
 760   SWI "OS_ReadEscapeState"
 770   BCC outerloop               ; exit if Escape pressed
 780   LDMFD(sp)!,{PC}
 790   ;
 800   .move
 810   LDR work,[tablebase,tableindex]
 820   STR work,[mainpointer,#4]   ; build up
 830   RSB R1,charcount,#(ASC"A"+numchars%)
 840   STRB R1,[mainpointer,#8]  ; VDU string
 850   ADD R0,mainpointer,#2
 860   MOV R1,#7
 870   SWI "OS_WriteN"             ; now print it
 880   CMP tableindex,#mark%-4     ; update char position
 890   MOVEQ work,#0
 900   LDRNE work,[charbase,charcount,LSL#2]
 910   ADDNE work,work,#4
 920   STR work,[charbase,charcount,LSL#2]
 930   CMP tableindex,#16          ; initiate next char?
 940   MOVNE PC,link
 950   CMP charcount,#0
 960   MOVLE PC,link
 970   MOV work,#0
 980   SUB charcount,charcount,#1
 990   STR work,[charbase,charcount,LSL#2]
1000   MOV PC,link
1010   .charadd
1020   EQUD char%
1030   EQUD table%
1040   EQUW 0     ; two dummy alignment bytes
1050   EQUB 25    ; plot
1060   EQUB 4     ; move
1070   EQUD 0     ; x co-ordinate
1080   EQUD 0     ; y co-ordinate
1090   EQUB 0     ; character
1100   ]
1110 NEXT
1120 ENDPROC
1130 :
1140 DEF PROCline(n%)
1150 FOR I%=0 TO n%
1160   x%+=12
1170   table%!mark%=x%
1180   mark%+=2
1190   table%!mark%=y%
1200   mark%+=2
1210 NEXT
1220 ENDPROC
1230 :
1240 DEF PROCcircle(rad%,dir%)
1250 start=-PI/2
1260 step=PI/20
1270 end=start+PI*2+step
1280 FOR I=start TO end STEP step
1290   table%!mark%=x%+COS(I*dir%)*rad%
1300   mark%+=2
1310   table%!mark%=y%+SIN(I*dir%)*rad%+rad%*dir%
1320   mark%+=2
1330 NEXT
1340 ENDPROC
```

In order to get the fastest possible character printing, the move and print commands are combined in a single VDU string. OS_WriteN uses the low-level VDU drivers, so is very much faster than individual calls to OS_WriteC would be.

As the Y coordinate in the VDU string follows the X coordinate only two bytes later, it makes sense to set up a single interleaved table for both. Values taken from this can then be handled as whole registers by the ARM routine. The fiddle with the address of the VDU string, ensures that we are working word aligned, and therefore most efficiently, for all instructions apart from the actual VDU call itself.

A point of interest is where I've used the load multiple registers instruction to set up the charbase and tablebase pointers, while leaving mainpointer at a sensible position for both filling and printing the VDU string. It is always worth trying to get as much out of your load and store instructions as you can, as these are the most time hungry operations.

10.3 Direct Screen Manipulation

In some earlier code fragments, there was a degree of screen handling but at a rather crude level. Once you start to use ARM code, the speed of execution is so much faster that detailed screen handling becomes practical. When selecting screen modes, with a view to using direct manipulation, you should bear in mind that it is far easier to handle the modes where one byte exactly represents one pixel. These are, of course, the 256 colour modes. In the lower resolution modes it may be two or four pixels and in Mode 0 it's eight. If you use these lower modes you will have to do a lot of bit manipulation to get full pixel control, so it may not be worth the effort. However, if your routines can usefully handle adjacent pairs of pixels, the 16 colour modes become as easy to handle as the 256 colour modes.

A delight to many programmers with experience of the older BBC Model B, is the fact that the Archimedes screen is handled as one continuous line of bytes, scanning from left to right, in a similar fashion to that of the VDU itself. To obtain access to this, the operating system allows you to read not only the current start of the screen in memory, but its size and the length of the lines in bytes, and therefore by simple division the number of lines in the screen.

The screen storing routine of Chapter 4 used this information to get all the information it needs to find correctly the start of the screen, allow for which screen bank was currently displayed, and then copy it line by line to the memory area reserved for it. However, you can do far more than this.

By now, you have probably seen a number of impressive demos that are passed around PD libraries. One of the commonest features of these are various starfields. Listing 10.2 is a sideways scrolling starfield generator, that is remarkably simple to implement. It uses a plot-move-rubout technique for a smooth scrolling effect without the need of banked screens. The twinkling effect that sometimes takes place, is due to the occasional plotting of one star directly on top of another, and actually enhances the effect.

Listing 10.2: Starfield generator

```
 10 REM > Starfield
 20 :
 30 PROCinitialise
 40 PROCassemble
 50 PROCfill
 60 GCOL%110101
 70 RECTANGLE FILL 0,0,1279,255
 80 RECTANGLE FILL 0,768,1279,1023
 90 PRINT TAB(10,27) "Press Escape to stop"
100 CALL code%
110 END
120 :
130 DEF PROCinitialise
140 MODE 13
150 OFF
160 PRINT TAB(12,7) "Please wait"
170 DIM block% 19
180 block%!0=148:          REM screen base address
190 block%!4=7:            REM screen size
200 block%!8=-1
210 SYS "OS_ReadVduVariables",block%,block%+12
220 screen%=block%!12
230 size%=block%!16>>1:  REM only want a half screen
240 screen%+=(size%>>1): REM offset by a quarter screen
250 stars%=2048
260 ENDPROC
270 :
280 DEF PROCassemble
290 DIM code% &50+stars%*3
300 fx=0
310 offset=1
320 speed=2
330 colour=3
340 screenbase=4
350 stack=5
360 end=6
370 new=7
380 blank=8
390 sp=13
400 link=14
410 FOR I%=0 TO 2 STEP 2
420   P%=code%
430   [ OPT I%
440   MOV     blank,#0
450   MOV     fx,#19
460   LDR     screenbase,start
470   LDR     end,stop
480   .scroll
490   ADR     stack,index
500   .pass
510   LDMIA  (stack)!,{offset,speed,colour}
520   SUBS   new,offset,speed
530   MOVMI  new,#size%<<1
540   STR    new,[stack,#-12]
550   STRB   blank,[screenbase,offset,LSR #1]
560   STRB   colour,[screenbase,new,LSR #1]
570   CMP    stack,end
580   BLT    pass
590   SWI    "OS_Byte"
600   SWI    "OS_ReadEscapeState"
610   BCC    scroll
620   MOV    PC,link
630   :
640   .start
650   EQUD   screen%
660   .stop
670   EQUD   index+stars%*3
680   .index
690   ]
700 NEXT
710 ENDPROC
720 :
730 DEF PROCfill
740 N%=index
750 FOR I%=1 TO stars%
760   R%=RND(4)
770   S%=RND(4)
780   IF R%=4 AND RND(3)>1 R%=1
790   IF R%=3 AND RND(2)=1 R%=0
800   !N%=RND(size%<1)
810   N%!4=R%+S%-1
820   CASE R% OF
830     WHEN 0:N%!8=3
840     WHEN 1:N%!8=3
850     WHEN 2:N%!8=S%+43
860     WHEN 3:N%!8=S%+207
870     WHEN 4:N%!8=S%+251
880   ENDCASE
890   N%+=12
900 NEXT
910 ENDPROC
```

The operation of this program relies on the fact that, as mentioned before, the screen can be regarded simply as a long line of bytes. Once you know where the start of this line is, and its length, you can put bytes directly to this area using the screen start, or base address, with an offset that you know is no greater than the size of the screen. To get movement sideways, all that is necessary is to add or subtract a small amount from the address you use for putting each byte. If you know how many bytes there are in a screen line, also available from the same OS call, adding and subtracting multiples of this will have the effect of moving points up and down.

There is quite a complex relationship between the on-screen colours and the byte values associated with them. The easiest way to find out which byte values to use, is simply to plot the colour you want to the top corner of the screen, then read it back with an indirection *peek*, having first found the start of the screen. This is shown in the following fragment:

```
DIM block% 11
!block%=148
block%!4=-1
SYS "OS_ReadVduVariables",block%,block%+8
screen%=block%!8
GCOL%100111 TINT &40: REM the colour you want to find
POINT 0,1023
PRINT ?screen%
```

10.4 ARM Sprites

If you've decided to program that ultimate fast action super invaders game that everyone will be amazed by, then I'm afraid you will have to abandon, not only Basic, but also the sprite handler itself. Acorn's handler is intended to be a general purpose tool, which will perform well over a wide range of screen modes, inside and outside the desktop. What you really need is a dedicated sprite controller, that has none of the overheads of the Risc Os sprite handler.

The sprites you want to define, will be intended to operate in only one mode, with a fixed palette, and in all probability at a fixed scale. Your sprites won't need names in string form, nor will they need to carry their size information, because you will access them from a lookup table of address offset screen positions, and possibly sizes as well. This table will have been created at the same time as the sprites, probably in some other generating program, and then saved as a data file along with your sprites.

10.4.1 Simple sprites

We will start by looking at the easiest to produce, that is, sprites that are intended to go only on a single colour background, where overlapping is unimportant, as these sprites don't require any complicated masking to be performed. Where you have sprites of different sizes, it is often worth using separate dedicated routines for each sprite size.

Another point worth consideration is whether you can fiddle the sprite size and screen positions so that you are always working word aligned. If you can do this, you can make dramatic gains in speed and efficiency. Listing 10.3 shows this in action.

Listing 10.3: ARMsprites

```
  10 REM > ARMsprite
  20 :
  30 ON ERROR PROCerror:END
  40 PROCinitialise
  50 PROCfill
  60 PROCassemble
  70 CALL code%
  80 END
  90 :
 100 DEF PROCerror
 110 MODE 12
 120 IF ERR<>17 PRINT REPORT$ " @ ";ERL
 130 ENDPROC
 140 :
 150 DEF PROCinitialise
 160 MODE 15
 170 MODE 13
 180 OFF
 190 PRINT TAB(12,7) "Please wait" TAB(11,11) "Escape to stop"
 200 COLOUR 130
 210 numsprites%=52
 220 spritewidth%=32
 230 spriteheight%=32
 240 colours%=8
 250 DIM block% 15,b2% 15
 260 block%!0=149:          REM screen base address
 270 block%!4=7:            REM screen size
 280 block%!8=6:            REM line length
 290 block%!12=-1
 300 SYS "OS_ReadVduVariables",block%,b2%
 310 screen%=b2%!0
 320 size%=b2%!4
 330 line%=b2%!8
 340 height%=size% DIV line%
 350 DIM table% numsprites%*20
 360 DIM spritearea% colours%*spritewidth%*spriteheight%
 370 ENDPROC
 380 :
 390 DEF PROCfill
 400 FOR K%=0 TO colours%-1
 410   FOR J%=0 TO spriteheight%-1
 420     FOR I%=0 TO spritewidth%-1 STEP 4
 430       V%=spritearea%+(K%*spriteheight%*spritewidth%+J%*spritewid
th%+I%)
 440       b%=K%*32+RND(16)+8
 450       IF I%>15 AND J%<15 OR I%<15 AND J%>15 !V%=b%+(b%<<8)+(b%<<
16)+(b%<<24) ELSE !V%=0
 460     NEXT
 470   NEXT
```

```
  480 NEXT
  490 FOR K%=0 TO numsprites%-1
  500   table%!(K%*20)=(RND(3)-2)*4
  510   table%!(K%*20+4)=(RND(3)-2)*line%
  520   table%!(K%*20+8)=RND((line%-spritewidth%)DIV 4-1)*4
  530   table%!(K%*20+12)=RND(height%-spriteheight%-3)*line%+line%
  540   table%!(K%*20+16)=(K% MOD colours%)*spriteheight%
*spritewidth%+spritearea%
  550 NEXT
  560 ENDPROC
  570 :
  580 DEF PROCassemble
  590 DIM code% &100
  600 work=0
  610 dX=2
  620 dY=3
  630 Xpos=4
  640 Ypos=5
  650 spritedata=6
  660 datablock=7
  670 dataline=8
  680 tablepointer=9
  690 screenbase=10
  700 tablebase=11
  710 bank=12
  720 sp=13
  730 link=14
  740 FOR I%=0 TO 2 STEP 2
  750   P%=code%
  760   [ OPT I%
  770   STMFD (sp)!,{link}
  780   MOV bank,#1
  790   ADR work,screenadd
  800   LDMIA (work)!,{screenbase,tablebase}
  810   .mainloop
  820   MOV R0,#19              ; usual screen bank
  830   SWI "OS_Byte"
  840   MOV R0,#113
  850   MOV R1,bank
  860   SWI "OS_Byte"
  870   EOR bank,bank,#3        ; display and write
  880   MOV R0,#112
  890   MOV R1,bank
  900   SWI "OS_Byte"
  910   SWI &10C                ; and clear screen
  920   CMP bank,#2
  930   ADDEQ screenbase,screenbase,#size%  ; ensure we write
  940   SUBNE screenbase,screenbase,#size%  ; to correct bank
  950   MOV tablepointer,#numsprites%-1     ; get count then x 20
  960   ADD tablepointer,tablepointer,tablepointer,LSL #2
  970   ADD tablepointer,tablebase,tablepointer,LSL #2
  980   .spriteloop
```

```
 990   BL plot
1000   SUB tablepointer,tablepointer,#20
1010   CMP tablepointer,tablebase
1020   BGE spriteloop
1030   SWI "OS_ReadEscapeState"
1040   BCC mainloop
1050   LDMFD (sp)!,{PC}
1060   ;
1070   .plot
1080   LDMIA (tablepointer),{dX,dY,Xpos,Ypos,spritedata}
1090   ;
1100   CMP Xpos,#0
1110   CMPNE Xpos,#line%-spritewidth%
1120   MVNEQ dX,dX
1130   ADDEQ dX,dX,#1          ; check X limits
1140   ;
1150   CMP Ypos,#line%
1160   ADD work,Ypos,#line%*(spriteheight%)
1170   CMPNE work,#size%
1180   MVNEQ dY,dY
1190   ADDEQ dY,dY,#1          ; check Y limits
1200   ;
1210   ADD Xpos,Xpos,dX        ; update positions
1220   ADD Ypos,Ypos,dY        ; and store
1230   STMIA (tablepointer),{dX,dY,Xpos,Ypos}
1240   ;
1250   ADD Xpos,Xpos,screenbase
1260   ADD Xpos,Xpos,Ypos
1270   ADD datablock,spritedata,#spritewidth%*spriteheight%
1280   ;
1290   .block
1300   ADD dataline,spritedata,#spritewidth%
1310   .line
1320   LDMIA (spritedata)!,{R0-R3}; slop it over in
1330   STMIA (Xpos)!,{R0-R3}      ; 4 register lumps
1340   CMP spritedata,dataline
1350   BLT line
1360   ADD Xpos,Xpos,#line%-spritewidth%
1370   CMP spritedata,datablock
1380   BLT block
1390   MOV PC,link
1400   .screenadd
1410   EQUD screen%
1420   EQUD table%
1430   ]
1440 NEXT
1450 ENDPROC
```

Because we are using our own dedicated sprites, in a known screen configuration, there is a lot of information that we don't need to store, that is essential in Acorn's generalised sprite handling routines.

Simple though it is, this sprite plotting routine is quite impressive, and certainly useful for creating backgrounds, or non colliding monsters. It can plot over 50 sprites of 32 x 32 pixels at 50 frames per second without any jitter. This is largely due to keeping everything word aligned, hence there are considerable movement restrictions. One result is that horizontal and vertical movement can only be in powers of two.

There is the added proviso that for smooth action, the minimum value for horizontal movement is four, so as it stands, vertical movement can be slower than horizontal movement, while remaining smooth. This is not necessarily a disadvantage. A close look at some commercial games will reveal the fact that other programmers have discovered this!

You should bear in mind, that the routine can only handle sprites that stay on the screen all the time, and that in this particular example only primitive edge detection has been used, so you can't reliably change the movement speeds at all.

The sprite plotter uses a single, word aligned table with five words for each sprite, as follows:

Byte offset	Function
+0	Horizontal movement
+4	Vertical movement
+8	Horizontal position
+12	Vertical position
+16	Address of sprite data

Further efficiency is gained by keeping the width of the sprites themselves to multiples of 16 bytes. This allows us to use multiple register loads and saves, using four registers at a time, in the main plotting loop.

You will see that all vertical position and movement calculations are kept in line multiples. This avoids any time wasting multiplication. Simple additive increments, calculated at the end of each plotting line, are all that's needed to keep the pointers in step.

In order to make the best use of the registers available, some of them are overwritten in the main plotting routine. This does no harm as they are finished with by then. It is a point you need to watch very carefully though. It is remarkably easy to forget, and try to re-use a register that was a pointer of some sort, only to get *Address exception*, or *Abort on data transfer* errors.

It is a false economy to try to make one plotting routine do several different jobs with extra, time wasting flags and tests needed to identify each type of sprite action. If, for example, you want to include some non-word aligned sprites, you should use a completely separate routine for these, bearing in mind that you won't be able to have anything like the same number for the same plotting speed.

These dedicated routines can be made so short that there probably is little overall difference in length of code, once you allow for the extra code that would have been needed for the identification of different sprite types. Added to this is the fact that the code length will probably be quite insignificant compared with the sprite data anyway.

10.4.2 Masking

Where you want your sprites to appear to sit on top of the background you need to identify which parts of your sprite are solid and which are transparent. This is masking, in exactly the same way as you might use a stencil to mask areas of lettering on a piece of paper.

There are two ways you can mask your sprites. One way, only really practical in 256 colour modes, is to scan your sprite byte by byte, comparing it with your mask. Wherever you have a 1 byte in the mask, you plot the corresponding byte from the true sprite to the screen. This is rather tedious and forces you to work only one byte at a time.

A better method seems slower at first, but is actually much faster, as you are back to using whole registers. In this case, it isn't the sprite that's masked, but the screen. Here you load a word from the screen at the address where the sprite is to be plotted. You AND this with a mask that has the negative of the sprite pattern. This means you have a shell of background screen which is then ORed with the corresponding word of the true sprite, and the composite word plotted back to the screen.

Figure 10.1 shows the two types of mask with their corresponding sprite patterns.

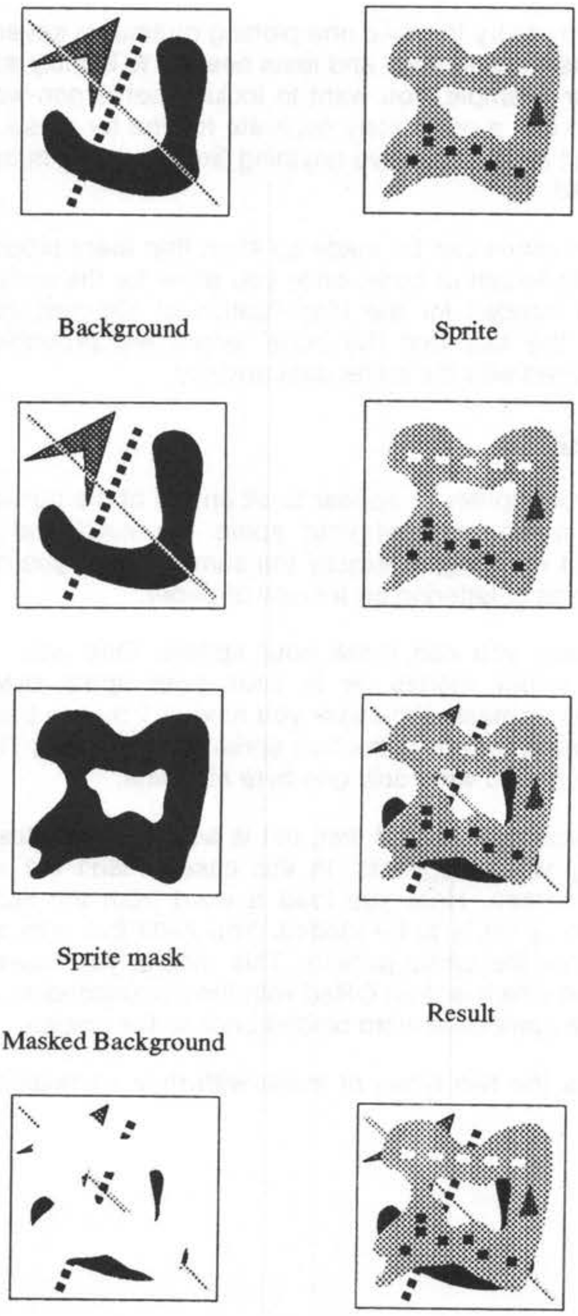

Figure 10.1: Masking sprites

10.4.3 ARM collisions

Although not obvious at first, using the background and merging it with your sprites, gets you half way towards a very accurate collision system. Indeed it can almost be called the ultimate pixel collision system.

When you load your background, instead of directly masking it for the sprites, you first, a word at a time, store a negatively masked copy of it in a spare register. This is then tested against a one word bit pattern. If it matches, the word you are about to plot on top of is a colliding object. This you flag, then carry on as before, merging the background with the sprite and plotting. When the sprite has been completely plotted you can then examine the flag, knowing the collision details.

Listing 10.4 shows this masked sprite technique. It is largely derived from our earlier sprite program but in this example only 20 sprites can be reliably plotted at the 50 frame rate. As a point of interest, if you don't have the collision tests, you still only get 25 sprites, so the main time overhead is the extra loading of data.

Colliding sprites are simply flagged as dead in our example. A dead sprite has its X coordinate set to &FF000000. This is an impossible coordinate value. You could, if you like, have a routine to restore a rational X coordinate and bring the sprite back to life. Also, instead of killing the sprite off, you may decide to insert a bounce routine.

As we are using all the registers very heavily, I had to pick a figure that is equally impossible as a value for work, or for bitpattern. The dead flag is actually stored outside the plotting routine. Using register values outside the subroutine that handles them is a bit naughty, so you have to be very careful to ensure that you know exactly what could be thrown back as you exit.

Listing 10.4: ARMmask

```
 10 REM > ARMmask
 20 :
 30 ON ERROR PROCerror:END
 40 PROCinitialise
 50 PROCfill
 60 PROCassemble
 70 CALL code%
 80 END
 90 :
100 DEF PROCerror
```

```
110 MODE 12
120 IF ERR<>17 PRINT REPORT$ " @ ";ERL
130 ENDPROC
140 :
150 DEF PROCinitialise
160 MODE 15
170 MODE 13
180 OFF
190 PRINT TAB(12,7) "Please wait" TAB(11,11) "Escape to stop"
200 COLOUR 130
210 numsprites%=20
220 spritewidth%=32
230 spriteheight%=32
240 colours%=8
250 DIM block% 15,b2% 15
260 block%!0=149:          REM screen base address
270 block%!4=7:            REM screen size
280 block%!8=6:            REM line length
290 block%!12=-1
300 SYS "OS_ReadVduVariables",block%,b2%
310 screen%=b2%!0
320 size%=b2%!4
330 line%=b2%!8
340 height%=size% DIV line%
350 DIM table% numsprites%*24
360 masks%=colours%*spritewidth%*spriteheight%
370 DIM spritearea% masks%*2
380 ENDPROC
390 :
400 DEF PROCfill
410 FOR K%=0 TO colours%-1
420   FOR J%=0 TO spriteheight%-1
430     FOR I%=0 TO spritewidth%-1 STEP 4
440       V%=spritearea%+(K%*spriteheight%*spritewidth%+J%*spritewid
th%+I%)
450       IF I%>15 AND J%<15 OR I%<15 AND J%>15 THEN
460         b%=K%*32+RND(16)+8
470         !V%=b%+(b%<<8)+(b%<<16)+(b%<<24)
480         V%!masks%=0
490         ELSE
500         !V%=0
510         V%!masks%=&FFFFFFFF
520       ENDIF
530     NEXT
540   NEXT
550 NEXT
560 FOR K%=0 TO numsprites%-1
570   table%!(K%*24)=(RND(3)-2)*4
580   table%!(K%*24+4)=(RND(3)-2)*line%
590   table%!(K%*24+8)=RND((line%-spritewidth%)DIV 4-1)*4
600   table%!(K%*24+12)=RND(height%-spriteheight%-3)*line%+line%
```

```
  610   table%!(K%*24+16)=(K% MOD colours%)*
spriteheight%*spritewidth%+spritearea%
  620   table%!(K%*24+20)=&EAEAEAEA  :REM collision bit pattern
  630 NEXT
  640 ENDPROC
  650 :
  660 DEF PROCassemble
  670 DIM code% &200
  680 work=0
  690 dX=2
  700 dY=3
  710 Xpos=4
  720 Ypos=5
  730 datablock=5
  740 mask=5
  750 dataline=6
  760 spritedata=7
  770 bitpattern=8
  780 tablepointer=9
  790 screenbase=10
  800 tablebase=11
  810 bank=12
  820 sp=13
  830 link=14
  840 maskdata=14
  850 FOR I%=0 TO 2 STEP 2
  860   P%=code%
  870   [ OPT I%
  880   STMFD (sp)!,{link}
  890   MOV bank,#1
  900   ADR work,screenadd
  910   LDMIA (work)!,{screenbase,tablebase}
  920   .mainloop
  930   MOV R0,#19                      ; usual screen bank
  940   SWI "OS_Byte"
  950   MOV R0,#113
  960   MOV R1,bank
  970   SWI "OS_Byte"
  980   EOR bank,bank,#3                ; display and write
  990   MOV R0,#112
 1000   MOV R1,bank
 1010   SWI "OS_Byte"
 1020   SWI &10C                        ; and clear screen
 1030   CMP bank,#2
 1040   ADDEQ screenbase,screenbase,#size%  ; ensure we write
 1050   SUBNE screenbase,screenbase,#size%  ; to correct bank
 1060   MOV tablepointer,#numsprites%-1     ; get count then x 24
 1070   ADD tablepointer,tablepointer,tablepointer,LSL #1
 1080   ADD tablepointer,tablebase,tablepointer,LSL #3
 1090   .spriteloop
 1100   BL plot
 1110   CMP work,#&FF0000         ; collision flag
```

```
1120   ;
1130   ;  bitpattern holds type of collision
1140   ;  tablepointer points to colliding sprite
1150   ;
1160   SWIEQ &107
1170   STREQ work,[tablepointer,#8]  ; kill sprite
1180   SUB tablepointer,tablepointer,#24
1190   CMP tablepointer,tablebase
1200   BGE spriteloop
1210   SWI "OS_ReadEscapeState"
1220   BCC mainloop
1230   LDMFD (sp)!,{PC}
1240   ;
1250   .plot
1260   LDMIA (tablepointer),{dX,dY,Xpos,Ypos,spritedata,bitpattern}
1270   TST Xpos,#&FF0000
1280   MOVNE PC,link                ; don't bother with dead ones
1290   ;
1300   CMP Xpos,#0                  ; check X limits
1310   CMPNE Xpos,#line%-spritewidth%
1320   MVNEQ dX,dX
1330   ADDEQ dX,dX,#1
1340   ;
1350   CMP Ypos,#line%              ; check Y limits
1360   ADD work,Ypos,#line%*(spriteheight%)
1370   CMPNE work,#size%
1380   MVNEQ dY,dY
1390   ADDEQ dY,dY,#1
1400   ;
1410   ADD Xpos,Xpos,dX             ; update positions
1420   ADD Ypos,Ypos,dY
1430   STMIA (tablepointer),{dX,dY,Xpos,Ypos}     ; store them
1440   ;
1450   ADD Xpos,Xpos,screenbase
1460   ADD Xpos,Xpos,Ypos
1470   ADD datablock,spritedata,#spritewidth%*spriteheight%
1480   ;
1490   STMFD (sp)!,{datablock,bitpattern,R9-R12,link}  ; grab more regs
1500   ADD maskdata,spritedata,#masks%        ; get mask address
1510   ;
1520   .block
1530   ADD dataline,spritedata,#spritewidth%
1540   .line
1550   LDMIA (maskdata)!,{R9-R12} ; load mask
1560   LDMIA (Xpos),{R0-R3}         ; get background but don't increment
1570   ;
1580   BIC mask,R0,R9               ; mask background to sprite shape
1590   TEQ mask,bitpattern          ; match bit pattern
1600   BICNE mask,R1,R10
1610   TEQNE mask,bitpattern
1620   BICNE mask,R2,R11
1630   TEQNE mask,bitpattern
```

```
1640   BICNE mask,R3,R12
1650   TEQNE mask,bitpattern
1660   ;
1670   MOVEQ bitpattern,#&FF0000  ; flag collision
1680   ;   (unlikely bit pattern)
1690   AND R9,R9,R0               ; mask out sprite shape
1700   AND R10,R10,R1
1710   AND R11,R11,R2
1720   AND R12,R12,R3
1730   ;
1740   LDMIA (spritedata)!,{R0-R3}; get sprite
1750   ;
1760   ORR R0,R0,R9               ; merge sprite with background
1770   ORR R1,R1,R10
1780   ORR R2,R2,R11
1790   ORR R3,R3,R12
1800   ;
1810   STMIA (Xpos)!,{R0-R3}      ; put to screen and increment
1820   CMP spritedata,dataline
1830   BLT line
1840   ADD Xpos,Xpos,#line%-spritewidth%
1850   LDR datablock,[sp]         ; need datablock again
1860   CMP spritedata,datablock
1870   BLT block
1880   MOV work,bitpattern
1890   LDMFD (sp)!,{datablock,bitpattern,R9-R12,PC} ; tidy up
1900   ;
1910   .screenadd
1920   EQUD screen%
1930   EQUD table%
1940   ]
1950 NEXT
1960 ENDPROC
```

There are two particularly significant points about this collision system. The first is that without any special effort on your part, it will only make the tests that are really necessary. This is because each sprite only looks at what is already on the screen. Therefore it is only going to see collisions with sprites that have been plotted. Any that are not being plotted for any reason don't figure at all in the tests.

The second point is that each sprite can be arranged to recognise a different bit pattern, so that colours can be used to group sprites into colliding and non-colliding sets. In our example I've set the bit pattern for all sprites to recognise only a pale blue. This has an interesting side effect. As the first few sprites to be plotted are reds and greens, these can never recognise blues that are all plotted later, nor can they ever be recognised as colliding objects.

Careful selection of plotting order and bit patterns can be used to produce highly detailed collision recognition, with very little extra code overhead. It is especially pleasing to be able to do this without making any further alteration to the plotting routine itself.

10.5 Reflected Images

In many games, you can produce quite startling effects by using mirror images of objects, part objects or backgrounds. While quite impractical in Basic, these effects are very easy to produce in ARM code.

Probably the simplest to consider is where the top half of the screen is folded down onto the bottom half. We know that the screen is just a continuous line of bytes, so if you divide the screen size by two you have the halves necessary for you mirroring. At first thought you'd probably be tempted to simply use two counters, one stepping forwards and the other backwards, until they meet in the middle. This will certainly give you a mirror image, but unfortunately it will also be reversed left to right.

Worse still, if you use blocks of registers for the copying, you will get stripes of forward image, but stepped in the wrong direction. That may actually be useful with a symmetrical layout, or for special effects, but the more likely need is for just vertical mirroring. To do this you need to reverse the order that the lines are counted in, but not the way the bytes on each line are.

As you are working in lines you may be inclined to keep a line count for both halves of the screen, the source and destination counters. In fact you only need to do this for one. Assuming you are copying from top to bottom, the source counter only need to be checked against reaching the mid point. This determines the end of the whole reflection. The destination however, needs to start at the last line of the screen. It is then checked against a marker that is exactly one line higher. When the two are equal, the destination counter has two lines subtracted from it (not one), and the marker has one line subtracted. This is repeated until the reflection is completed.

Left to right mirroring is much harder. This is because the order of bytes needs to be reversed. While it is easier to do a byte at a time it is also slow, and if you use anything other than a 256 colour mode, you will still have to reverse some of the bits in each byte.

The solution is to load whole registers, swap their bits into other registers then store the reversed registers. If you look at Figure 10.2 you will see just how this looks with a Mode 13 screen. This is marked out in pixel blocks. Obviously, only the first few bytes of the lines are shown, and only for three lines. You can see that not only do the registers themselves swap, but as one byte represents exactly one pixel, the bytes themselves are in reverse order.

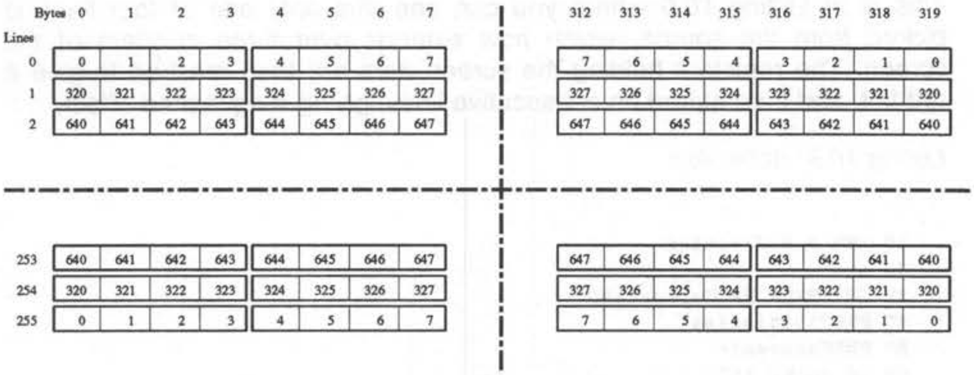

Figure 10.2: Mode 13 screen reflections

A method of making this reversal, for a 256 colour mode, is shown in the fragment below. In this case we need to reverse the order of whole bytes. The register to be reversed is R0 and the result is obtained in R1. R0 itself remains unaltered.

```
MOV R1,#0              ; clear results
MOV R3,#&FF
ORR R3,R3,R3,LSL#16    ; create mask
AND R2,R0,R3
ORR R1,R1,R2,ROR#8
MVN R3,R3              ; invert mask
AND R2,R0,R3
ORR R1,R1,R2,ROR#24
```

You can use this method to reverse a group of registers by interleaving them. This will improve overall efficiency as the mask in R3 will only have to be built up once, and inverted once for the whole group of registers.

Having resolved the detail, we can look at the overall reflection. What we need to do, assuming left to right reflecting, is to count up for half a line from the left for the source, while counting down from the end of the same line for the destination. When the two counters are equal, half a line is added to the source register and a line and a half to the destination. This is repeated until the half line addition on the source exceeds the end of the screen.

With a bit of thought you can produce a routine that will take just one quadrant of the screen and reflect it both ways, but I'll leave the subject for now, with a novel reflection program that adds both distortion and tinting. This is in Listing 10.5 where you can see that only one of four lines is picked from the source, which now extends over three quarters of the screen. The registers holding the screen data are then masked to give a red tint, and then stored as consecutive lines, giving a squashed effect.

Listing 10.5: Reflection

```
   10 REM > Reflection
   20 :
   30 ON ERROR PROCerror:END
   40 PROCinitialise
   50 PROCassemble
   60 IF INKEY 150
   70 REPEAT
   80   FOR I%=0 TO num%
   90     GCOL c%(I%)
  100     CIRCLE FILL x%(I%),y%(I%),s%(I%)
  110     IF ABS x%(I%)>640 dx%(I%)=-dx%(I%)
  120     IF ABS y%(I%)>512 dy%(I%)=-dy%(I%)
  130   NEXT
  140   x%()=x%()+dx%()
  150   y%()=y%()+dy%()
  160   F%=USR code%
  170 UNTIL FALSE
  180 END
  190 :
  200 DEF PROCerror
  210 MODE 12
  220 IF ERR<>17 PRINT REPORT$ " @ ";ERL
  230 ENDPROC
  240 :
  250 DEF PROCinitialise
  260 MODE 15
  270 MODE 13
  280 OFF
  290 PRINT TAB(9,9) "Screen Reflection" TAB(12,13) "Please wait" TAB(
11,17) "Escape to stop"
  300 COLOUR 128+%100000
  310 DIM block% 27
  320 block%!0=149:          REM screen base address
  330 block%!4=6:            REM line length
  340 block%!8=7:            REM screen size
  350 block%!12=-1
```

```
360 SYS "OS_ReadVduVariables",block%,block%+16
370 A%=block%!20
380 B%=A%*4:              REM 4 line step
390 G%=block%!24:         REM screen size
400 C%=G%+block%!16:      REM end of screen
410 D%=B%+C%-G% DIV 4:    REM last quarter screen
420 E%=&10101010:         REM reflection mask
430 F%=1:                 REM screen bank
440 num%=5
450 DIM x%(num%),y%(num%)
460 DIM dx%(num%),dy%(num%)
470 DIM s%(num%),c%(num%)
480 FOR I%=0 TO num%
490   c%(I%)=RND(31)+31
500   s%(I%)=RND(32)+16
510   x%(I%)=RND(1279)-640
520   y%(I%)=RND(1023)-512
530   dx%(I%)=RND(2)*4
540   dy%(I%)=RND(2)*4
550 NEXT
560 ORIGIN 640,512
570 ENDPROC
580 :
590 DEF PROCassemble
600 DIM code% &100
610 line=0
620 fourline=1
630 end=2
640 reflect=3
650 mask=4
660 bank=5
670 size=6
680 linend=7
690 screen=8
700 link=14
710 FOR I%=0 TO 2 STEP 2
720   P%=code%
730   [ OPT I%
740   CMP bank,#2                     ; check screen bank
750   ADDEQ reflect,reflect,size
760   ADDEQ end,end,size
770   MOV screen,reflect
771   ;
780   .frameloop
790   SUB screen,screen,fourline      ; step back four lines
800   ADD linend,screen,line          ; set end of line marker
801   ;
810   .lineloop
820   LDMIA (screen)!,{9,12}          ; grab 4 reg blocks
830   ORR 9,9,mask                    ; mask them
840   ORR 10,10,mask
850   ORR 11,11,mask
```

```
 860    ORR 12,12,mask
 870    STMIA (reflect)!,{9,12}         ; slop them back
 880    CMP screen,linend
 890    BLT lineloop
 900    CMP reflect,end                 ; all done ?
 910    BLT frameloop
 911    ;
 920    MOV R0,#19
 930    SWI "OS_Byte"                   ; wait
 940    MOV R0,#113
 950    MOV R1,bank
 960    EOR bank,bank,#3                ; swap banks
 970    SWI "OS_Byte"
 980    MOV R0,#112
 990    MOV R1,bank
1000    SWI "OS_Byte"
1010    MOV R0,bank                     ; clear screen
1020    SWI &10C
1030    MOV PC,link
1050    ]
1060 NEXT
1070 ENDPROC
```

10.6 ARM Scrolling

In Chapter 5 I referred briefly to scrolling in ARM code. Although there are a great many possibilities for this, I'll only be covering a single basic method. This is a windowing technique. Listing 10.6 is a program that produces three such windows onto the same Mode 13 scene, controlled quite independently. The scrollable area is exactly one screen in size. This allows us to use bank switching and normal graphic commands to produce the scene, and even update it later. The simple circle plotting is duplicated in some cases. This is so that objects that overlap the screen edges appear to wrap round, giving the appearance of an infinite area.

Listing 10.6: ARMscroll

```
  10 REM > ARMscroll
  20 :
  30 REM A% = start position in stored screen
  40 REM B% = display start
  50 REM C% = display width
  60 REM D% = display height
  70 REM E% = screen base
  80 REM F% = screen size
  90 REM G% = line length
 100 :
 110 ON ERROR PROCerror:END
 120 PROCinitialise
 130 PROCassemble
 140 PROCdraw
 150 REPEAT
 160   IF INKEY-98 X%+=4:IF X%>edge% X%-=G%
 170   IF INKEY-67 X%-=4:IF X%<0 X%+=G%
 180   IF INKEY-80 Y%+=vert%:IF Y%>top% Y%-=F%
 190   IF INKEY-105 Y%-=vert%:IF Y%<0 Y%+=F%
 200   IF INKEY-122 PROCshiftx(4)
 210   IF INKEY-26 PROCshiftx(-4)
 220   IF INKEY-42 PROCshifty(vert%)
 230   IF INKEY-58 PROCshifty(-vert%)
 240   auto1%+=G%:IF auto1%>top% auto1%-=F%
 250   auto2%+=4:IF auto2%>top% auto2%-=F%
 260   WAIT
 270   PROCscroll(X%+Y%,frame0%,width0%,height0%)
 280   PROCscroll(auto1%,frame1%,width1%,height1%)
 290   PROCscroll(auto2%,frame2%,width2%,height2%)
 300 UNTIL FALSE
 310 END
 320 :
 330 DEF PROCerror
 340 MODE 12
 350 IF ERR<>17 PRINT REPORT$ " @ ";ERL
 360 ENDPROC
 370 :
 380 DEF PROCinitialise
 390 mode%=13            :REM only thing to change for different modes
 400 SYS"OS_ReadModeVariable",mode%,7 TO ,,F%  :REM checks the mode
 410 SYS"OS_ReadModeVariable",mode%,3 TO ,,col% :REM we want not the
 420 IF col%<63 ERROR 0,"Not a 256 colour Mode" :REN current one
 430 DIM block% 19
 440 block%!0=150
 450 block%!4=-1
 460 SYS "OS_ReadVduVariables",block%,block%+8
 470 IF block%!8<2*F% ERROR 0,STR$(F%DIV512)+"k needed for screen"
 480 :
 490 MODE mode%
 500 OFF
 510 block%!0=149  :REM Now we want info for
 520 block%!4=6    :REM the current mode
 530 block%!8=-1
 540 SYS "OS_ReadVduVariables",block%,block%+12
 550 E%=block%!12
 560 G%=block%!16
 570 COLOUR 131
 580 CLS
 590 PRINT TAB(9,2) "Windowed Scrolling"
 600 PRINT TAB(5,13) "Scroll" SPC3  "Z  X  '  /"
 610 PRINT TAB(5,14) "Move" SPC5 CHR$136 "  " CHR$137 "  " CHR$
139 "  " CHR$138
 620 PRINT TAB(5,15) "Escape to Stop"
 630 xmult%=1280 DIV G%
 640 ymax%=F% DIV G%         :REM everything hangs on G% and F%
 650 ymult%=1024 DIV ymax%
 660 vert%=G%<<2
 670 edge%=G%-1
 680 top%=F%-1
 690 X%=0
 700 Y%=0
 710 auto1%=0
 720 auto2%=0
 730 frame2%=G%*32+G% DIV 8
 740 frame1%=G%*32+G% DIV 2
 750 frame0%=G%*136+G% DIV 8
 760 width2%=G% DIV 4
 770 width1%=G% DIV 4
 780 width0%=G% DIV 2
 790 height2%=G%<<6
 800 height1%=G%<<6
 810 height0%=G%<<6
 820 ENDPROC
 830 :
 840 DEF PROCdraw
 850 *FX 112 2
 860 CLG
 870 FOR I%=0 TO 99
 880   GCOL 15+RND(47)
 890   R%=8+RND(100)
 900   x%=R%+RND(1279-R%)
 910   y%=R%+RND(1023-R%)
 920   CIRCLE FILL x%,y%,R%
 930   IF y%+R%*2>1023 CIRCLE FILL x%,y%-1024,R%
 940   IF x%+R%*2>1279 THEN
 950     CIRCLE FILL x%-1280,y%,R%
 960     IF y%+R%*2>1024 CIRCLE FILL x%-1280,y%-1024,R%
 970   ENDIF
 980 NEXT
 990 *FX 112 1
1000 GCOL 3 TINT 0
1010 ENDPROC
1020 :
1030 DEF PROCassemble
1040 DIM code% &100
1050 storestart=0
1060 storeindex=0
1070 dispstart=1
1080 dispindex=1
1090 dispwidth=2
1100 inc=2
1110 dispheight=3
1120 dispend=3
1130 screenbase=4
1140 storeend=4
1150 screensize=5
```

```
1160 linelength=6
1170 endofline=7
1180 storebase=8
1190 lowreg=9
1200 stepcount=10
1210 highreg=12
1220 link=14
1230 FOR I%=0 TO 2 STEP 2
1240   P%=code%
1250   [ OPT I%
1260   ADD dispindex,screenbase,dispstart
1270   ADD dispend,dispindex,dispheight
1280   ADD endofline,dispindex,dispwidth
1290   SUB inc,linelength,dispwidth
1300   ADD storebase,screenbase,screensize
1310   ADD storeindex,storebase,storestart
1320   ADD storeend,storebase,screensize
1330   SUB storeend,storeend,#16
1340   ;
1350   .loop
1360   CMP storeindex,storeend          ; could enter with screen address
1370   BGE fudge                        ; right on the boundary
1380   LDMIA(storeindex)!,{lowreg-highreg}
1390   STMIA(dispindex)!,{lowreg-highreg} ; slop 4 regs over
1400   ;
1410   .stopfudge
1420   CMP dispindex,endofline
1430   ADDGE storeindex,storeindex,inc
1440   ADDGE dispindex,dispindex,inc
1450   ADDGE endofline,endofline,linelength
1460   CMP dispindex,dispend
1470   BLT loop
1480   ;
1490   .end
1500   MOV PC,link
1510   ;
1520   .fudge                  ; deals with last & first
1530   ADD storeend,storeend,#16 ; 4 regs at screen ends
1540   MOV stepcount,#8        ; could be 1,2 or 3
1550   ;
1560   .fudgeloop
1570   CMP storeindex,storeend
1580   SUBGE storeindex,storeindex,screensize
1590   CMP dispindex,endofline
1600   ADDGE storeindex,storeindex,inc
1610   ADDGE dispindex,dispindex,inc
1620   ADDGE endofline,endofline,linelength
1630   CMP dispindex,dispend
1640   BGE end
1650   LDR lowreg,[storeindex],#4     ; only move
1660   STR lowreg,[dispindex],#4      ; 1 reg over
1670   SUBS stepcount,stepcount,#1
```

```
 1680   BNE fudgeloop
 1690   B stopfudge
 1700   ]
 1710 NEXT
 1720 ENDPROC
 1730 :
 1740 DEF PROCshiftx(x%)
 1750 le%=(frame0% MOD G%)*xmult%
 1760 ri%=((frame0%+width0%)MOD G%)*xmult%
 1770 IF x%<0 AND le%=0 OR x%>0 AND ri%=0 ENDPROC
 1780 lo%=(ymax%-((frame0%+height0%)DIV G%))*ymult%
 1790 hi%=(height0% DIV G%)*ymult%
 1800 IF x%>0 RECTANGLE FILL le%,lo%,4*xmult%,hi% ELSE RECTANGLE FILL
ri%-4*xmult%,lo%,8*xmult%,hi%
 1810 frame0%+=x%
 1820 ENDPROC
 1830 :
 1840 DEF PROCshifty(y%)
 1850 lo%=(ymax%-((frame0%+height0%)DIV G%))*ymult%
 1860 hi%=(ymax%-(frame0% DIV G%))*ymult%
 1870 IF y%>0 AND lo%=0 OR y%<0 AND hi%>=ymax%*2 ENDPROC
 1880 le%=(frame0% MOD G%)*xmult%
 1890 ri%=(width0% MOD G%)*xmult%
 1900 IF y%<0 RECTANGLE FILL le%,lo%,ri%,4*ymult% ELSE RECTANGLE FILL
le%,hi%-4*ymult%,ri%,4*ymult%
 1910 frame0%+=y%
 1920 ENDPROC
 1930 :
 1940 DEF PROCscroll(A%,B%,C%,D%)
 1950 CALL code%
 1960 ENDPROC
```

The heart of scroll action itself, consists of maintaining a pointer to the start of the displayed portion of the scroll area, which is then modified, by four byte values for horizontal movement, or 320 bytes, or one line values for vertical movement. For greater flexibility, the actual display position on the screen can be altered by modifying its pointer in a similar way.

These parameters, along with the display width and height, are used by the ARM code routine to transfer the data from the scroll area to the visible screen. The routine first calculates the screen base offsets for the start and end points of the display, along with the end of line marker, and a difference figure used after each display line has been plotted. This sets the display pointer to the start of the next line. The scroll area pointers and limits are set in a similar manner, then the main transfer loop is entered.

The main problem here is that of alignment against speed. As before, byte alignment has been abandoned completely, so that whole registers can be used simply. However, even this is not good enough for manipulating the huge amounts of memory that have to be transferred quickly. Four word handling gives the necessary speed, but being simplistic and only allowing four word scrolling steps, is unreasonable.

A difficulty arises when the window on the scroll area overlaps the end of the screen end. The main loop handles only four registers at a time, or 16 bytes, if there is say, a one word overlap. If there was no special allowance made, the routine would try to write three words into a prohibited memory area. This is the reason for the fudge routine. To allow for all contingencies, the end of store marker is reduced by four words, and if the store index reaches this value, eight registers are transferred one at a time, continually checking for the true end of the scroll area. Once this has been reached the store index is decremented by an entire screen so that the routine correctly manages a wrap and carries on from this point. Once all eight register transfers have been made the main loop is re-entered and four register blocks are transferred as before.

The final result is that there is a smooth fast scrolling action, with register transfers being made for all but an insignificant amount of time. There is however, still one problem. If you scroll the user window carefully sideways, you will see that every so often it jumps one line up or down, depending on the scroll direction. This is due to the fact that I've used a simple line adding system for the store memory. What is happening, is that, as the store index is incremented it makes no allowance for the fact that at the end of a line the screen should wrap back to the same line, not the next line as happens when you just keep counting up through the screen. To cure this, you need to change the program so that the store index is incremented by re-calculating the start of each line, then adding an extra check to subtract a whole line from the index, if it goes through a line boundary.

As a point of interest, the entry to the code itself is wrapped up in a procedure, so that the three separate calls can be made to it with independent parameters, while the code just accepts register values taken, apparently, from the same resident integers.

10.7 Some Final Points

Quite often, integer multiplication is required within ARM code. There is a temptation just to use the MUL instruction regardless, but this can be very inefficient. In the first place, you should try to plan your routines so that wherever possible, powers of two are employed. All you then need to do is barrel shift left for multiplication, or right for division.

A point well worth keeping in mind is the way you can produce quite a range of seemingly complex fast multiplications with combinations of barrel shifting, addition and subtraction. Look at the examples below. All of them, with just one instruction, take the value in R0 and without corrupting it produce a result in R1.

```
ADD R1,R0,R0,LSL#1   ; x2+x1   gives x3
ADD R1,R0,R0,LSL#2   ; x4+x1   gives x5
RSB R1,R0,R0,LSL#3   ; x8-x1   gives x7
ADD R1,R0,R0,LSL#3   ; x8+x1   gives x9
```

10.7.1 Random numbers

Below is a practical application that also introduces a simple ARM code random number generator. The number generator itself creates a random integer, whereas we want a one-of-five selection. Your first temptation might be simply to mask this to get the range you want. However this won't work except where the wanted range is a power of two. With some values masking can appear to work, but actually doubles the chance of certain numbers appearing. The answer is to mask out one byte, multiply it by the range we want then divide by 256. In the example given we end up with a random number between 0 and 4, a range of 1 to 5, as we wanted.

```
.random
   LDR R0,seed
   CMP R0,#0                 ; traps zero seed
   MOVEQ R0,#255             ; omit seed is never zero
   MOV R2,#17
;
.random_loop
        MOV R1,R0,ASR #13
        EOR R1,R1,R0,ASR #24
        MOVS R1,R1,ROR #1
        ADCS R0,R0,R0
        SUBS R2,R2,#1
   BNE random_loop
   STR R0,seed
   AND R0,R0,#&FF            ; mask 0 - 255
   ADD R0,R0,R0,LSL#2        ; * 5
   MOV R0,R0,LSR#8           ; / 256
   MOV PC,link
;
.seed
   EQUD 1234                 ; any number except zero
```

10.7.2 Square roots

One problem with ARM code programming is that important, very complex routines provided within Basic are no longer available to you. One of these is for calculating square roots. There are a number of algorithms for this, but I'll stick to the two simplest methods. The first is the iterative method. For this, you simply guess at the number, then multiply it by itself to square it. If the result is too high, you make a downward adjustment to the guess, and if the result is too low, you adjust up. The adjustment steps are made progressively smaller and smaller until you reach the required accuracy.

The second method is a simple additive method. All you do is keep adding progressively higher and higher odd numbers, keeping a count of how many you add, until your total exceeds the number to be square rooted. The count is then the integer part of the square root. This method is extremely fast for small numbers, but gets progressively slower as the numbers increase. If you add even numbers instead of odd numbers you get a more useful figure. This is an integer value for the nearest square root, rather than the value below the correct floating point figure.

Listing 10.7 shows both of these methods and a comparison with Basic, along with their timings for significant numbers.

Listing 10.7: Roots

```
 10 REM > Roots
 20 :
 30 MODE 12
 40 PROCassemble
 50 :
 60 M%=6000
 70 PROCtest(25,M%)
 80 PROCtest(&419,M%)
 90 PROCtest(&10133,M%)
100 PROCtest(&FFFFFF,M%)
110 END
120 :
130 DEF PROCtest(B%,M%)
140 PRINT"M% " iterations finding root of ";B%
150 :
160 TIME=0
170 FORI%=0TOM%:A%=SQRB%:NEXT
180 PRINT TIME " centi-seconds Basic SQR" SPC11 "Result is ";SQR B%
190 :
200 TIME=0
210 FORI%=0TOM%:A%=USRC%:NEXT
```

```
220 PRINT TIME " centi-seconds Iterative method" SPC4 "Result is ";A%
230 :
240 TIME=0
250 FORI%=0TOM%:A%=USRD%:NEXT
260 PRINT TIME " centi-seconds Additive method" SPC5 "Result is ";A%
270 :
280 TIME=0
290 FORI%=0TOM%:A%=USRE%:NEXT
300 PRINT TIME " centi-seconds Dummy code"
310 ENDPROC
320 :
330 DEF PROCassemble
340 DIM C% &100
350 root=0
360 number=1
370 test=2
380 step=3
390 comp=4
400 shift=5
410 mult=6
420 link=14
430 FOR I%=0 TO 2 STEP 2
440   P%=C%
450   [ OPT I%
460   MOV root,number,LSR#1
470   MOV step,root,LSR#1
480   CMP number,#&15000  ; adjust for big numbers
490   MOVGT shift,#1
500   MOVLE shift,#0
510   CMPGT number,#&2D000  ; very big numbers
520   ADDGT shift,shift,#1
530   CMPGT number,#&5A000  ; even bigger ones
540   ADDGT shift,shift,#1
550   CMPGT number,#&B5000  ; enormous numbers
560   ADDGT shift,shift,#1
570   CMPGT number,#&168000 ; stupendous ones
580   ADDGT shift,shift,#1
590   CMPGT number,#&2D4000 ; wow!
600   ADDGT shift,shift,#1
610   CMPGT number,#&5A0000 ; phew!
620   ADDGT shift,shift,#1
630   CMPGT number,#&B40000 ; the limit
640   ADDGT shift,shift,#1
650   MOV comp,number,LSR shift
660   MOV comp,comp,LSR shift
670   ;
680   .iterloop
690   MOV mult,root,LSR shift
700   MUL test,mult,mult
710   CMP test,comp
720   ADDLT root,root,step
730   SUBGT root,root,step
740     MOVGTS step,step,LSR#1
750     BNE iterloop
760     MOV PC,link
770     ;
780     .D%
790     MOV root,#0
800     ;
810     .addloop
820     ADD root,root,#1
830     SUBS number,number,root,LSL #1
840     BGE addloop
850     ;
860     .E%
870     MOV PC,link
880     ]
890 NEXT
900 ENDPROC
```

When you run the program you will see that the Basic algorithm is pretty consistent for both speed and accuracy, bearing in mind it handles floating point numbers. The additive method starts out by far the fastest, but gets dramatically slower with the largest numbers. Handling purely integers, its accuracy increases proportionately with the larger numbers. The iterative method holds its speed quite well, but the accuracy is rather variable. This is because of the ranging adjustments that are made. The worst case error caused by these adjustments is nearly 5% for numbers just higher than the last adjustment value.

10.7.3 Fast circles

It would seem fairly logical to assume that you could devise a circle plotting routine that was considerable faster than the one built into Risc Os. As you don't have to make the Mode and colour translation tests, there should be quite an improvement in speed. If you limit the circles to those fully on screen there isn't the problem of edge testing either. However for circle plotting, you can't avoid some byte manipulation, and if you just perform a byte by byte plot, you won't make much improvement in speed.

The answer is to perform a variation on the fudge idea that was introduced earlier. Starting each line from the left, you perform a byte by byte plot until your plotting index is word aligned. You then carry on plotting whole words until there are no more complete words that can be fitted in. Finally you finish off the line, plotting byte by byte again.

Listing 10.8 achieves just this, and manages to plot circles in about half the time taken by the Risc Os plotter. The algorithm used is an interesting adaptation of Pythagoras that gives us the perimeter of the circle without needing any polar calculations. When plotting small circles, the implementation breaks down a bit, and the circles become rather angular, so you may be better off reverting to the Risc Os plotter for these. For the tiniest ones, you should just plot a handful of bytes from a look up table.

Listing 10.8: Circles

```
   10 REM > Circles
   20 :
   30 REM A% = X co-ordinate in bytes
   40 REM B% = Y co-ordinate in lines
   50 REM C% = radius in bytes
   60 REM D% = absolute colour
   70 :
   80 ON ERROR PROCerror:END
   90 PROCinitialise
  100 PROCassemble
  110 REPEAT
  120   D%=15+RND(47)*4
  130   C%=5+RND(128)
  140   A%=C%DIV2+RND(F%-C%)
  150   B%=C%DIV2+RND(G%-C%)
  160   IFINKEY-99 WAIT
  170   CALL code%
  180 UNTIL FALSE
  190 :
  200 DEF PROCerror
  210 MODE 12
  220 IF ERR<>17 PRINT REPORT$ " @ ";ERL
  230 ENDPROC
  240 :
  250 DEF PROCinitialise
  260 MODE 13
  270 OFF
  280 PRINT TAB(13,6) "Fast Circles" TAB(8,12) "Press any key to start
" TAB(6,14) "Hold Spacebar to slow down" TAB(12,16) "Escape to stop"
  290 IF GET
  300 ENDPROC
  310 :
  320 DEF PROCassemble
  330 DIM block% 27
  340 !block%=149
  350 block%!4=6
  360 block%!8=7
  370 block%!12=-1
  380 SYS "OS_ReadVduVariables",block%,block%+16
  390 E%=block%!16          : REM screen base
  400 F%=block%!20          : REM bytes per line
  410 G%=(block%!24)DIV F% : REM number of lines
  420 DIM code% &180
```

```
430 horiz=0
440 count=0
450 vert=1
460 linesqr=1
470 radius=2
480 colour=3
490 screenadd=4
500 lines=4
510 screenline=5
520 radsqr=6
530 upper=7
540 lower=8
550 diff=9
560 shift=10
570 upperleft=11
580 lowerleft=0
590 blockright=12
600 right=1
610 link=14
620 FOR I%=0 TO 2 STEP 2
630   P%=code%
640   [ OPT I%
650   ADD colour,colour,colour,LSL#8
660   ADD colour,colour,colour,LSL#16
670   ADD upper,screenadd,horiz
680   MUL vert,screenline,vert
690   ADD upper,upper,vert
700   SUB lower,upper,screenline
710   MOV lines,#0
720   MUL radsqr,radius,radius
730   ;
740   .vertloop
750   MUL linesqr,lines,lines
760   SUB count,radsqr,linesqr
770   MOV diff,#1
780   ;
790   .widthloop
800   ADD diff,diff,#1
810   SUBS count,count,diff,LSL #5  ; adjust for width
820   BGT widthloop
830   ADD right,lower,diff
840   SUB upperleft,upper,diff
850   SUB lowerleft,lower,diff
860   ANDS shift,lowerleft,#3
870   BEQ block
880   RSB shift,shift,#4
890   SUB diff,right,lowerleft
900   CMP diff,shift
910   MOVLT shift,diff
920   ;
930   .leftloop
940   STRB colour,[lowerleft],#1
```

```
 950    STRB colour,[upperleft],#1
 960    SUBS shift,shift,#1
 970    BNE leftloop
 980    CMP diff,#4
 990    BLT nomore
1000    ;
1010    .block
1020    SUB blockright,right,#4
1030    CMP lowerleft,blockright
1040    BGE rightloop
1050    ;
1060    .mainloop
1070    STR colour,[lowerleft],#4
1080    STR colour,[upperleft],#4
1090    CMP lowerleft,blockright
1100    BLT mainloop
1110    ;
1120    .rightloop
1130    STRB colour,[lowerleft],#1
1140    STRB colour,[upperleft],#1
1150    CMP lowerleft,right
1160    BLT rightloop
1170    ;
1180    .nomore
1190    ADD upper,upper,screenline
1200    SUB lower,lower,screenline
1210    ;
1220    ADD lines,lines,#4
1230    CMP lines,radius
1240    BLE vertloop
1250    MOV PC,link
1260    ]
1270 NEXT
1280 ENDPROC
```

10.7.4 Clipping

All the examples in this chapter have assumed that the objects being plotted will remain within the screen boundaries. While this is highly desirable from the programming point of view, there are times when you want to define sprites that go off the edge of the screen. If you just let them overlap the edges, the results will be pretty disastrous. What you have to do is to calculate the minimum and maximum values for the start and end points of your screens, both vertically and horizontally, then compare the sum of the sprite coordinates and its size, with the boundary figures you calculated. If your sprite is outside these limits you adjust its perimeter accordingly. Don't forget that you will also need to adjust the byte or line count to ensure that you don't just push the sprite to a different position.

To make this clearer, Figure 10.3 shows a number of clipping situations.

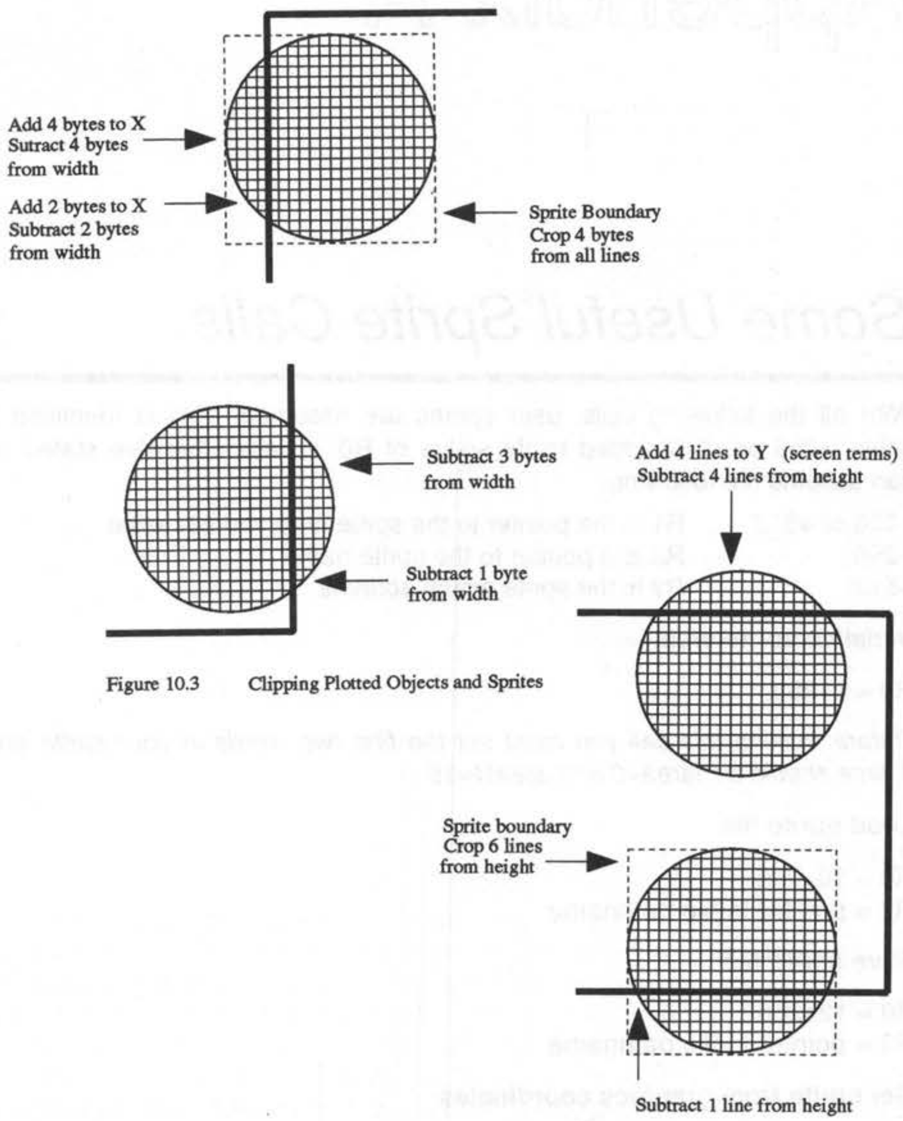

Figure 10.3: Clipping plotted objects and sprites

Appendix A

Some Useful Sprite Calls

With all the following calls, user sprites are assumed. This is identified by either +256 or +512 added to the value of R0. Unless otherwise stated you can assume the following:

+256 or +512	R1 is the pointer to the sprite area you set aside
+256	R2 is a pointer to the sprite name
+512	R2 is the sprite actual address

Initialise sprite area

R0 = 9+256

Before making this call you must set the first two words in your sprite area. These should be !area=0 and area!4=16

Load sprite file

R0 = 10+256
R2 = pointer to file pathname

Save sprite file

R0 = 12+256
R2 = pointer to file pathname

Get sprite from graphics coordinates

R0 = 14+256
R2 = pointer to sprite name
R3 = 0 to exclude palette data, 1 to include it
Return
R2 = address of sprite

Create empty sprite

R0 = 15+256
R2 = pointer to sprite name
R3 = 0 to exclude palette data, 1 to include it
R4 = width in pixels
R5 = height in pixels
R6 = screen Mode number

Get sprite from user coordinates
R0 = 16+256
R2 = pointer to sprite name
R3 = 0 to exclude palette data, 1 to include it
R4 = left edge OS screen coordinate
R5 = bottom edge OS screen coordinate
R6 = right edge OS screen coordinate
R7 = top edge OS screen coordinate

Return
R2 = address of sprite

Put sprite to current graphics cursor

R0 = 28+512
R5 = plot action

Plot actions are as follows:

0 overwrite on screen
1 OR with colour on screen

2 AND with colour on screen

3 EOR with colour on screen

4 invert colour on screen

5 leave colour on screen unchanged

6 AND with colour on screen with NOT of sprite pixel colour

7 OR with colour on screen with NOT of sprite pixel colour

&08 if set, use mask, otherwise don't

&10 ECF pattern 1

&20 ECF pattern 2

&30 ECF pattern 3
&40 ECF pattern 4
&50 giant ECF pattern

Create mask with all pixels set (solid)

R0 = 29+512

Put sprite at user coordinates

R0 = 34+512
R3 = X coordinate
R4 = Y coordinate
R5 = plot action

Read pixel from sprite

R0 = 42+512
R3 = X coordinate
R4 = Y coordinate
Return
R5 = colour
R6 = tint

Write pixel to sprite

R0 = 42+512
R3 = X coordinate
R4 = Y coordinate
R5 = colour
R6 = tint

Read mask pixels

R0 = 43+512
R3 = X coordinate
R4 = Y coordinate
Return
R5 = status, 0 for transparent, 1 for solid

Write mask pixels

R0 = 44+512
R3 = X coordinate
R4 = Y coordinate
R5 = status

Put scaled sprite

R0 = 52+512
R3 = X coordinate to plot
R4 = Y coordinate to plot
R5 = plot action
R6 = pointer to scale control block
R7 = pointer to pixel translation table, 0 to use sprite colours

The scale control block byte offsets are as below

0	X multiplier
4	Y multiplier
8	X divisor
12	Y divisor

For pixel translation, a sprite pixel colour of N will be plotted as the colour represented by the Nth byte in the table.

Switch output to sprite

R0 = 60+512
R2 = sprite address as usual, but if zero will restore screen output
R3 = save area (generally not needed, set to 0)

Switch output to mask

R0 = 61+512
R2 = sprite mask address, 0 for screen output
R3 = save area, normally 0

Appendix B

VDU Variables (Sub-set)

Number	Meaning
1	Maximum column number for printing
2	Maximum row number for printing
4	Barrel shift right to convert X coordinate to screen pixels
5	Barrel shift right to convert Y coordinate to screen pixels
6	Screen line length in bytes
7	Size in bytes of the visible screen
148	Address of screen start use by VDU drivers
149	Address of displayed screen start
150	Total number of bytes currently available to screen

Appendix C

Screen Modes for Standard Monitors

Mode	Text	Graphics	Colours	Memory
0	80x32	640x256	2	20k
1	40x32	320x256	4	20k
2	20x32	160x256	16	40k
3	80x25	Text only	2	40k
4	40x32	320x256	2	20k
5	20x32	160x256	4	20k
6	40x25	Text only	2	20k
7	40x25	TELETEXT	16	80k
8	80x32	640x256	4	40k
9	40x32	320x256	16	40k
10	20x32	160x256	256	80k
11	80x25	640x250	4	40k
12	80x32	640x256	16	80k
13	40x32	320x256	256	80k
14	80x25	640x250	16	80k
15	80x32	640x256	256	160k
16	132x32	1056x256	16	132k
17	132x25	1056x250	16	132k

Appendix D

Plot Codes

Plot codes are in groups of eight. Below are the base numbers of each group.

Dec	Hex	Meaning
0	00	Solid line including both end points
8	08	Solid line excluding final point
16	10	Dotted line including both end points
24	18	Dotted line excluding final point
32	20	Solid line excluding first point
40	28	Solid line excluding both end points
48	30	Dotted line excluding first point
56	38	Dotted line excluding both end points
64	40	Point plot
72	48	Horizontal line fill to non-background (left and right)
80	50	Triangle fill
88	58	Horizontal line fill to background (right only)
96	60	Rectangle fill
104	68	Horizontal line fill to foreground (left and right)
112	70	Parallelogram fill
120	78	Horizontal line fill to non-foreground (right only)
128	80	Flood fill to background
136	88	Flood fill to foreground
144	90	Circle outline
152	98	Circle fill
160	A0	Circular arc
168	A8	Segment
176	B0	Sector
184	B8	Block copy/move
192	C0	Ellipse outline
200	C8	Ellipse fill

Within each block the offset from the base number has the following meaning:

0	Move cursor relative to last point visited
1	Draw relative, using current foreground colour
2	Draw relative, using logical inverse colour
3	Draw relative, using current background colour
4	Move cursor to absolute coordinates
5	Draw absolute, using current foreground colour
6	Draw absolute, using logical inverse colour
7	Draw absulute, using current background colour

Block copy and move are exceptions whose codes are:

0	Move only, relative
1	Move rectangle relative
2	Copy rectangle relative
3	Copy rectangle relative
4	Move only, absolute
5	Move rectangle absolute
6	Copy rectangle absolute
7	Copy rectangle absolute

Appendix E

VDU Commands

Code	Ctrl	Bytes	Meaning
0	@	0	Does nothing
1	A	1	Sends next character to printer only
2	B	0	Enables printer
3	C	0	Disables printer
4	D	0	Writes characters at text cursor
5	E	0	Writes characters at graphics cursor
6	F	0	Enables VDO driver
7	G	0	Generates bell sound
8	H	0	Moves cursor back one character or deletes previous character
9	I	0	Moves cursor on one space
10	J	0	Moves cursor down one line
11	K	0	Moves cursor up one line
12	L	0	Clears text area
13	M	0	Moves cursor to start of current line
14	N	0	Enables page mode
15	O	0	Disables page mode
16	P	0	Clears graphics area
17	Q	1	Defines text colour
18	R	2	Defines graphics colour
19	S	0	Defines logical colour
20	T	0	Restores default colours
21	U	0	Disables VDU driver or deletes current line
22	V	1	Selects screen mode
23	W	9	General purpose command
24	X	8	Defines graphics window
25	Y	5	PLOT
26	Z	0	Restores default windows
27	[	0	Does nothing
28	\	4	Defines text window
29	]	4	Defines graphics origin
30	^	0	Homes text cursor
31	_	2	Moves text cursor
32-255			Display characters

Appendix F

Negative Inkey Values

Key	Number	Key	Number
Cursor up	–58	Mouse select	–10
Cursor down	–42	Mouse menu	–11
Cursor left	–26	Mouse adjust	–12
Cursor right	–122	=	–94
<Print>	–33	'	–103
<F1>	–114	–	–24
<F2>	–115	.	–104
<F3>	–116	/	–105
<F4>	–21	[	–57
<F5>	–117	\	–121
<F6>	–118	]	–89
<F7>	–23	;	–88
<F8>	–119	<Esc>	–113
<F9>	–120	<Tab>	–97
<F10>	–31	<Caps Lock>	–65
<F11>	–29	<Scroll Lock>	–32
<F12>	–30	<Num Lock>	–78
A	–66	<Break>	–47
B	–101	~\‘	–46
C	–83	Currency	–47
D	–51	<Back Space>	–48
E	–35	<Insert>	–62
F	–68	<Home>	–63
G	–84	<Page Up>	–64
H	–85	<Page Down>	–79
I	–38	‘\"	–80

J	–70	<Shift> (either/both)	–1
K	–71	<Alt> (either/both)	–3
L	–87	<Shift> (left/right)	–4/–7
M	–102	<Ctrl> (left/right)	–5/–8
N	–86	<Alt> (left/right>	–6/–9
O	–55	Space bar	–99
P	–56	<Delete>	–90
Q	–17	<Return>	–74
R	–52	<Copy>	–106
S	–82	Keypad 0	–107
T	–36	Keypad 1	–108
U	–54	Keypad 2	–125
V	–100	Keypad 3	–109
W	–34	Keypad 4	–123
X	–67	Keypad 5	–124
Y	–69	Keypad 6	–27
Z	–98	Keypad 7	–28
0	–40	Keypad 8	–48
1	–49	Keypad 9	–44
2	–50	Keypad +	–59
3	–18	Keypad –	–60
4	–19	Keypad .	–77
5	–20	Keypad /	–75
6	–53	Keypad #	–91
7	–37	Keypad *	–92
8	–22	Keypad <Enter>	–61
9	–39		

INDEX

Archimedes Game Maker's Manual

A worthy companion to our STOS and AMOS Game Maker guides, this shows how to use Archimedes BASIC 5 to build high quality games for both educational and recreational purposes. Although 'old hands' will find this book invaluable, it is ideal for those new to the Archimedes, and probably new to games programming, although some knowledge of programming in BASIC is expected. Contents include:

- ★ Planning a game
- ★ Critical game stages
- ★ Graphics and animation
- ★ Arcade games
- ★ Role Play games
- ★ Board Games and other games of strategy
- ★ Arm Code – professional tips and tricks

Numerous fully tested program listings are supplied for the user PLUS the option of a disc with all the major listings!

ISBN 1-85058-241-6
9 781850 582410

SIGMA