simulation techniques on the commodore 64

practical routines for home and business use

john cochrane



simulation techniques on the commodore 64

practical routines for home and business use

john cochrane

First published 1984 by:
Sunshine Books (an imprint of Scot Press Ltd.)
12–13 Little Newport Street
London WC2R 3LD

Copyright © John Cochrane, 1984

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording and/or otherwise, without the prior written permission of the Publishers.

British Library Cataloguing in Publication Data
Cochrane, John
Simulation techniques on the Commodore 64.
1. Commodore 64 (Computer)—Programming
I. Title
001.64'2 QA76.8.C64

ISBN 0–946408–58–0

Cover design by Grad Graphic Design Ltd.
Cover illustration by Stuart Hughes.
Text illustrations by Linda Jeffrey.
Typeset and printed in England by Commercial Colour Press, London E7.

CONTENTS

		Page
Foreword		vii
Program Notes		ix
	PART 1: THEORETICAL BACKGROUND	
1	Made to Measure Introduction	3
2	When I was a Boy A background to simulations	9
3	But What Does This Little Black Button Do? What simulations are and how they are used	25
4	Getting it Together Practical techniques	35
5	I'll Do It My Way What the 64 can do	51
	PART 2: PRACTICAL	
6	Laughing All the Way to the Bank Money models	61
7	Testing Testing Testing Product development	73

8 What We Want Is Information 87
What do you want to know?

9 And Out of the Unknown . . . 103
Simulating the unknown

10 Campaign For Real-time 123
Real-time simulation

Appendix A: Simplespread 139

Glossary 155

Contents in detail

PART 1: THEORETICAL BACKGROUND

A down-to-earth guide to the world of simulation.

CHAPTER 1

Made to Measure

Introduction — who the book is for — what the book is about — who can use simulations — who develops simulations — which techniques can be used on micros — what is the scope of this book — organisation of the book.

CHAPTER 2

When I was a Boy...

A background to simulations — what are simulations — music program — systems thinking — the user — early hardware and simulations — operational analysis — Monte Carlo simulation — industrial simulations.

CHAPTER 3

But What Does This Little Black Button Do?

What simulations are and how they are used — the basic types of simulation — emulation simulations — real-time clock — analytical simulations — predictors — use and abuse of simulations.

CHAPTER 4

Getting it Together

Practical techniques — the importance of defining the requirement — diagrams — feedback — simulation structure — choosing the basic techniques — Monte Carlo — deterministic — Markov chain — technique to suit the simulation — development of the simulation — three movement programs — verifying the model.

CHAPTER 5

I'll Do It My Way

What the C64 can do — basic description — overview — 6510 — 6566/9 — 6581 — 6526 — memory — external hardware — operating system — simulations on microcomputers: general comments — C64 benefits.

PART 2: PRACTICAL

Five full simulations to demonstrate the techniques.

CHAPTER 6

Laughing All the Way to the Bank

Money models, using trend analysis to make financial predictions — background — example — simulation background — spreadsheet — simulation development — results.

CHAPTER 7

Testing Testing Testing

Product development, physical simulation — background — example — simulation development — results — program description — program listing.

CHAPTER 8

What We Want Is Information

What do you want to know?, project planning, project management, and management information — background — example — simulation development — results — program description and use — program improvements — program listing.

CHAPTER 9

And Out of the Unknown...

Simulating the unknown, what to do when you don't know everything — background — example — simulation development — results — program description — program improvements — program listing.

CHAPTER 10

Campaign For Real-time

Real-time simulation, programs designed around the user — background — example — simulation development and results —program description — program use and further development — program listing.

Appendix A

Simplespread

A general-purpose spreadsheet program.

Foreword

```
10 REM "FORWARD"
20 DIM L(11)
30 POKE 54296,15:POKE 54277,130
40 POKE 54278,40:POKE 54272,37
50 POKE 53280,3:POKE 53281,3:PRINT "[CLR
]"
60 PRINT "[CLR]"
70 FOR N=1 TO 25:R$="[CD]"+R$:NEXT
80 FOR N=1 TO 40:C$="[CR]"+C$:NEXT
90 L(0)=83:L(1)=73:L(2)=77:L(3)=85
100 L(4)=76:L(5)=65:L(6)=84:L(7)=69
110 L(8)=45:L(9)=73:L(10)=84:L(11)=33
120 SX=1:SY=1:X=4:Y=4:K=0
130 IF Y<6 THEN SY=1:SX=5*RND(0)-2:B=1
140 IF Y>14 THEN SY=-1:SX=5*RND(0)-2:B=1
150 IF X<6 THEN SX=1:SY=5*RND(0)-2:B=1
160 IF X>30 THEN SX=-1:SY=5*RND(0)-2:B=1
170 IF Y=9 THEN GOSUB 240
180 IF K=0 THEN GOSUB 270
190 GOSUB 280
200 B=0:OX=X:X=INT(X+SX)
210 OY=Y:Y=INT(Y+SY)
220 M=113:J=5:P=32
230 GOTO 130
240 IF X<12 OR X>23 THEN RETURN
250 O=X-12:M=L(O):B=1:P=32:J=144
260 RETURN
270 C=OX:R=OY:N=P:GOTO 290
280 C=X:R=Y:N=M
290 PRINT "[HOME]";LEFT$(R$,R);LEFT$(C$,
C);
300 K=0:IF N>32 AND N<101 THEN K=1
310 PRINT CHR$(J);CHR$(N)
320 IF B=0 THEN RETURN
330 POKE 54276,17:POKE 54273,X+Y
340 FOR T=1 TO 40:NEXT
```

```
350 POKE 54276,33
360 FOR T=1 TO 20:NEXT
370 POKE 54276,0:B=0
380 RETURN
```

Program Notes

For the sake of clarity, control characters in the program lines which are presented throughout this book have been printed as follows:

CURSOR UP	[CU]
CURSOR DOWN	[CD]
CURSOR LEFT	[CL]
CURSOR RIGHT	[CR]
CLEAR SCREEN	[CLR]
HOME CURSOR	[HOME]
REVERSE ON	[RVS]
REVERSE OFF	[RVO]
INSERT	[INST]
DELETE	[DEL]

Colours are represented by the colour name in square brackets. Thus the control character for purple is:

[PUR]

Program listings have been produced using the renumber and list routines prepared by David Lawrence and Mark England in their book *Commodore 64 Disk Companion.*

I have attempted to keep the programs in a form most suitable for easy understanding by using structured principles as far as possible, by using subroutines for each functional part of a program, by using short program lines, and by including remark lines.

Data storage on tape or disk is allowed for.

PART 1

Theoretical Background

This book is in two parts.

In Part 1 I introduce you to the concept of computer simulations, in all its guises, and describe the basic approaches required for the development of practical programs. My guide to this complex and growing field is necessarily an introduction; I hope an understandable one. I hope to give you food for thought, to start you on the road to a full working knowledge of the techniques available for simulation programming and how to apply them. Example programs are used throughout Part 1.

Part 2 presents a number of practical examples and applies the approaches noted in Part 1. Five full simulations are presented in Part 2, all written in BASIC, covering topics such as financial prediction, weather prediction, and production management. I finish with a real-time game played against the computer.

CHAPTER 1

Made to Measure

Introduction



Who the book is for

This book is intended to give the less experienced computer user a theoretical and practical introduction to the development and use of computer simulations. In particular, simulation techniques appropriate for use on the Commodore 64 microcomputer are presented. The reader is expected to have a basic understanding of the principles of computing and, preferably, to have read the Commodore 64 manuals. Practical examples are given throughout to build up general understanding and to illustrate particular points of interest.

What the book is about

So, what are computer simulations? What are they used for? What are the benefits?

None of these questions have simple answers, although Chapters 3 and 4 go some way towards answering them. For now, we can think of computer simulations as special-purpose programs designed to duplicate the workings of the real world and to provide us with output in a very particular form. Every simulation must be developed to fulfil the specific requirements of the user (the user is quite simply the person, or group, who wishes to use the results produced by a computer simulation) and a great variety of types of simulation have been developed. Well-known simulations based on computers include the control routines for full-scale flight simulators used in the training and testing of airline pilots, weather forecasting programs (a particularly difficult simulation task — see Chapter 4), video games of many types, the programs used by television producers to predict election results on live broadcasts, and the (sometimes) hugely complex programs used by engineers to help in the design of such diverse structures as off-shore oil rigs, aircraft, and modern motor vehicles. In addition, much of the research into artificial intelligence and expert systems centres on the simulation of the workings of the human brain.

Who can use simulations

Until very recently, the number of viable simulations which could be used by 'the man in the street' have bebn very few, and very expensive. Typically, a finite-element model (used for analysing the strength of large and small mechanical structures, such as bridges, machine parts, and buildings) would only be available to the largest engineering companies or, at high cost, to other companies through computer bureaux. The running of such a model would cost the user many hundreds of pounds for every run, and the total cost of designing a product using computer simulation techniques would run to many tens or hundreds of thousands of pounds. Obviously these models have been beyond the reach of many companies

and most individuals. The big changes which have occurred recently are the dramatic fall in cost, and rise in capability, of the microcomputer and the development of a number of simulation techniques suitable for the smaller user.

Recently, many new applications for simulations on microcomputers have been recognised, particularly in the video games area. Simulation-based programs for the home computer market, other than games, include sporting predictors, voice synthesizers, and some of the newer expert systems. These applications are of restricted use however; both memory and processing speed limitations are against the commercial exploitation of most practical simulations in the home. Video games, of course, provide a huge market for the skilled simulation writer. With a brief stretch of the imagination a basic technique can be used to create hurtling asteroids, mutant sparrows, or undersea fishes. Flight simulators, motor racing games, and tank-battle simulators take a little more effort, but can successfully be implemented on even the cheapest of home micros.

Business simulations on the other hand are an entirely different kettle of fish. Simulations can be of immense value to a thriving business, but each simulation is likely to be unique. This limits the value of general-purpose programs, and most simulations are developed by professional consultants or by the most enthusiastic employees. Obviously, in this situation, the development of useful programs lags behind the potential demand for those programs. This book seeks to extend the knowledge of 'ordinary' programmers to include the basic techniques required to develop functional simulations.

Who develops simulations

The principal current users of large non-game simulations are development engineers, university research workers, and financial go-getters. The simulations used tend to be experimental; they are used to test out different ideas and determine alternative possible futures. Engineers use detailed simulations, usually home-grown, of prospective constructions, to investigate the potential problems/cost-savings of alternative designs. Universities either work on a contract basis for local industries or use incredibly advanced (at least they think so) techniques for the pure pleasure of the development. In fact, these techniques can lead to genuine advances in 'simulator technology'. Financial modellers are usually interested in developing either sophisticated and large database programs for the rapid analysis of trends or specialised simulations of specific commercial organisations.

Which techniques can be used on micros

Unfortunately, some of the most striking simulations demand very high access speeds to huge reserves of memory. It is still the case that microcom-

puters have severe limitations in the area of memory access. Real-time simulations demand high processing speeds, another weak area for micros. Despite the restricted capabilities of most microcomputers when compared to mainframes, many simulation techniques are usable on microcomputers. However, it is not possible to pack a quart into a pint pot.

Real-time simulations when run on microcomputers remain rather simple. Large database programs often require expensive hardware add-ons, such as hard-disk memories, and can be exasperatingly slow to run. Similarly, applications such as some of the latest artificial intelligence techniques are usually of limited practical value unless kept very basic and then tailored to meet a particular need.

The display capabilities of microcomputers are a particular problem area. Display resolution is kept low in order to preserve memory and to reduce the processing demands on the microprocessor. The following program demonstrates, in a very graphic manner, one of the problems facing the computer engineer. The program draws a low resolution line from a point on the lefthand side of the screen to another point on the screen marked by a flashing cursor. By pressing the cursor keys the flashing cursor can be moved up, down, left, or right. Pressing RETURN draws the line.

```
10 REM==========LINE==================
20 REM DISPLAYS A LOW-RESOLUTION LINE
30 REM POKES TO SCREEN MEMORY USED
40 REM================================
50 POKE 649,1:REM KEYBOARD BUFFER TO 1
60 POKE 53280,0:REM BORDER=BLACK
70 POKE 53281,0:REM SCREEN=BLACK
80 PRINT "[CLR]":REM CLEAR SCREEN
90 CX=39:REM CURSOR AT RIGHT OF SCREEN
100 CY=12:REM CURSOR HALF WAY DOWN
110 REM SHOW START OF LINE AT LEFT OF
120 REM SCREEN AND HALF WAY DOWN
130 POKE 1504,160:REM WHITE SQUARE
1000 REM#=======CURSOR================
1010 REM DISPLAY AND MOVE CURSOR
1020 REM================================
1030 POKE (1024+CX+40*CY),160:REM WHITE
1040 FOR N=1 TO 20:NEXT N:REM DELAY
1050 POKE (1024+CX+40*CY),32:REM BLACK
1060 GET A$:REM GET KEYPRESS
1070 REM IF NO KEY THEN FLASH CURSOR
1080 IF A$="" THEN GOTO 1030
1090 REM IF "RETURN" THEN DRAW LINE
```

```
1100 IF A$=CHR$(13) THEN GOSUB 2000
1110 REM MOVE CURSOR UP?
1120 IF A$="[CU]" THEN CY=CY-1
1130 IF CY<0 THEN CY=0
1140 REM MOVE CURSOR DOWN?
1150 IF A$="[CD]" THEN CY=CY+1
1160 IF CY>24 THEN CY=24
1170 REM MOVE CURSOR LEFT?
1180 IF A$="[CL]" THEN CX=CX-1
1190 IF CX<12 THEN CX=12:REM WHY 12?==?
1200 REM MOVE CURSOR RIGHT?
1210 IF A$="[CR]" THEN CX=CX+1
1220 IF CX>39 THEN CX=39
1230 GOTO 1030
2000 REM#=======DRAW==================
2010 REM CONNECT START TO CURSOR
2020 REM================================
2030 PRINT "[CLR]":REM CLEAR SCREEN
2040 FOR X=0 TO CX:REM STEP FROM LEFT
2050 REM CALCULATE Y FOR EACH X
2060 Y=INT(12.5+X*(CY-12)/CX)
2070 REM SET POINT ON LINE TO WHITE
2080 POKE 1024+X+40*Y,160
2090 NEXT X:REM LOOP TO END OF LINE
2100 RETURN
```

You will notice that, at some angles, the line appears quite uneven because of the poor resolution. (*Note:* Can you see why I have put a lower limit of 12 on the cursor x-position? Try reducing it and see what happens if you move to the bottom of the screen.) Exactly the same thing happens when using high resolution graphics, but the effect is less obvious. The higher the resolution, the more effective computer graphics are. As each pixel (dot) on the screen requires memory to store it, it is obvious that if we wanted, say, twice the number of pixels across the screen and twice the number down the screen, we would need four times the memory to store screen data (2×2).

Many useful techniques can be implemented on microcomputers, however, as described in this book. Allowance must be made for the limitations of the machine being used, of course: the Commodore 64, with its 6510 processor, dedicated sound and video chips, and large on-board memory is particularly well suited to the running of certain types of simulations. Real-time simulations — even when written in BASIC, a notoriously slow-running language — can be developed if the screen display is

kept very simple. Financial and developmental programs of quite a high standard can be implemented.

What is the scope of this book

As a basic introduction, this book gives a guide to the most useful techniques available and the way in which the techniques can be incorporated into practical simulations. Techniques are explained by way of a series of examples, covering a wide range of possible application areas. These examples are kept as simple as possible in order better to demonstrate the techniques used.

It is not possible in this book to describe fully all potential applications, and it is left to you, the reader, to assess your own particular requirement for such techniques. Further books in this series will give full listings of a number of developed simulations for business and personal use.

I draw on my own experience throughout the book and the examples presented in Chapters 6 to 10 are taken from my own case-book.

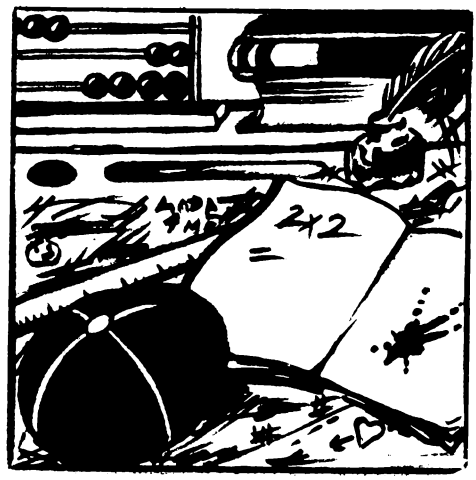
Organisation of the book

The general theories of simulation development and use are presented in Chapters 2, 3 and 4. Chapter 5 indicates the points of practical interest when programming the Commodore 64 to run simulations, and examines the potential of the machine. The remaining chapters examine the particular techniques available for typical applications areas and give example simulations. A glossary provides a reference list of words and phrases which may be unfamiliar to you. A general-purpose spreadsheet program is presented in Appendix A: this is used as the basis for a financial simulation in Chapter 6.

CHAPTER 2

When I was a Boy...

A background to simulations



"...the first computers..."

In this chapter I shall begin to introduce you to the concept of computer simulation. That is, what simulations are, where they came from, and why they are used.

Firstly, however, a short word of warning. Inevitably, there is a fair amount of jargon associated with this particular brand of computer science (you see, it's started already!). I'm not especially fond of jargon for jargon's sake, but some is bound to creep in from time to time — for which you must forgive me. More importantly, I shall be introducing you to a number of new concepts and must include some technical description. Don't worry about either of these, the concepts are based on the application of logic and the technical descriptions only need be remembered if you want to delve deeper into the formal side of simulations.

What are simulations

Although I have just called simulations a branch of computer science, it would be more truthful to say that what I am talking about is the 'art' of computer simulation. I say this because despite attempts being made (successfully!) from the earliest days of computers to run simulations, there is still no widely- accepted formal approach to their creation, little information is shared between users, and no yardstick exists to measure performance. Hence, in strict terms , there is no such thing as 'simulation technology'. Every new simulation is started afresh, and the final success of the program will depend on the inspiration and experience of the simulation designer. In this book I attempt to describe the vital first few steps required for the development of practical and useful simulations.

Already in this chapter I have innocently touched on two subjects which are fundamental to my concept of computer simulation. Firstly, the computer simulations covered by this book are programs. This may seem a little obvious to most of us but it is important to think this way right from the start — I shall explain further presently. The second message is that simulations are run on computers, and that computers are tools.

Computers mean different things to different people. Some people want instant access to stored information; the computer is a memory device. Some people want entertainment; the computer is a video-game machine. Some people want to produce documents; the computer is a word processor. Some people want to manipulate data; the computer is a programmable calculator. And so on and so on. Given sufficient time and imagination you could fill a library of books with descriptions of what computers can do for different people. The point, with all these varied uses, is that the computer is a tool. Versatile and expensive perhaps, when compared to a hammer, but a tool none the less. Computers can be made to do what you want and in the way you want (there are limitations of course). What is needed at first is a clear idea of just what it is that you require. You

can't get anything out of a computer unless you (or somebody else who has sold you a computer package) tell it exactly what to do.

This is where the programming comes in. All the applications (uses) noted above are really just examples of different computer programs. In exactly the same way, there are a host of different types of computer simulation but all are just computer programs. The techniques of computer simulation are used to develop such programs. By applying these techniques, some of which are more straightforward than others, practical programs can be produced which satisfy the criteria of doing what is required in the way that is wanted.

But what is a computer simulation? — what separates it from all the other computer programs? I shall give a basic description and then go on to explain the important points. A computer simulation is a program which duplicates or copies some of the features of something else (a system — the first bit of terminology to get to grips with) and predicts what will happen given various possible occurrences in a manner which is acceptable to the user. You can see now why I avoided attempting to define computer simulations in the Introduction — the definition is so vague as to be almost useless without further explanation.

I can give you a practical example of a program which may at first sight appear to be working as a simulation. The intention is to use the first of the Commodore 64 sound channels (voice 1) to produce computer-generated music; to simulate the workings of a composer. RND(0) is used to produce a series of sounds.

```
10 REM==========MUSIC================
20 REM PRODUCES RANDOM TUNE EFFECTS
30 REM==============================
40 PRINT "[CLR]PRESS S TO STOP"
50 GET K$:IF K$="S" THEN GOSUB 2000:END
60 GOSUB 4000:REM SET UP
70 GOSUB 1000:REM SOUND ON
80 GOSUB 3000:REM DELAY
90 GOSUB 2000:REM SOUND OFF
100 GOSUB 3000:REM DELAY
110 GOTO 50:REM LOOP
1000 REM#========SOUND-ON============
1010 REM SET REGISTERS FOR VOICE1
1020 REM==============================
1030 FH=INT(FZ/256):REM HIGH BYTE OF FZ
1040 FL=FZ-256*FH:REM LOW BYTE OF FZ
1050 POKE 54272,FL
1060 POKE 54273,FH
1070 PH=INT(PW/256):REM HIGH BYTE OF PW
```

```
1080 PL=PW-256*PH:REM LOW BYTE OF PW
1090 POKE 54274,PL
1100 POKE 54275,PH
1110 POKE 54277,16*AT+DE
1120 POKE 54278,16*SU+RE
1130 POKE 54296,PA
1140 POKE 54276,FC
1150 RETURN
2000 REM#==========SOUND-OFF==========
2010 REM DISENABLE VOICE1
2020 REM================================
2030 POKE 54276,0
2040 RETURN
3000 REM#==========DELAY=============
3010 REM DELAY LOOP
3020 REM================================
3030 FOR N=1 TO TS:NEXT N
3040 RETURN
4000 REM#==========SET-UP============
4010 REM SET VARIABLES
4020 REM================================
4030 FZ=1000+2000*RND(0):REM FREQUENCY
4040 PW=4095*RND(0):REM PULSE WIDTH
4050 AT=2:REM ATTACK
4060 DE=2:REM DECAY
4070 SU=15:REM SUSTAIN
4080 RE=2:REM RELEASE
4090 PA=15:REM PEAK AMPLITUDE
4100 FC=65:REM WAVEFORM (PULSE)
4110 TS=10+40*RND(0):REM DELAY LENGTH
4120 RETURN
```

The result does not sound like music. We need to define the difference between 'noise' and 'music'. Rhythm is a quality intrinsic to most types of music. Changing line 4110 to:

```
4110 TS=40:REM DELAY LENGTH
```

deleting line 100, and running improves our noise by adding rhythm, but there is still something missing!

Typing in the following lines:

```
45  K=1
55  K=K+1
```

```
56 IF K=8 THEN GOSUB 6000:REM SET SEED
6000 REM#==========SET-SEED==========
6010 REM SET RANDOM SEED
6020 REM==============================
6030 K=0
6040 FOR N=139 TO 143
6050 POKE N,1
6060 NEXT N
6070 RETURN
```

and changing lines 4030 and 4040 to:

```
4030 FZ=1000+2000*RND(1):REM FREQUENCY
4040 PW=4095*RND(1):REM PULSE WIDTH
```

gives us repetition and rhythm but is now rather boring. Now change line 4030 again to:

```
4030 FZ=1000+200*A*RND(1):REM FREQUENCY
```

and add:

```
6065 A=A+10:IF A>30 THEN A=0
```

Now we have rhythm, repetition, and the basis for a simple tune. The tune may not appeal to you very much (try changing some of the variables to get different effects) but it's a whole lot better than the random noises we started with!

However, should the program we have created be called a simulation of a composer? We are simply using a modified set of random responses, and no composer worthy of his name would call it a rival. It would perhaps be more accurate to call the program a sound generator.

Additionally, and more importantly, we have not even attempted to duplicate the process of composing; we have merely taken advantage of some of the built-in features of the computer. If we played around with the program long enough we might end up with something which was not too objectionable. We could not, however, credit the program with having composed a masterpiece, and so the program is not a simulation of a composer.

Systems thinking

In order better to define simulations, let me introduce the concept of systems thinking. Calling something a 'system' is a technique adopted by

analysts (people who analyse things for one reason or another — systems analysts, for example, specialise in computer systems and optimise such systems in terms of hardware and software to best meet particular requirements) to contain and describe the thing which they are analysing. For example, a car designer may think of a whole car as a system. Describing it thus provides a guide as to what is contained within the system (the car) and what is not contained within the system (the road, other vehicles, the local bank, etc.). There are two important qualities of a system: the system itself, and the interface of the system (the way in which the system interacts and communicates) with the non-system. In the case of a car these two features are both highly complex. The car itself consists of thousands of moving and non-moving parts, all the result of many years research and development. The interface between the car and the outside world is multi-faceted (don't bother remembering that one — I don't intend using it again!), that is there are actually several interfaces: the car with the undulations on the road, the car with the air that it is moving through, the driver's eyes and the road ahead and the constant pushing and pulling of various control levers, and so on.

Now let's extend the idea of systems a little further. Think of the specialist engineer, working as part of a design team engaged in the development of new cars (our previous system), who may be responsible for just a single sub-system of the car. A sub-system, in this context, is itself a system (say one wheel and its suspension) but is recognised as part of the whole system (the car). This sub-system can be described conceptually in the same way as the major system. That is, the sub-system has an interface (relative movement of the car and the road) and it has qualities (spring rate, damping, mass of moving parts, dimensions etc...) which define the way the sub-system reacts to the things around it. Similarly, a part of the suspension can be described as a sub-system in its own right. Thus we have levels of systems and sub-systems which fully define the car design and are useful in different ways. At the more detailed levels, specialist engineers can consider individual components or functional groups of components. These can be designed, tested, modified and developed almost in isolation from each other (careful here; obviously sub-systems must be compatible with each other — this compatibility is achieved through careful definition of the sub-system interfaces and by placing constraints on the system qualities, such as size and weight).

Now comes the pay-off for thinking in terms of systems and sub-systems. Each system can be simulated, once properly defined on a computer. The sub-systems can be simulated and these simulations combined together to give a complete simulation of the whole thing. A car simulation can thus be developed which will allow the designer to test ideas and optimise designs. Using a computer simulation makes sense economically. The cost of hardware development for something as

complex as a car is very high indeed whereas computer simulations are relatively cheap. Computer simulations also make sense from the point of view of time expended. Several ideas can be tested in one day, using a simulation, which would take several months without it. Nearly all car designers and manufacturers make extensive use of simulations. Let's not get carried away, though, the simulations can only be used as design aids (tools for the use of the designer) and are only as good as the programming. There are many areas of technology used in car manufacture which are not easy to incorporate into a simulation, and in many cases the best solution from a technical point of view is not acceptable for safety or aesthetic reasons. The simulation is only useful up to a point.

With a simulation available, the coarse-level decisions about a car design, such as engine type and size, suspension type, passenger seating, and so on, can be evaluated and practicable concepts identified at a very early stage. Further work, from the basic concepts, at a more detailed level of component and component-group development will use sub-system simulations to indicate the most appropriate hardware designs. Thus, a new car can rapidly be outlined and defined, to quite a detailed level, in terms of hardware design and performance — all on a computer.

So now I have outlined the basic concept of 'systems thinking' and its relevance to computer simulations. Before we move on, I want to give you something else to think about on the subject of systems. This is an extension to the use of the technique, which many people find difficult to understand fully at first. Don't worry too much if you have difficulty with the next paragraph — most practical simulations can be fully understood by the application of common sense, and the full details of 'systems thinking' are not required. If, however, you intend to spend much time on simulations involving people or institutions then read on most carefully.

Anything can be described as a system, not just items of hardware as described above. Thus, for example, a company board of executives can be described as a system, as can a classroom full of kids. Anything which can be defined as a system can be used as the subject of a computer simulation. The success of a computer program in correctly simulating a system is dependent principally on how well the qualities and interfaces of the system are understood and described. Further, the characteristic which ultimately defines a system, as separate from other possible systems, is the description of the limits to the system. Thus the system of the car wheel plus suspension is actually defined by the functional qualities of the suspension; anything not contributing to the function of suspending the wheel from the car (for example, the wheel arch) is not, strictly, part of the system. The system limits may be functional, spatial, temporal, imaginary, or some combination of these and others. A space-based video game is a simulation of a system which is totally imaginary!

The user

But let's get back to the simulation business. We can now re-state the description of computer simulations thus: a computer simulation is a computer program which duplicates the workings of a system and predicts the responses of that system to external and internal events, giving output in a manner acceptable to the user.

OK. So now we can cope with the system bit, and the internal and external 'events' are just things which happen inside the system or outside the system, but what's this business about the output? In fact, this is one of the things which separates good simulations from bad simulations. It is vital that the simulation provides the information required of it in the most suitable form. Remember that the simulation is supposed to be a tool: that means that, even though it may be a masterpiece of programming, it will be useless if it cannot do what the user wants. A user is the person (or group of people) who requires to use the simulation. This is not quite the same thing as the computer user. Usually, the author of the simulation is not the user of the simulation but is probably the user of the computer, or, in the case of commercial software, the author may be writing for unknown users. This is of importance because the workings of the simulation will often have to be designed around the output requirements of the user and these are often not well defined at the start of program design. This makes things that much tougher for the programmer, who must chose a balance between the versatility of the simulation in meeting possible future requirements and the complexity, and hence cost, of the program.

Thus, simulations are designed and developed to do a specific job and in a specific way. They are in practical terms very limited copies of the systems which they represent, often only representing a single characteristic of the system. Simulations can be described as 'functional' representations of systems. That is, they copy some feature of the functioning of the system. For example, when we looked at the system consisting of a car suspension and wheel, there are a number of possible simulations. One of these may indicate the dimensions of alternative designs, whilst another will simply indicate how the forces from the suspension are transmitted to the car, and yet another will investigate in detail the forces within the components of the suspension itself.

Although we now have a description of computer simulations, I will not attempt to develop this into a formal definition. I do not believe that it is possible rigidly to define computer simulations as separate from other computer programs. Many programs not usually thought of as simulations use the same techniques, and can, in certain situations, be used as the basis for simulations. Later in this book I give an example based on the use of the a spreadsheet program (Chapter 6).

Early hardware and simulations

The first computers were designed as number-crunchers. They performed relatively simple (by today's standards anyway), but highly repetitive calculations. Typical of the first applications were the generation of mathematical functions and tables, and the calculation of shell trajectories for artillery during the Second World War.

The UNIVAC 1 was the first commercially available electronic digital computer (UNIVAC was purchased by IBM in 1949 — a very astute business move indeed!) but it was not really until 1965 that commercial computers really became big business, with the introduction of the IBM 360 series.

Many more mainframe computers are bought for business than for laboratory or scientific use. Business computers are used, in the main, to process a great volume of data whereas scientific computers tend to be used to process less raw data but a great many calculations are performed per datum. It is the latter set of program applications which tend to include simulations. The upshot of all this is that the simulation business is still a young business. In practical terms, although much has been talked of the more glamorous simulations, the majority of simulations are under 10 years old.

As is to be expected, the first simulations were rather rudimentary and simplistic. This was for three main reasons. Firstly, the computers available before the mid-1970s were restricted in terms of speed and memory availability. Secondly, simulation techniques and the approach required to develop satisfactory simulations took time to evolve. Thirdly, simulations need data and this data was often simply not available.

An example of this last point is the aircraft industry. It is possible to simulate the aerodynamics (the interaction of a body with the air it is passing through and the resultant forces produced on the body) of quite complex designs. Such simulations use equations and formulae which have been available, at least in principle, since the middle of the century. When computers of sufficient power to run such simulations became available to the aircraft industry there was no immediate rush to transfer research funds into the development of such simulations. Basic aerodynamic research was at that time very much an empirical discipline. Knowledge of the aerodynamic qualities of particular shapes of body was built up through the testing of models, full-size test rigs, and production hardware in wind tunnels, water channels, and in flight tests. Combinations of shapes (eg wings plus aircraft body) again were tested in model form to obtain basic aerodynamic data. The data thus obtained was not directly usable in simulations — it was never intended for such a use. The development of simulations as useful tools for research and development was a long, slow, and expensive exercise. The traditional methods had to be adapted slowly to provide data and to verify the accuracy of the simu-

lations as they started to produce results. These days, however, no self-respecting aircraft/missile/car designer is without a sophisticated suite of aerodynamic simulations.

An interesting example of a simple but very effective simulation from the seventies is the well-known game called 'LIFE'. Invented at Cambridge University in 1970, this game simulates successive generations of a simple life-form as it flourishes or dies, depending on a few basic rules governing the life or death of individuals. In its simplest form, each member of a 'community' is represented as a letter O (or an asterisk or anything else will do) set on a grid pattern. A member will survive if it has two or three neighbours but die in any other circumstance (either of over-population or under-population). A new member is born in any empty space which has three neighbours. These rules are applied to the full grid area (typically a VDU screen or teletype page) and repeated to represent succeeding generations. The simulation is based on very simple rules but does simulate some of the important aspects of life. The interest in the game comes from attempting to set up an initial population which will eventually survive in some stable form. This game, along with the original Adventure game, was rumoured to be so popular with computer programmers at one stage that many companies banned it from their computers.

Listed below is a version of 'LIFE' which I have produced to run on the Commodore 64. A 15 by 15 grid is displayed, with a population initially generated randomly. Each grid location is represented in memory by an array (P%( ) or Q%( )) where P%(X) = 1 if a being is alive at location X (or zero otherwise). By using a one-dimensional array to represent a two-dimensional grid we avoid possible 'edge effects'. Cells which are shown at the edge of the grid area on the screen are treated by the program as if they are in fact joined face on to the opposite face (much as the patterns in wallpaper repeat themselves). You will see some patterns displayed which 'bridge' from one edge to another.

```
10 REM==========LIFE==============
20 REM 15 BY 15 VERSION OF LIFE
30 REM================================
40 GOSUB 3000:REM SET UP RANDOM POPLN.
1000 REM#---SET UP LOOP1-Q%( ) TO P%( )--
1010 J%(224)=Q%(223)+Q%(224)+Q%(0)
1020 D%=210:E%=15
1030 REM LOOP1111111111111111111111111
1040 N%=J%(D%)+J%(X%)+J%(E%)-Q%(X%)
1050 P%(X%)=0
1060 IF N%=2 THEN P%(X%)=Q%(X%)
1070 IF N%=3 THEN P%(X%)=1
1080 POKE 1236+40*INT(X%/15)+X%-15*INT(X
```

```
%/15),160+10*P%(X%)
1090 POKE 1236+40*INT(E%/15)+E%-15*INT(E
%/15),32
1100 I%(B%)=P%(A%)+P%(B%)+P%(X%)
1110 A%=B%:B%=X%:X%=C%
1120 C%=C%+1:IF C%=225 THEN C%=0
1130 IF X%=0 THEN GOTO 2010
1140 D%=D%+1:IF D%=225 THEN D%=0
1150 E%=E%+1:IF E%=225 THEN E%=0
1160 GOTO 1040:REM END LOOP111111111111
2000 REM#---SET UP LOOP2-P%() TO Q%()--
2010 I%(224)=P%(223)+P%(224)+P%(0)
2020 D%=210:E%=15
2030 REM LOOP22222222222222222222222222
2040 N%=I%(D%)+I%(X%)+I%(E%)-P%(X%)
2050 Q%(X%)=0
2060 IF N%=2 THEN Q%(X%)=P%(X%)
2070 IF N%=3 THEN Q%(X%)=1
2080 POKE 1236+40*INT(X%/15)+X%-15*INT(X
%/15),160+10*Q%(X%)
2090 POKE 1236+40*INT(E%/15)+E%-15*INT(E
%/15),32
2100 J%(B%)=Q%(A%)+Q%(B%)+Q%(X%)
2110 A%=B%:B%=X%:X%=C%
2120 C%=C%+1:IF C%=225 THEN C%=0
2130 IF X%=0 THEN GOTO 1010
2140 D%=D%+1:IF D%=225 THEN D%=0
2150 E%=E%+1:IF E%=225 THEN E%=0
2160 GOTO 2040:REM END LOOP222222222222
3000 REM#=======RANDOM================
3010 REM SET UP RANDOM POPULATION
3020 REM================================
3030 DIM P%(224),Q%(224),I%(224),J%(224)
3040 PRINT "[CLR]"
3050 Q%(0)=INT(1.5*RND(0))
3060 POKE 1236,160+10*Q%(0)
3070 Q%(1)=INT(1.5*RND(0))
3080 POKE 1237,160+10*Q%(1)
3090 FOR X=2 TO 224
3100 Q%(X)=INT(1.5*RND(0))
3110 POKE 1236+40*INT(X/15)+X-15*INT(X/1
5),160+10*Q%(X)
3120 J%(X-1)=Q%(X-2)+Q%(X-1)+Q%(X)
3130 NEXT X
```

```
3140 J%(0)=Q%(224)+Q%(0)+Q%(1)
3150 A%=223:B%=224:C%=1:X%=0
3160 RETURN
```

You may like to modify the set-up routine to allow the user to choose the initial population, which would be closer to the original conception of the simulation.

The program, which is not too easy to understand at first sight, consists of two virtually identical loops and a set-up routine. LOOP1 sets up a new population of beings and records the presence of each being in the array P%( ). This new population is displayed as it is set up by POKEing characters directly to screen. The variables A%, B% and X% indicate the last-but-one, the last, and the current location in P%( ). These are used in line 1100 to calculate I%( ) which is simply the sum of beings at and to the left and right of a given position. Adding I%( ) for a particular location and for locations directly above and below will indicate the number of beings neighbouring that location (variables D% and E% give locations above and below the current location X%).

LOOP2 does the same job as LOOP1 but P%( ) is replaced by Q%( ) and I%( ) is replaced by J%( ).

I have used integer variables where I can throughout the program to try and speed things up a bit and have only used a few REMs for the same reason.

Operational analysis

Operational analysis is the study of the operations of a system. Often the systems analysed are financial or commercial systems although the more famous examples include traffic flow on roads and around airports, critical-path analysis, and time and motion studies. This has been the happy hunting-ground for simulation engineers for a number of years.

A well-known development of the late sixties and early seventies much used in operational analysis is the 'Monte Carlo simulation'. This is in fact one of the basic techniques used in the simulation business (see Chapter 10) which was rapidly and successfully implemented. The Monte Carlo process really came into its own with the arrival of high-speed digital computers. The technique is used to predict the final results of a series of possible happenings. For example, if you wanted to find out how long it might take to write a computer program, you might choose to simulate the process using a Monte Carlo model. Suppose the program will consist of ten subroutines, which must be written and tested in sequence, each of which will take a minimum of one hour to write. Also, having written a subroutine there will only be a 10% chance of the subroutine working as intended first time. There is a 50% chance that the subroutine will contain

bugs which will take a further hour to sort out and a 40% chance that the subroutine will have bugs which will take two hours to find and eradicate. How long will the program take to complete? We know that the 10 subroutines will take a minimum of 10 hours to write. Debugging time will vary from nothing to a maximum of 20 hours. Thus we can say that the program will take from 10 to 30 hours to complete. But this is no good for planning purposes, we need to know the average time as well as the possible extremes.

A Monte Carlo simulation considers each of the activities (in this case the writing of a subroutine is an activity) and selects a particular outcome at random but determined according to the in-built probabilities (10, 50, or 40%). Thus, for each subroutine a particular writing and debugging time is determined at random and a running total is built up of the overall programming time. At the end of the simulation you are given a single example of the time taken to write a 10-subroutine program. Unfortunately you only have a single example so far. The simulation must be run many times before you can be reasonably sure that you have sufficient data, in the form of a whole series of simulated programming tasks, to derive an average time. The average time taken is found by adding all the results and dividing by the number of results. If you are particularly keen to demonstrate this for yourself then you can write a simple simulation as described above and run it a few times. There are a number of ways that such a simulation can be written and I include below a listing of one of the most obvious.

```
10 REM=========MONTE CARLO=============
20 REM SIMPLE MONTE CARLO SIMULATION
30 REM================================
40 REM SET UP BASIC PROBABILITIES OF
50 REM 3 POSSIBLE EVENTS
60 P1=0.1:P2=0.5:P3=0.4
70 TM=0:REM INITIAL TIME SET ZERO
80 REM LOOP FOR TEN "SUBROUTINES"
90 FOR N=1 TO 10
100 REM SET RANDOM PROBABILITY RP
110 REM THIS IS USED TO DECIDE WHICH
120 REM OF THREE POSSIBLE EVENTS OCCURS
130 RP=RND(0)
140 REM SELECT EVENT EV USING RP
150 IF RP<=P1 THEN EV=1
160 IF RP>P1 AND RP<=(P1+P2) THEN EV=2
170 IF RP>(P1+P2) THEN EV=3
180 REM NOW SET TIME DT TAKEN BY EVENT
190 IF EV=1 THEN DT=1
```

```
200 IF EV=2 THEN DT=2
210 IF EV=3 THEN DT=3
220 REM NOW UPDATE TOTAL TIME
230 TM=TM+DT
240 NEXT N:REM NEXT "SUBROUTINE"
250 PRINT "PROGRAM TAKES";TM;
260 PRINT "HOURS TO WRITE"
```

Running this program around 20 times is sufficient to give a result within one hour of the true average time (around 23 hours). You will notice a few other points here which explain why Monte Carlo simulations were quick to appear, what their benefits are, and what their drawbacks are. They are simple to create: the simulation follows very closely the actions of the system being simulated. But a large number of runs may be required before a reliable average result is obtained, although this may not be important. After a number of runs have been made additional data is available, such as the range of resultant times in the example above, which can be very difficult to obtain any other way. Of particular note is the value of Monte Carlo techniques to real-time simulations because of the close representation of the actual workings of the simulated system.

Obviously, the Monte Carlo process requires a large number of calculations to be made to complete a single cycle of the simulation. This is where computers came to the aid of a process already in use as the basis for many non-computer simulations and really made it into a feasible method for widespread utilisation (I'm sorry, I'm beginning to slip back into jargon — computerspeak you might say!).

Industrial simulations

Another early application area for the use of simulations came from the engineering industries. I have already talked of the car and aircraft industries. These two examples are perhaps the most widely quoted, but the general impact of computers on engineering as a whole has been very pronounced. Many of the basic tools of the engineer are built around the laborious repetition of various calculations. The desk calculator and, more recently, the computer have changed the way in which engineers work.

In order to design a product (whether a bridge or a typewriter or anything else) an engineer will always simulate the finished article. Until the advent of the computer such simulations consisted of drawings and hand calculations. These were, in principle at least, readily transferable to computer. In practice, of course, things are a little different. Simulations are certainly big business in the engineering industry but a lot of diagrams are still drawn and desk calculators are well used. Simulations for today's engineer take the design and development process into a completely new

area. One of the truisms of computing is that computers do not replace people, but they do change the way that people do things. Ten years ago it would not have been practical to use finite-element techniques to simulate the time-transient behaviour of an off-shore oil rig in a force 10 gale. Today it is considered by many to be dangerous not to. In the main, the simulations developed for and used by engineers tend to be complex and large.

The great growth of simulations has really occurred in the last ten years. Early work tended to be done in universities, in some of the larger engineering companies, and in support of various studies by operational analysts. As 'computer literate' graduates started to appear and computers were made available to many more companies, the use of simulations became quite common. Recently, serious financial simulations have started to appear (other than clandestine programs produced by some governments!), expert systems are available commercially, video games are becoming more and more sophisticated, and educational software is not far behind.

CHAPTER 3

But What Does This Little Black Button Do?

What simulations are and how they are used



The basic types of simulation

There are three basic functions for simulations — emulation, analysis, and prediction. In the following sections I describe each of these, the uses to which they are put, and some of the more important approaches adopted in developing simulations.

Emulation simulations

Emulations are direct copies of real-life systems and are used either as low-cost substitutes for use of the real thing or to represent situations which would be impracticable to achieve using real hardware. Examples of emulators include flight simulators, video games, war games, and some engineering simulations.

The purpose of the emulation is to represent as realistically as possible the time-dependent workings of the simulated system. It is this time-dependency which is important, and programs are written around the requirement to simulate the state or condition of the system at regular intervals. The time taken between successive time intervals is known as the simulation time-step. This time-step is not the time taken to perform one loop of calculations but the time represented in the real system by the calculations. In many situations the complexity of the simulation will lead to a computational time far in excess of the time-step. For example, a simulation of the flight of a rocket may have a time-step of, say, a tenth of a second whereas the computing time for each time-step may be several seconds. The time-step chosen for a simulation of the tidal effects in a river estuary, for example the River Thames in London upstream of the flood barrier, may be of the order of 15 minutes but the computing time may be from under a second to a few minutes depending on the complexity of the representation. In special circumstances it may be desired that the simulation time-step and the computational time should be the same. Such simulations are real-time simulations and are intended for direct interaction with a human operator.

Usually the sophistication of a real-time simulation is limited by the time available for computing. If a long time-step is chosen, then the response of the simulator to the actions of the operator will be slow. This may not be important if you are engaged in the education of nuclear power station operators where time-steps of the order of one second would be more than adequate. A space-invader game must respond very much more quickly, however.

Obviously, all the computations for a single time-step must be completed before the end of the time-step. All inputs must be made, responses of the simulated system to the inputs must be estimated, outputs must be recorded, and the simulation set up for the next time-step. Very often, simply writing down the list of operations to be performed for each time-

step will provide a structure for the program itself. Much more care is required for real-time simulations, however, particularly with the timing of inputs and outputs.

I have an example program here which is not, to my way of thinking, a true simulation (it has no elements of the unknown and does not interact with the user) but does nicely demonstrate some of the characteristics of a real-time emulation program. The program displays enlarged versions of the Commodore 64 character set to produce a digital clock. Hours and minutes are displayed and a sprite is used to show a 'second hand' moving back and forth every second.

Note that there are two delays built into the program, at lines 250 and 410, which ensure that the second hand sweeps in one second and that the clock keeps good time. A complication here is that any changes to the program, such as inserting additional program lines or removing remark lines, changes the computing time and adjustments are required to the delays.

```
10 REM========REAL-TIME==============
20 REM DIGITAL CLOCK
30 REM SHOWS TIME IN HOURS AND MINS.
40 REM USES SPRITE #0 AS SECOND HAND
50 REM================================
60 GOSUB 6000:REM COPY ROM CHARACTERS
70 DIM S0%(63):REM USED FOR SPRITE #0
80 H=12:M=15:REM START TIME (HRS MIN)
90 GOSUB 5000:REM INITIALISE CLOCK
100 GOTO 320:REM DISPLAY TIME
110 REM DISPLAY SECOND HAND ----------
120 REM LOOP FOR 1 MIN.
130 POKE 53269,1:REM SPRITE #0 ON
140 FOR S=1 TO 48
150 X=114:REM SET SPRITE START POS'N
160 REM LOOP FOR 1/2 SEC.
170 FOR N=1 TO 10
180 X=X+11:GOSUB 1000:REM SPRITE RIGHT
190 NEXT N
200 REM LOOP FOR 1/2 SEC.
210 FOR N=1 TO 10
220 X=X-11:GOSUB 1000:REM SPRITE LEFT
230 NEXT N
240 REM ADJUST TO TIME TO 1 SEC.
250 FOR N=1 TO 39:NEXT N
260 NEXT S
270 POKE 53269,0:REM SPRITE #0 OFF
```

```
280 REM UPDATE TIME BY 1 MIN. ---------
290 M=M+1
300 IF M=60 THEN M=0:H=H+1
310 IF H=24 THEN H=0
320 REM DISPLAY NEW TIME --------------
330 PRINT "[CLR][YEL][CD][CD][CD][CD]";
340 A$=CHR$(A+INT(H/10)):GOSUB 2000
350 A$=CHR$(A+H-10*INT(H/10)):GOSUB 2000
360 A$=":":GOSUB 2000
370 A$=CHR$(A+INT(M/10)):GOSUB 2000
380 A$=CHR$(A+M-10*INT(M/10)):GOSUB 2000
390 REM DELAY BEFORE STARTING NEXT LOOP
400 REM TO GIVE CORRECT TIMING
410 FOR N=1 TO 400:NEXT N
420 GOTO 110:REM REPEAT ---------------
1000 REM#========SPRITE================
1010 REM MOVE SPRITE (SECOND HAND)
1020 REM================================
1030 XH=INT(X/256):XL=(X-256*XH)
1040 POKE 53248,XL:POKE 53264,XH
1050 RETURN
2000 REM#======BIGCHR==================
2010 REM DISPLAY LARGE CHARACTER (C)
2020 REM AND RESET CURSOR FOR NEXT
2030 REM================================
2040 PRINT A$;"[CL]";:REM PRINT CHR
2050 REM GET CHARACTER CODE FROM SCREEN
2060 C=PEEK(256*PEEK(210)+PEEK(209)+PEEK
(211))
2070 PRINT " [CL]";
2080 REM LOOP FOR 8 LINES OF CHARACTER
2090 FOR BN=0 TO 7
2100 GOSUB 4000:REM GET BYTE CB
2110 REM LOOP FOR 8 BITS OF BYTE CB
2120 FOR N=0 TO 7
2130 GOSUB 3000:REM NEXT BIT OF CB
2140 REM PRINT SPACE OR CHARACTER
2150 PRINT CHR$((1-BT)*32+BT*ASC(A$));
2160 NEXT N
2170 PRINT B$;:REM NEXT ROW OF CHR
2180 NEXT BN
2190 PRINT U$;:REM NEXT CHARACTER
2200 RETURN
3000 REM#======CHRBIT=================
```

```
3010 REM GET NEXT BIT OF CB
3020 REM================================
3030 BT=0
3040 IF CB<128 THEN GOTO 3060
3050 BT=1:CB=CB-128
3060 CB=CB*2
3070 RETURN
4000 REM#======CHRBYTE================
4010 REM GET BYTE BN OF CHARACTER C
4020 REM================================
4030 CB=PEEK (12288+8*C+BN)
4040 RETURN
5000 REM#======SET-UP==================
5010 REM INITIALISATION
5020 REM================================
5030 A=48:REM ASCII CODE=A+NUMBER
5040 B$="[CL][CL][CL][CL][CL][CL][CL][CL
][CD]":REM CURSOR CONTROL
5050 U$="[CR][CR][CR][CR][CR][CR][CR][CR
][CU][CU][CU][CU][CU][CU][CU][CU]":REM
DITTO
5060 REM SET-UP SPRITE FOR SECOND HAND-
5070 REM MAKE SURE S0% ARRAY IS CLEAR
5080 FOR N=1 TO 63:S0%(N)=0:NEXT N
5090 REM DATA FOR SECOND HAND
5100 FOR N=2 TO 62 STEP 3
5110 S0%(N)=24:REM BITS 3 & 4 SET
5120 NEXT N
5130 REM POKE SPRITE #0 SHAPE DATA
5140 REM (USE CASSETTE BUFFER)
5150 FOR N=1 TO 63
5160 POKE 831+N,S0%(N)
5170 NEXT N
5180 POKE 2040,13:REM POINTS TO DATA
5190 REM SPRITE X,Y POS'N (OFF SCREEN)
5200 POKE 53248,0:REM X LOW-BYTE
5210 POKE 53264,0:REM X HIGH-BYTE
5220 POKE 53249,170:REM Y POS'N FIXED
5230 POKE 53271,1:REM DOUBLE HEIGHT
5240 POKE 53287,1:REM SPRITE WHITE
5250 RETURN
6000 REM#======SETCHAR================
6010 REM MOVES 64 CHARACTERS FROM ROM
6020 REM TO RAM AT 12288
```

```
6030 REM SEE P.110 CBM PROGRAMMERS
6040 REM REFERENCE GUIDE
6050 REM================================
6060 REM FIRST SET POINTERS TO RESERVE
6070 REM MEMORY FOR CHARACTERS
6080 POKE 52,48:POKE 56,48
6090 REM KEYSCAN INTERRUPT TIMER OFF
6100 POKE 56334,(PEEK(56334) AND 254)
6110 REM SWITCH ROM INTO 6510 MEMORY
6120 POKE 1,(PEEK(1) AND 251)
6130 REM LOOP FOR 64 CHRS. (=512 BYTES)
6140 FOR I=0 TO 511
6150 POKE I+12288,PEEK(I+53248)
6160 NEXT I:REM LOOP END
6170 REM RETURN SYSTEM TO NORMAL
6180 POKE 1,(PEEK(1) OR 4)
6190 POKE 56334,(PEEK(56334) OR 1)
6200 RETURN
```

Analytical simulations

The second type of simulation is that used for system analysis. Examples include financial models, expert systems, management information systems, computer-aided design packages, and reliability assessment programs. Note here that not all the applications noted above need be simulations, but simulations may well be involved one way or another. It is rather dangerous to generalise about this group of simulations; they are designed for a specific job and their program structures are very varied. Some simulations are similar to the emulators described above but adapted for a different use and modified accordingly. For example, a financial package may be based on the emulation of the Stock Market, representing the buying and selling processes, but modified to draw attention to particular points of interest, perhaps to highlight shares which are likely to change rapidly in value. Rather than simply performing a time-step and dumping a full set of data, additional calculations will be performed on the latest set of output to analyse for points of particular interest. Also, comparative runs with different starting assumptions and different input data may be used to indicate not just the absolute response of the system to varying inputs but also to investigate the system sensitivity to inputs. This is the 'rate' at which the system response varies with a changing input. Real-time emulators are not usually used directly for analysis but the results of a number of runs of such a program may be recorded for future analysis.

Reliability assessment and critical-path analysis are worth a bit of further consideration here. These are good examples of applications for

programs which take input in the form of a number of very similar data items (all related in some way), and manipulate the data in such a way that high-level information is output. In the case of reliability assessment, raw data on the number and type of breakdowns of a product is combined and processed to give an assessment of the overall reliability of the product, or to identify particular problem areas, or to identify how the reliability has been changing during development. Critical-path analysis takes a planner's estimates for the duration and requirements of a number of tasks and combines these to indicate the overall project timing and resource requirements. In addition, the set of tasks which are most important to the completion of the project (the critical path of tasks) are shown and the program may indicate the best way to sequence tasks to minimise risks and to make the best use of available resources. Large projects, such as building works or product development programmes, rely heavily on careful planning of the individual tasks. Raw materials must be ordered in advance, a stable workforce must be maintained and provided for, use of special equipment (such as expensive test equipment) must be planned in advance, and so on. The difficulty of doing all this planning by hand and being able rapidly to update such plans if required leads to a high demand for programs which can automatically perform the necessary analyses.

Predictors

Finally we come to simulations designed to make predictions. Included in the list are weather predictors, profit and loss predictors, reliability-growth programs, and crisis forecasters. Yet again, these are often based on emulation programs but are further developed in two important areas. Firstly, these simulations take a starting situation (typically representing time now) and simulate a number of time-steps into the future. Stored data may be used to provide inputs as the program runs. The second important feature of predictors is that they are usually developed to be able to make use of poor data. This is data which is not well defined, may be non-existent, or is known to be unreliable. As simulations predict further into the future, or into other areas where knowledge is lacking, it is important that the simulation give some indication of the confidence which can be placed in the validity of its output. By making special provision for the use of data which is known to be poor, estimates can be automatically made of the reliability of the output. Emulations without the added sophistications noted above can be used to predict future events but must be used with great caution.

A second major form of predictor is that which is based on the analysis of trends in past data to predict forwards. Horse race and football predictors take data in the form of recent results, select those results of particularly relevant meetings and use these to indicate the current performance of

the competitors involved. Armed with this indicator and data on where the event is to be held, the program predicts the likely outcome of the match, usually along with an indication of the probability of a surprise result.

Another example of a sophisticated trend-analysis predictor comes from the financial world. Knowledge of the variation in value of stocks, shares, commodities, etc., available on the world financial markets is vital for investors and up-to-date knowledge can be the making of fortunes. Many individuals and groups maintain predictors for various uses and of various types. The most obvious are trend analysers, working on past data and predicting forward days or even months. Other predictors remove the time-dependency by relating one factor, for example the value of a particular commodity (eg gold or oil), to another, for example some measure of overall level of trade. In times of recession or low levels of trading on certain markets, such predictors can be useful in indicating the value of moving invested funds from one market to another.

Use and abuse of simulations

In the preceding paragraphs I have given a brief introduction to the simulation business, its history, and some of the major approaches adopted. I may have made it sound as if simulations can be used for almost any purpose. In a simplistic sense this is in fact true — you can if you wish simulate almost anything.

However, there are many reasons why you shouldn't bother. Many problems are more easily solved without going to the expense of creating a special simulation program. The example I gave of the simulation of writing a computer program is a case in point. That particular problem can be easily solved with a hand calculation, so why bother with a simulation that requires to be run at least 20 times? Often, the simulator required to mimic a given system would be too complex or difficult to verify (check the correct operation of) and any results obtained would be suspect. This does not mean that simulators should not be considered, it just means that other means of obtaining the desired results should be investigated as well. Weather prediction programs fall into this category at present — there is simply not enough data and knowledge available at present to make a very large-scale simulation which produces acceptable long-term results, but work goes on to develop just such simulations. It must also be remembered that simulations are simplified representations of systems and thus cannot fully replace the real thing. Indeed, most simulators are very approximate representations of systems which themselves are not fully understood and thus simulators can only be used to give a rough idea of the system behaviour.

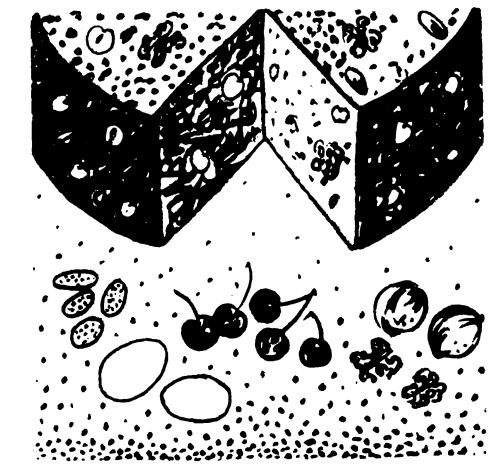
I want to make two more important points here. Firstly, as for most computer programs, a simulation is only as good as its programmer and is

only as useful as its initial design constraints. Secondly, the major limitation on most simulations is the basic understanding of the system to be simulated. I don't want to say more than this at present, but please remember what I said at the beginning of this chapter, that simulations are tools. Tools can be used or abused but if used properly they can be of great benefit.

CHAPTER 4

Getting it Together

Practical techniques



This chapter looks at the basic techniques which I suggest should be adopted if you want to develop simulations which work as you want. I'm not going to give you a great long list of points to check off, rather a set of basic approaches which will help you avoid the major pitfalls and will bypass the common causes of delay.

The importance of defining the requirement

First define the requirement. This is the message that applies to all forms of programming: any half-way decent guide to programming should stress this one. Starting to write any computer program without defining the ultimate requirement in some way is a bit like trying to train a monkey to drive you to work — bits of the trip may be of interest but the journey is likely to end in tears.

But what is a requirement and how do you go about defining it? The requirement is based on what the user (the user, not the programmer!) wants from the simulation. This simple observation becomes very clouded when you actually start work on a new simulation, it is usual for the user to have a very loose idea of what they want at the start of development and to revise these wants as development continues. This in fact is one reason why it is important to try and define as much as possible of the requirement right from the start. If changes do occur, it is possible to go back to the original requirement and decide whether modifications can be accepted or whether the original requirement should be kept.

A first step is to try and write down the requirement. If this can't be done then forget the whole thing at this stage: if you can't manage even a description in written English then how can you expect to develop a concise description in the very much more formal medium of a computer program?

The description should contain the following:

A definition of the system to be simulated, its limits, its functioning, and the interfaces (remember? the inputs and outputs to the non-system).

What is desired from the simulation.

What data is, or will be, available and in what form.

What form of output will be possible and what will be most desirable.

How much time/effort is available for program development.

Other points.

This last is a catch-all and will vary from case to case. Some of the points which may be important are: what computing hardware will be used for development, what computing hardware will be used to run the developed simulation, what (if any) other hardware will be involved, what sort of organisation will be using the simulation, is the simulation to be sold commercially, what sort of testing of the simulation will be done, what developments might be required in the future, and will the user require continued software support. Some of these points are covered in the following paragraphs but I make no apologies for avoiding discussion of each of them in detail. In certain cases each of the points above may become important — just how important and what form the importance may take will vary from case to case.

Diagrams

Diagrams can be very useful for defining and analysing the system to be simulated. These can start off as simple lists of the system components but can include many useful factors. For example, computer systems analysts may use 'rich pictures' (**Figure 4.1**) which show the parts of the organi-

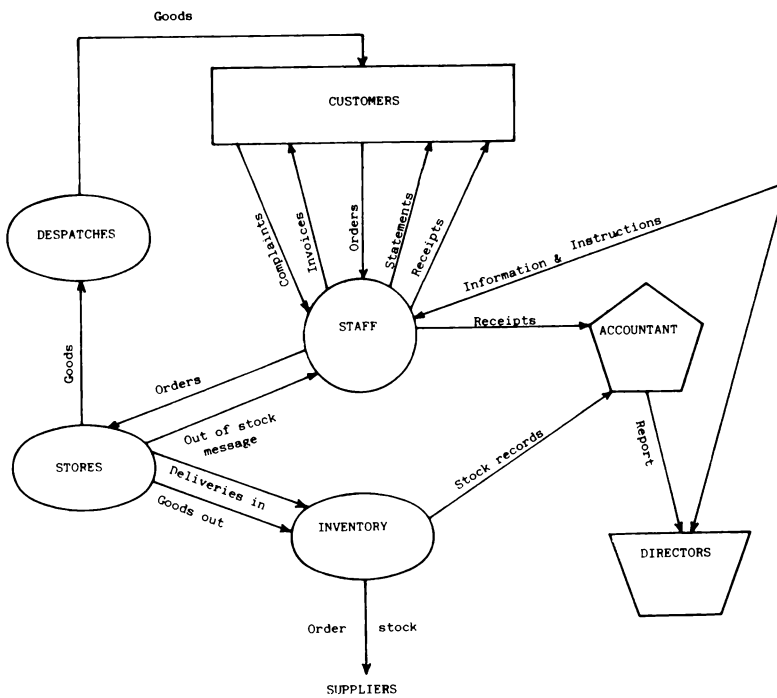


Figure 4.1: Rich Picture

sation which is to be provided with a computer system, say a wholesaler of electrical components, on which communications can be indicated as arrows. Thus the rich picture can show all the actions from receipt of an order through the post, to retrieval of the desired order from the store, to packaging the order, sending out the goods, recording remaining stock levels in store, sending out bills, and so on. It is vital to identify these communication lines if a computer simulation is to be installed, as the computer will be providing many of them. Also, it is very likely that improvements to the existing communications will be required and these are most easily identified using a communications diagram.

Another form of chart used by systems analysts and simulation engineers is the 'function chart' (**Figure 4.2**). This quite simply is a means of noting down, in a reasonably formal manner, a breakdown of the functions of an organisation. Thus, using our example above, the function of Joe Bloggs of the packing department may be to collect a store's order form and pass this to Bert (stores), wait for the items to be dug out of the box at the back of the warehouse (or take back an out-of-stock message), sign a receipt for the goods, obtain packaging materials (probably with more paper work involved), package the order, and so on. The function chart, when expressed in this way, has obvious connections with the rich picture.

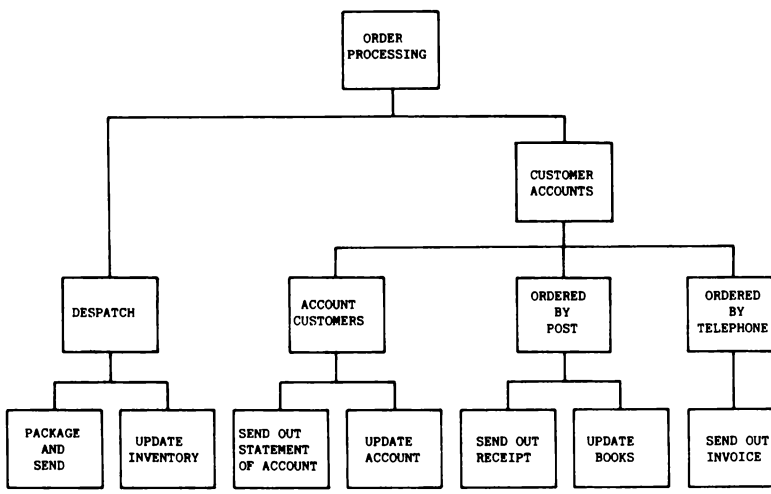


Figure 4.2: Function Chart

A more basic diagram technique, which I often use when struggling to define the operation of a new system, is the 'causal diagram'. This diagram

concentrates on the possible responses of a system to different situations. For example, a very simple causal diagram might consist of:

Fire → Smoke → Alarm

This means fire causes smoke which causes an alarm. The arrows show the 'causal links' between possible system characteristics. If we are interested in looking more closely at the consequences of fire then we can extend the diagram as in **Figure 4.3**.

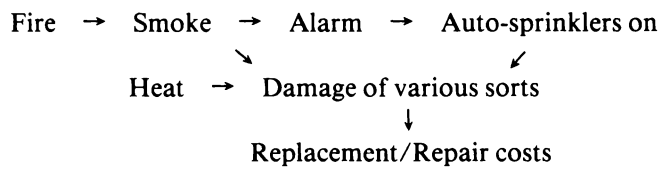


Figure 4.3: Casual Diagram.

Notice that I am mixing several types of item on the diagram: some real components of the system, some financial, and some just describing responses of the system.

Feedback

I'm going to digress a little here, while I'm on the topic of causal diagrams, and introduce the concept of feedback and feedback loops of various types. In the example above, the switching on of the sprinklers should result in a reduction of the intensity of the fire, less heat and smoke, and so less damage (but some water damage). Here is a simple form of loop — the fire causes the sprinklers to come on which damps down the fire. This loop is called a closed loop, the loop operates automatically and is contained within the system itself. An open loop requires the action of something external to the system, for example the local fire brigade, to complete the loop.

The program listed below demonstrates a feedback loop. If you type in and run the program you will see a figure in the centre of the screen. You can move it using the cursor keys. Notice how the program reacts to the movement!

```
10 REM==========FEEDBACK============
20 REM FEEDBACK DEMONSTRATOR
30 REM================================
40 DIM S0%(63)
50 GOSUB 2000:REM SET UP SPRITE
```

```
60 POKE 53281,1:REM BACKGROUND COLOUR
70 POKE 53287,0:REM SPRITE COLOUR
80 PRINT "[CLR]":REM CLEAR SCREEN
90 PRINT "[PUR]PROGRAM FEEDBACK : PRESS
S TO STOP"
100 X=172:Y=147:REM SPRITE TO CENTRE
110 GX=-0.06:GY=-0.1:REM FEEDBACK GAINS
120 GOSUB 1000:REM SHOW SPRITE ON SCREEN
130 GET K$:REM SCAN KEYBOARD
140 IF K$="[CU]" THEN Y=Y-10
150 IF K$="[CR]" THEN X=X+10
160 IF K$="[CD]" THEN Y=Y+10
170 IF K$="[CL]" THEN X=X-10
180 IF K$<>"S" THEN 200
190 POKE 53269,0:POKE 53281,6:END
200 X=X+GX*(X-172)
210 Y=Y+GY*(Y-147)
220 IF X<24 THEN X=24:REM KEEP SPRITE
230 IF X>320 THEN X=320:REM ON SCREEN
240 IF Y<50 THEN Y=50
250 IF Y>229 THEN Y=229
260 GOTO 120
1000 REM#==========SHOW-SPRITE=======
1010 REM MOVE AND DISPLAY SPRITE
1020 REM================================
1030 XH=INT(X/256):REM HIGH BYTE OF X
1040 XL=INT(X-256*XH):REM LOW BYTE OF X
1050 POKE 53248,XL
1060 POKE 53264,XH
1070 POKE 53249,INT(Y)
1080 POKE 53269,1:REM SPRITE #0 ON
1090 RETURN
2000 REM#==========SETUP-SPRITE=======
2010 REM SET UP SPRITE #0
2020 REM================================
2030 FOR N=1 TO 63:S0%(N)=0:NEXT N
2040 FOR N=23 TO 41 STEP 3
2050 READ S0%(N):REM DATA FOR 8*7 SHAPE
2060 NEXT N
2070 FOR N=1 TO 63:REM SPRITE DATA
2080 POKE 12287+N,S0%(N)
2090 NEXT N
2100 POKE 2040,192:REM DATA POINTER
```

```
2110 RETURN
2120 REM SPRITE DATA (CENTRE ONLY)
2130 DATA 66,164,136,207,162,162,66
```

A sprite is used to display a '64' at the centre of the screen. Lines 130 to 170 look for the cursor keys pressed and move the sprite accordingly. Lines 200 and 210 perform the feedback function by comparing the actual sprite position with the desired (or 'demand') position at the centre of the screen (172, 147) and move it back a distance proportional to the offset (or 'error'). The rate at which the sprite is moved back to the centre is controlled by GX and GY, which are known as feedback gains. These are set in line 110. Try varying these and see what happens.

This is an example of feedback being used to control something, in this case the position of the '64'.

Many simulations involve feedback of one form or another and this can become a major problem during the development of the program. Feedback can be represented in a number of different ways. If the wrong approach is adopted, then it is possible for the program to become numerically unstable (the output starts to oscillate and will usually produce results which are obviously incorrect) or the feedback item will be ignored. If you experience such problems then all I can realistically suggest here is to read an introductory book on control theory — I cannot do justice to the practical representation of all possible forms of feedback. Some odd comments though: a common source of error in feedback representation is selection of too long a time-step, try a shorter time-step if you suspect errors; sometimes a process of iterating on to a valid result, by repeating a feedback operation several times over for each time- step, will be required; pay very careful attention to the limits within which your various equations and so on are valid, many problems only appear when you approach or accidentally exceed these limits — a knowledge of mathematics is very useful here.

Now back to the main feature.

Simulation structure

Having written down the requirement, supported by diagrams, a number of possible actions are open to you. You can, if you so desire and you want to take the risk, dive straight into computing. This is only advisable, however, if the simulation is very simple (like the example given in Chapter 2) or you are very familiar with all aspects of the simulation techniques to be used and the system being simulated. In most cases the next stage after defining the initial requirement is either to re-write the requirement in greater and greater detail, until you are satisfied that you can do no better, or else to begin to define the structure of the simulation and the techniques

which you are going to use. The first course is better if the simulation requirement is particularly well defined or particularly badly defined. If well defined, then the simple process of repeatedly writing down the requirement will very often also serve as a pattern for the computer program. A straight translation of the text will give the functional parts of the simulation, all that need be added will be the supervisor routines (to guide the simulation activities) and input/output routines. If the basic requirement is particularly badly defined, the repeated re-writing of the requirement will help to concentrate the mind on those features which are probably more important.

Defining the overall structure of the simulation and beginning to define the techniques to be used is a logical precursor to the writing of the program. I have the rather bad habit of attempting to do most of this in my head, with the consequence that I often lose 'brilliant' ideas somewhere 'twixt brain and keyboard. My advice to you is to start off at least by getting something down on paper. Start at a high level, that is decide on the overall workings of the simulation before looking at the detail of any one part. Use diagrams, words, notes, whatever helps you think about the problems of programming. Also bear in mind that you will probably want to refer back to these scribblings at a later date to try and find out what has gone wrong or to find out just what it was you were thinking of in the first place!

Choosing the basic technique

Having gone through the process of defining just what is wanted from the simulation, it is time to look at how to get the thing working.

The descriptions given in Chapter 3 of the basic types of simulation which can be developed will have given you an indication of the range of possible approaches which will be needed. Very few simulations are of systems which will react in only one way to a given situation. In nearly all cases, we must allow for the system to react in several possible ways to any particular set of circumstances. It is normal to define the system reactions in terms of probabilities. To each possible reaction, a probability of that reaction is defined. Let's put that into simpler terms. If a system can react in one of several ways, we must know which reaction is most likely and by how much before we can simulate that system.

Monte Carlo

The differences between the basic types of simulation centre on the way in which we deal with these probabilities. The Monte Carlo process described in Chapter 2 is one way of simulating system responses to various

situations. In the Monte Carlo process a single response is selected randomly according to the probabilities of all possible responses.

Deterministic

Another process which is often used instead of the Monte Carlo process is the deterministic process. In the deterministic process, each system response is determined from the most probable response. If, for example, three responses are possible in a given situation with probabilities 0.15, 0.45, and 0.4, the deterministic process will always select the response with a probability of 0.45. The other system responses will never be simulated. The deterministic process is useful in emulations of large systems, particularly real-time simulations, where a single run-through of the simulation is all that will be made. Examples of simulations often based on the deterministic process are weather predictors, election forecasters, video games, financial models, and some of the simulators used for training personnel.

A note here about deterministic models. Often a deterministic approach is the only way of realistically simulating large and complex systems but it can lead, in some cases, to unreliable conclusions. A good example is in the long-term prediction of the weather. A huge quantity of data is required and much of this is difficult to obtain. In consequence it is difficult fully to describe the starting weather conditions. At each time-step in the simulation process a new set of conditions is derived. However, real life does not always follow the most probable course and thus there will be differences between the prediction and the reality. These differences will grow with each time-step and it has been found that the long-term prediction of weather patterns is a very difficult task. The real-life system is simply too complex and possible responses too varied to permit reliable simulation using this approach, even on the most sophisticated computers available.

A common extension to the deterministic process is to incorporate a form of accumulator which records system responses and biases future probabilities accordingly. Thus the system response can be prevented from entering into a long series of actions which can be justified individually but which are unlikely when combined together.

Markov chain

Both Monte Carlo and deterministic processes, and the simulators based on them, result in serial operations. One action follows another, and each action leads on to a further operation. There is another class of processes which result in parallel operations. These are processes which simulate more than one possible response of the system to a given situation. Obviously, a simulation such as this must maintain a running record of several system-states — the current condition of the system is known as the system-

state. One process which keeps a record of all potential system-states is the Markov chain process. In its most useful form, two matrices are set up (I'm not about to explain matrices to you — in this case they are used as a means of storing probabilities and formulae, have a look in the Glossary for more information on matrices). The first matrix holds a running record of the various possible system-states. There must be a limit to the total number of possible system-states and the state matrix holds a set of probabilities of the system being in each of these possible states. At the start of the simulation the state matrix will usually hold a probability of one in the location representing a single starting state and zero probabilities representing all the other possible system-states. The second matrix is called the transformation matrix and holds a series of equations which determine the way in which the system can change from one state to another. Often the equations in the transformation matrix are reduced to simple probabilities. Possible system-states are numbered and the equations of the transformation matrix give the individual relationships of each numbered system-state to each of the other system-states.

At each time-step, or point in the simulation where a system response is to be modelled, the state matrix and the transformation matrix are multiplied together to give a new state matrix. The probability of the system being in any of the possible system-states is thus continuously available.

There are other parallel-processes, such as the branch- search sometimes used in chess-playing computers, but the Markov process is the most widely known.

Technique to suit the simulation

It is not unusual, particularly in the analysis field, to create a new process for each new simulation. This is because the requirements of the simulation are likely to be too individual and the computations too complex to make use of one of the text-book processes. For example, many practical parallel-processes do not keep track of all the possible system-states but only those which are of most interest. This can greatly reduce the computing time.

Another possibility is, at each stage in the simulation, to calculate an 'average' system response or state. Typically this will use a number of separate system responses, much as used in a Markov process, which are multiplied by their relevant probability of occurring, and added together. This technique is valuable in situations where the system is free to respond in a range of ways rather than in a limited number of discrete ways. An example, which also shows up the dangers of this particular technique, could be a simulation of firing arrows at a target. Suppose we represent the firing action and the flight of the arrow as a simple probability distribution

— that is we define a set of probabilities (adding up to one if summed) of the arrow ending up at any position on or off the target for any particular shot. The averaging process will typically calculate the distances of each possible position on the target from the centre and the relative angle of each position from the upright. Each such distance/angle is multiplied by a probability provided from the input probability distribution and added into a total figure which directly gives an average result. If the probability distribution is symmetric — that is, there is an equal chance of hitting above or below the centre and left is as likely as right — then the 'average shot' will land dead centre. Obviously this result is of little use to us when stated in this form, we know that the important thing about archery is not what the average shot gives but what happens on average over a series. It is the spread of results about the average (or mean) position which is more likely to be of interest, and some additional calculations are necessary to obtain this information.

The example just given is a useful one with which to compare the responses of the various processes which I have been discussing. A Monte Carlo simulation would give a random arrow position, different for each run of the simulation. This would give the closest representation of the actual system in action. A deterministic simulation would give a single arrow position each time, dependent on the highest probability. If the archer has a tendency to drop every fifth arrow in their foot then this is the result which would always be indicated by the deterministic simulation. The averaging process, as indicated above, will show what is the average result of a number of runs. A parallel-process simulation will give an output indicating all possible arrow positions and their likelihood (in this simple example this will just be an output of the probability distribution which we input in the first place) and will probably also give some further analysis of the most likely arrow positions.

Some simulations combine characteristics of Monte Carlo, deterministic, averaging, and parallel-processing, using the appropriate techniques in different parts of the simulation. The choice of a particular technique will depend on the requirements of the simulation, the nature of the system, and the various restrictions on computing power, time available, data availability, and so on. The specialist chapters of this book give examples of my decision-making process and the consequent adoption of various techniques. One of the important restrictions on the simulations covered by this book is that I must keep the programs as short as I can whilst demonstrating the techniques. In your simulations you will not have this constraint but it is likely that some other factor will be important: you must think long and carefully about the basic techniques available to you and just what the advantages/disadvantages of each will be.

Development of the simulation

You know what you want to do, you know how you want to do it, but how do you actually set out to create a program to make the thing real?

There is more than one way to skin a cat (or write a program!). Here is a program which moves a '*' around the screen:

```
10 REM==========MOVE================
20 REM MOVES * AROUND THE SCREEN
30 REM================================
40 GOSUB 1000:REM SET UP
50 X=12
60 FOR Y=5 TO 19
70 GOSUB 2000:REM PRINT
80 NEXT Y
90 Y=20
100 FOR X=12 TO 26
110 GOSUB 2000:REM PRINT
120 NEXT X
130 X=27
140 FOR Y=20 TO 6 STEP -1
150 GOSUB 2000:REM PRINT
160 NEXT Y
170 Y=5
180 FOR X=27 TO 13 STEP -1
190 GOSUB 2000:REM PRINT
200 NEXT X
210 GOTO 50
1000 REM#==========SET-UP============
1010 REM SET UP R$ AND C$
1020 REM================================
1030 FOR N=1 TO 25:R$="[CD]"+R$:NEXT N
1040 FOR N=1 TO 40:C$="[CR]"+C$:NEXT N
1050 RETURN
2000 REM#==========PRINT============
2010 REM PRINT * AT X,Y
2020 REM================================
2030 PRINT "[CLR]";LEFT$(R$,Y);LEFT$(C$,
X);
2040 PRINT "*"
2050 GOSUB 3000
2060 RETURN
3000 REM#==========PAUSE============
3010 REM DELAY LOOP
3020 REM================================
```

```
3030 FOR N=1 TO 50:NEXT N
3040 RETURN
```

This program clears the whole screen display and then, using cursor movements to get the new print position, prints a '*'. Run it and note how fast the '*' moves.

A second version of the program is obtained by changing subroutines SET-UP and PRINT (lines 1000 and 2000), and changing the subroutine calls as follows:

```
70 GOSUB 2030:REM PRINT
110 GOSUB 2035:REM PRINT
150 GOSUB 2040:REM PRINT
190 GOSUB 2045:REM PRINT
1000 REM#==========SET-UP============
1010 REM SET INITIAL POSITION
1020 REM================================
1030 PRINT "[CLR]";
1040 PRINT "[CD][CD][CD][CD]";
1045 PRINT "[CR][CR][CR][CR][CR][CR][CR]
[CR][CR][CR][CR]";
1050 RETURN
2000 REM#==========PRINT=============
2010 REM PRINT * AT X,Y
2020 REM================================
2030 PRINT "[CL] [CL][CD]*";:GOTO 2050
2035 PRINT "[CL] *";:GOTO 2050
2040 PRINT "[CL] [CL][CU]*";:GOTO 2050
2045 PRINT "[CL][CL]* [CL]";
2050 GOSUB 3000:REM PAUSE
2060 RETURN
```

Make sure you include the spaces in lines 2030, 2035, 2040, and 2045.

This version uses the cursor control characters to overwrite the last printed '*' with a space and then move to the new location to print the next '*'. Notice any difference in the speed?

Finally, the following version of the program replaces the PRINTed '*' with a sprite, moved in the same pattern. Replace subroutines SET-UP at 1000, and PRINT at 2000, in the original program with:

```
1000 REM#==========SET-UP============
1010 REM SET UP SPRITE DATA
1020 REM================================
1030 DIM S0%(63):REM FOR SPRITE DATA
```

```
1040 PRINT "[CLR]"
1050 FOR N=1 TO 63:S0%(N)=0:NEXT N
1060 FOR N=23 TO 41 STEP 3
1070 READ S0%(N):REM READ DATA
1080 NEXT N
1090 FOR N=1 TO 63
1100 POKE 12287+N,S0%(N)
1110 NEXT N
1120 POKE 2040,192:REM DATA POINTER
1130 POKE 53287,1:REM SPRITE WHITE
1140 POKE 53269,1:REM SPRITE #0 ON
1150 PRINT "PRESS S TO STOP"
1160 RETURN
2000 REM#==========PRINT=============
2010 REM PRINT * AT X,Y
2020 REM================================
2030 GET K$
2040 IF K$="S" THEN POKE 53269,0:END
2050 XH=INT((8*X+17)/256):REM HIGH X
2060 XL=INT((8*X+17)-256*XH):REM LOW X
2070 YL=INT(7*Y+43):REM Y POSITION
2080 POKE 53248,XL
2090 POKE 53264,XH
2100 POKE 53249,YL
2110 GOSUB 3000:REM PAUSE
2120 RETURN
```

and add

```
3050 DATA 0,34,28,127,28,34,0
```

Thus we have at least three ways of performing the same task but each runs at its own rate and each is different in complexity and program length.

It is important when writing programs to consider all possible alternatives.

Different programmers work in different ways: some draw up detailed flowcharts (diagrams showing the functions of each part of the program and indicating connections between different parts of the program) and only convert these into a program when happy that they have the whole thing sorted out. Other programmers work very much from their heads and start writing program lines right from the start. I tend to be one of the latter, but I can see the value in being more systematic about programming.

At the very least, I think that it is well worthwhile writing down a basic block diagram of the major subroutines which you are going to have to use.

This diagram will, in the first case, be based on your description of the requirement and your choice of simulation technique. You will need a main or controlling routine: this is a routine which determines the overall functioning of the program, calling specialist subroutines as required. Next come the input and output subroutines, based on data availability and the user requirements. One or more routines will be required to set up initial values for various program variables and to do any manipulation of the input data required before the main functional parts of the simulation are started. These main functional routines will be determined by the techniques to be used and the system to be simulated. A small number of simple subroutines may be required to be called directly from the main routine or, at the other end of the scale, many levels of subroutine may be required representing different possible responses of the system. Whichever the case, this is where the value of a block diagram is really shown: keep a close eye on how the simulation is developing and you are much less likely to suffer unpleasant surprises.

How you go about writing the subroutines themselves (programmers call this coding, despite the fact that it has been many years since computers would only accept machine or assembler code instructions) is up to you. I think that the most logical method is to keep on dividing up the description of the functioning of the subroutine until you can't describe it in any greater detail. At this stage each functional description can be converted into a program line. That is the logical approach but not the one which I usually adopt — I tend to start at the beginning and work my way through to the end. This is not always a good idea and I would advise you to attempt to do things the logical way rather than pick up my bad habits!

Keep notes on your use of variables: try to make the names of the variables meaningful so that you will be able easily to understand the program when reading through a listing, but do keep those notes. Note down the description of each variable and, if possible, the relationship of that variable to other variables. Note how I have included, at the end of each of the major programs in this book, a section which describes the variables used. I do this to make sure that I don't lose track of what the variables are supposed to be doing whilst I develop the program and also to help me understand the program if I want to come back to it after a few months and modify it. If the description of the variables is part of the program then it can't get lost.

Verifying the model

As each subroutine is completed it should be possible to make up simple tests to make sure that the subroutines work as they are supposed to. A more important level of testing must be performed when you get the whole program together. The full workings of the program must be verified

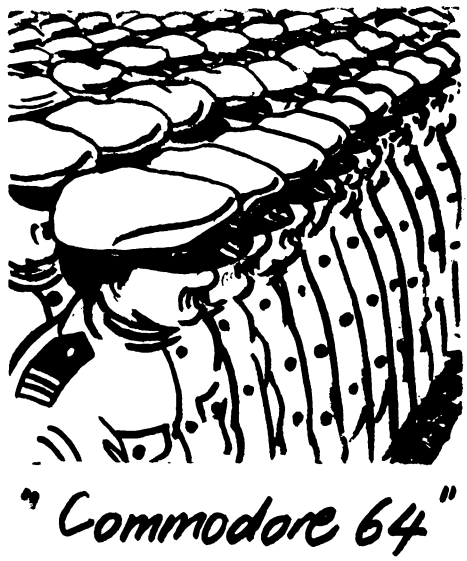
before the simulation can be used for its intended purpose. This isn't just a simple process of performing various runs and correcting any 'bugs' that appear. A systematic series of test runs must be planned and executed. Records must be kept of the results of these tests and the results carefully evaluated to detect any sources of inaccuracy. Tests to be completed include such things as running the simulation with zero inputs, in the expectation that the simulated system will remain in a stable state, running the simulation with very much simplified inputs, and running the simulation with a set of very extreme inputs. These tests will indicate any major errors but no test can be guaranteed to reveal everything; this is why careful planning and evaluation of the test runs is important.

One last piece of information, before I quit this chapter, concerning the use of a simulation for analysing the sensitivity of a system to changes in input. I am bringing this to your attention here because a simple form of sensitivity analysis can also be used to indicate possible errors in the program which would otherwise go unnoticed. This sensitivity analysis is typically performed by keeping all but one of the inputs constant for a short series of runs. The remaining input must be given several different values as the test proceeds and the responses of the simulated system noted. By plotting the results of such a series of tests on graphs for each of the various inputs, any particular sensitivity of the system can be readily identified. For test purposes watch out for unexpected sensitivities, behaviour which appears unpredictable, and any response which otherwise does not agree with your understanding of the system. If any of these circumstances applies then get suspicious and check the workings of the program once more, preferably setting additional runs.

CHAPTER 5

I'll Do It My Way

What the C64 can do



Although simulations are programs, they are also designed around the particular characteristics of the computer hardware being used. Obviously, not all simulations can be run on all computers. The power, memory, operating system and hardware peripherals all determine what can and what cannot be simulated on a given computer. This chapter considers the Commodore 64 and determines its strengths and weaknesses.

Basic description

If you have a 64, or have thought seriously about getting one, then you will already know the basic construction details of the machine. I want to go over these once more, however, and point out those features which are particularly important to the budding simulation engineer.

The Commodore 64 is a microcomputer. This means that it is based around a special integrated circuit (usually called a chip because the circuit itself is formed on a very small chip of silicon) called a microprocessor. This device is special because it permits logical operations to be programmed by a user and can hence be used as the basis for a number of types of equipment, including computers. The microprocessor is also important because it is relatively cheap. A microcomputer can provide many of the facilities of a full-scale computer (now usually called a mainframe computer, or a minicomputer if it can be easily installed in an ordinary room) but at a fraction of the cost.

Overview

A microcomputer is very much more than just a microprocessor in a box: additional circuitry is needed to produce a screen image, to hold programs and data, to produce sounds, to decode inputs from the keyboard, and to allow efficient communications with additional external devices such as disk drives, joysticks, printers, modems, and so on. The microcomputer also needs some means of controlling all this circuitry, an operating system, in a manner which allows the computer user to get the thing to do what is required with a minimum of fuss.

The vital bits of hardware in the Commodore 64, at least from the simulations point of view, are:

1) The 6510A microprocessor.
2) The 6566 (USA)/6569 (Europe) video interface chip II.
3) The 6581 sound interface device.
4) Two 6526 complex interface adaptors.
5) 64 kilobytes of random access memory.
6) 20 kilobytes of read only memory holding the operating system.

I shall discuss the importance of each of these in turn.

6510

Different microprocessors have different capabilities and can perform at different speeds. The 6500 family of microprocessors has been used in a number of commercial and research microcomputers and offer the benefits of high-speed operation and the ability to control a wide variety of other hardware items.

The 6510 is an eight-bit microprocessor. These bits are actually a string of ones or zeros (or electrical charges if you want to be pedantic) used by the machine to represent numbers. With more bits, larger numbers can be represented directly and time will not be lost pushing data into and out of memory whilst computing. Early microprocessors were four-bit machines and could thus represent numbers between zero and fifteen (one less than two to the power four). When numbers are larger than this the microprocessor must be programmed to perform a series of chained operations shunting data backwards and forwards internally and in memory. This takes time and uses up the available memory.

The home computer business has been founded on the use of eight-bit microprocessors. These can represent numbers between zero and 255 (one less than two to the power eight). For a typical microcomputer application, between four and six of these eight-bit numbers are combined to represent realistic numbers and special algorithms are programmed in to perform number operations. Some modern microprocessors use numbers internally which are up to 32 bits long and can thus be used directly to represent real numbers from zero to over four thousand million: obviously with this level of direct representation it is easier to construct short algorithms to perform operations on real numbers. The number of computing steps required to perform a given operation which is arithmetic can be reduced as the number of bits increases and hence the time taken to perform such operations comes down. However, many of the operations performed by the microprocessor in a microcomputer are not arithmetic by nature, for example much time is spent interpreting or translating the user's program written in BASIC into a form usable by the microprocessor. These tasks can be performed as quickly by an eight-bit processor as by a 16- or 32-bit processor.

I used the short 'vanilla-standard' BASIC program given below to give a comparison of a low-cost Z80-microprocessor-based home microcomputer (eight-bit processor), with the C64, and a contemporary 16-bit business machine (16-bit processor).

```
10 DIM O(1000)
20 FOR N=1 TO 1000
```

```
30 LET M=N
40 IF (M>500) THEN LET M=500
50 LET O(N)=INT(M*N*LOG(N)+.5)
60 NEXT N
70 PRINT "LOOP FINISHED"
80 STOP
```

The low-cost Z80 based machine took 1 minute 33 seconds to run, the C64 took 47 seconds to run, and the business machine took 24 seconds to run. Re-writing the program specifically to optimise the run time on the C64:

```
10 DIMO(1000)
20 FORN=1TO500:O(N)=INT(N*N*LOG(N)+.5):NEXT
30 FORN=501TO1000:O(N)=INT(N*500*LOG(N) +.5)
:NEXT
40 PRINT"FINISHED LOOP":STOP
```

and re-running reduces the C64's time to 39 seconds. Cutting out spaces and remarks, if there are any, will decrease the run time of your programs, although it will not do anything for their legibility. Using integer variables where possible, as in the program 'LIFE' in Chapter 2, will also save on run-time. Note that using the abbreviated keywords available on the C64 does not affect the run-time.

6566/9

The video interface chip II (as Commodore call it) is the circuit which works with the 6510 microprocessor to produce the television, or monitor, display.

Three important facilities are provided by the Commodore 64, through the use of the 6566/9 chip.

The basic screen display is based on a text-screen of 40 columns and 25 lines. Printed text and various predefined characters can be readily placed anywhere on the screen in a variety of colours. Simple commands in BASIC are sufficient to control this display mode and it is used throughout this book to produce various effects.

The display of detailed drawings, the plotting of graphs, or the creation of animated scenes however require a second display mode allowing high-resolution images to be produced. The 6566/9 chip can be used to produce a display of 320 (horizontal) by 200 (vertical) pixel dots. The major drawback with this display mode is that it can only realistically be used with machine code programs or after writing a number of special routines in BASIC. The Commodore 64 V2 BASIC does not provide commands which use the high-resolution display mode. In addition, the resolution of 320 × 200 pixels cannot be considered as particularly high by today's

standards and is certainly a limiting factor when producing diagrams or pictures.

Last, but not least, the 6566/9 provides the capability for displaying moveable sprites on-screen on top of the ordinary display (low- or high-resolution). Sprites have become one of the basic tools for video-games programmers and permit complex, high-speed moving sequences to be built up.

In general, the display facilities provided by the 6566/9 video interface chip II are as good as, or better than, most eight-bit home microcomputers. The home computer is designed to be compatible with ordinary television receivers and this puts a limitation on the detail which can be displayed. Special-purpose microprocessor systems are available which can produce very high-resolution displays but they are very expensive, relying on special hardware for the storage and display of screen information. Such machines are used for computer-aided design, network manipulation, and graphic representation.

6581

Known as the sound interface device, the 6581 chip can simultaneously produce three separate sound effects. Control of frequency, amplitude, waveform, and time-envelope provides a very versatile sound production capability.

Once again, the drawback here (depending on how you look at it of course) is that it is difficult to produce useful sounds from BASIC. The machine code programmer can simulate common musical instruments, provide rudimentary speech capabilities, and make all kinds of interesting special effect noises to add interest to games.

The 6581 is in fact a very advanced piece of electronics and is not usually used to the full in the Commodore 64. Provision has been made on the chip for input of analogue data (as produced by most conventional forms of electronic test equipment and audio equipment), for input of and manipulation of external audio signals, and for the interconnection of additional 6581s to create more complex sound synthesizers.

All this sophistication is useless to most forms of simulation however as little use is made of sound. Music and speech synthesis are the major 'users' of sound.

6526

Most communications between the Commodore 64 and the outside world (tape, disk, printer, etc.) pass through two special chips known as complex interface adaptors. The significance of these chips to the simulations engineer is simply that they are there. They contribute nothing to the simu-

lation capability of the 64 itself but make communications with external hardware easier.

Memory

Of particular note on the Commodore 64 is that a full 64 kilobytes of random access memory (RAM) is provided. This is as much memory as can usually be used by an eight-bit microprocessor and is considerably more than many other microcomputers provide. The RAM is the useful memory which can be used for the storage of data and programs and which is used to store the information used by the 6566/9 to produce the screen display.

Ordinarily, almost half (26 kilobytes) of the RAM is used up by the operating system and to store the display data, leaving 38 kilobytes (less if using high-resolution display) for programs and data. This is sufficient for many tasks but definitely limits the quantity of data which can be stored.

External hardware

Most users of the Commodore 64 have used the tape cassette system. This is noted for its reliability but not for its speed. The tape can at least be used to store and retrieve data however and this is a useful facility when running most serious simulations, which tend to be 'data-hungry'.

Disk drives improve the system considerably, allowing named files to be searched for and used with little wasting of time.

I shall not discuss the relative merits of other hardware items as they have less relevance to the running of simulations. Data storage is important and can mean the difference between usable and unusable simulations. The reduced data-transfer times and the facility for random access of data available with disk-drives can be vital.

Operating system

The operating system is a machine language program which controls the operations of the 6510 microprocessor and is the means of providing the computer user with the commands necessary to write and execute programs.

It is the operating system which provides the BASIC programming language and all of the commands available under BASIC. This is the major problem area for the Commodore 64, as the V2 BASIC used is not very sophisticated and thus the development of some types of simulation is made more difficult than it need be. For example, it is very often useful to provide some form of graphical output during, or at the end of, a simulation run. This is not impossible on the Commodore 64 but could be made very much easier by the inclusion of a few appropriate additional commands.

The operating system of the Commodore 64 also provides a library of machine code routines which can be called from programs written by the user. This is particularly important when developing machine language simulations.

Thus, in conclusion, the operating system does not do any favours for the BASIC programmer and rather limits the use of the hardware available. The machine code programmer is quite well catered for, however, and can fully control the sophisticated Commodore hardware.

Simulations on microcomputers: general comments

Simulations are greedy programs. They nearly always demand a lot of memory for the program or the data and also require a fast processing speed. This explains why microcomputers are not usually the ideal form of hardware on which to try to develop and run simulations, as micros are usually rather slow and have very small memories.

Some generalisations may be in order here. Complex BASIC programs can be written in perhaps 10 kilobytes of memory but additional memory is required for data. This additional requirement could range from perhaps two to many tens of kilobytes. Thus, if a micro has less than, say, 20 kilobytes of memory readily available, then you can predict that problems with memory are going to be a prime concern with simulation programs.

Speed is required not just by real-time simulations but also by most other simulations in order to permit a large number of runs to be made and, of course, to stop the operator getting too bored. If a program takes much more than an hour to run, and this is not so unusual for many applications on microcomputers, then the development and testing stage can be something of a trial. There are various ways of speeding up computing on microcomputers, including using machine code and adopting special programming techniques. Such considerations are beyond the scope of this book and I can only say here that the faster the programming speed of the microcomputer the better.

Arithmetic accuracy can be a bit of a problem with micros, particularly as simulations often rely on the precise calculation of probabilities. Generally, however, as long as the programmer is aware of the limitations of the software it is relatively easy in most cases to compensate for any potential problems. The Commodore 64 allows for the selection of integer numbers or real numbers with the simpler data offering (in theory) higher processing speeds.

Display capabilities affect output from the simulation, and a great variety of display formats are available on microcomputers, ranging from simple text displays in black and white only to quite good resolution in colour. A point to note here, though, is that the more complex displays usually require more memory and this may well eat into the memory availa-

ble for the programmer's use. Real-time simulations, and in particular video games, make the heaviest demand on display capabilities for most home-based micros, whilst the display of complex designs and networks demand very high resolution capabilities for business use.

Facilities for communication with external devices, such as measuring equipment, data loggers, other computers, and so on, are usually rather minimal on microcomputing systems. Some micros are designed to be used in laboratory environments as well as in the office or at home, and suitable interfaces are provided. This tends to be an exception rather than the rule, however.

C64 benefits

Any computer has its own series of potential benefits and restrictions. At the start of this chapter I quickly described some of the important characteristics of the Commodore 64 and then went on to note some of the general comments which can be made about micros as a class. In the next few paragraphs I will summarize these comments and pick out the most important factors, as I see them, when using the Commodore 64 for running simulations.

The Commodore 64 is a good general-purpose eight-bit machine with a relatively large memory. Memory is vital for many simulations. The processing speed of the Commodore 64 is neither particularly exceptional nor particularly poor when compared with other home microcomputers.

Storage facilities (tape or disk) are slow but reliable. Data used by programs can be stored on tape or disk relatively easily — this is important for many simulations.

Software available on the machine, in the form of the standard BASIC, is not well suited to the rapid development of complex simulations but is adequate. Machine code programming can overcome many of the limitations of the operating system but is not really available to the majority of users.

Access to external devices is quite well catered for, making practical use of the computer with the other equipment a real possibility.

PART 2
Practical

Each chapter in Part 2 presents a practical example and develops a program to provide the required simulation.

Each example is taken from one of the major application areas for simulations, thus I get a chance to talk about financial simulations, simulations used for technology and development, simulations used for commercial analysis to provide management information and control, special-purpose simulations used to analyse systems with unknown characteristics and predict their behaviour, and real-time simulations.

Looking back at Chapter 3 you will recognise the following types of simulation:

prediction (Chapter 6)
emulation optimisation (Chapter 7)
analytical optimisation (Chapter 8)
analytical prediction (Chapter 9)
emulation (Chapter 10).

Notice that analytical simulators are rarely used simply to analyse systems. In most cases, the information is further manipulated to predict behaviour or to optimise the system performance.

CHAPTER 6

Laughing All the Way to the Bank

Money models



Financial modelling is the first area which I consider and one which most users show interest in at some time or another.

Background

Although expressed in a variety of ways, most businesses and individuals want the same basic questions answered about their finances.

The most popular question is: 'How much money have I got now?'. Next comes: 'How much money will I have tomorrow?'. These two questions tend to form the basis for the vast majority of financial models (and also keep a lot of accountants very busy), although companies will talk about assets rather than money.

There are, however, a few other questions which can be asked, especially by those seeking to become healthy, wealthy and wise. These typically are: 'What happened to the money that I had yesterday?', 'How can I make the most money tomorrow from that which I have today?', and 'How much money has everybody else got, and what are they going to do with it?'. It is these last few questions that often need some form of simulation to answer properly. The investigation of what will happen if alternative possible courses of action are taken is, in principle, a very straightforward problem. In fact, financial simulations often grow into very complex programs when all factors are allowed for.

Some examples of real applications which are based on the questions noted above are:

How much have I got?	— Stock maintenance program
How much tomorrow?	— Profit and loss projections
What happened to it?	— Personal accounts
How can I make more?	— Investment modelling
What about the others?	— Stock Market models

The last two tend to require not only a very good understanding of the system which is being modelled but also access to a very large amount of reliable data. Think carefully before you decide to take the Stock Market by storm with your Commodore 64 — some very clever people using some very expensive hardware are already operating there and have found that their simulations are mainly of use in aiding them to make their own decisions; the programs cannot usually be used very reliably for unaided prediction in this complex environment.

Example

As my example in this chapter, I am going to look at the second question, predicting future finances based on current assets and past performance.

The case that I have dug out of my case-book concerns the prediction of monies held in various accounts belonging to a working person (not me, let me hasten to add, but a friend of mine who wanted to visit a computer exhibition in Los Angeles), with allowance made for income from a fixed source (the main employment) and from a second source (a part-time business). Outgoings are also allowed for and broken down into major types of expense. The simulation part of the program comes in when I define the rules for predicting future figures from known historical data. The basic approach that I outline here can equally well be used for the analysis of company performance or the monitoring of any form of funds, such as club assets.

Simulation background

This is one case which lends itself to development on a spreadsheet and thus provides me with a chance to describe the potential use of a common software package not normally thought of as the basis of a simulation. But I am jumping the gun a little, am I not?

As I described in Chapter 4, we must first define the simulation requirements. I shall of necessity keep this bit brief: when you come to developing real simulations you will have to be more thorough. The system which we are going to simulate consists of the money possessed by a single person, Penny Greenback. The money is kept in a current account at her local bank, National Vault, a deposit account in a building society, Humpty Securities, and as cash. The cash is almost incidental as Penny keeps an average of about £10 in cash at all times. There is a little more to the system than just the money, however, as we must also look at the ways in which money enters and leaves the accounts and moves from one place to another.

The month is May, Penny has just had a pay rise and each month (calendar month rather than the often used four-week month) Penny is paid £600, after tax and other deductions have been made, which is paid directly into her bank. As a means of building up her savings, Penny has arranged a standing order of £60 a month from her bank to her building society account. All major bills, such as electricity and phone bills, are paid by cheque from the current account at the bank. Day-to-day bills are paid in cash which Penny withdraws from the bank as she needs it. Penny has, very conveniently, kept all her old cheque stubs and can thus say what she has paid, and to whom, over the last 16 months. I have made a note of the most important figures below.

Is that all that we need to know? We must also note the important changes in the past. Penny has just had a rise in pay — until last month she was only getting £550 a month. She has also been doing some freelance work and her income from this is given below; she expects this income

(which is quoted after tax, etc.) to increase at about 10% every month for the next few months. Every year she takes out most of her savings from the building society and goes on holiday. She wants to know what is likely to happen to her money over the next six months and whether or not she can go to an exhibition, costing a minimum of £950, in three months' time at Computaland in Los Angeles.

Now we are getting a better idea of the simulation requirement and the system to be simulated. The requirement is a prediction for each month for the next six months, with a special test for three months time. The predictions are to be based on monthly data which covers the past 16 months. There are a number of techniques for predicting values based on historical data. We could plot the historical data and visually pick out a trend. We could 'fit' a simple mathematical formula to the data and use this. These techniques, and others, could all be attempted, but I'm going for something a little simpler.

The value which we are going to predict for a given expenditure is to be based on the value for that expenditure for 12 months ago, but modified by an estimate of inflation. This estimate of inflation is based on a general comparison of the previous month's outgoings with those of twelve months before. This may not sound too simple, but will become much clearer when we get to setting up the spreadsheet.

Of course, I have already chosen the spreadsheet as the basic means of setting up the simulation and have thus avoided having to select between the various options introduced in Chapter 4. Use of the spreadsheet gives a simple means of storing and manipulating data. The predictions are quite straightforward, not involving any decisions or alternative paths. The output requirements are also very simple, being a minimum of a single figure (total money) for each of six months plus one special figure for the middle of the six month period. In fact, we shall actually be taking more interest than this in the output, but I want to set up the simulation first.

Spreadsheet

The first thing we need is a spreadsheet program. If you happen to have one already then use it, there are several good spreadsheets available for the Commodore 64. If you do not have a commercial program then use the program that I have prepared in Appendix A, which gives the basic facilities required of a spreadsheet but is written in BASIC and hence is a little slow to run.

A spreadsheet is a general-purpose program which holds data and sets of instructions on what is to be done with the data. The value of a spreadsheet comes from two fundamental features:

1) The data is displayed clearly on the screen in rows and columns, where each item of data is held in a 'cell' and rows or columns of cells can be

given some special significance by the user (for example a column of cells may represent income and each row may represent a month in which the income was received).

2) It is possible to move any cell on the spreadsheet, modify the data (or the formula which is used to produce the data for that particular cell) and then ask the program to re-calculate the whole spreadsheet, thus showing rapidly just what happens if any changes are made to any of the data.

The program presented in Appendix A, 'Simplespread', provides the following features:

1) Input T to move TO a particular cell, when using the cursor keys would be too slow.
2) Cursor keys control movement from cell to cell.
3) Input F to set up a FORMULA at the current cell.
4) Input N to input a NUMBER to the current cell.
5) Input H to input a HEADING to the current cell.
6) Input C to COPY a formula from the current cell to any other cell. Any references to other cells will be modified in relation to the new cell position.
7) Input X to EXECUTE each formula in turn and thus re-calculate the whole spreadsheet.
8) Input L to LOAD a set of spreadsheet data from tape/disk.
9) Input S to SAVE a set of spreadsheet data to tape/disk.
10) Input I to display a list of permissible INSTRUCTIONS.
11) Input Q to QUIT the current spreadsheet and set up a new blank spreadsheet.

Commercial spreadsheets provide additional facilities but these are the basics. My description, given below, of how to set up the spreadsheet to simulate Penny Greenback's money situation is based on Simplespread but will be understandable when using any commercial spreadsheet program.

Note that Simplespread is set up to display data in the conventional monetary format with two decimal places shown.

Simulation development

The following data has been supplied by Penny Greenback and should be set up in columns A to G of a blank spreadsheet. Use the titles given as

column headings in row one and the month numbers as references in column A. Figures give expenditure for a full month.

	A	B	C	D	E	F	G
01		RENT	GASELEC	CAR	HOLIDAY	PHONE	MISC
02	FEB-A	148.00		58.00	200.00		261.00
03	MAR-A	148.00	51.00				312.00
04	APR-A	148.00	55.00	22.00		21.00	264.00
05	MAY-A	148.00					230.00
06	JUN-A	148.00	42.00				265.00
07	JUL-A	148.00	40.00	47.00		23.00	278.00
08	AUG-A	148.00		95.00			324.00
09	SEP-A	163.00	34.00	85.00			330.00
10	OCT-A	163.00	24.00			23.00	211.00
11	NOV-A	163.00			550.00		320.00
12	DEC-A	163.00	31.00				278.00
13	JAN-B	163.00	22.00	44.00		25.00	256.00
14	FEB-B	163.00					300.00
15	MAR-B	163.00	56.00				289.00
16	APR-B	163.00	61.00			35.00	268.00
17	MAY-B	163.00		12.00			268.00

Note: I have not reproduced the display of Simplespread exactly. Only four columns of data are displayed by Simplespread — I have had to use more than this to aid clarity.

In addition to expenditure, we must also set down Penny's income and the money which remains in her bank and building society accounts. We shall ignore cash — Penny keeps around £10 as cash, but this is a relatively constant figure and can be excluded from this system. The relevant figures, to be entered in columns H to K of the spreadsheet, are as follows:

	A	H	I	J	K
01		INCOME-A	INCOME-B	BLDSOC	BANK
02	FEB-A	550.00		10.00	71.00
03	MAR-A	550.00		70.00	43.00
04	APR-A	550.00		130.00	70.00
05	MAY-A	550.00		190.00	84.00
06	JUN-A	550.00		250.00	161.00
07	JUL-A	550.00		310.00	183.00
08	AUG-A	550.00		370.00	91.00
09	SEP-A	550.00		430.00	8.00
10	OCT-A	550.00		490.00	5.00
11	NOV-A	550.00			-35.00
12	DEC-A	550.00		60.00	14.00

13	JAN-B	550.00		120.00	54.00
14	FEB-B	550.00	35.00	180.00	25.00
15	MAR-B	550.00	50.00	240.00	113.00
16	APR-B	550.00	56.00	300.00	172.00
17	MAY-B	600.00	64.00	360.00	249.00

So far, we have set up the data to be used in the simulation but nothing more. The next thing we need to do is to set up any modifications that may be required to the raw data. (In case you were wondering, raw data is data that has not been modified or analysed in any way. For some reason, which you may be able to work out for yourself, it is unheard of for people to talk of modified data as cooked data, it is more usual to refer to it as processed data.) One of the things that we want to know is how much money Penny actually has at the end of each month. This quantity is found simply by adding the two account figures. We also need to estimate the inflation in spending over the previous 12 months, based on actual expenditure, which in turn requires that we find out, month by month, just what the expenditure has been. We thus have additional columns to add to the spreadsheet: spending, inflation, and money. In fact two columns are needed for spending when using Simplespread, because there is a limit to the length of any formula which can be entered into a cell. Thus, we need a total four columns set up in L, M, N, and O, headed SPEND1, SPEND2, INFLATN, and MONEY respectively.

SPEND1 is a simple sum of money out in any month, not including miscellaneous. Move the cursor to cell C(L-02) on the spreadsheet and enter the formula:

F(C(B-02) + C(C-02) + C(D-02) + C(F-02))

Note that I have left the holiday expenditure out of the sum. This is because we are going to use spend to generate an inflation figure and the large expenditure on infrequent holidays would play havoc with this figure.

Note also that I have used a two-figure entry for each reference to a cell row. I do this to ensure that the total length of the formula will not exceed the maximum allowable length when the formula is copied to another part of the spreadsheet, as we are about to do with this one.

The spend formula given above must be C(opied) down to each cell in column L to row 17 (including cell 17) in order to give the spend for each month.

Now move to cell C(M-02) and we shall set up SPEND2, which is the total spending including MISC. Enter the formula:

F(C(L-02) + C(G-02))

C(opy) this formula down to each cell in column M to row 17.

Inflation is to be set in column N, but it can only be set where at least 12 months of spend figures are available. We shall leave this blank for the moment. Column O is to be the money column. The formula to enter in cell C(O-02) is:

F(C(J-02) + C(K-02)

Again this must be C(opied) down the column.

Enter X(ecute) to execute the formulae. Columns L, M, N, and O should now look as follows:

	L	M	N	N	O
01		SPEND1	SPEND2	INFLATN	MONEY
02	FEB-A	206.00	467.00	81.00	
03	MAR-A	199.00	511.00	113.00	
04	APR-A	246.00	510.00	200.00	
05	MAY-A	148.00	378.00	274.00	
06	JUN-A	190.00	455.00	411.00	
07	JUL-A	258.00	536.00	493.00	
08	AUG-A	243.00	567.00	461.00	
09	SEP-A	282.00	612.00	438.00	
10	OCT-A	210.00	421.00	495.00	
11	NOV-A	163.00	483.00	-35.00	
12	DEC-A	194.00	472.00	74.00	
13	JAN-B	254.00	510.00	174.00	
14	FEB-B	163.00	463.00	205.00	
15	MAR-B	219.00	508.00	353.00	
16	APR-B	259.00	527.00	472.00	
17	MAY-B	175.00	443.00	609.00	

Now we can set up the simulation part of the spreadsheet. This means setting up formulae which will predict the future performance of the system from stored data.

Extend column A downwards to give months JUN-B; JUL-B; AUG-B; SEP-B; OCT-B; NOV-B. This will give predictions for the next six months (remember, when I first set up this simulation the current month was May). Now we must enter the formulae, column by column, to fill rows 18 to 23.

First, rent. Enter:

F(C(B-17)

into cell C(B-18) and C(opy) down the column to all the cells in row 23. This will keep the rent at a fixed level.

The general formula we shall use for predicting unknown expenditure for each month is:

Expenditure = Expenditure from 12 months before * annual inflation calculated to the previous month

For example, the precise formula to enter into cell C(C-18) and C(opy) down the column into all cells to C(C-23) is:

F(C(C-06)*C(N-17))

Similar formulae must be set up for columns C, D, F, and G.

As for rent, the predicted values in columns E and H will remain constant. Use the formula:

F(C(E-17))

in cell C(E-18) and C(opy) down to cell C(E-23). Enter:

F(C(H-17))

in cell C(H-18) and C(opy) down to cell C(H-23).

INCOME-B is predicted by Penny to rise at 10% per month. Hence the formula to enter at cell C(I-18) is:

F(C(I-17)*1.1)

This also must be C(opied) down the column.

Money in the building society account is added to (from the bank account) by £60 per month, and is only removed if required for holidays. Enter the following formula in cell C(J-18) and C(opy) down to cell C(J-23):

F(C(J-17)+60)

The ordinary bank account is a little more complex. Money in is the income figure from columns H and I. £60 goes to the building society. Miscellaneous expenses are paid for, in general, in cash which Penny withdraws as required from the account; hence the figure given in column G is subtracted directly from the bank account. Other major outgoings are paid by cheque. One point about cheques is that they take time to process. In order to give you an idea of how to deal with this I am assuming that cheques written in one month are not actually cleared until the following month. This is an exaggeration, of course, but is a simple rule to apply and

will do for this simulation. Perhaps you might like to think of other approaches, for instance to back up the assumption that it takes an average of six days to post and process a cheque.

Simplespread will not allow a single formula long enough to work out the full bank account sum in one go. Hence we must set up some 'dummy' data cells which will each hold part of the information required to calculate the bank sum. Move to cell C(P-01) and enter the heading BANKD1 and in cell C(Q-01) enter BANKD2. These are the headings for our dummy data cells and will hold sums of the money moving into and out of the bank account.

Move to cell C(P-17) and enter the formula:

F(C(K-17)-60-C(B-17)-C(I-17))

and C(opy) this down the column to cell C(P-23).

Enter:

F(C(P-17)-C(D-17)+C(F-17))

into cell C(Q-17) and C(opy) down to cell C(Q-23).

Now we can set up the proper entry for the bank account. Move to cell C(K-18) and enter:

F(C(Q-17)+C(H-18)+C(I-18))-C(G-18))

this is now C(opied) down column K to cell C(K-23).

Notice how I set up the dummy cells in the same row as the actual expenditure or income rather than the row in which we want to calculate BANK. This is because of the order in which spreadsheets calculate the formulae, which is generally left to right along each row moving down through the spreadsheet. If BANKD1/D2, etc., were set up in the same row as BANK, which uses them, then BANK would take data from BANKD1/D2 before these had been calculated.

Let's move on.

Columns L, M, and O already have formulae set up, these merely need extending to row 23.

The last column to fill is the inflation column. This must be calculated from MAY-B to NOV-B. The formula is simple:

F(C(M-17)/C(M-05))

is entered at cell C(N-17) and C(opied) down to cell C(N-23). Again, this is a gross simplification and there are other approaches which could be adopted which would be more realistic and less prone to error, but much

more complex to implement. The major problem with the method I have adopted is that the inflation figure can be distorted by any odd highs or lows in the data. Also, recent trends in data are not identified using this approach. For example, the most recent phone bill was unusually high, due, I suspect, to Penny's part-time job. Our method of prediction will not carry this high rate forward.

Results

The simulation is more or less complete now. If you X(ecute) you should get the following:

	A	B	C	D	E	F	G
01		RENT	GASELEC	CAR	HOLIDAY	PHONE	MISC
18	JUN-B	163.00	49.22				310.57
19	JUL-B	163.00	45.96	54.00		26.43	319.42
20	AUG-B	163.00		107.90			368.01
21	SEP-B	163.00	38.31	95.78			371.86
22	OCT-B	163.00	26.23			25.14	230.63
23	NOV-B	163.00					338.25

	A	H	I	J	K
01		INCOME-A	INCOME-B	BLDSOC	BANK
18	JUN-B	600.00	70.40	420.00	373.83
19	JUL-B	600.00	77.44	480.00	459.63
20	AUG-B	600.00	85.18	540.00	427.41
21	SEP-B	600.00	93.71	600.00	418.36
22	OCT-B	600.00	103.07	660.00	533.70
23	NOV-B	600.00	113.38	720.00	634.46

	A	L	M	N	N	O
01		SPEND1	SPEND2	INFLATN	MONEY	
18	JUN-B	212.22	522.79	1.15	793.83	
19	JUL-B	289.39	608.81	1.14	939.63	
20	AUG-B	270.90	638.92	1.13	967.41	
21	SEP-B	297.09	668.95	1.09	1018.36	
22	OCT-B	214.37	445.01	1.06	1193.70	
23	NOV-B	163.00	501.25	1.04	1354.46	

If you do not get this result, then check through your entries from the start.

Now we have our answer. Penny can just about afford to go to Los Angeles in August — or can she? We have modelled the system and predicted forward for six months in steps of one month. The indications are that

Penny will have over £900 in August and will not suffer drastically in the following months. However, in our haste to set up the simulation we have overlooked a few vital points. These are as follows:

1) Penny's rent is due to go up in September, probably by 10%. Enter F(C(B-20)*1.1) in cell C(B-21).

2) We have not actually allowed for the cost of the trip in August. Set cell C(E-20) to say 950. Modify the formulae in cells C(J-20) and C(K-20) to pay for the trip, £540 from the building society account and £410 from the bank.

3) Although Penny can take paid leave from her full-time job, she will lose her potential earnings from her part-time job, and will also lose some of her increased earnings in the following months. In fact, by talking to Penny I found out that she would lose all her money in August (enter zero in cell C(I- 20)) and her earnings in September would be 10% up on July (enter F(C(I-19)*1.1) in cell C(I-21)).

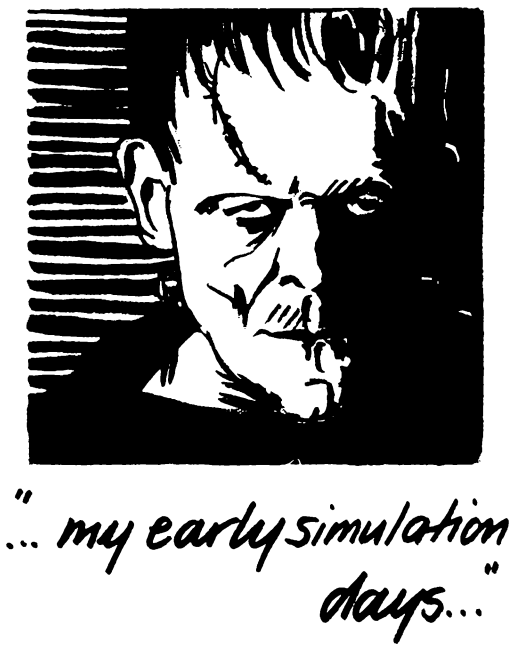
Now X(ecute)again. Do you still think Penny can afford to go to Los Angeles? If she does, go she will go £85 into the red in her bank account.

I'm not going to tell you what Penny finally decided to do — all the best mystery stories leave the ending open — but you see how careful you must be in setting up even a simple simulation and in making sure that all relevant information is allowed for. You must constantly be checking the workings of the simulation and the assumptions on which it is based. If at any point the simulation gives you a marginal result, ie one which is very close to a decision-making value, then be especially careful.

CHAPTER 7

Testing Testing Testing

Product development



My second practical chapter dives into the world of hardware development.

Background

The potential roles for simulations in support of hardware development are so many and so varied that there is little that I can meaningfully add to my general comments in Chapters 2 and 3.

Most of my own experience has been concerned with the evaluation of different hardware solutions to meet a given need. This involves creating a simulation of the environment in which the hardware is to be used and then using this to compare the different performances of possible designs using the simulation. This form of simulation is usually developed at the initial planning stage of a hardware programme. In parallel, a financial assessment of the hardware designs will be made, possibly based on financial simulations of the development and production operations and the potential commercial market, which will lead to the selection of a preferred design.

Development of hardware typically follows the pattern: research, design, development, production and update. Comparison of alternative possible designs is generally completed during the research stage. The design stage is based on the evolution of a detailed technical evaluation and description of the chosen concept, often based on the use of computer-aided design programs (CAD). The design process is based on the use of theoretical calculations and the application of old, proven technologies into new products. Use of computer assistance during the past few years has lead to a considerable reduction in the time taken to achieve a detailed design, a more thorough evaluation of the design prior to the manufacture of prototypes, and savings in product weight without any reduction in strength.

During the development stage, the designs are converted into hardware, tested, improved and evaluated against the original requirement. Reliability assessment and prediction will identify problem areas and indicate whether or not the product will be fully ready for production when planned. Especially important during development is the setting up of the production facility. This may mean just making adjustments to existing machines, or creating new moulds and dies for casting work, or it may mean building a new factory, ordering and buying new machinery, employing and training staff, and so on. Obviously, it is vital to get this planning, which may be extremely complex, done right, or large amounts of money can be lost. Extensive use is made of project-planning computer programs, such as critical-path analysis, and these programs are becoming more and more sophisticated. At the heart of most of these is a simple simulation of the sequencing of the project plan.

Less use is made of simulations once production is under way: Chapter 8 looks at simulations for management information.

Updating the design may take place some time after the product has first entered production. This may be to deal with some defect appearing after production is under way, to reduce production costs, to improve performance, or simply to make the product look more modern. The updating process is, in theory, very similar to the first design phases, with research, design, development and production. Computing requirements are also, in principle, the same.

In summary, greater and greater emphasis is being placed on the use of computers during development and many of these uses involve the simulation of either the hardware under development, the system in which the hardware is to be used, or the system which is to be used to make the hardware. I titled this chapter 'Testing Testing Testing', because hardware development has traditionally required considerable effort to be put into prototype testing. The emphasis is gradually changing — perhaps I should have called this chapter 'Evaluating Evaluating Evaluating'.

Example

There are many examples that I could have chosen; most of these however would have been too lengthy to include in a book of this size. The example that I have chosen takes me back to my early simulation days. When I was first learning to use computers I was also keenly interested in motor racing, particularly go-kart racing and motor-cycle racing. Engine capacity ranged from 100 to 500cc (cubic centimeters), and the engines, usually two-strokes, were tuned to give maximum power outputs. Don't worry if you do not know much about engines, I shall shortly explain all you need to know.

One of the interesting things about these small capacity engines is that, once you start to look at them in detail, they turn out to be very simple (often to keep weight to a minimum) and rather under-developed. Advances in design have been rapid, but mainly because a large number of people have been trying out different ideas; some are bound to work and these are rapidly copied by all the manufacturers. It struck me that I could use my A-level physics knowledge to simulate part of the engine and thus 'try out' different ideas without having to spend huge amounts of time and money testing hardware. (You may like to note that the Japanese motor-cycle industry adopted a similar approach with some success.) I was particularly interested in simulating the exhaust system (a vital component for racing engines — still under-developed) and the ignition system.

My example for this chapter is based on my early efforts to develop a simple electronic ignition system for a 100cc racing engine.

Simulation development

Piston engines work by using a piston moving in a cylinder to compress a mixture of air and fuel (in this case petrol); burning the mixture and thus using the chemical energy present in the fuel to heat the air/fuel mixture which in turn increases the pressure of the mixture; and using this high pressure to push the piston back down the cylinder. It is the pressure acting on the piston which does the work and provides power. Maximising the power output of the engine depends principally on burning the air/fuel mixture in such a way that the pressure achieved as the piston moves down is controlled to produce useful work whilst not overheating or overstressing the piston and other engine parts.

A sparking plug (simply a screw-in plug which produces a high-voltage spark in the combustion chamber — where the air/fuel mixture is compressed and burnt) is used to ignite the fuel, with the timing of the spark set by the ignition system. If the spark is too early the pressure rise may force the piston back down the cylinder or may overheat the engine; if too late, then some of the work potential of the burnt air/fuel mixture will be lost. The fuel takes time to ignite (there is a timing delay, between the spark and the mixture starting to burn, known as the ignition delay) and the fuel takes time to burn, thus maximum pressure occurs some time after the spark. As the engine speeds up the spark must be set to occur earlier and earlier to ensure that the peak pressure comes just after the piston has reached its highest point in the cylinder and is on its downward, or power stroke.

In its simplest form, and remember that racing engines are often very simple, the ignition should automatically provide the correct ignition advance as the engine accelerates through its speed range to give maximum power. My simulation is of the compression, burning and power-producing expansion of the air/fuel mixture. The desired ignition advance is found by gradually changing the simulated ignition set-point until a maximum power output is found. This process is repeated for various engine speeds until a full picture is built up of the required ignition characteristics.

Thus I have defined the basic simulation requirement. In order to make the simulation suitably short for inclusion in this book, I have kept the simulation as simple as possible, as explained below. The technique to be adopted for the simulation is obtained by a process of elimination. I cannot use a spreadsheet program as I did in the last chapter, as the detail required would rapidly use up the limited space available on a spreadsheet. Also, the resulting simulation would not be versatile enough to permit future development of the simulation, and the finding of optimum ignition advance would probably have to be done by hand. Thus a special program must be written.

There is no decision-making required within the simulation, as the air/fuel mixture can be taken to follow known laws of fluid dynamics. As the mixture is compressed, burned and expanded we can uniquely define its state at all times; there is no question of assigning probabilities to alternative possible states. Thus the techniques available for handling probabilities, such as Monte Carlo, Markov chain, deterministic and special-purpose, cannot be used. We could possibly use a model which splits up the engine cycle into a small number of phases — eg a compression phase, a burning phase, an expansion phase, etc., with calculations giving the engine state at the end of each phase, depending on the conditions at the start of each phase. This is essentially the approach adopted for hand calculations but is not sufficiently accurate for our needs.

The only approach which I think merits use is to base the simulation on a model of the air/fuel mixture and to step through the engine cycle from start to finish. Here we do have a choice, however; we can step through the cycle using a constant time-step or we can use the rotation of the crankshaft. (See **Figure 7.1** for the relationship between the piston and the crankshaft. The crankshaft is used to convert the linear motion of the piston into rotational motion required for driving wheels.) It is much simpler to base the model on crankshaft position than on time, hence I have chosen to base the model steps on the rotation of the crankshaft.

As I mentioned above, I have had to use a very much simplified model of an engine, and have had to adopt a very simple model of the behaviour of the air/fuel mixture. Those of you who are familiar with engines and fluid thermodynamics will probably realise that the representation, as it stands, can only be used to give a very rough guide to engine performance. The model can be extended and improved quite easily, however.

The model steps through a single engine cycle consisting of compression of the air/fuel mixture, ignition and burning of the mixture, and expansion of the burnt mixture. Other parts of a real engine cycle — expelling the burnt mixture and bringing in a fresh charge of air and fuel — are ignored. A total is calculated for the work output of the engine, which is obtained by continuously calculating the force acting on the piston, multiplied by the distance moved by the piston from one crankshaft position to the next. As the air/fuel mixture is being compressed this work term will be negative, ie the piston must supply work to compress the mixture. As the hot burnt gas expands, the work is positive — work is done on the piston. The total work available at the end of the cycle indicates, when multiplied by the engine speed, the power produced by the engine.

I am not going to explain the program development in any great detail. **Figure 7.2** gives a flowchart for the program (this is the next item to produce once the simulation technique has been selected) and below I give a brief description of each of the program routines. Use these and work

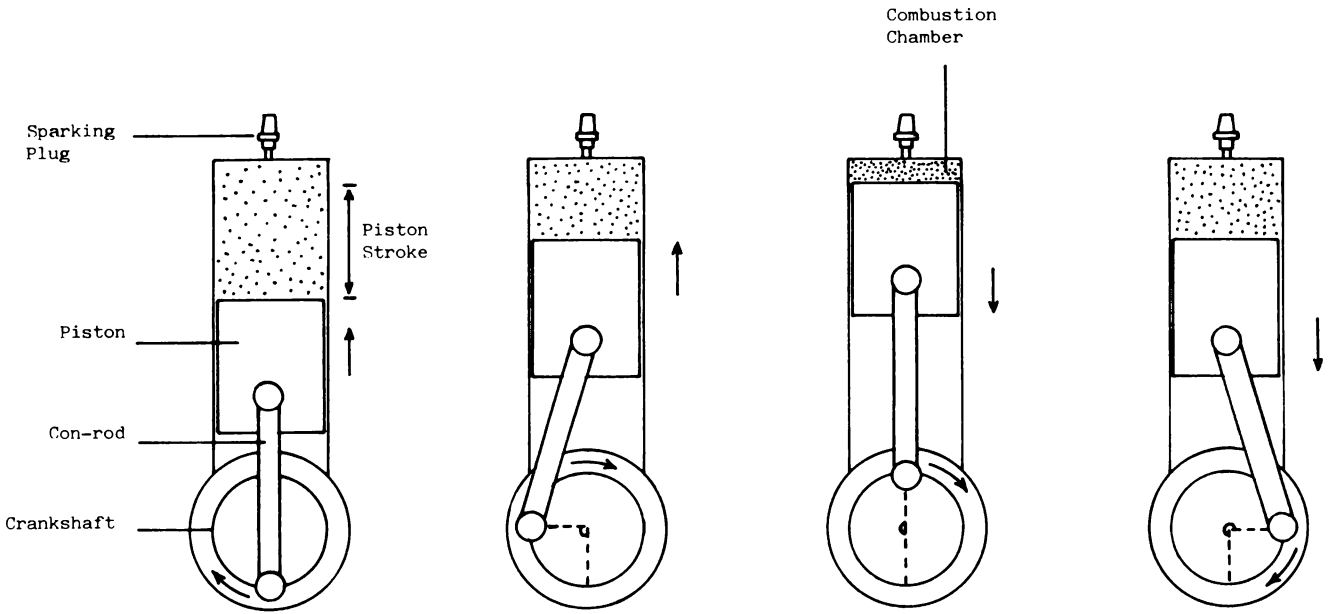


Figure 7.1: Crankshaft Angle and Piston Position.

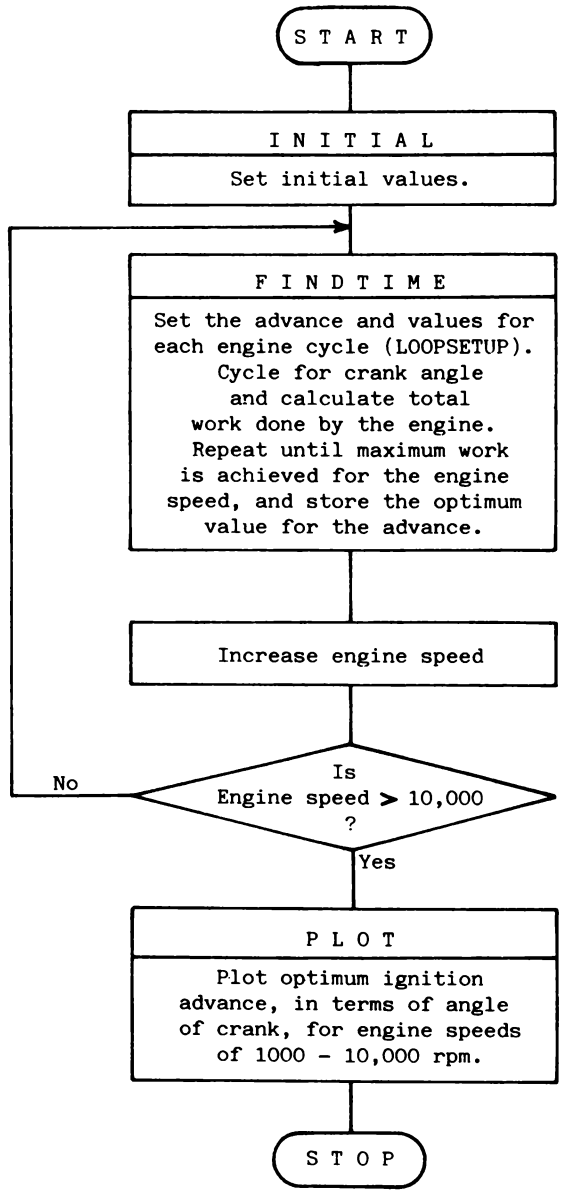


Figure 7.2: Flowchart for the Program 'Ignition'.

through the program listing to see how I have developed the program: you should have no real difficulty. Program variables are described at the end of the listing.

Results

The output from the program (**Figure 7.3**) is very simple. It gives the desired ignition advance, in terms of the crankshaft angle before the piston reaches the top of the cylinder, for engine speeds from 1,000 to 10,000 revolutions per minute.

Basically this is all I need in order to begin to design a full electronic ignition system. In practice, however, I would want to improve the simulation considerably before using the results, and I would want to start investigating factors such as what happens when the air inlet temperature varies (hot and cold weather), what happens if the engine compression ratio is changed, what happens if the ratio of fuel to air is changed, and so on, before deciding what I want from the ignition system.

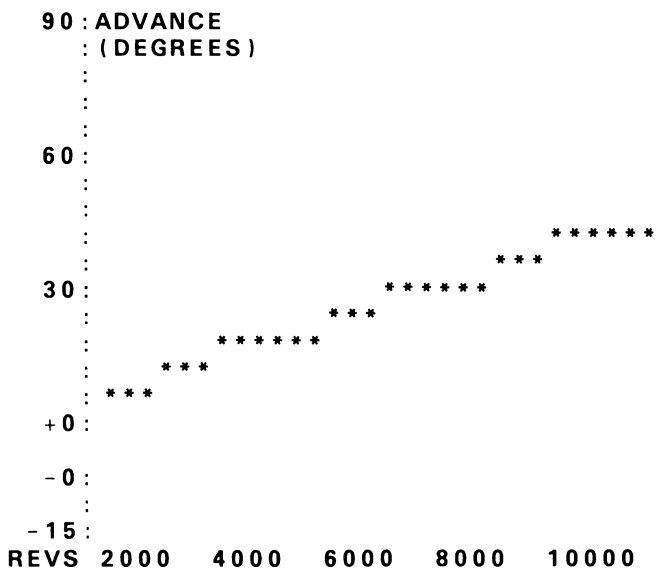


Figure 7.3: Output Produced by Program 'Ignition'.

Program description

Figure 7.2 presents an overall flowchart for the program 'Ignition'.

The routines used in the program have the following functions:

Subroutine INITIAL: Sets the initial values of program variables.

Subroutine FINDTIME: Searches for the optimum ignition advance for each engine speed. This is done by setting the advance, rotating the crankshaft and working out the total work output, and then modifying the ignition advance until an optimum is found. At first the ignition setting is reduced. If a drop in work is recorded then the ignition is progressively advanced and the optimum point is taken as the point just before a second fall in work is indicated.

Subroutine LOOPSETUP: Sets variables for each change of advance, setting the record of old values from the current values, and increasing or decreasing the advance as required.

Subroutine PLOT: Gives a simple graphical display of the results on screen.

FuNction PS(V): Calculates engine internal pressure at a given crankshaft angle.

Program listing

```
1000 REM#========IGNITION==============
1010 REM ENGINE COMBUSTION SIMULATION
1020 REM================================
1030 GOSUB 2000:REM INITIAL
1040 REM LOOP FOR ENGINE SPEED
1050 GOSUB 3000:REM FINDTIME
1060 OA(INT(RV/1000+.5))=OV
1070 RV=RV+1000
1080 IF RV>10000 THEN GOTO 1100:REM EXIT
 SPEED LOOP
1090 GOTO 1040:REM SPEED LOOP
1100 GOSUB 5000:REM PLOT
1110 END
2000 REM#========INITIAL==============
2010 REM SET INITIAL VALUES
2020 REM================================
```

```
2030 DIM OA(10)
2040 PI=3.1415927
2050 CV=500000
2060 CA=0.1075
2070 CB=4.135E-5
2080 PN=101300
2090 TN=273
2100 VM=2.24E-2
2110 R=8.314
2120 KG=35
2130 GA=1.3
2140 DY=3E-4
2150 KR=2.5E-3
2160 RV=1000
2170 SK=5.38E-2
2180 BO=4.82E-2
2190 CP=SK*PI*BO*BO/4
2200 RD=0.1
2210 VC=9.5E-6
2220 OA(0)=2
2230 WK=0
2240 DEF FN PS(V)=R*T1/(V-CB)-CA/(V*V)
2250 RETURN
3000 REM#========FINDTIME==============
3010 REM FIND OPTIMUM ADVANCE
3020 REM================================
3030 AV=OA(INT(RV/1000-0.5))
3040 II=0
3050 REM LOOP FOR TIMING
3060 GOSUB 4000:REM LOOPSETUP
3070 PRINT "[CLR]OPTIMISING IGNITION ADV
ANCE"
3080 PRINT "REVS = ";RV;" ADVANCE = ";AV
;" DEGREES"
3090 REM LOOP TO FIND WORK DONE
3100 LA=5
3110 IF CR<=218 AND CR>=140 THEN LA=2
3120 CR=CR+LA
3130 IF CR>360 THEN GOTO 3330:REM EXIT W
ORK LOOP
3140 V0=V1
3150 T0=T1
3160 P0=P1
```

```
3170 V2=CR*2*PI/360
3180 V3=SQR(RD*RD-(SK/2*SIN(V2))^2)-SK/2
*COS(V2)-(RD-SK/2)
3190 V1=(CP+VC-(PI*BO*BO/4)*V3)
3200 BH=KG*BU
3210 VI=(V0/V1)^(GA-1)
3220 T1=(T0+BH)*VI
3230 V=V1*VM/(CP+VC)
3240 P1=FN PS(V)
3250 WK=WK+((P0+P1)/2*(V1-V0))
3260 IF CR<(BA+DY*RV*360/60) THEN BU=0:G
OTO 3310
3270 IF RA=0 THEN BU=0:GOTO 3310
3280 RT=KR*CV*CP*LA/(RV*V1)
3290 IF RA>RT THEN BU=RT:GOTO 3310
3300 BU=RA
3310 RA=RA-BU
3320 GOTO 3090:REM WORK LOOP
3330 B=II
3340 IF II=0 THEN B=1
3350 IF II=1 AND WK>OW THEN B=2
3360 IF II=1 AND WK<=OW THEN B=3
3370 IF (II=2 OR II=3) AND WK<OW THEN RE
TURN:REM TO SPEEDLOOP
3380 II=B
3390 GOTO 3050:REM TIMING LOOP
4000 REM#========LOOPSETUP============
4010 REM SET VALUES FOR EACH ADVANCE
4020 REM==============================
4030 CR=0
4040 BU=0
4050 V1=CP+VC
4060 T1=TN
4070 P1=PN
4080 OW=WK
4090 WK=0
4100 OV=AV
4110 DV=5
4120 IF II=3 AND (AV>=-40 AND AV<=38)THE
N DV=2
4130 IF II=3 AND (AV<=40 AND AV>=-38)THE
N DV=2
4140 IF II=1 OR II=2 THEN AV=AV-DV
```

```
4150 IF II=3 THEN AV=AV+DV
4160 BA=180-AV
4170 RA=CV*CP
4180 RETURN
5000 REM#========PLOT=================
5010 REM PRINT RESULTS
5020 REM================================
5030 PRINT "[CLR]"
5040 PRINT "  90:ADVANCE"
5050 PRINT "    :(DEGREES)"
5060 FOR N=1 TO 16
5070 IF N=3 THEN PRINT "  60";:GOTO 5130
5080 IF N=8 THEN PRINT "  30";:GOTO 5130
5090 IF N=13 THEN PRINT "  +0";:GOTO 5130
5100 IF N=14 THEN PRINT "  -0";:GOTO 5130
5110 IF N=16 THEN PRINT "-15";:GOTO 5130
5120 PRINT "    ";
5130 PRINT ":";
5140 FOR M=1 TO 10
5150 A=72-6*N
5160 B=A+5
5170 C=OA(M)
5180 IF C>=A AND C<=B THEN PRINT "***";
5190 IF C>=-25 AND C<=A-1 THEN PRINT "
  ";
5200 IF C>=B+1 AND C<=73 THEN PRINT "    ";
5210 NEXT M
5220 PRINT
5230 NEXT N
5240 PRINT " REVS 2000  4000  6000  8000
 10000"
5250 RETURN
6000 REM VARIABLES====================
6010 REM AV=IGNITION TIMING ANGLE BEFORE
TOP OF PISTON CYCLE
6020 REM BA=IGNITION TIMING ANGLE AFTER
BOTTOM OF PISTON CYCLE
6030 REM BH=BURNHEAT (TEMP RISE DUE TO
FUEL BURNING)
6040 REM BO=ENGINE BORE
6050 REM BU=RATE OF INCREASE IN GAS TEMP
BASED ON RATE OF FUEL BURN
6060 REM CA=CONSTANT IN VAN DER WAALS
```

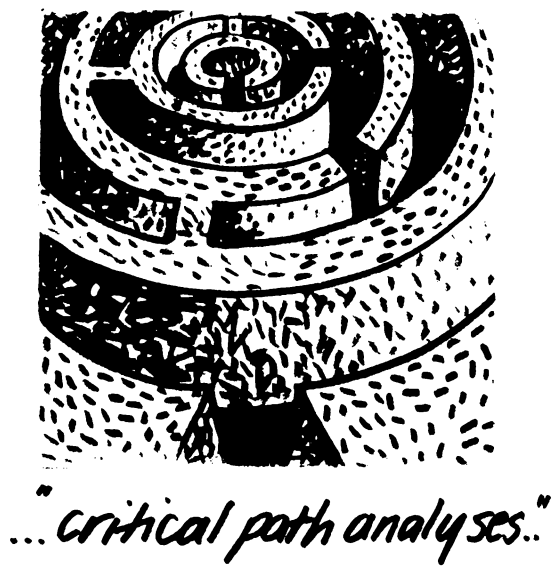
```
EQUATION
6070 REM CB=CONSTANT IN VAN DER WAALS
EQUATION
6080 REM CP=ENGINE CAPACITY
6090 REM CR=ANGLE OF CRANK
6100 REM CV=CALORIFIC DENSITY OF AIR/
FUEL MIXTURE AT INLET CONDITIONS
6110 REM DV=DELTA ADVANCE
6120 REM DY=IGNITION DELAY
6130 REM GA=RATIO OF SPECIFIC HEATS
6140 REM II=INDICATOR
6150 REM  0-1ST TIME ROUND FINDTIME LOOP
6160 REM  1-SECOND TIME ROUND LOOP
6170 REM  2-CONTINUE REDUCING ADVANCE
6180 REM  3-CONTINUE INCREASING ADVANCE
6190 REM KG=CONSTANT RELATING TEMP RISE
TO ENERGY FROM FUEL
6200 REM KR=CONSTANT
6210 REM LA=ANGLE CRANK MOVES THROUGH IN
ONE INCREMENT
6220 REM LW=WORK DONE IN ENGINE IN 1
CYCLE
6230 REM NV=NEW VOLUME OF AIR/FUEL
MIXTURE
6240 REM OA(10)=OPTIMUM ADVANCE AT REVS
6250 REM OV=LAST VALUE OF ADVANCE
6260 REM OW=LAST VALUE OF WORK
6270 REM P0=OLD GAS PRESSURE
6280 REM P1=NEW GAS PRESSURE
6290 REM PN=INITIAL PRESSURE
6300 REM R=UNIVERSAL GAS CONSTANT
6310 REM RA=REMAINDER OF AIR/FUEL TO BE
BURNT
6320 REM RD=LENGTH OF CON ROD
6330 REM RT=RATE OF BURNING FUEL
6340 REM RV=ENGINE SPEED RPM
6350 REM SK=ENGINE STROKE
6360 REM T0=OLD GAS TEMPERATURE
6370 REM T1=NEW GAS TEMPERATURE
6380 REM TN=INITIAL TEMPERATURE
6390 REM V0=VOLUME OF GAS OLD
6400 REM V1=VOLUME OF GAS NEW
6410 REM V2,V3=USED TO CALCULATE NV
```

```
6420 REM V=VARIABLE USED IN FN PS
6430 REM VC=VOLUME OF COMBUSTION CHAMBER
6440 REM VI=TEMP CHANGE INDEX DUE TO
CHANGE IN VOLUME
6450 REM VM=VOLUME OF ONE MOLE AT STP
6460 REM WK=WORK DONE
```

CHAPTER 8

What We Want Is Information

What do you want to know?



There is a significant role for simulations in the provision of information. The bulk of such information applications are in the area of project planning and management support. This chapter considers the use of simulations in this role.

Background

It is well recognised that the information business is big business. Newspapers, television, publishing, reporting: these represent the public side of a business that has continued to grow at a sustained high rate for the past 50 years. There is another, less well known, aspect which has been highlighted recently by the introduction of small computers into business. This is the provision of information services for management.

The management of modern companies is not usually based on the old idea of a single boss or small board making many day-to-day decisions and simply handing down the results of these decisions as actions to be carried out by employees. Such companies do still exist but tend to be small or in financial difficulties. In order to respond flexibly to the demands of a changing world, it is necessary for a larger proportion of any company to have a direct understanding of management aims and to have a responsibility for making management decisions. This approach can only work if good communications exist and information systems are built up.

Computer systems often provide the means for building up information facilities. This does not, however, mean that there is an important role for simulations. Requirements vary from company to company and different management groups work in different ways. Various financial and commercial models may be used but are not common. Representations of the internal workings of a company are becoming popular, usually to keep track of manpower and stock, but few of these use simulation techniques as yet.

The major area of interest for the implementation of simulations is currently in project management. This involves the long-term planning of programmes of work and the implementation of such plans on a day-to-day basis. In this application, the system being simulated is the project itself and the techniques used for the simulation range from simple critical-path analyses to highly sophisticated resource-optimisation programs.

Example

My example for this chapter is a simple project planning task to be carried out using a general-purpose program. The program is a basic critical-path analysis program which builds up a model of the whole project, identifies those parts of the project which are most important, and estimates an overall manpower requirement.

I could have chosen from a number of everyday examples for this chapter, including preparing for a party, a year plan for a garden, writing a book, or re-decorating a kitchen. Almost anything which requires forward planning, in fact. It is more usual, however, to apply critical-path analysis to business problems and I have selected a simple manufacturing process as my example.

Lookitthat Ltd. manufacture kites. The company is small (normally no more than six employees, two of whom are part-time) but devotes a high proportion of its effort to the development of new designs. Before going into full production with a new kite design, Lookitthat Ltd. usually prepares a special batch of 20 kites in order to test out production methods, to give them a small stock to show their retailers, and for use as test models. Such small batches are often used by manufacturers and are known as pre-production prototypes.

TASK	TITLE OF TASK (20 CHARACTERS)	NO OF TIME STEPS	MANPOWER REQUIREMENT	PRECEDING TASKS TO BE COMPLETED
A	Cut material	7	2	—
B	Stitch material	13	1	A
C	Cut struts	4	1	—
D	Cut tails	2	1	—
E	Wind tails	2	1	D
F	Cut control strings	1	1	—
G	Wind control strings	2	1	F
H	Fold kites	3	2	B
I	Collect components	4	1	H G E C
J	Package	3	2	I

No of tasks = 10
Unit of time = 15 minutes
Total manpower available = 2

Figure 8.1: List of Tasks Required to Manufacture 20 Pre-production Prototypes for Lookitthat Ltd.

Two people are available to make the kites and both can complete any of the tasks required. **Figure 8.1** gives a list of the tasks. What we want to know is: how long will the manufacture take, which are the important tasks (the critical path), and how much of the available effort is actually used.

Simulation development

We are going to use a general-purpose program, so we will not go through the full selection technique in this case. What I shall do, however, is des-

cribe the program itself, how to use it, and how it might be extended.

The program 'Critical', takes information on a number of tasks and works out a plan for completing the tasks within a given manpower limitation. Output consists of the plan itself, the actual manpower requirement, and an indication of those tasks which are critical to complete the planned work on time. Any delays occurring on one of these critical tasks will result in a delay in the project completion time.

Several basic approaches to the writing of a critical-path model are possible. The differences are principally concerned with the way in which the project plan is built up and also the control over this process desired by the user. Tasks may be given different relative priorities, or particular start dates. Resources other than manpower may be added into the calculation, which must be kept within pre-set limits. It may be required to smooth out the use of certain resources, including manpower, as much as possible. Another common requirement is to build in an allowance for problems and delays and to have the program indicate just how sensitive the project plan is to such things.

Our program is a basic one, however, albeit with a capability for extension. The core of the program is a routine which builds up the project plan. This is achieved by initially considering all tasks, discounting those which cannot be commenced yet as other tasks on which they are dependent are still to be completed, and selecting the task which is of the longest duration from those which can be started within the manpower currently available. The principle behind selecting the longest-duration task is that such a task is most likely to be on the critical path and thus should not be delayed. A more sophisticated approach might analyse the forward dependency of tasks, ie establish which tasks in the future will be dependent on current tasks, and, using this information, derive a relative priority for tasks. As I have already indicated, though, this is a basic program lacking many of the possible sophistications. If spare effort is available then additional tasks may be selected, on the same basis of priority given to long duration.

The duration of tasks is input as a multiple of some fixed base unit, which will be 15 minutes in our example. In building up the project plan, the program steps through the project using the base-unit-of-time. This time unit is also used to display the completed project plan. The display shows all of the input tasks, their relative position in the plan, and indicates if they are on the critical path by marking their duration with a C. Tasks not on the critical path are marked with '*'.

Results

Figure 8.2 is the output provided by the critical path program.

```
TASK (PAGE 1;0 TO 450 MINUTES)

A : CCCCCCC
B :         CCCCCCCCCCCCC
C :         * * * *
D :             * *
E :               * *
F :                 *
G :                   * *
H :                     CCC
I :                       CCCC
J :                           CCC

SCALE IS ONE CHARACTER = 15 MINUTES
PROJECT LENGTH = 450 MINUTES
MANPOWER UTILISATION = .9 x 2
```

Figure 8.2: Production Plan Critical-Path Analysis.

Tasks A, B, H, I, and J provide the critical path for this particular project. A total time of 7.5 hours is indicated for the production test run, just right for a full day's work. Also, the utilisation (the proportion of time for which the manpower available is actually used) is quite high, and little benefit can be gained from playing around with the plan.

Note that Task I, collecting together the components in preparation for final packaging, is marked as a single task dependent on the completion of all manufacturing tasks. In reality, of course, this task could be spread out and time saved. This task should really be broken down into sub-tasks.

Program description and use

Figure 8.3 presents an overall flowchart for the program 'Critical'.

Definitions of the variables used in the program are given in REM statements at the end of the progam listing.

The routines used in the program have the following functions:

Subroutine INITIAL: Sets up dimension statements and initialises variables.

Subroutine DATAIN: Inputs data on the project, including how many tasks there are, the time-step to be used for the project plan, total manpower available, the title of each task along with task duration and manpower requirement. Initial data on each task is also set.

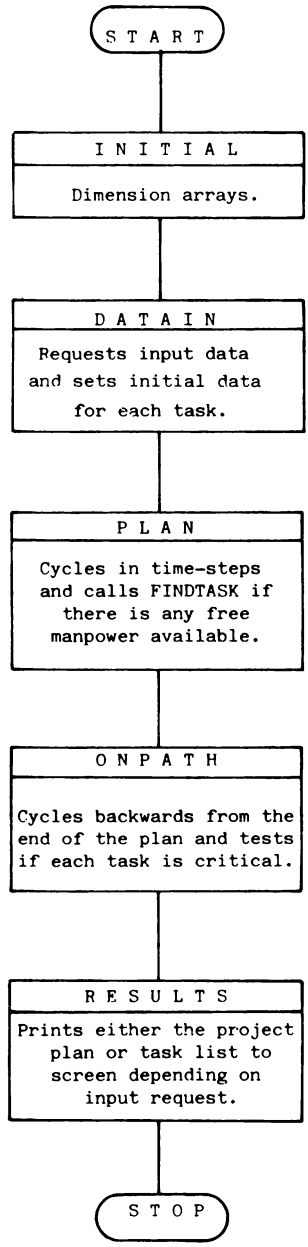


Figure 8.3: Flowchart for the Program 'Critical'.

Subroutine PLAN: Steps through the project plan one time-step at a time and gradually extends the project plan. Tasks are started as early as possible within the manpower available.

Subroutine ONPATH: This takes a completed plan and works backwards through it, considering each task in turn. For each task, a check is made for other dependent tasks starting immediately after the current task ends. If no such dependent task exists and manpower is available, the task is moved on one time-step. This process is repeated until the task can be moved no further. All tasks are tested in this way. Those which cannot be moved lie on the critical path and IT(n) is set to 4.

Subroutine RESULTS: This prints the results to screen. The user can choose between a listing of the project tasks or a diagram of the project plan itself. If the project plan is too long to be shown on a single screen, then several 'pages' are displayed.

Subroutine REVISE: At the start of a new project time-step, this routine checks to see if tasks previously in progress have ended, and marks them as completed, using the indicator variable I(n), if so. The routine then checks all the tasks which are dependent on the completion of previous tasks and determines whether or not these can now be started.

Subroutine TESTEND: Tests for the completion of the project plan. Returns TD with a value of 0 if so.

Subroutine FINDTASK: If some manpower is available, then a new task is selected (longest duration given priority) from those tasks remaining.

Subroutine DEFFORT: Returns a total estimated manpower requirement for all tasks assuming latest possible start-time (DF).

Subroutine EFFORT: Returns a total manpower figure (EF) for tasks in progress.

Program use is relatively straightforward. The program works in a linear fashion, taking a full set of input, producing a project plan, and providing a set of results.

To run the example problem, simply enter the data provided in Figure 8.1. Several minutes are required to run the complete program.

Program improvements

'Critical' obviously has much room for improvement. As it stands, the

program is not very 'user-friendly' — the raw data is not stored and cannot be modified once entered, mistakes on entering data can cause the program to stop with an error message, the linear data entry used does not encourage manipulation of the raw data to answer 'what if?' type questions, hardcopy of the input data and the resultant project plan is not available, and the limit of 15 tasks (chosen to simplify the subroutine RESULTS) restricts practical applications.

Replacing the current main program with a menu-driven program could provide a means of providing additional facilities. Defined processes for tape, disk and printer operations and for updating data could easily be added.

Also, notice how I have included a variable AI(1,n) which stores the starting time of a task. This is currently calculated as part of the project planning process. It could equally well be input by the user, thus giving more control over the project plan.

A final facility which I think it would be useful to add is a means of defining a relative priority of tasks. This would force, or encourage the program (depending on how the facility is implemented) to give some tasks priority over others.

The addition of the facilities noted above would allow the user to enter data in a simple form, produce a first project plan, and modify the plan as desired to suit requirements.

Program listing

```
1000 REM#========CRITICAL==============
1010 REM CRITICAL PATH PROGRAM
1020 REM================================
1030 GOSUB 2000:REM INITIAL
1040 GOSUB 3000:REM DATAIN
1050 GOSUB 4000:REM PLAN
1060 GOSUB 5000:REM ONPATH
1070 GOSUB 6000:REM RESULTS
1080 STOP
2000 REM#========INITIAL===============
2010 REM DIMENSION ARRAYS
2020 REM================================
2030 DIM IT(15)
2040 DIM AI(4,15)
2050 DIM A$(2,15)
2060 O$="":U$=""
2070 RETURN
3000 REM#========DATAIN================
```

```
3010 REM INPUT DATA
3020 REM================================
3030 REM LOCAL VARS N,M,B
3040 PRINT "[CLR]":INPUT "HOW MANY TASKS
 ARE THERE";TL
3050 PRINT "DO YOU WANT TO PLAN IN"
3060 INPUT "(D)AYS, (H)OURS, OR (M)INUTE
S";T$
3070 IF T$="D" THEN U$="DAYS"
3080 IF T$="H" THEN U$="HOURS"
3090 IF T$="M" THEN U$="MINUTES"
3100 IF U$="" THEN GOTO 3060:REM ERROR O
N INPUT
3110 PRINT "[CLR]HOW MANY ";U$;
3120 INPUT " IN ONE TIME-STEP";LN
3130 INPUT "[CLR]WHAT IS THE MANPOWER AV
AILABLE";MP
3140 FOR N=1 TO TL
3150 PRINT "[CLR]INPUT TITLE OF TASK ";C
HR$(N+64)
3160 PRINT " (UP TO 20 CHARACTERS)"
3170 INPUT A$(1,N)
3180 PRINT:PRINT "HOW MANY TIME-STEPS OF
 ";LN;" ";U$
3190 PRINT "IS TASK ";CHR$(N +64);
3200 INPUT AI(2,N)
3210 PRINT:PRINT "WHAT IS THE MANPOWER R
EQUIREMENT"
3220 PRINT "FOR TASK ";CHR$(N+64);
3230 INPUT AI(3,N)
3240 REM LOOP FOR DEPENDENT TASKS
3250 PRINT:PRINT "WHAT TASKS IS TASK ";C
HR$(N+64);" DEPENDENT UPON"
3260 PRINT "(INPUT STRING OF LETTERS THE
N ENTER)":INPUT A$(2,N)
3270 B=LEN(A$(2,N))
3280 IF B=0 THEN GOTO 3330:REM NO DEPEND
ENT TASKS
3290 FOR M=1 TO B
3300 C$=MID$(A$(2,N),M,1)
3310 IF ASC(C$)<65 OR ASC(C$)>64+TL THEN
 GOTO 3240:REM ERROR-REPEAT LOOP
3320 NEXT M
```

```
3330 AI(4,N)=0
3340 AI(1,N)=0
3350 IT(N)=0
3360 NEXT N
3370 RETURN
4000 REM# =======PLAN==================
4010 REM DEVELOP PROJECT PLAN
4020 REM================================
4030 PRINT "[CLR]PLANNING"
4040 UT=0
4050 TM=-1
4060 REM LOOP THROUGH PLAN
4070 TM=TM+1
4080 GOSUB 7000:REM REVISE
4090 GOSUB 8000:REM TESTEND
4100 IF TE=0 THEN GOTO 4200:REM EXIT LOO
P
4110 REM LOOP STARTTASK
4120 GOSUB 11000:REM EFFORT
4130 MF=MP-EF
4140 IF MF=0 THEN 4180:REM EXIT LOOP
4150 GOSUB 9000:REM FINDTASK
4160 IF CT=0 AND MF<>0 THEN 4180:REM EXI
T LOOP
4170 GOTO 4110:REM STARTTASK LOOP
4180 UT=UT+(MP-MF)
4190 GOTO 4060:REM PLAN LOOP
4200 PD=TM-1
4210 UT=UT/(MP*(PD+1))
4220 RETURN
5000 REM#========ONPATH===============
5010 REM SEE IF TASKS ARE CRITICAL
5020 REM================================
5030 REM LOCAL VARS N4,M4,P1,Q
5040 PRINT "[CLR]FINDING CRITICAL PATH"
5050 FOR N4=1 TO TL
5060 AI(4,N4)=AI(1,N4)
5070 NEXT N4
5080 M4=TL+1
5090 REM BACKWARDS LOOP
5100 M4=M4-1
5110 IF M4=0 THEN 5270:REM EXIT BACKWARD
S LOOP
```

```
5120 N4=(ASC(MID$(O$,M4,1)))-64
5130 REM LOOP FOR MOVETASK
5140 TM=AI(4,N4)+AI(2,N4)
5150 IF TM>PD THEN 5260:REM EXIT MOVETAS
K
5160 FOR P1=1 TO TL
5170 IF AI(4,P1)<>TM OR LEN(A$(2,P1))<=0
 THEN GOTO 5210:REM NEXT P1
5180 FOR Q=1 TO LEN(A$(2,P1))
5190 IF ASC(MID$(A$(2,P1),Q,1))=N4+64 TH
EN 5260:REM EXIT MOVETASK
5200 NEXT Q
5210 NEXT P1
5220 GOSUB 10000:REM DEFFORT
5230 MF=MP-DF
5240 IF AI(3,N4)>=MF THEN AI(4,N4)=AI(4,
N4)+1
5250 GOTO 5130:REM MOVETASK LOOP
5260 GOTO 5090:REM BACKWARDS LOOP
5270 FOR N4=1 TO TL
5280 IF AI(1,N4)=AI(4,N4) THEN IT(N4)=4
5290 NEXT N4
5300 RETURN
6000 REM#========RESULTS==============
6010 REM PRINT RESULTS
6020 REM================================
6030 REM LOCAL VARS V,N6,M6,P3,Q1,S,W,R
6040 REM RESLOOP
6050 INPUT "[CLR]ENTER P FOR PROJECT PLA
N, T FOR LIST OF TASKS ";T$
6060 V=ASC(T$)
6070 IF V=80 THEN GOTO 6140:REM PRINT PL
AN
6080 REM PRINT LIST OF TASKS
6090 PRINT "[CLR]":FOR N6=1 TO TL
6100 PRINT "TASK ";CHR$(N6+64);" = ";A$(
1,N6)
6110 NEXT N6
6120 GOTO 6500
6130 REM PRINT PLAN
6140 PRINT "[CLR]":P3=0
6150 REM LOOP FOR PAGEPLAN
6160 P3=P3+1
```

```
6170 M6=30*(P3-1)
6180 Q1=M6+29
6190 IF Q1>PD THEN Q1=PD
6200 PRINT "[CLR]TASK   (PAGE ";P3;"; ";
M6*LN;" TO ";(Q1+1)*LN;" ";U$;")"
6210 PRINT
6220 FOR N6=1 TO TL
6230 E$=" "+CHR$(N6+64)+":"
6240 S=AI(1,N6)+AI(2,N6)-1
6250 W=AI(1,N6)
6260 IF W>=0 AND W<=M6 THEN R=0
6270 IF W>=M6+1 AND W<=Q1 THEN R=W-M6
6280 IF W<0 OR W>Q1 THEN R=-1
6290 IF S>=0 AND S<=M6 THEN W=-1:R=-1
6300 IF S>=M6+1 AND S<=Q1 THEN W=S-M6
6310 IF S<0 OR S>Q1 THEN W=Q1-M6
6320 IF (S<0 OR S>Q1) AND R=-1 THEN W=-1
6330 S=W
6340 W=IT(N6)
6350 IF W=3 THEN C$="*"
6360 IF W=4 THEN C$="C"
6370 IF R<=0 THEN GOTO 6390
6380 FOR W=1 TO R:E$=E$+" ":NEXT W
6390 IF R<>-1 THEN E$=E$+C$
6400 IF S<=R THEN GOTO 6420
6410 FOR W=R TO S-1:E$=E$+C$:NEXT W
6420 PRINT E$
6430 NEXT N6
6440 PRINT "SCALE IS ONE CHARACTER = ";L
N;" ";U$
6450 PRINT "PROJECT LENGTH = ";(PD+1)*LN
;" ";U$
6460 PRINT "MANPOWER UTILISATION = ";0.0
1*INT(UT*100+.5);"X";MP
6470 IF Q1>=PD THEN GOTO 6500
6480 INPUT "PRESS RETURN FOR NEXT PAGE O
F PLAN";E$
6490 GOTO 6150:REM NEXT PAGE
6500 INPUT "PRESS RETURN TO CONTINUE, O
R E TO EXIT   PROGRAM";E$
6510 IF E$="E" THEN GOTO 1080:REM STOP
6520 GOTO 6040:REM REPEAT RESLOOP
6530 RETURN
```

```
7000 REM#========REVISE===============
7010 REM REVISE STATUS OF TASKS IT()
7020 REM================================
7030 REM LOCAL VARS N2,M2,K,P
7040 FOR N2=1 TO TL
7050 IF IT(N2)=2 AND AI(1,N2)+AI(2,N2)-1
<TM THEN IT(N2)=3
7060 NEXT N2
7070 FOR N2=1 TO TL
7080 IF IT(N2)<>0 THEN 7180:REM NEXT N2
7090 K=1:M2=0
7100 REM LOOP DEPENDS
7110 M2=M2+1
7120 IF LEN(A$(2,N2))=0 THEN K=1:GOTO 71
70:REM EXIT LOOP
7130 IF M2>LEN(A$(2,N2)) THEN GOTO 7170:
REM EXIT LOOP
7140 P=ASC(MID$(A$(2,N2),M2,1))
7150 IF IT(P-64)<>3 THEN K=0
7160 GOTO 7100:REM DEPEND LOOP
7170 IT(N2)=K
7180 NEXT N2
7190 RETURN
8000 REM#========TESTEND==============
8010 REM TEST FOR END OF PLAN
8020 REM================================
8030 REM LOCAL VAR N1
8040 TE=0
8050 FOR N1=1 TO TL
8060 IF IT(N1)<>3 THEN TE=1
8070 NEXT N1
8080 RETURN
9000 REM#========FINDTASK=============
9010 REM SELECT NEXT TASK
9020 REM================================
9030 REM LOCAL VARS P2,N3,M3
9040 P2=0:M3=0:CT=0
9050 FOR N3=1 TO TL
9060 IF IT(N3)=1 AND AI(3,N3)<=MF AND AI
(2,N3)>M3 THEN M3=AI(2,N3):P2=N3:CT=1
9070 NEXT N3
9080 IF P2<>0 THEN IT(P2)=2:AI(1,P2)=TM:
O$=O$+CHR$(P2+64)
```

```
9090 RETURN
10000 REM#======= DEFFORT==============
10010 REM TOTAL MANPOWER REQUIREMENT
10020 REM==============================
10030 REM LOCAL VARS N5,M5
10040 M5=0
10050 FOR N5=1 TO TL
10060 IF TM>=AI(4,N5) AND TM<=(AI(4,N5)+
AI(2,N5)-1) THEN M5=M5+AI(3,N5)
10070 NEXT N5
10080 DF=M5
10090 RETURN
11000 REM#========EFFORT==============
11010 REM MANPOWER FOR CURRENT TASK
11020 REM======= ======================
11030 REM LOCAL VARS N7,M7
11040 M7=0
11050 FOR N7=1 TO TL
11060 IF IT(N7)=2 THEN M7=M7+AI(3,N7)
11070 NEXT N7
11080 EF=M7
11090 RETURN
12000 REM VARIABLES==================
12010 REM A$(1,15)=TITLES OF TASKS
12020 REM A$(2,15)=RECORD OF TASKS WHICH
12030 REM MUST BE  COMPLETED BEFORE
12040 REM CURRENT TASK
12050 REM AI(1,15)=START OF TASK (UNITS
12060 REM OF TIME FROM START OF PROJECT)
12070 REM AI(2,15)=DURATION OF TASK
12080 REM (UNITS OF TIME)
12090 REM AI(3,15)=MANPOWER REQUIREMENT
12100 REM OF TASK (PEOPLE)
12110 REM AI(4,15)=LATEST START TIME
12120 REM POSSIBLE FOR TASK NOT DELAYING
12130 REM PROJECT END
12140 REM CT=INDICATOR (T=0 THEN CAN'T
12150 REM    START NEW TASK)
12160 REM DF=TOTAL ESTIMATED MPOWER REQD
12170 REM EF=MANPOWER FOR CURRENT TASK
12180 REM IT(15)=INDICATOR OF TASK
12190 REM         STATUS AS PROGRAM
12200 REM         STEPS THROUGH PLAN
```

```
12210 REM    0-TASK WAITING FOR END OF
12220 REM      OTHER TASKS
12230 REM    1-TASK WAITING FOR EFFORT
12240 REM    2-TASK IN PROGRESS
12250 REM    3-TASK COMPLETE
12260 REM    4-TASK ON CRITICAL PATH
12270 REM LN=LENGTH OF TIMESTEP
12280 REM MF=MANPOWER NOT YET ALLOCATED
12290 REM MP=MAXIMUM AVAILABLE MANPOWER
12300 REM O$=RECORDS ORDER OF TASKS AS
12310 REM    PLAN IS BUILT UP
12320 REM PD=END OF PROJECT PLAN
12330 REM T$=INPUT CHOICE FOR TASK LIST
12340 REM    OR PLAN
12350 REM TE=TEST END OF PLAN 0=END
12360 REM TL=TOTAL NO OF TASKS (TO 15)
12370 REM TM=CURRENT TIME
12380 REM U$=UNITS FOR TIMESTEP
12390 REM UT=OVERALL UTILISATION OF
12400 REM    AVAILABLE MANPOWER
12410 REM PLOT VARIABLES
12420 REM E$=HOLDS SINGLE TASK PLOT & MISC.
12430 REM M6=START TIME FOR PLOT
12440 REM P3=PAGE NO
12450 REM Q1=END TIME FOR PLOT
12460 REM R=START TIME FOR TASK
12470 REM S=END TIME FOR TASK
```

CHAPTER 9

And Out of the Unknown . . .

Simulating the unknown



How is it possible to simulate the unknown?

Background

Nearly all simulations involve elements of the unknown. One of the major classes of simulation, however, as described in Chapter 3, is the analytical simulation which is designed to investigate the unknown characteristics of a system. Analytical simulations take known features of the system in question and, by making various assumptions about the workings of the system, fill in the unknown areas to give a practical working model of the system.

Analytical simulations of unknown system characteristics are extremely experimental in nature: although you may have theories about how the simulation will work, it is only when you actually start to test the thing out that you begin to verify and extend those theories. Trial and error is the order of the day, and this sets the conditions under which a technique can be adopted. A highly detailed emulation technique is most unlikely to be useful if the unknown parts of the system are a significant part of the whole. It is better to start off with a simple simulation which models the major system responses only and gradually to extend this into a more complex model as you gain knowledge of the system.

Another well-used approach is to use knowledge of other similar systems to provide a model of the system to be investigated, and thus to predict its characteristics. For example, if you are the manager of a Formula 1 car racing team and wish to analyse the performance of a competitor's car, then probably the best starting point would be to model your own car first and use this as a basis for your evaluation. A further example of the use of existing knowledge to predict the characteristics of something new is the development and use of commercial expert systems. These are specifically designed to build up a store of knowledge in such a way that new systems with unknown characteristics can be emulated and described.

Example

One of the more interesting examples from my case-book is a simple simulation which attempts something which I have described, in previous chapters, as extremely difficult and unreliable. The interest comes not only from the practical aspect of the simulation, but also in the techniques used.

Over the past few years I have worked on several projects for the famous kite makers, Lookitthat Ltd. Their research department — consisting of the company founder, Ivor Problem, and his secretary, Sophie Aye — were having difficulties with their hardware development because they could never predict whether or not there would be sufficient wind to try out their latest kites. They were having trouble planning their work around our

changeable weather and asked for a method for predicting the weather tomorrow given a simple description of today's weather.

My solution to their problem, which I originally designed for use on the Sinclair ZX81 but which becomes much more versatile on the Commodore 64, is a simple simulation of the weather changes from day to day, with a prediction of the likelihood that the weather tomorrow will be suitable for kite flying.

Simulation development

The requirement is for a model of weather conditions which vary from day to day, which will indicate the likely conditions one day ahead. Obviously, I cannot expect Ivor Problem to spend a lot of time taking wind and temperature readings or constantly phoning the local meteorological office for information — data input must be simple. Also, I should be able to give Ivor a direct indication as to how probable it is that he will be able to fly his kites — output must be simple.

Any of the techniques mentioned in Chapter 4 for modelling probabilities could be adopted. I chose a Markov chain approach as the basic means of predicting weather changes (see Chapter 4), because of the relative simplicity of the model itself and the ease of running the developed simulation. The system-state matrix is based on a very much simplified description of the weather, concentrating on three fundamental attributes of temperature, wind strength, and rainfall (or other forms of precipitation, eg snow). The most important attributes from Ivor's point of view are wind and rain. I also include temperature, as this gives an easily understood indication of the general weather conditions, much easier than air pressure or humidity. Another possibility which I have not included, in order to keep the input as simple as possible, is the percentage cloud cover. I list below the possible weather attributes and the resultant possible system-states. Notice how I have chosen descriptions for the various weather conditions which are immediately understandable by Ivor and useful to him.

WEATHER ATTRIBUTE	DESCRIPTION	LABEL
WIND	Too light for kite flying	W0
	OK for kite flying	W1
	Too strong for kite flying	W2
RAIN	None: OK for kites	R0
	Light or infrequent: OK	R1
	Heavy:No good for kites	R2
TEMPERATURE	Hot: OK	T0
	Intermediate:OK	T1
	Cold: Rather stay at home	T2

SYSTEM-STATE	ATTRIBUTES	SYSTEM-STATE	ATTRIBUTES
0	W0 R0 T0	14	W1 R1 T2
1	W0 R0 T1	15	W1 R2 T0
2	W0 R0 T2	16	W1 R2 T1
3	W0 R1 T0	17	W1 R2 T2
4	W0 R1 T1	18	W2 R0 T0
5	W0 R1 T2	19	W2 R0 T1
6	W0 R2 T0	20	W2 R0 T2
7	W0 R2 T1	21	W2 R1 T0
8	W0 R2 T2	22	W2 R1 T1
9	W1 R0 T0	23	W2 R1 T2
10	W1 R0 T1	24	W2 R2 T0
11	W1 R0 T2	25	W2 R2 T1
12	W1 R1 T0	26	W2 R2 T2
13	W1 R1 T1		

The weather conditions which are suitable for kite flying are represented by system-states 9, 10, 12 and 13.

But what about the transition matrix? If it's warm, dry, and there's a steady wind blowing today, what will it be like tomorrow? The transition matrix is a collection of probabilities of the weather changing from its current state today into one of the 27 possible states tomorrow. The problem is that we don't know what these probabilities are. My solution to this problem is to give the simulation a capability to learn as more and more data is input, and thus build up a store of knowledge. This is where the analysis part of the program resides, the simulation builds up a model of the weather. In this way the program also works like an extremely simple expert system — the more information you give it, the more expert it becomes (up to a point).

Two more points must be covered: we must set up initial values (guesses) for the transition matrix and we must decide what we want in terms of input/output.

Initial values for the transition matrix could be determined in a number of different ways and then input manually. I have decided that it would be better to give the simulation a running start and attempt to base the initial transition matrix on a logical set of assumptions. Also, I want the program itself to set up the initial transition matrix.

My first assumptions for the transition matrix probabilities are based on the weather attributes of wind, rain and temperature. Each of these can be described in one of three states. I think it reasonable to assume that there is a 60% chance that a particular attribute will stay the same from one day to the next and that, if it changes, it will not change from one extreme to another. Thus, if it is raining lightly today, the chances of rain states R0, R1 and R2 tomorrow are 0.2, 0.6 and 0.2 respectively. Similarly, if it is hot

today, the probabilities of T0, T1 and T2 tomorrow are 0.6, 0.4 and 0.

Finally, input and output. Each day the program needs to be informed of the current weather conditions and will make a prediction based on the estimated probability of system-states 9, 10, 12 or 13. It would not be very kind to demand of Ivor that he remember all the system-states, which would be the simplest form of input. The simulation asks about each of the weather attributes in turn and, from this, determines the full system- state.

Output is in the form of a prediction of the suitability of the weather tomorrow for kite flying and also an indication as to how much confidence can be placed on the prediction. This confidence factor is an important element of predictor simulations, as I have pointed out in Chapter 3. In this case, it is not possible easily to derive a statistical confidence level for the prediction because we do not know what the random element is, if any, in the change of one weather state to another. We could make an assumption based on a binomial distribution, or a Poisson distribution, or one of the other common methods, but I see no justification for any of these with this low-level simulation. What the program that I developed for Ivor does is to keep track of the number of times that the weather has been in a given state and use this to give a rough indication of the reliability of the relevant transition probabilities. If fewer than 10 previous similar weather-states have been recorded then the prediction will be at best unreliable. Between 10 and 30 then not too bad, over 30 and the results are getting to be as good as the simulation can give. This defines the estimation of confidence and the way of outputting that confidence level.

Storage of the transition matrix and data is important and is done automatically (on request). Facilities are included for both tape and disk storage.

Results

This is a simple simulation which relies on the slow acquisition of data. It thus takes some time to start producing reasonable results (typically one to three months). This process can be shortened by making a trip to the local library, and getting daily weather reports from the newspapers stored there, and thus building up the stored data using historical data.

Once the database starts to build up, the predictions made by the simulation are quite good, considerably better than just guessing. Of course the output is very simple, which helps the prediction process, but the output is also tailored to the specific requirements of Ivor Problem and he finds the results more useful than trying to interpret the published weather forecasts.

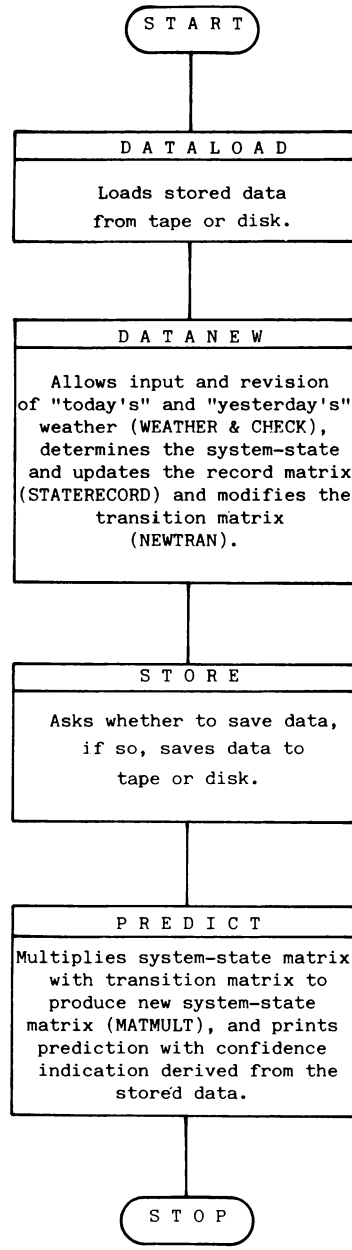


Figure 9.1: Flowchart for Program 'Itmaybe'.

Program description

Figure 9.1 presents an overall flowchart for the program 'Itmaybe'.

Definitions of program variables are given in REM statements at the end of the program listing.

The routines used in the program have the following functions:

Subroutine DATALOAD: Loads in the stored data from tape or disk and sets values for yesterday's weather from a set of temporary stores.

Subroutine DATANEW: Requests input for today's weather, verifies both today's and yesterday's weather, determines the appropriate system-state, updates the record matrix, and updates the transition matrix.

Subroutine STORE: Asks if data is to be stored and, if so, saves the new data.

Subroutine PREDICT: Multiplies the system-state matrix and the transition matrix, and prints out the prediction for tomorrow's weather, together with an indication of the reliability of the prediction, derived from the historic data stored.

Subroutine WEATHER: Prints the request for weather data input, and gives the options available.

Subroutine CHECK: Prints out the data stored for today's and yesterday's weather and asks if it is correct. If not, allows for the modification of the data.

Subroutine STATERECORD: Determines the system-state from the data input, and updates the system-state matrix and the record matrix.

Subroutine NEWTRAN: Modifies the probabilities held in the transition matrix gradually, according to the historical data held in the record matrix.

Subroutine DATASAVE: Sets today's data into a temporary store, and saves all data files on to tape or disk as 'WDATA'.

Subroutine MATMULT: Produces predicted system-states by multiplying the system-state and transition matrices together.

Subroutine WPRINT: Converts stored numerical values for the weather into a verbal description.

Subroutine KEYIN1: Waits for keyboard input and checks if it is 0, 1 or 2.

Subroutine KEYIN2: Waits for keyboard input and checks if it is Y or N.

Subroutine TRANINIT: Calculates the probabilities of transition from a particular system-state into any other. Represents a single row of the initial transition-matrix before modification by historical data.

Subroutine SETPROB: Calculates the initial probabilities of the change of one system-attribute into another. Used in the initialisation and update of the transition-matrix.

Subroutine STARTUP: Initialises and saves the program data, creating the file 'WDATA' on tape or disk.

Subroutine INITIAL: Sets up initial data for the transition matrix.

Subroutine INITRAN: Initialises the transition-matrix.

Subroutine INITMISC: Requests intial data for today and yesterday, and sets up the record matrix.

Subroutine DISK ERROR CHECK: Checks for disk error, and prints error messages.

To run the program, the program must first initialise and save the data arrays and data files, including the transition matrix. In order to do this, you must run the initialisation routine by running subroutine STARTUP directly. Simply type:

GOTO 17000(return)

and the program (with some assistance from you) will set up the initial arrays. You will be asked for information on yesterday's and today's weather, and the data can be saved on to disk or tape.

Once the data is saved, the main program can be used and the initialisation routine need not be used again.

Program improvements

I don't think that there is a lot of point trying to add to the sophistication of this model. If you want to fly a kite, sail a boat, go hang-gliding, or you want to know whether to water the garden or to do something else which requires prediction of some particular aspect of the weather, then use the program at its current level. My feeling (note that I'm resorting to a quali-

tative argument here) is that increased complexity will not improve the simulation and may, in fact, make it less reliable.

The basic technique, however, is very useful and can be applied in a number of other fields. Some applications will demand a more detailed approach. Watch out for any increase in the number of system-states — my experience is that computing time goes up by the fourth power of the number of system-states.

Program listing

```
1000 REM#==========ITMAYBE============
1010 REM WEATHER PREDICTION PROGRAM
1020 REM================================
1030 REM TYPE GOTO 17000 (START-UP) TO S
ET UP THE INITIAL DATA
1040 PRINT:PRINT "[CLR]*** WEATHER PREDI
CTION PROGRAM ***":PRINT:PRINT
1050 GOSUB 2000:REM DATALOAD
1060 GOSUB 3000:REM DATANEW
1070 GOSUB 4000:REM STORE
1080 GOSUB 5000:REM PREDICT
1090 END
2000 REM#==========DATALOAD===========
2010 REM LOAD SAVED DATA
2020 REM================================
2030 REM LOCAL VARS I$,E$,N5,M5
2040 PRINT "PRESS T TO LOAD FROM TAPE, O
R D TO LOAD FROM DISK"
2050 GET I$:IF I$="" GOTO 2050
2060 IF I$="D" THEN GOTO 2150
2070 IF I$="T" THEN GOTO 2090
2080 GOTO 2040:REM ERROR ON INPUT
2090 REM OPEN INPUT CHANNEL FROM TAPE
2100 PRINT "[CLR]WIND ON TAPE AS DESIRED
"
2110 PRINT "THEN PRESS RETURN"
2120 GET E$:IF E$="" THEN 2120
2130 OPEN 1,1,0,"WDATA"
2140 GOTO 2250:REM LOAD DATA
2150 REM OPEN INPUT CHANNEL FROM DISK
2160 PRINT "[CLR]PUT CORRECT DISK IN DRI
VE"
```

```
2170 PRINT "THEN PRESS RETURN"
2180 GET E$:IF E$="" THEN 2180
2190 CLOSE 1:CLOSE 15:OPEN 15,8,15
2200 OPEN 1,8,2,"WDATA,S,R":GOSUB 21000:
REM DISK ERROR CHECK
2210 IF E1=0 THEN GOTO 2250:REM LOAD DAT
A
2220 GET E$:IF E$="" THEN GOTO 2220
2230 IF E$="R" THEN GOTO 2150:REM RE-TRY
2240 GOTO 2000:REM START AGAIN
2250 REM LOAD DATA-FILE "WDATA"
2260 PRINT "[CLR]READING HISTORIC DATA"
2270 DIM TN(26,26),RC(26,26),MS(2),SC(26
),SD(26)
2280 FOR M5=0 TO 26
2290 FOR N5=0 TO 26
2300 INPUT#1,TN(M5,N5)
2310 NEXT N5
2320 NEXT M5
2330 FOR M5=0 TO 26
2340 FOR N5=0 TO 26
2350 INPUT#1,RC(M5,N5)
2360 NEXT N5
2370 NEXT M5
2380 FOR N5=0 TO 2
2390 INPUT#1,MS(N5)
2400 NEXT N5
2410 CLOSE 1
2420 IF I$="D" THEN CLOSE 15
2430 YW=MS(0):YR=MS(1):YT=MS(2)
2440 RETURN
3000 REM#=========DATANEW==============
3010 REM ENTER NEW DATA
3020 REM================================
3030 PRINT "[CLR]INPUT DATA FOR TODAY'S
WEATHER"
3040 GOSUB 6000:REM WEATHER
3050 TW=W1:TR=R1:TT=T1
3060 GOSUB 7000:REM CHECK
3070 GOSUB 8000:REM STATERECORD
3080 GOSUB 9000:REM NEWTRAN
3090 RETURN
4000 REM#=========STORE===============
```

```
4010 REM REQUEST WHETHER TO SAVE DATA
4020 REM================================
4030 E1=0
4040 PRINT "[CLR]DO YOU WISH TO SAVE THE
 DATA (Y/N)?":GOSUB 14000:REM KEYIN2
4050 IF YN=1 THEN GOSUB 10000:REM DATASA
VE
4060 IF E1<>0 THEN GOTO 4030:REM TRY AGA
IN
4070 RETURN
5000 REM#=========PREDICT=============
5010 REM PRINT PREDICTION
5020 REM================================
5030 REM LOCAL T,M,N
5040 PRINT "[CLR]WORKING ON RESULTS"
5050 GOSUB 11000:REM MATMULT
5060 PRINT"[CLR]THE PROBABILITY OF SUIT
ABLE WEATHER FOR KITE FLYING TOMORROW IS
;"
5070 PRINT SD(9)+SD(10)+SD(12)+SD(13)
5080 PRINT
5090 PRINT
5100 M=9*TW+3*TR+TT
5110 T=0
5120 FOR N=0 TO 26
5130 T=T+RC(M,N)
5140 NEXT N
5150 IF T>=0 AND T<=9 THEN PRINT "THE PR
EDICTION IS UNRELIABLE"
5160 IF T>=10 AND T<=30 THEN PRINT "THE
PREDICTION IS NOT TOO BAD"
5170 IF T>30 THEN PRINT "THE PREDICTION
SHOULD BE OK"
5180 RETURN
6000 REM#=========WEATHER=============
6010 REM REQUEST WEATHER STATUS
6020 REM================================
6030 PRINT "[HOME][CD]PRESS 0 IF WIND =
TOO LIGHT"
6040 PRINT "PRESS 1 IF WIND = OK"
6050 PRINT "PRESS 2 IF WIND = TOO STRONG
"
6060 GOSUB 13000:REM KEYIN1
```

```
6070 W1=KV
6080  PRINT "[HOME][CD]PRESS 0 = NO RAIN
ETC      "
6090 PRINT "PRESS 1 = RAIN NOT TOO BAD"
6100 PRINT "PRESS 2 = RAIN TOO HEAVY
"
6110 GOSUB 13000:REM KEYIN1
6120 R1=KV
6130 PRINT "[HOME][CD]PRESS 0 IF TEMP =
HOT"
6140 PRINT "PRESS 1 IF TEMP = OK      "
6150 PRINT "PRESS 2 IF TEMP = TOO COLD"
6160 GOSUB 13000:REM KEYIN1
6170 T1=KV
6180 PRINT "[CLR]"
6190 RETURN
7000 REM#=========CHECK===============
7010 REM VERIFY INPUT DATA
7020 REM==============================
7030 PRINT "[CLR]THE WEATHER YESTERDAY W
AS;"
7040 W1=YW:R1=YR:T1=YT
7050 GOSUB 12000:REM WPRINT
7060 PRINT "IS THIS OK (Y/N)?"
7070 GOSUB 14000:REM KEYIN2
7080 IF YN=1 THEN GOTO 7130:REM YESTERDA
Y'S WEATHER OK
7090 PRINT "[CLR]INPUT NEW DATA FOR YEST
ERDAY'S WEATHER"
7100 GOSUB 6000:REM WEATHER
7110 YW=W1:YR=R1:YT=T1
7120 GOTO 7000:REM RE-CHECK WEATHER
7130 PRINT "[CLR]TODAY'S WEATHER IS;"
7140 W1=TW:R1=TR:T1=TT
7150 GOSUB 12000:REM WPRINT
7160 PRINT "IS THIS OK (Y/N)?"
7170 GOSUB 14000:REM KEYIN2
7180 IF YN=1 THEN GOTO 7230:REM TODAY'S
WEATHER OK
7190 PRINT "[CLR]INPUT NEW DATA FOR TODA
Y'S WEATHER"
7200 GOSUB 6000:REM WEATHER
7210 TW=W1:TR=R1:TT=T1
```

```
7220 GOTO 7130:REM RE-CHECK TODAY'S WEAT
HER
7230 PRINT "[CLR]"
7240 RETURN
8000 REM#=========STATERECORD==========
8010 REM EVALUATE STATE MATRIX
8020 REM================================
8030 REM LOCAL M1,N1
8040 PRINT "[CLR]EVALUATING STATE MATRIX
"
8050 FOR N1=0 TO 26
8060 SC(N1)=0
8070 NEXT N1
8080 N1=9*TW+3*TR+TT
8090 SC(N1)=1
8100 M1=9*YW+3*YR+YT
8110 RC(M1,N1)=RC(M1,N1)+1
8120 RETURN
9000 REM#=========NEWTRAN=============
9010 REM UPDATE TRANSITION MATRIX
9020 REM================================
9030 REM LOCAL M2,N2,T2,P2
9040 PRINT "[CLR]REVISING TRANSITION MAT
RIX"
9050 M2=9*YW+3*YR+YT
9060 T2=0
9070 FOR N2=0 TO 26
9080 T2=T2+RC(M2,N2)
9090 NEXT N2
9100 IF T2<>0 THEN GOTO 9150
9110 FOR N2=0 TO 26
9120 GOSUB 15000:REM TRANINIT
9130 TN(M2,N2)=IT
9140 NEXT N2:GOTO 9240:REM RETURN
9150 IF T2>99 THEN GOTO 9210
9160 P2=T2/100
9170 FOR N2=0 TO 26
9180 GOSUB 15000:REM TRANINIT
9190 TN(M2,N2)=(1-P2)*IT+P2*RC(M2,N2)/T2
9200 NEXT N2:GOTO 9240
9210 FOR N2=0 TO 26
9220 TN(M2,N2)=RC(M2,N2)/T2
9230 NEXT N2
```

```
9240 RETURN
10000 REM#========DATASAVE============
10010 REM SAVE DATA
10020 REM============================
10030 REM LOCAL VARS I$,E$,N5,M5
10040 PRINT "[CLR]PRESS T TO SAVE TO TAP
E, OR D TO SAVE TO DISK"
10050 GET I$:IF I$="" GOTO 10050
10060 IF I$="D" THEN GOTO 10150
10070 IF I$="T" THEN GOTO 10090
10080 GOTO 10040:REM ERROR ON INPUT
10090 REM OPEN OUTPUT CHANNEL TO TAPE
10100 PRINT "[CLR]WIND ON TAPE AS DESIRE
D"
10110 PRINT "THEN PRESS RETURN"
10120 GET E$:IF E$="" THEN GOTO 10120
10130 OPEN 1,1,1,"WDATA"
10140 GOTO 10270:REM SAVE DATA
10150 REM OPEN OUTPUT CHANNEL TO DISK
10160 PRINT "[CLR]PUT CORRECT DISK IN DR
IVE"
10170 PRINT "THEN PRESS RETURN"
10180 GET E$:IF E$="" THEN GOTO 10180
10190 CLOSE 1:CLOSE 15:OPEN 15,8,15
10200 OPEN 1,8,2,"WDATA,S,W":GOSUB 21000
:REM DISK ERROR CHECK
10210 IF E1=0 THEN 10270
10220 IF E1>0 AND E1<>63 THEN GOTO 10450
:REM RETURN
10230 IF E1=63 THEN PRINT "PRESS R TO OV
ERWRITE THE EXISTING FILE"
10240 GET E$:IF E$="" GOTO 10240
10250 IF E$="R" THEN CLOSE 1:PRINT#15,"S
0:WDATA":GOTO 10200:REM RE-OPEN FILE
10260 GOTO 10450:REM RETURN
10270 REM WRITE FILE WDATA
10280 PRINT "[CLR]WRITING HISTORIC DATA"
10290 MS(0)=TW:MS(1)=TR:MS(2)=TT
10300 FOR M5=0 TO 26
10310 FOR N5=0 TO 26
10320 PRINT#1,TN(M5,N5)
10330 NEXT N5
10340 NEXT M5
```

```
10350 FOR M5=0 TO 26
10360 FOR N5=0 TO 26
10370 PRINT#1,RC(M5,N5)
10380 NEXT N5
10390 NEXT M5
10400 FOR N5=0 TO 2
10410 PRINT#1,MS(N5)
10420 NEXT N5
10430 CLOSE 1
10440 IF I$="D" THEN CLOSE 15
10450 RETURN
11000 REM#=========MATMULT============
11010 REM MULTIPLY MATRICES SC & TN
11020 REM==============================
11030 REM LOCAL M3,N3
11040 FOR N3=0 TO 26
11050 SD(N3)=0
11060 FOR M3=0 TO 26
11070 SD(N3)=SD(N3)+SC(M3)*TN(M3,N3)
11080 NEXT M3
11090 NEXT N3
11100 RETURN
12000 REM#=========WPRINT=============
12010 REM PRINT CURRENT WEATHER STATES
12020 REM==============================
12030 ON W1 GOTO 12050,12060
12040 PRINT "WIND TOO LIGHT":GOTO 12070
12050 PRINT "WIND OK":GOTO 12070
12060 PRINT "WIND TOO STRONG"
12070 ON R1 GOTO 12090,12100
12080 PRINT "NO RAIN":GOTO 12110
12090 PRINT "NOT TOO MUCH RAIN":GOTO 121
10
12100 PRINT "TOO RAINY"
12110 ON T1 GOTO 12130,12140
12120 PRINT "TEMP=HOT":GOTO 12150
12130 PRINT "TEMP=INTERMEDIATE":GOTO 121
50
12140 PRINT "TOO COLD"
12150 RETURN
13000 REM#=========KEYIN1============
13010 REM INPUT VALUE KV=0,1,OR 2
13020 REM==============================
```

```
13030 GET A$
13040 IF A$="" GOTO 13030
13050 IF ASC(A$)<48 OR ASC(A$)>50 THEN G
OTO 13030:REM INPUT ERROR
13060 KV=VAL(A$)
13070 RETURN
14000 REM#=========KEYIN2==============
14010 REM INPUT YES OR NO (YN=1=YES)
14020 REM================================
14030 GET A$
14040 IF A$="" GOTO 14030
14050 IF ASC(A$)=89 THEN YN=1:GOTO 14080
14060 IF ASC(A$)=78 THEN YN=0:GOTO 14080
14070 GOTO 14030:REM INPUT ERROR
14080 RETURN
15000 REM#========= TRANINIT============
15010 REM UPDATES TRANSITION MATRIX
15020 REM================================
15030 REM USES M2,N2 FROM NEWTRAN
15040 REM LOCAL X,S,U,C1,C2,C3
15050 X=INT(N2/9)
15060 S=INT((N2-X*9)/3)
15070 U=N2-X*9-S*3
15080 C=X-W:D=X:GOSUB 16000:REM SETPROB
15090 C1=PS
15100 C=S-R:D=S:GOSUB 16000:REM SETPROB
15110 C2=PS
15120 C=U-T:D=U:GOSUB 16000:REM SETPROB
15130 C3=PS
15140 IT=C1*C2*C3
15150 RETURN
16000 REM#=========SETPROB============
16010 REM CALCULATES PROBABILITIES
16020 REM================================
16030 REM USES C&D FROM TRANINIT & INITR
AN
16040 REM RETURNS PS
16050 IF C=0 THEN PS=0.6:GOTO 16100:REM
RETURN
16060 IF C=2 OR C=-2 THEN GOTO 16090
16070 IF D=2 THEN PS=0.2:GOTO 16100:REM
RETURN
16080 PS=0.4:GOTO 16100:REM RETURN
```

```
16090 PS=0
16100 RETURN
17000 REM#=========STARTUP============
17010 REM INITIALISATION ONLY
17020 REM============================
17030 PRINT "[CLR]INITIALISING DATA"
17040 GOSUB 18000:REM INITIAL
17050 GOSUB 19000:REM INITRAN
17060 GOSUB 20000:REM INITMISC
17070 GOSUB 8000:REM STATERECORD
17080 GOSUB 9000:REM NEWTRAN
17090 GOSUB 10000:REM DATASAVE
17100 PRINT "INITIALISATION COMPLETE":EN
D
18000 REM#=========INITIAL============
18010 REM DIMENSION ARRAYS
18020 REM============================
18030 DIM TN(26,26),RC(26,26),MS(2),SC(2
6),SD(26)
18040 RETURN
19000 REM#=========INITRAN============
19010 REM INITIALISE TRANSITION MATRIX
19020 REM============================
19030 REM LOCAL VARS I1,I2,I3,I4,I5,I6,
S1,S2,S3
19040 FOR I1=0 TO 2
19050 FOR I2=0 TO 2
19060 FOR I3=0 TO 2
19070 FOR I4=0 TO 2
19080 FOR I5=0 TO 2
19090 FOR I6=0 TO 2
19100 C=I4-I1:D=I4:GOSUB 16000:REM SETPR
OB
19110 S1=PS
19120 C=I5-I2:D=I5:GOSUB 16000:REM SETPR
OB
19130 S2=PS
19140 C=I6-I3:D=I6:GOSUB 16000:REM SETPR
OB
19150 S3=PS
19160 TN(9*I1+3*I2+I3,9*I4+3*I5+I6)=S1*S
2*S3
19170 NEXT I6
```

```
19180 NEXT I5
19190 NEXT I4
19200 NEXT I3
19210 NEXT I2
19220 NEXT I1
19230 RETURN
20000 REM#=========INITMISC===========
20010 REM SET UP WEATHER STATUS
20020 REM==============================
20030 REM LOCAL VARS N4,M4
20040 PRINT "[CLR]SET-UP WEATHER FOR YES
TERDAY"
20050 GOSUB 6000:REM WEATHER
20060 YW=W1:YR=R1:YT=T1
20070 PRINT "[CLR]SET-UP WEATHER FOR TOD
AY"
20080 GOSUB 6000:REM WEATHER
20090 TW=W1:TR=R1:TT=T1
20100 GOSUB 7000:REM CHECK
20110 RETURN
21000 REM#=========DISK ERROR CHECK====
21010 REM  RETURNS E1
21020 REM==============================
21030 REM LOCAL EM$
21040 INPUT#15,E1,EM$
21050 IF E1<20 THEN E1=0:GOTO 21110:REM
NO ERROR
21060 IF E1=63 THEN GOTO 21110:REM FILE
EXISTS
21070 PRINT "DISK ERROR"
21080 PRINT "ERROR NUMBER ";E1
21090 PRINT EM$
21100 PRINT "PRESS R TO TRY AGAIN, OR AN
Y OTHER KEY TO ABORT"
21110 RETURN
22000 REM  VARIABLES==================
22010 REM  EM$=ERROR MESSAGE
22020 REM  KV=VALUE OF KEY PRESS
22030 REM  MS()=STORES TODAY'S WEATHER
22040 REM   ATTRIBUTES TO BE RETRIEVED
22050 REM   AS YESTERDAY'S NEXT TIME
22055 REM  R1=TEMPORARY RAIN INDICATOR
22060 REM  RC()=MATRIX STORING RECORD OF
```

```
22070 REM   EACH SYSTEM-STATES NO OF
22080 REM   OCCURRENCES
22090 REM  SC()=CURRENT SYSTEM-STATE
MATRIX
22100 REM  SD()=DERIVED MATRIX FOR
22110 REM   FOLLOWING DAY HOLDING
22120 REM   PREDICTED PROBABILITIES
22130 REM   FOR EACH SYSTEM-STATE
22140 REM  TN()=TRANSITION MATRIX
22150 REM  TR=TODAY'S RAIN INDICATOR
22160 REM  T1=TEMPORARY TEMPERATURE
INDICATOR
22170 REM  TT=TODAY'S TEMPERATURE
INDICATOR
22180 REM  TW=TODAY'S WIND INDICATOR
22190 REM  YN=YES OR NO INDICATOR
22200 REM  YR=YESTERDAY'S RAIN INDICATOR
22210 REM  YT=YESTERDAY'S TEMPERATURE
INDICATOR
22220 REM  YW=YESTERDAY'S WIND INDICATOR
22230 REM  W1=TEMPORARY WIND INDICATOR
```

CHAPTER 10

Campaign For Real-time

Real-time simulation



Training simulators and computer games: real-time simulations are a fast-growing business.

Background

Real-time simulators are usually designed to interact with humans. The interaction is often sophisticated in nature: complex images are produced, inputs of various kinds are recognised and acted upon, sounds are produced, and equipment external to the computer may be activated. Because of the inherent complexity of most real-time simulations and the speed of computation required, it is normally necessary to resort to the use of machine code when using a microcomputer. Thus the bulk of real-time simulations are beyond the scope of this book.

Having said that most real-time simulations require the use of machine code programming, let me now indicate the sort of things that can be done with a high-level language.

In the training field, it is often not necessary to provide a complex screen image and programming time can be saved by having only a small proportion of the image moving at any one time. Thus, where dials and gauges, for example, are represented, they should be presented in as simple a form as possible and only the moving part of the image may need to be simulated. If hardware in addition to the microcomputer is to be used, then it is quite possible that the high-level language will be fast enough.

Simulators may be used in the testing of hardware, duplicating in a controlled and repeatable manner the extended use of a product. Specialised machinery is usually required for such testing, but a low-cost test facility based on a microcomputer can be a practical proposition in some cases. Again, the high-level language will often be sufficient.

Games written in high-level languages will always compare badly with rivals produced in machine code, and thus will have a limited commercial value (the reason most computer games are produced). Some concepts are simple enough to allow the high-level language to be used but will almost certainly demand considerable skill from the programmer.

The simulation technique most often used in real-time is based on the Monte Carlo approach. Obviously, a real-time simulation will normally only represent a single (though changing) system-state. When alternative system responses are possible, a random response will be more useful than a deterministic response, unless repeatability is required. Thus the Monte Carlo approach is often used.

There is much more to a real-time simulation than just predicting the system response, however. A large portion of such a simulation program may be devoted to the presentation of the system, the display, sound, or movement of external hardware. It is often this portion of the program

which can give the programmer the greatest difficulties, partly because it is here that fast-executing routines will be required and also because much of the programming task will demand the solution of unique problems.

Normally it is possible to trade off memory for execution speed by making use of pre-stored images, using look-up tables rather than performing calculations, and using repetitive blocks of program rather than calling subroutines. A detailed knowledge of the execution times of various program statements can indicate the quickest ways of doing things.

Real-time simulations use a time-step to move from one system-state to the next. A longer time-step will provide more computing time but may lead to unrealistic behaviour (jerky or slow responses). Normally the time-step is constant and it is important to make sure that, where alternative system-reponses are possible, the time taken by the program to represent such responses remains constant. Careful attention must be paid to program execution speed because of this, and it is common to use delays of variable length to keep all execution speeds constant. Such delays can also be used to control the overall speed of the running of the simulation. A reduction in an in-built delay whilst the simulation is running will reduce the time-step and hence speed up the simulation.

Example

My final example is a rudimentary game. I have chosen this to demonstrate some of the points made above.

The game is based on the concept of volleyball. A high net divides the screen into two. On each side of the net, there is a single player who can move from side to side and hit a ball which must be knocked over the net. Players are represented as a hand-shaped paddle.

The object of the game is to force the opposing player (in this case the computer) to drop the ball. Each player may hit the ball up to three times before passing it over the net; if the player hits the ball a fourth time then this counts as a dropped ball. If the ball is hit too hard and goes right over the opposition's playing area then again this counts as a dropped ball.

Points are scored when a ball is dropped. You score one point if your opponent drops the ball after you have served. If your opponent served and also dropped the ball, then you do not score a point, but gain the right to serve and thus have a chance of scoring the next point.

One more thing. The game is being played on the moon! Why does it have to be on the moon? The answer is quite simply that I could not get the thing to run fast enough to represent realistically a genuine game of volleyball and hence had to change reality a bit to fit the simulation.

Simulation development and results

Video games tend to be written around the image which is to be produced. This is not a universal rule but often is the most important consideration when planning and creating the simulation.

In this case (as with most real-time simulations) we are producing an emulation simulation. Where choices in action are possible then these will either be made by following the commands of the user (move player's paddle left, move player's paddle right, hit ball hard or soft) or by performing a Monte Carlo selection of a single outcome from all possible choices using a random-number process.

Our first consideration is the ball: how is this to be represented, how is it to move? This is an important consideration and I am going to go into a fair amount of detail here.

The Commodore 64 has a sprite facility for moving objects and so it is logical to represent the ball and other moving objects (the paddles) using sprites. The data describing the shapes of the ball and paddles is held in DATA statements from line 11000 on. The ball comes out as more of a diamond than a volleyball using the current data: you might like to change the ball data to get a better shape, to show a face for example.

Once in flight, what will happen to the ball? It will tend to drop downwards due to gravity and (normally) air resistance will slow it down. I am ignoring air resistance because it will be a minor effect and, in any case, didn't I say that we are playing on the moon? There are standard equations-of-motion which will describe the way in which an object behaves in a gravitational field (look in any school textbook on physics or applied mathematics) and we could set up a well-reasoned argument to determine an exact scale representation of a ball in a lunar gravitational field, being hit at a particular and realistic speed, and in a playing court of known dimensions. A simpler approach is to develop the model using simple equations-of-motion, using approximate values for ball speeds and the effect of gravity, and then using a process of trial and error to produce a practical simulation. I employed the second approach in this case. Had I been producing, say, a flight simulator where an exact representation was required, then the first approach would have been the one to choose.

The ball is to move around the screen, up and down, left and right. It is usually easier to represent the actions of a moving body in terms of horizontal movement (movement in the x-direction, a convention for describing horizontal displacements to the right of a fixed point) and vertical movement (movement in the y-direction) rather than using the velocity (speed in a given direction) of the body. In this case it is also sensible to use Commodore's definition of x and y for the positioning of sprites. In this way we don't have to do any conversion of ball position to screen position, because they are the same thing.

In one time-step (from one screen image to the next) we must get input from the player, make decisions for the computer's play, move the ball, and check for the ball hitting anything. Obviously this takes time, and the more things which need to be done the longer the time spent. During this time the ball must be moved to its new location. This movement is not a nice smooth glide but the removal of the ball image from one position and the production of a new image in a second position. If the distance between consecutive positions is too great then the effect will be jerky and unrealistic. Thus high-speed objects can only be represented if the time-step is very short. This is the point at which I ran into trouble when developing a volleyball simulation, I could not get the ball moving realistically given the amount of computing required. Thus I decided to slow the ball down a little and call the simulation 'Lunaball'.

A second, and possibly more important, reason to move objects only a small distance each time-step is because the tests which must be performed to find out if various things shown on-screen are hitting each other are usually done on the basis of the new position rather than the line of movement joining old and new positions. This is particularly true of the Commodore sprites, which automatically provide an indication of collisions between sprites and between sprites and the background screen display. The collision indication can only be provided on an actual displayed image. Thus, if the distance moved by the sprite in one time-step is large, it is possible for one object to pass straight through another without registering a collision. The ball could pass through the net, for example, if the ball were to be moved too far.

Line 1240 of Lunaball is the line which controls the ball, and it was the first line of the program written when I began developing the simulation. XB is the position of the ball in the x-direction and YB in the y-direction. The ball-speed terms, VX and VY, represent the distance the ball will travel in the x and y directions in a single time-step. (If you are not familiar with representing movement as a combination of velocities in x and y then think of the ball moving first in the x-direction, horizontally across the screen, a short distance VX and then in the y-direction, vertically down the screen a distance VY. This stepping motion, if small enough and fast enough, will reproduce a movement diagonally across the screen.) VY is modified each time-step to allow for gravity, GR. You can see that, if nothing else happens, VY will gradually be increasing, the ball will drop faster and faster. What does happen in this instance is that the ball is hit back up again by a paddle or falls to the base of the court and play ends.

As I said at the start of this section, the simulation is based around the production of a screen display. This screen display is in fact the end result of the simulation. No great analysis of play is provided, predictions are not made of possible future performances, and no graphs are drawn. The

simulation results are, quite simply, something you must see for yourself.

Program description

Before going into a blow-by-blow description of the program I must just list down some of the more important programming techniques which should be remembered for real-time simulation. (These are not in any particular order.)

1) In order to maintain a constant time-step, the program instructions performed each time round the loop should be the same or should be carefully written to take the same time. For example, even though we know that the ball will only hit a paddle when it is moving down, we cannot 'save time' by missing out the test when the ball is moving up — this would cause the ball to appear to move quickly upwards and slowly downwards.

2) It is often quicker (in terms of program running speed) to forget about subroutines in the main part of a real-time simulation (this will annoy the purists but is a fact of life when using BASIC). This is because the subroutine call and return instructions take time to execute, time is wasted transferring to and from the subroutines, and exchanging variables to use a general-purpose subroutine takes time. If there is sufficient memory available, then it may be better (from the point of view of speed) to duplicate lines of program where needed.

3) Once out of the main loop, in this case when the ball hits something, we can usually stop worrying too much about timing. We should, however, make sure that those cases where further action is required (if the ball hits a paddle or the net) are considered first in order to avoid any unnecessary break in the action.

4) Where subroutines are used, and time is important, the call should be directly to an executable statement and not a REM. This saves time. Also, on the same topic, the program will run significantly faster if REMs are not included. You will notice that I have not included REMs in the main program loop.

5) The most used routines should appear at the start of the program. This is because BASIC has to 'search' for each subroutine every time it is used, starting from the beginning of the program. The longer the search the longer the time taken.

6) One last point when using sprites. Care must be taken when using the automatic collision indicators. These must be cleared (line 1290) of old data before being re-read but you must also allow sufficient time for the 6566/9 chip to re-register any collisions. Line 1300 is a delay designed for this purpose.

I have not prepared a flowchart for this program. The program logic is simple — after initialising, the program really consists of two loops. In the first loop a new ball is set up and play started (the second loop): this continues until the ball hits something and the appropriate action is taken. Eventually the ball is dropped, or hit too many times, or is knocked off court. At this stage the score is displayed and the loop restarted for a new ball.

The inner loop simulates the ball in play. The vital steps here are:

Lines 1140 – 1170: Get input from keyboard for player's turn and move player's paddle.

Lines 1180 – 1230: Move computer's paddle depending on position of ball — I have tried to make the computer react in a reasonable manner, it is all to easy to ruin a good game by making the computer unbeatable.

Lines 1240 – 1280: Move ball.

Lines 1290 – 1340: Check for the ball hitting something then loop back to 1140.

Here are some notes on the rest of the program:

Lines 1000 – 1600: Perform main program loops as above. When the ball hits something then decide what has been hit (1360 – 1510) and take appropriate action (1520 – 1580).

Subroutine INITIALISE: Initialise various things for whole game. The keyboard buffer needs to be reduced to 1 to give a reasonable speed of response.

Subroutine NEW BALL: Sets positions of ball and paddles for each new ball.

Subroutines MAKE DISPLAY: Clears old display (eg display of score), sets up court and net as a background screen display, and switches on the sprites representing the paddles and ball. Line 4120 is a variable-length delay which holds the ball before serving.

Subroutine HIT NET: If the ball hits the net this moves the ball image to the most likely side of the net and allows it to drop.

Subroutine HIT PADDLE: This routine is in two halves, for the player's paddle and computer's paddle. New ball speeds are set to knock the ball back up and, hopefully, over the net. Line 6190 moves the ball up and clear of the paddle in order to re-set the collision indicators.

Subroutine PLAY ENDED: Depending on how the ball has been lost, this routine determines whether a point has been scored or if a change in server is to take place.

Subroutine SCORE: Updates and displays latest score.

Subroutine NOSCORE: Changes server.

Subroutine SPRITES: Reads sprite data from DATA statements and POKEs the data into the sprite memory area. Sets up colours for background and sprites.

Program use and further development

The program runs automatically and even performs the serve (for both players). Use the Q and W keys to move your paddle (the lefthand one) to the left and right, and the '+' and '−' keys to hit the ball hard or soft. You must press only one key at a time, so make sure that you are in the right position before hitting the ball. If you do not press + or −, then the ball will simply bounce off your paddle.

The paddles have been shaped to give the impression of hands. These cannot move out of their own side.

Potential improvements to the game are many: the court could be coloured differently on each side, players could be given more control of the direction in which a ball is hit, the shape of the paddles and the ball could be improved, sound could be added, a high-score table included, and so on. I leave these to you, but remember my comments about keeping down the run-time — if you are too ambitious the game may become too slow to play!

That's about it, I hope you have found this book as interesting to read as I found it to write. All I can do now is to wish you success with your simulations.

Program listing

```
1000 REM#=======LUNABALL=============
1010 REM REAL-TIME GAME
1020 REM=============================
1030 GOSUB 2000:REM INITIALISE
1040 REM LOOP FOR EACH NEW BALL--------
1050 REM------------------------------
1060 GOSUB 3000:REM SETUP FOR NEW BALL
1070 GOSUB 4000:REM PREPARE DISPLAY
1080 REM TIMESTEP LOOP, BALL IN PLAY+++
1090 REM++++++++++++++++++++++++++++++++
1100 REM GET PLAYER & COMPUTER ACTIONS
1110 REM MOVE AND DISPLAY BALL
1120 REM CHECK FOR BALL HIT ANYTHING
1130 REM AND TAKE APPROPRIATE ACTION
1140 GET A$
1150 IF A$="Q" AND XL>32 THEN XL=XL-8
1160 IF A$="W" AND XL<152 THEN XL=XL+8
1170 POKE (V+2),XL
1180 N=-8-8*(XR<200)
1190 IF VX>=0 AND XB>XR-8 THEN N=8
1200 IF XR<290 THEN XR=XR+N
1210 P=0:IF XR>255 THEN P=1
1220 POKE (V+4),XR-P*255
1230 POKE (V+16),(PEEK(V+16)AND251)+P*4
1240 XB=XB+VX:VY=VY+GR:YB=YB+VY
1250 P=0:IF XB>255 THEN P=1
1260 POKE (V+1),YB
1270 POKE V,XB-P*255
1280 POKE (V+16),(PEEK(V+16)AND254)+P
1290 N=PEEK(V+30)+PEEK(V+31)
1300 FOR N=1 TO 10:NEXT N
1310 HI=(PEEK(V+30)AND6)/2
1320 IF HI>0 THEN GOTO 1520
1330 IF (PEEK(V+31)AND1)=1 THEN GOTO 142
0
1340 HI=0:GOTO 1140:REM HIT NOTHING++++
1350 REM++++++++++++++++++++++++++++++++
1360 REM IF GOT THIS FAR THEN BALL HAS
1370 REM HIT SOMETHING. WE HAVE TESTED
1380 REM FOR HIT PADDLE AND HIT
1390 REM BACKGROUND. NOW WE MUST FIND
```

```
1400 REM EXACTLY WHAT HAS BEEN HIT
1410 REM HIT LEFT SIDE?
1420 IF XB<=32 THEN HI=4:GOTO 1520
1430 REM HIT RIGHT SIDE?
1440 IF XB>=312 THEN HI=5:GOTO 1520
1450 REM HIT LEFT BASE?
1460 IF YB>=221 AND XB<=162 THEN HI=6:GO
TO 1520
1470 REM HIT RIGHT BASE?
1480 IF YB>=221 AND XB>=181 THEN HI=7:GO
TO 1520
1490 REM IF GOT THIS FAR THEN MUST HAVE
1500 REM HIT NET. SET HI=3
1510 HI=3
1520 IF HI=1 OR HI=2 THEN GOSUB 6000:REM
 HIT PADDLE
1530 IF (HI=1 OR HI=2) AND BC>3 THEN GOS
UB 7000:GOTO 1580:REM TOO MANY HITS
1540 IF HI=3 THEN GOSUB 5000:REM HIT NET
1550 IF HI=4 OR HI=5 THEN GOSUB 7000:GOT
O 1580:REM BALL OUT
1560 IF HI=6 OR HI=7 THEN GOSUB 7000:GOT
O 1580:REM DROPPED BALL
1570 GOTO 1140:REM BALL BACK IN PLAY+++
1580 GOTO 1040:REM NEW BALL----------
1590 REM--------------------------------
1600 END
2000 REM########INITIALISE##########
2010 REM SET INITIAL VALUES
2020 REM################################
2030 POKE 649,1:REM SET KEYBOARD BUFFER
2040 POKE 650,255:REM ALL KEYS REPEAT
2050 REM SET UP S$=37 SPACES
2060 S$=""
2070 FOR N=1 TO 37:S$=S$+" ":NEXT N
2080 V=53248:REM ADDRESS OF VIDEO CHIP
2090 GR=3
2100 LS=0:RS=0
2110 IS=INT(RND(0)+.5)
2120 GOSUB 10000:REM SET UP SPRITES
2130 RETURN
3000 REM########NEW BALL#############
3010 REM SET UP DATA FOR EACH NEW BALL
```

```
3020 REM================================
3030 BI=IS:BC=1
3040 IF IS=1 GOTO 3080:REM RIGHTHAND SE
RVE
3050 REM LEFTHAND SERVE
3060 VY=-24:VX=10+2*RND(0):XB=80
3070 GOTO 3100
3080 REM RIGHTHAND SERVE
3090 VY=-24:VX=-10-2*RND(0):XB=270
3100 REM DATA FOR EITHER SERVE
3110 YB=190:XL=72:XR=272
3120 REM POKE SPRITE DATA
3130 POKE (V+2),XL
3140 POKE (V+4),XR-255*INT(XR/255)
3150 POKE (V+16),(PEEK(V+16)AND251)+4*IN
T(XR/255)
3160 POKE (V+1),YB
3170 POKE V,XB-255*INT(XB/255)
3180 POKE (V+16),(PEEK(V+16)AND254)+INT(
XB/255)
3190 RETURN
4000 REM#========MAKE DISPLAY==========
4010 REM SCREEN DISPLAY
4020 REM================================
4030 PRINT "[CLR]"
4040 FOR N=1 TO 18
4050 PRINT CHR$(166);S$;CHR$(166)
4060 NEXT N
4070 FOR N=1 TO 6
4080 PRINT CHR$(166);LEFT$(S$,18);CHR$(9
8);LEFT$(S$,18);CHR$(166)
4090 NEXT N
4100 FOR N=1 TO 39:PRINT CHR$(166);:NEXT
 N
4110 POKE (V+21),7:REM ENABLE SPRITES
4120 FOR N=1 TO 1000*RND(0):NEXT N
4130 RETURN
5000 REM#======HIT NET===============
5010 REM BALL HAS HIT THE NET
5020 REM================================
5030 IF VX>=0 THEN XB=162
5040 IF VX<0 THEN XB=182
5050 IF YB>221 THEN YB=221
```

```
5060 VX=0:VY=0
5070 RETURN
6000 REM#======HIT PADDLE============
6010 REM UPDATE VARIABLES
6020 REM==============================
6030 IF HI=2 THEN GOTO 6090
6040 IF BI=0 THEN BC=BC+1
6050 IF BI=1 THEN BC=1:BI=0
6060 IF A$="+" THEN VY=-24:VX=INT(RND(0)
*-VY/2-VY/4):GOTO 6190
6070 IF A$="-" THEN VY=-12:VX=INT(RND(0)
*-VY/2-VY/4):GOTO 6190
6080 VY=-VY*(1-.5*RND(0)):VX=.5*VY*(.5-R
ND(0)):GOTO 6190
6090 RP=INT(3*RND(0)+1)
6100 IF BI=1 THEN BC=BC+1
6110 IF BI=0 THEN BC=1:BI=1
6120 ON RP GOTO 6130,6150,6170
6130 VY=-24:VX=-INT(RND(0)*-VY/2-VY/4)
6140 GOTO 6190
6150 VY=-12:VX=-INT(RND(0)*-VY/2-VY/4)
6160 GOTO 6190
6170 VY=-VY*(1-.5*RND(0))
6180 VX=.5*VY*(.5-RND(0))
6190 YB=190:POKE (V+1),YB
6200 RETURN
7000 REM#=======PLAY ENDED===========
7010 REM UPDATE SCORES
7020 REM==============================
7030 POKE (V+21),0:REM SPRITES OFF
7040 N=IS+1
7050 IF HI=4 AND BI=1 THEN GOTO 7100
7060 IF HI=5 AND BI=0 THEN GOTO 7080
7070 IF HI=2 OR HI=5 OR HI=7 THEN GOTO 7
100
7080 ON N GOSUB 9000,8000
7090 GOTO 7110
7100 ON N GOSUB 8000,9000
7110 RETURN
8000 REM#=======SCORE================
8010 REM UPDATE SCORE
8020 REM==============================
8030 PRINT "[CLR]THIS IS THE LATEST SCOR
```

```
E"
8040 IF IS=0 THEN LS=LS+1
8050 IF IS=1 THEN RS=RS+1
8060 PRINT "YOU HAVE ";LS
8070 PRINT "I HAVE ";RS
8080 FOR N=1 TO 1000:NEXT N
8090 RETURN
9000 REM#=======NO SCORE==============
9010 REM CHANGE SERVE
9020 REM============================
9030 PRINT "[CLR]NO SCORE"
9040 PRINT "CHANGE SERVICE"
9050 FOR N=1 TO 1000:NEXT N
9060 IF IS=1 THEN IS=0:GOTO 9080
9070 IS=1
9080 RETURN
10000 REM#=======SPRITES==============
10010 REM SET UP SPRITE DATA
10020 REM============================
10030 FOR SN=0 TO 2
10040 POKE (2040+SN),(192+SN)
10050 FOR N1=0 TO 20
10060 FOR M1=0 TO 2
10070 READ D
10080 POKE ((192+SN)*64)+(M1+N1*3),D
10090 NEXT M1
10100 NEXT N1
10110 NEXT SN
10120 POKE (V+39),1:REM COLOURS
10130 POKE (V+40),7
10140 POKE V+41,4
10150 POKE V+28,6:REM MULTICOLOUR ON
10160 POKE V+37,3:REM MULTICOLOURS
10170 POKE V+38,5
10180 POKE 53280,0:REM BORDER
10190 POKE 53281,0:REM SCREEN COLOUR
10200 POKE 53275,0:REM BACKGROUND PRIORI
TY
10210 POKE (V+3),218:REM PADDLE Y POSNS
10220 POKE (V+5),218
10230 RETURN
11000 REM#========SPRITE DATA==========
11010 REM DATA
```

```
11020 REM================================
11030 REM SPRITE #0 BALL
11040 DATA 0,0,0,0,0,0,0,0,0
11050 DATA 0,0,0,0,0,0,3,192,0
11060 DATA 3,192,0,15,240,0,15,240,0
11070 DATA 63,252,0,63,252,0,255,255,0
11080 DATA 255,255,0,255,255,0,255,255,0
11090 DATA 63,252,0,63,252,0,15,240,0
11100 DATA 15,240,0,3,192,0,3,192,0
11110 REM SPRITE #1 LEFT PADDLE
11120 DATA 0,0,0,0,0,0,0,0,0
11130 DATA 0,0,0,0,255,208,3,239,208
11140 DATA 63,172,0,62,175,232,58,175,23
2
11150 DATA 58,172,0,58,175,244,58,175,24
4
11160 DATA 58,172,0,62,175,232,63,175,23
2
11170 DATA 63,172,0,0,255,160,0,63,160
11180 DATA 0,0,0,0,0,0,0,0,0
11190 REM SPRITE #2 RIGHT PADDLE
11200 DATA 0,0,0,0,0,0,0,0,0
11210 DATA 0,0,0,11,255,0,11,247,192
11220 DATA 0,53,252,23,245,124,23,245,92
11230 DATA 0,53,92,47,245,92,47,245,92
11240 DATA 0,53,92,23,245,124,23,245,252
11250 DATA 0,55,252,5,255,0,5,252,0
11260 DATA 0,0,0,0,0,0,0,0,0
11270 REM END OF DATA
12000 REM#=======VARIABLES============
12010 REM A$=PLAYER INPUT
12020 REM  "Q"-MOVE PADDLE LEFT
12030 REM  "W"-MOVE PADDLE RIGHT
12040 REM  "+"-HARD HIT
12050 REM  "-"-SOFT HIT
12060 REM BC=COUNT OF CONSECUTIVE HITS
12070 REM BI=INDICATOR OF WHO HIT BALL
12080 REM  0-PLAYER ON LEFT
12090 REM  1-PLAYER ON RIGHT (COMP.)
12100 REM HI=HIT INDICATOR
12110 REM  0-NOTHING HIT
12120 REM  1-HIT LEFT PADDLE
12130 REM  2-HIT RIGHT PADDLE
```

```
12140 REM  3-HIT NET
12150 REM  4-BALL OFF LEFT
12160 REM  5-BALL OFF RIGHT
12170 REM GR=EFFECT OF GRAVITY
12180 REM IS=INDICATOR OF SERVICE
12190 REM  0-LEFT SERVED
12200 REM  1-RIGHT SERVED
12210 REM LS,LR=SCORE FOR LEFT AND RIGHT
12220 REM RP=USED FOR COMPUTER COMMANDS
12230 REM VX,VY=BALL SPEEDS
12240 REM XB=BALL X-POSITION
12250 REM XL,XR=X-POSITION OF PADDLES
12260 REM YB=Y-POSITION OF BALL
```

APPENDIX A
Simplespread

Simplespread is, as its name suggests, a simple general-purpose spreadsheet program. Although simple in concept you can see that the program is quite lengthy, but worth typing in.

Program description

Briefly, the functions of the routines used in Simplespread are as follows:

SETUP: Dimensions arrays F$( ) and N( ) to hold formulae and numbers for each cell on the spreadsheet. At present the number of cell rows is limited to a maximum (RM) of 25. This is sufficient for the example given in Chapter 6, but may be increased. Two points to note: the memory requirement for F$ will vary depending on the number of cells actually used, and long formulae will use up lots of memory, hence a strict limitation on RM cannot really be defined. Once data is entered, RM cannot be changed and so you must keep a careful note of RM if you decide to increase it — data saved to tape or disk with one value for RM must be re-loaded to the same value.

WAIT: The main control routine. Waits for an input and acts appropriately by calling the relevant subroutine. Permissible inputs are stored in I$.

GETKEY: Waits for a key to be pressed. Ignores RETURN.

DISPLAY: A collection of subroutines which produce the screen display or modify the existing display. VD holds the vertical position for the current cell (marked by inverse cell) and LR holds the horizontal position. P$ is a string used for displaying the cell contents (either a number or a text heading).

N(C,R): This routine works through F$(C,R) for a given cell and calculates a numerical value to be stored in N(C,R).

NEXT.F$( ): Decodes any formula held in F$( ) an element at a time and calculates a numerical value, CC, for the next identifiable element.

FORMULA: Controls the entry of numerical data, headings, or formulae. Formulae are not evaluated as they are entered.

COPY: Allows a formula from one cell to be copied into another and maintains the relative relationship of any references in the formula to other cells.

EXECUTE: Goes through each of the formula in turn and calculates a numerical result.

LOAD/SAVE: Controls the transfer of data to and from tape and disk.

INSTRUCT: Displays a list of the accepted inputs.

QUIT: Clears the spreadsheet.

VARIABLES: Describes the program variables.

Notes on use

The best way to get to know what can and can't be done on a spreadsheet is simply to spend some time playing around with it.

Inputs are indicated by pressing a single key. For example pressing H leads to 'HEADING (' being printed to screen and you can then type in up to eight characters of text. The heading (or formula, or number) is entered by entering a ')'. RETURN is generally ignored and cannot be used to enter data onto the spreadsheet, although some specific requests for input may require it.

The INST/DEL key should *not* be used when entering data as the program has all sorts of problems trying to interpret the final cell contents. Cursor left will erase characters from the entry line in order to make corrections whilst entering formulae, headings, and numeric data.

Do not enter ',' or ';' (comma or semicolon) into a cell heading, number (see below for entering comments into numerical data), or formula because these cause confusion during save and load operations and you will lose data.

Do not enter cursor controls other than cursor left, which deletes one character, as these will confuse the display.

Do not use the colour controls for the same reason.

The symbols '+', '−', '*', and '/' are supported for entry into formu-

lae. The formulae are interpreted from left to right. You should be careful that these are set up in the correct order.

The format 'C(N,NM)' should be used from within a formula to refer to the numerical contents of other cells, where N is the column letter and NM is the row number. The program will accept a single digit for NM but it is recommended that a double digit number (eg 05) be used if the formula is likely to be copied to other cells.

When entering numerical data, the program stores the data both as a string (on input) and as the numerical value of the string. Avoid entering useless data into the string (eg use '45' rather than '45.00') in order to save memory. If you are not worried about running out of memory then it is possible to use the numerical data entry to store comments. For example, NUM(38.25-24 JULY.MR GRAB) will enter a numerical value of 38.25 but will also perhaps remind you that the money is to be paid to Mr Grab by 24 July.

Although I have tried to cover most situations, the program is not entirely foolproof. For example, if you attempt to C(opy) a formula to the current cell (the cell highlighted by the cursor) you will lose the formula, if you C(opy) to a cell using a negative row number the program will stop with an error, and so on.

Program listing

```
10 REM INITIALISE VARIABLES ETC
20 REM================================
30 POKE 649,1:REM KEYBOARD BUFFER=1
40 RM=25:REM MAX NO. OF ROWS
50 DIM F$(26,RM),N(26,RM)
60 CD$="":CR$="":CL$="":S$=""
70 FOR N=1 TO 40
80 CD$=CD$+"[CD]"
90 CR$=CR$+"[CR]"
100 CL$=CL$+"[CL]"
110 S$=S$+" "
120 NEXT N
130 SL$="[HOME]"+LEFT$(CD$,23)+S$+CL$
140 I$="T[CU][CD][CL][CR]FNHCXLSIQ"
150 C=1:R=1:LR=1:UD=1
160 GOSUB 3430:REM DISPLAY SPREADSHEET
170 PRINT SL$;"PRESS I=INSTRUCTIONS, ELS
E=CONTINUE";
180 GOSUB 2000:REM GET T$
190 IF T$="I" THEN GOSUB 11000:GOTO 170
```

```
1000 REM#======WAIT====================
1010 REM WAIT FOR COMMAND & ACT
1020 REM================================
1030 PRINT SL$;"INPUT?";
1040 GOSUB 2000:REM GET T$
1050 I=0
1060 I=I+1
1070 IF T$=MID$(I$,I,1) THEN GOTO 1090
1080 IF I<15 THEN GOTO 1060
1090 IF I>8 THEN GOTO 1120
1100 ON I GOSUB 3030,3150,3220,3290,3360
,7000,7000,7000
1110 GOTO 1030
1120 J=I-8
1130 ON J GOSUB 8000,9000,10000,10000,11
000,12000
1140 GOTO 1030
2000 REM#======GETKEY==================
2010 REM WAIT FOR KEYPRESS
2020 REM================================
2030 GET T$
2040 GET T$:IF T$="" THEN GOTO 2040
2050 IF T$=CHR$(13) THEN GOTO 2040
2060 RETURN
3000 REM#======DISPLAY=================
3010 REM MOVE TO NEW CELL AND DISPLAY
3020 REM================================
3030 REM MOVE TO CELL C,R--------------
3040 PRINT SL$;
3050 INPUT "INPUT COLUMN ADDRESS FOR MOV
E";T$
3060 IF ASC(T$)<65 OR ASC(T$)>90 THEN GO
TO 3030
3070 C=ASC(T$)-64
3080 PRINT SL$;
3090 INPUT "ROW ADDRESS FOR MOVE";R
3100 IF R>RM THEN GOTO 3080
3110 UD=1:IF R>RM-19 THEN UD=20-(RM-R)
3120 LR=1:IF C>23 THEN LR=C-22
3130 GOSUB 3430
3140 RETURN
3150 REM CURSOR UP--------------------
3160 IF R=1 THEN RETURN
```

```
3170 GOSUB 3770:GOSUB 3610:GOSUB 3660
3180 R=R-1
3190 UD=UD-1:IF UD=0 THEN UD=1:GOSUB 343
0:RETURN
3200 GOSUB 3770:GOSUB 3610:GOSUB 3690:GO
SUB 3720
3210 RETURN
3220 REM CURSOR DOWN------------------
3230 IF R=RM THEN RETURN
3240 GOSUB 3770:GOSUB 3610:GOSUB 3660
3250 R=R+1
3260 UD=UD+1:IF UD=21 THEN UD=20:GOSUB 3
430:RETURN
3270 GOSUB 3770:GOSUB 3610:GOSUB 3690: G
OSUB 3720
3280 RETURN
3290 REM CURSOR LEFT------------------
3300 IF C=1 THEN RETURN
3310 GOSUB 3770:GOSUB 3610:GOSUB 3660
3320 C=C-1
3330 LR=LR-1:IF LR=0 THEN LR=1:GOSUB 343
0:RETURN
3340 GOSUB 3770:GOSUB 3610:GOSUB 3690:GO
SUB 3720
3350 RETURN
3360 REM CURSOR RIGHT-----------------
3370 IF C=26 THEN RETURN
3380 GOSUB 3770:GOSUB 3610:GOSUB 3660
3390 C=C+1
3400 LR=LR+1:IF LR=5 THEN LR=4:GOSUB 343
0:RETURN
3410 GOSUB 3770:GOSUB 3610:GOSUB 3690:GO
SUB 3720
3420 RETURN
3430 REM REMAKE FULL DISPLAY----------
3440 CK=C:RK=R
3450 SC=C+1-LR:SR=R+1-UD
3460 PRINT "[CLR]";
3470 PRINT "    ";:REM 4 SPACES
3480 FOR N=SC TO SC+3
3490 PRINT "    ";CHR$(N+64);"    ";:REM
 4 SPACES
3500 NEXT N
```

```
3510 FOR N=SR TO SR+19
3520 C1=INT(N/10)
3530 PRINT " ";CHR$(C1+48);CHR$(N-10*C1+
48);" ";
3540 FOR M=0 TO 3
3550 C=SC+M:R=N:GOSUB 3770:GOSUB 3660
3560 NEXT M:NEXT N
3570 GOSUB 3610
3580 C=CK:R=RK
3590 GOSUB 3770:GOSUB 3690:GOSUB 3720
3600 RETURN
3610 REM SCREEN CURSOR TO CURRENT CELL-
3620 PRINT "[HOME]";
3630 PRINT LEFT$(CD$,UD);
3640 PRINT LEFT$(CR$,9*LR-5);
3650 RETURN
3660 REM DISPLAY CURRENT CELL -NORMAL--
3670 PRINT P$;
3680 RETURN
3690 REM DISPLAY CURRENT CELL -R.VIDEO-
3700 PRINT "[RVS]";P$;"[RVO]";
3710 RETURN
3720 REM PRINT COMMENTS----------------
3730 PRINT "[HOME]";LEFT$(CD$,21);
3740 PRINT S$;CL$;"CURRENT CELL =";CHR$(
C+64);"-";R;" CONTAINING;"
3750 PRINT S$;CL$;F$(C,R)
3760 RETURN
3770 REM SET P$ FOR CURRENT CELL-------
3780 IF LEFT$(F$(C,R),1)="H" THEN GOSUB
3970:GOTO 3940
3790 IF LEFT$(F$(C,R),1)="N" THEN GOSUB
5240
3800 IF N(C,R)=0 THEN P$=LEFT$(S$,8):GOT
O 3940
3810 P$=""
3820 NG=0:IF N(C,R)<0 THEN NG=1:N(C,R)=-
N(C,R)
3830 N1=INT(1000*N(C,R)+5)
3840 N1=INT(N1/10)
3850 N2=N1-10*INT(N1/10)
3860 IF N1=0 THEN N2=-16
3870 P$=CHR$(N2+48)+P$
```

```
3880 IF LEN(P$)=2 THEN P$="."+P$
3890 IF LEN(P$)<8 THEN GOTO 3840
3900 IF N1<10 THEN GOTO 3940
3910 IF N1>10000 THEN P$="OVERFLOW":GOTO
 3940
3920 P$=LEFT$(P$,7)
3930 GOTO 3840
3940 IF NG=1 THEN P$="-"+P$:N(C,R)=-N(C,
R):NG=0:GOTO 3960
3950 P$=" "+P$
3960 RETURN
3970 REM HEADING=8 CHARACTERS LONG
3980 P$=MID$(F$(C,R),9,9)
3990 P$=LEFT$(P$,LEN(P$)-1)
4000 IF LEN(P$)<8 THEN P$=P$+" ":GOTO 40
00
4010 RETURN
5000 REM#========N(C,R)================
5010 REM DECODE F$(C,R) TO GIVE N(C,R)
5020 REM================================
5030 T$=LEFT$(F$(C,R),1)
5040 IF T$="N" THEN GOSUB 5240:GOTO 5070
5050 IF T$="" OR T$="H" OR T$=" " THEN G
OSUB 5290:GOTO 5070
5060 GOSUB 5080
5070 RETURN
5080 REM CALCULATE F$(---) ------------
5090 N(C,R)=0
5100 LI=LEN(F$(C,R))
5110 FI=3
5120 GOSUB 6000
5130 IF EI<>0 THEN GOSUB 5320:GOTO 5230
5140 N(C,R)=CC
5150 FI=FI+1:IF FI>=LI THEN GOTO 5230
5160 Y$=T$:GOSUB 6000
5170 IF EI<>0 THEN GOSUB 5320:GOTO 5230
5180 IF Y$="+" THEN N(C,R)=N(C,R)+CC
5190 IF Y$="-" THEN N(C,R)=N(C,R)-CC
5200 IF Y$="*" THEN N(C,R)=N(C,R)*CC
5210 IF Y$="/" THEN N(C,R)=N(C,R)/CC
5220 GOTO 5150
5230 RETURN
5240 REM CALCULATE NUM(---) -----------
```

```
5250 LI=LEN(F$(C,R))
5260 X$=MID$(F$(C,R),5,LI-5)
5270 N(C,R)=VAL(X$)
5280 RETURN
5290 REM F$(C,R) BLANK OR HOLDS HEADING
5300 N(C,R)=0
5310 RETURN
5320 REM ERROR IN F$(C,R)--------------
5330 LR=1:UD=1:GOSUB 3430
5340 PRINT "[HOME]";LEFT$(CD$,21);
5350 PRINT S$;CL$;"ERROR AT CURRENT CELL
 =";CHR$(C+64);"-";R;" CONTENTS;"
5360 PRINT S$;CL$;F$(C,R);
5370 RETURN
6000 REM#========NEXT.F$()==============
6010 REM GET NEXT ELEMENT OF F$(C,R)
6020 REM================================
6030 EI=0
6040 T$=MID$(F$(C,R),FI,1)
6050 IF T$="C" THEN GOSUB 6180:GOTO 6070
6060 X$=T$:GOSUB 6080
6070 RETURN
6080 REM GET VALUE OF NUMERIC ELEMENT
6090 FI=FI+1:IF FI>LI THEN EI=1:GOTO 617
0
6100 T$=MID$(F$(C,R),FI,1)
6110 IF ASC(T$)>40 AND ASC(T$)<44 THEN G
OTO 6160
6120 IF T$="-" OR T$="/" THEN GOTO 6160
6130 IF ASC(T$)<45 OR ASC(T$)>57 THEN EI
=1:GOTO 6170
6140 X$=X$+T$
6150 GOTO 6080
6160 CC=VAL(X$)
6170 RETURN
6180 REM GET CELL ADDRESS AND CONTENTS-
6190 FI=FI+2:IF FI>LI THEN EI=1:GOTO 635
0
6200 T$=MID$(F$(C,R),FI,1)
6210 IF ASC(T$)<65 OR ASC(T$)>90 THEN EI
=1:GOTO 6350
6220 N1=ASC(T$)-64
6230 FI=FI+2:IF FI>LI THEN EI=1:GOTO 635
```

```
0
6240 T$=MID$(F$(C,R),FI,1)
6250 FI=FI+1:IF F!>LI THEN EI=1:GOTO 635
0
6260 X$=MID$(F$(C,R),FI,1)
6270 IF X$=")" THEN GOTO 6300
6280 T$=T$+X$
6290 GOTO 6250
6300 N2=VAL(T$)
6310 IF N2<1 OR N2>RM THEN EI=1:GOTO 635
0
6320 CC=N(N1,N2)
6330 FI=FI+1
6340 T$=MID$(F$(C,R),FI,1)
6350 RETURN
7000 REM#=======FORMULA================
7010 REM ENTER FORMULA AT CURRENT CELL
7020 REM================================
7030 J=0
7040 IF I=6 THEN X$="F(":LM=39
7050 IF I=7 THEN X$="NUM(":LM=39
7060 IF I=8 THEN X$="HEADING(":LM=16
7070 PRINT SL$;X$;
7080 GOSUB 2000:REM GET T$
7090 IF T$="C" AND I=6 THEN J=1
7100 IF T$=")" THEN X$=X$+T$:GOTO 7180
7110 IF T$="[CL]" THEN GOTO 7140
7120 IF LEN(X$)<LM THEN X$=X$+T$
7130 GOTO 7070
7140 REM DELETE 1 CHARACTER-----------
7150 IF I=6 AND RIGHT$(X$,1)=")" THEN J=
1
7160 X$=LEFT$(X$,LEN(X$)-1)
7170 GOTO 7070
7180 REM CHECK FOR END----------------
7190 IF I<>6 OR J=0 THEN GOTO 7220
7200 J=0
7210 GOTO 7070
7220 REM RESET DISPLAY & RETURN--------
7230 F$(C,R)=X$:GOSUB 3770
7240 GOSUB 3610:GOSUB 3690:GOSUB 3720
7250 RETURN
8000 REM#=======COPY==================
```

```
8010 REM COPY FORMULA TO NEW CELL
8020 REM================================
8030 EI=0
8040 REM GET COLUMN--------------------
8050 PRINT SL$;
8060 INPUT "COPY FORMULA TO CELL COLUMN"
;T$
8070 IF ASC(T$)<65 OR ASC(T$)>90 THEN GO
TO 8000
8080 CK=ASC(T$)-64
8090 REM GET ROW-----------------------
8100 PRINT SL$;
8110 INPUT "ROW";RK
8120 IF RK>RM THEN GOTO 8090
8130 REM COPY F$() IF NOT FORMULA------
8140 T$=LEFT$(F$(C,R),1)
8150 IF T$<>"F" THEN F$(CK,RK)=F$(C,R):G
OTO 8250
8160 REM COPY F$ CHAR. BY CHAR.--------
8170 F$(CK,RK)=T$
8180 LI=LEN(F$(C,R)):FI=2
8190 T$=MID$(F$(C,R),FI,1)
8200 IF T$="C" THEN GOSUB 8260
8210 IF EI<>0 THEN GOSUB 5320:GOTO 8250
8220 F$(CK,RK)=F$(CK,RK)+T$
8230 FI=FI+1
8240 IF FI<=LI THEN GOTO 8190
8250 RETURN
8260 REM CHANGE CELL REFERENCES--------
8270 GOSUB 6180
8280 IF EI<>0 THEN GOTO 8380
8290 FI=FI-1
8300 N1=N1+CK-C:N2=N2+RK-R
8310 T$="C("+CHR$(N1+64)+"-"
8320 X$=""
8330 N1=N2-10*INT(N2/10)
8340 X$=CHR$(N1+48)+X$
8350 N2=INT(N2/10)
8360 IF N2>0 THEN GOTO 8330
8370 T$=T$+X$+")"
8380 RETURN
9000 REM#=======EXECUTE==============
9010 REM RE-CALCULATE FULL SPREADSHEET
```

```
9020 REM================================
9030 PRINT SL$;"THINKING - PLEASE WAIT"
9040 CK=C:RK=R:EI=0
9050 FOR R=1 TO RM
9060 FOR C=1 TO 26
9070 GOSUB 5000:REM CALC. N(C,R)
9080 IF EI<>0 THEN C=26:R=RM
9090 NEXT C:NEXT R
9100 C=CK:R=RK
9110 IF EI<>0 THEN I=6:GOSUB 7000:GOTO 9
040
9120 GOSUB 3430:REM DISPLAY SHEET
9130 RETURN
10000 REM#=======LOAD/SAVE=============
10010 REM LOAD/SAVE TO TAPE/DISK
10020 REM================================
10030 PRINT "[CLR]";LEFT$(CD$,10);
10040 PRINT "PRESS 1 WHEN USING TAPE"
10050 PRINT "       2 FOR DISK (DEV=8)"
10060 PRINT "       3 TO QUIT NOW"
10070 GOSUB 2000:REM GET T$
10080 XI=ASC(T$)-48
10090 IF XI=3 THEN GOTO 10410
10100 IF XI<1 OR XI>2 THEN GOTO 10070
10110 XI=INT(XI+2.1*(J-3))
10120 PRINT "[CLR]";LEFT$(CD$,5);LEFT$(C
R$,7);
10130 PRINT "ENTER NAME OF DATA-FILE"
10140 IF XI=1 THEN PRINT "[RVS]RETURN[RV
O] WILL LOAD FIRST FILE ON TAPE"
10150 PRINT LEFT$(CD$,2):INPUT X$
10160 IF T$="" AND XI<>1 THEN GOTO 10120
10170 ON XI GOSUB 10440,10510,10650,10720
10180 IF EI<>0 THEN GOTO 10410
10190 IF XI>2 THEN GOTO 10300
10200 REM LOAD DATA--------------------
10210 FOR C=1 TO 26
10220 FOR R=1 TO RM
10230 INPUT#1,F$(C,R)
10240 INPUT#1,N(C,R)
10250 NEXT R:NEXT C
10260 CLOSE 1
10270 IF XI=2 OR XI=4 THEN CLOSE 15
```

```
10280 C=1:R=1:LR=1:UD=1
10290 GOTO 10410
10300 REM SAVE DATA--------------------
10310 CK=C:RK=R
10320 FOR C=1 TO 26
10330 FOR R=1 TO RM
10340 IF F$(C,R)="" THEN F$(C,R)=" "
10350 PRINT#1,F$(C,R)
10360 PRINT#1,N(C,R)
10370 NEXT R:NEXT C
10380 CLOSE 1
10390 IF XI=2 OR XI=4 THEN CLOSE 15
10400 C=CK:R=RK
10410 REM RESTORE AND RETURN-----------
10420 GOSUB 3430:REM DISPLAY SHEET
10430 RETURN
10440 REM OPEN INPUT CHANNEL FROM TAPE--
10450 PRINT "[CD][CD]WIND ON TAPE AS DES
IRED"
10460 PRINT "THEN PRESS [RVS]RETURN[RVO]"
10470 GET T$:IF T$<>CHR$(13) THEN GOTO 1
0470
10480 EI=0
10490 OPEN 1,1,0,X$
10500 RETURN
10510 REM OPEN INPUT CHANNEL FROM DISK--
10520 PRINT "[CD][CD]PUT CORRECT DISK IN
 DRIVE"
10530 PRINT "THEN PRESS [RVS]RETURN[RVO]
"
10540 GET T$:IF T$<>CHR$(13) THEN GOTO 1
0540
10550 T$=X$+",S,R"
10560 OPEN 15,8,15
10570 OPEN 1,8,2,T$:GOSUB 10920
10580 IF EI=0 THEN GOTO 10640
10590 PRINT LEFT$(CD$,4)
10600 PRINT "PRESS R TO RETRY"
10610 PRINT "ANYTHING ELSE ABORTS"
10620 GOSUB 2000:REM GET T$
10630 IF T$="R" THEN GOTO 10510
10640 RETURN
10650 REM OPEN OUTPUT CHANNEL TO TAPE---
```

```
10660 PRINT "[CD][CD]WIND ON TAPE AS DES
IRED"
10670 PRINT "THEN PRESS [RVS]RETURN[RVO]
"
10680 GET T$:IF T$<>CHR$(13) THEN GOTO 1
0680
10690 EI=0
10700 OPEN 1,1,1,X$
10710 RETURN
10720 REM OPEN OUTPUT CHANNEL TO DISK---
10730 PRINT "[CD][CD]PUT CORRECT DISK IN
 DRIVE"
10740 PRINT "THEN PRESS [RVS]RETURN[RVO]
"
10750 GET T$:IF T$<>CHR$(13) THEN GOTO 1
0750
10760 T$=X$+",S,W"
10770 OPEN 15,8,15
10780 OPEN 1,8,2,T$:GOSUB 10920
10790 IF EI=0 THEN GOTO 10910
10800 PRINT LEFT$(CD$,4)
10810 PRINT "PRESS R TO RETRY"
10820 PRINT "ANYTHING ELSE ABORTS"
10830 PRINT "R WILL OVERWRITE IF FILE EX
ITS"
10840 GOSUB 2000:REM GET T$
10850 IF T$<>"R" THEN GOTO 10880
10860 IF EI<>63 THEN GOTO 10880
10870 PRINT#15,"S0:"+X$
10880 CLOSE 1:CLOSE 15
10890 PRINT "[CLR]"
10900 GOTO 10720
10910 RETURN
10920 REM CHECK DISK ERROR STATUS-------
10930 INPUT#15,EI,EM$
10940 IF EI<20 THEN EI=0:GOTO 10980
10950 PRINT "[CLR][RVS]DISK ERROR"
10960 PRINT LEFT$(CD$,6);"ERROR NUMBER";
EI
10970 PRINT EM$;"[RVO]"
10980 RETURN
11000 REM#=======INSTRUCT=============
11010 REM DISPLAY INSTRUCTIONS
```

```
11020 REM================================
11030 PRINT "[CLR]     [RVS]INSTRUCTIONS[
RVO]"
11040 PRINT "[CD]THE FOLLOWING INPUTS AR
E ALLOWED;"
11050 PRINT "[CD][CD]T - MOVE TO ANY DES
IRED CELL"
11060 PRINT "[CD]CURSOR KEYS - MOVE AROU
ND  SPREADSHEET"
11070 PRINT "[CD]F - INPUT FORMULA TO CU
RRENT CELL"
11080 PRINT "[CD]N - INPUT A NUMBER TO C
URRENT CELL"
11090 PRINT "[CD]H - INPUT A HEADING TO
CURRENT CELL"
11100 PRINT "[CD]C - COPY A FORMULA TO A
NOTHER CELL"
11110 PRINT "[CD]X - EXECUTE: RE-CALCULA
TE SPREADSHEET"
11120 PRINT "[CD]Q - QUIT: CLEAR THE SPR
EADSHEET"
11130 PRINT "[CD]  PRESS ANY KEY TO CONT
INUE"
11140 GOSUB 2000:REM GET T$
11150 GOSUB 3430:REM DISPLAY SHEET
11160 RETURN
12000 REM#=======QUIT==================
12010 REM SET UP BLANK SPREADSHEET
12020 REM================================
12030 PRINT SL$;"CLEAR SPREADSHEET : [RV
S]Z[RVO] TO ZAP"
12040 GOSUB 2000:REM GET T$
12050 IF T$<>"Z" THEN GOTO 12120
12060 PRINT "[CLR]NEW SHEET COMING UP"
12070 FOR C=1 TO 26
12080 FOR R=1 TO RM
12090 F$(C,R)="":N(C,R)=0
12100 NEXT R:NEXT C
12110 C=1:R=1:LR=1:UD=1:GOSUB 3430
12120 RETURN
13000 REM#=======VARIABLES=============
13010 REM VARIABLES
13020 REM================================
```

```
13030 REM C=COLUMN NO. CURRENT CELL
13040 REM  =1 TO 26 (=A TO Z)
13050 REM CC=NUMERICAL ELEMENT OF F$()
13060 REM CD$=40*"[CD]"
13070 REM CK=TEMPORARY STORE FOR C
13080 REM CL$=40*"[CR]"
13090 REM CR$=40*"[CL]"
13100 REM EI=ERROR INDICATOR (0=OK)
13110 REM EM$=DISK ERROR MESSAGE
13120 REM FI=POS'N IN F$() WHEN DECODING
13130 REM F$(C,R)=FORMULA FOR CELL C,R
13140 REM I=INDICATOR: TYPE OF INPUT/F$
13150 REM I$=HOLDS PERMISSIBLE INPUTS
13160 REM J=")" ENDS F$() IF J=0
13170 REM LI=LENGTH OF F$()
13180 REM LM=MAX. LENGTH F$() ON INPUT
13190 REM LR=CURRENT CELL ON SCREEN
13200 REM   =1 TO 3 FOR LEFT/RIGHT
13210 REM N=LOOP VARIABLE
13220 REM N(C,R)=NUMERICAL VALUE OF F$()
13230 REM N1=TEMPORARY VALUE OF C OR R
13240 REM N2=TEMPORARY VALUE OF R OR C
13250 REM NG=INDICATOR FOR -VE N(C,R)
13260 REM P$=STRING FOR CELL DISPLAY
13270 REM R=ROW NO. OF CURRENT CELL
13280 REM  =1 TO RM
13290 REM RK=TEMPORARY STORE FOR R
13300 REM RM=MAX NO. ROWS ON SPREADSHEET
13310 REM S$=40*" "
13320 REM SC=TOP LEFT C FOR DISPLAY
13330 REM SL$=CURSOR MOVES ETC. TO WRITE
13340 REM     TO BOTTOM LINE
13350 REM SR=TOP LEFT R FOR DISPLAY
13360 REM T$=KEYBOARD INPUT NO.1
13370 REM UD=CURRENT CELL ON SCREEN
13380 REM   =1 TO 21 FOR UP/DOWN
13390 REM X$=KEYBOARD INPUT NO.2
13400 REM Y$=KEYBOARD INPUT NO.3
```

Glossary

ALGORITHM: A step-by-step definition of the solution to a problem, often used as the basis of a program.

ARTIFICIAL INTELLIGENCE: A field of research into the simulation of aspects of human intelligence.

ASSEMBLER CODE: The set of binary instructions that the processor chip uses, a 'low level' programming language.

AVERAGING PROCESS: The arithmetic average or 'mean' is obtained by adding all the values and then dividing the result by the number of values.

BASIC: Beginners All-purpose Symbolic Instruction Code.

BINOMIAL DISTRIBUTION: A common mathematical distribution.

BITS: A binary digit — a 0 or a 1 — the smallest piece of information a computer can handle.

BLOCK DIAGRAM: A coarse representation of a program, indicating the major subroutines, such as the main controlling routines, input and output routines, and specialist subroutines. Useful in program development.

CAUSAL DIAGRAM: A diagram which represents the possible responses of a system to different situations, and shows the links between possible system characteristics.

CLOSED LOOP: A loop which operates automatically and is contained within the system itself.

COMPILER: Software that can translate a 'high level' language into assembler code which the computer executes.

COMPRESSION RATIO: The ratio of the volume of uncompressed air above the piston at the bottom of its cycle, to that of compressed air when the piston is at the top of its cycle, ie maximum to minimum volume.

COMPUTER-AIDED DESIGN (CAD): Using a computer to design products.

CONFIDENCE LEVEL: An indication of the confidence with which the results of a calculation or prediction should be viewed.

CONTROL THEORY: The mathematics and techniques used to analyse control systems.

CRITICAL-PATH ANALYSIS: An analysis of a number of dependent and independent items, designed to indicate which of the items are critical to the optimised execution of all the items.

DETERMINISTIC PROCESS: A process in which the most probable outcome is always chosen.

EMPIRICAL: Defined by experimental results; not theoretical.
EXPERT SYSTEM: A program that imitates some aspects of human knowledge.
EXTERNAL EVENTS: Events which occur outside a defined system, and which may or may not affect the system.

FEEDBACK: The modification or control of a process or system by reference to its results or effects.
FINITE-ELEMENT MODEL: A model of a structure which represents the whole structure as a number of smaller, simple elements.
FLOWCHART: Diagrams which show the logical paths followed by a program. They can vary in detail, from a presentation of all the routines/processes and the order in which they are called, to an almost line-by-line description of a single routine or process.
FLUID DYNAMICS: A branch of physical science, concerned with the behaviour of fluids.
FORWARD DEPENDENCY: An indication of how events in the future will be affected by those happening in the present.
FUNCTION CHART: A chart or diagram which describes the components of a system in terms of their function within the system.
FUNCTIONAL DESCRIPTION: A description of a component of a system in terms of its function within that system.

GANTT CHART: A chart which shows a project plan broken down into tasks. Named after Henry Gantt.

HARDCOPY: Jargon for a printout.

INTEGER: A whole number, either positive or negative, ie 2, 100, −23 are integers, whereas −0.5, 3/2, 3006.10 are not.
INTERFACE: The interaction of a system with anything not defined as part of the system.
INTERNAL EVENTS: Events which occur within the defined system.
ITERATION: Repetition.

LIMITS: The definition of the extent of a system, in mathematical, spatial, logical, physical, or other appropriate terms.

LINEAR: A sequence of events in which one event is directly followed by another, ie in a line.

MACHINE CODE: See ASSEMBLER CODE.

MAINFRAME: A large computer.

MARKOV CHAIN PROCESS: A method of handling probabilities so that a number of outcomes can be modelled.

MATRIX OPERATIONS: A matrix is simply a rectangular array of numbers or formulae. Matrices of the same dimensions can be added to or subtracted from each other by adding or subtracting corresponding elements, eg:

$$\begin{bmatrix} 2 & 4 & 6 \\ 8 & 10 & 12 \end{bmatrix} + \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 2+1 & 4+2 & 6+3 \\ 8+4 & 10+5 & 12+6 \end{bmatrix} = \begin{bmatrix} 3 & 6 & 9 \\ 12 & 15 & 18 \end{bmatrix}$$

$$\begin{bmatrix} 2 & 4 & 6 \\ 8 & 10 & 12 \end{bmatrix} - \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 2-1 & 4-2 & 6-3 \\ 8-4 & 10-5 & 12-6 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

Matrices can also be multiplied together, which can be very useful if the matrices contain definitions of various system-states and the probabilities of transition between one state and another (Markov chain process).

Matrices can only be multiplied together if the number of rows in one is equal to the number of columns in the other, ie a 3 by 2 matrix may be multiplied by a 2 by 2 matrix, and the resultant matrix will have dimensions 3 by 2.

The multiplication process can be shown as follows:

$$\begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix} * \begin{bmatrix} A & B \\ C & D \end{bmatrix} = \begin{bmatrix} [a*A]+[b*C] \\ [c*A]+[d*C] \\ [e*A]+[f*C] \end{bmatrix} \begin{bmatrix} [a*B]+[b*D] \\ [c*B]+[d*D] \\ [e*B]+[f*D] \end{bmatrix}$$

each element of the product matrix comes from a row in the first matrix combined with a column in the second matrix.

MEAN: See AVERAGING PROCESS.

MENU-DRIVEN: Program which offers a list of possible choices from which the user can select.

MONTE CARLO PROCESS: In the Monte Carlo process, a single response is selected randomly according to the probabilities of all possible responses. In order to obtain an average response, the process is repeated (each cycle is termed a pass), and the results of all passes added and then divided by the number of passes performed. See Chapters 4 and 10.

OPEN LOOP: A loop which requires something external to complete the loop. See Chapter 4.

OPERATING SYSTEM: Software that takes care of all the operations required by most programs, such as displaying characters on the screen, accepting characters input from the keyboard, and operating micro or disk drives.

OPERATIONAL ANALYSIS: The study of the operations of a system, eg traffic flow, critical-path analysis, and time and motion studies.

OPTIMUM: The best or most favourable; not necessarily the maximum.

PARALLEL PROCESS: A process in which more than one outcome or state can be represented at any one time, eg Markov chain.

PERIPHERALS: Devices that are separate to the main computer, such as printers and joysticks.

POISSON DISTRIBUTION: A common mathematical distribution.

PREDICTION: A forecast of the future based on a knowledge of past events.

PROBABILITY: The probability of an event happening is defined as:

$$\frac{\text{The number of outcomes leading to the event}}{\text{The total number of possible outcomes}}$$

Assuming that all outcomes are equally likely. *Note:*

1) The probability of an event lies between 0 and 1. The probability of a certain event is 1, and that of an impossible event is 0.

2) If the probability of an event happening is p, then the probability of it not happening is 1 – p.

3) If two independent events have the probabilities p1 and p2, then the probability of both the events happening is p1*p2.

REAL-TIME PROCESS: A process for which the simulated time-step is identical to the actual time taken.

RELIABILITY: A measure of the failure of a system, usually as a probability or percentage, derived from the results of tests of the system or a record of failures encountered whilst using the system.

RELIABILITY GROWTH: An indication of trends in reliability, generally produced for a new product or piece of hardware, in order to estimate when it will be sufficiently reliable to be manufactured.

RICH PICTURES: Usually a freehand drawing depicting parts of a system.

SENSITIVITY ANALYSIS: Series of tests designed to indicate how responsive a simulation is to its various parameters.

SERIAL OPERATIONS: Operations that take place consecutively.

SUB-SYSTEM
SYSTEMS
SYSTEMS ANALYSTS } See Chapter 2.

SYSTEM-STATE: Definition of the attributes of a system such that its state can be identified. See Chapters 4 and 9.

SYSTEM-STATE MATRIX: An array (matrix) which contains information about various attributes of a system, and uniquely defines a number of possible states of the system. See Chapters 4 and 9.

THERMODYNAMICS: A branch of physical science concerned with the effects of heat energy.

TIME-STEP: Time between consecutive system states as represented by the simulation.

TIME-TRANSIENT: Changing with time.

TRANSFORMATION MATRIX: An array (matrix) which contains the probabilities of a system changing from one state to another. Multiplication of the transition matrix with the system-state matrix will give an update of the system-state. See Chapters 4 and 9.

UTILISATION: A measure of the actual use of available resources.

VERIFICATION: Confirmation of correctness.

Other titles from Sunshine

SPECTRUM BOOKS

Artificial Intelligence on the Spectrum Computer
Keith & Steven Brain — ISBN 0 946408 37 8 — **£6.95**

Spectrum Adventures
Tony Bridge & Roy Carnell — ISBN 0 946408 07 6 — **£5.95**

Machine Code Sprites and Graphics for the ZX Spectrum
John Durst — ISBN 0 946408 51 3 — **£6.95**

ZX Spectrum Astronomy
Maurice Gavin — ISBN 0 946408 24 6 — **£6.95**

Spectrum Machine Code Applications
David Laine — ISBN 0 946408 17 3 — **£6.95**

The Working Spectrum
David Lawrence — ISBN 0 946408 00 9 — **£5.95**

Inside Your Spectrum
Jeff Naylor & Diane Rogers — ISBN 0 946408 35 1 — **£6.95**

Master your ZX Microdrive
Andrew Pennell — ISBN 0 946408 19 X — **£6.95**

COMMODORE 64 BOOKS

Graphic Art for the Commodore 64
Boris Allan — ISBN 0 946408 15 7 — **£5.95**

DIY Robotics and Sensors on the Commodore Computer
John Billingsley — ISBN 0 946408 30 0 — **£6.95**

Artificial Intelligence on the Commodore 64
Keith & Steven Brain — ISBN 0 946408 29 7 — **£6.95**

Machine Code Graphics and Sound for the Commodore 64
Mark England & David Lawrence — ISBN 0 946408 28 9 — **£6.95**

Commodore 64 Adventures
Mike Grace — ISBN 0 946408 11 4 — **£5.95**

Business Applications for the Commodore 64
James Hall — ISBN 0 946408 12 2 — **£5.95**

Mathematics on the Commodore 64
Czes Kosniowski — ISBN 0 946408 14 9 — **£5.95**

Advanced Programming Techniques on the Commodore 64
David Lawrence — ISBN 0 946408 23 8 — **£5.95**

Commodore 64 Disk Companion
David Lawrence & Mark England ISBN 0 946408 49 1 **£7.95**

The Working Commodore 64
David Lawrence ISBN 0 946408 02 5 **£5.95**

Commodore 64 Machine Code Master
David Lawrence & Mark England ISBN 0 946408 05 X **£6.95**

Machine Code Games Routines for the Commodore 64
Paul Roper ISBN 0 946408 47 5 **£6.95**

Programming for Education on the Commodore 64
John Scriven & Patrick Hall ISBN 0 946408 27 0 **£5.95**

Writing Strategy Games on your Commodore 64
John White ISBN 0 946408 54 8 **£6.95**

ELECTRON BOOKS

Graphic Art for the Electron Computer
Boris Allan ISBN 0 946408 20 3 **£5.95**

Programming for Education on the Electron Computer
John Scriven & Patrick Hall ISBN 0 946408 21 1 **£5.95**

BBC COMPUTER BOOKS

Functional Forth for the BBC Computer
Boris Allan ISBN 0 946408 04 1 **£5.95**

Graphic Art for the BBC Computer
Boris Allan ISBN 0 946408 08 4 **£5.95**

DIY Robotics and Sensors for the BBC Computer
John Billingsley ISBN 0 946408 13 0 **£6.95**

Essential Maths on the BBC and Electron Computer
Czes Kosniowski ISBN 0 946408 34 3 **£5.95**

Programming for Education on the BBC Computer
John Scriven & Patrick Hall ISBN 0 946408 10 6 **£5.95**

Making Music on the BBC Computer
Ian Waugh ISBN 0 946408 26 2 **£5.95**

DRAGON BOOKS

Advanced Sound & Graphics for the Dragon
Keith & Steven Brain ISBN 0 946408 06 8 **£5.95**

Artificial Intelligence on the Dragon Computer
Keith & Steven Brain ISBN 0 946408 33 5 **£6.95**

Dragon 32 Games Master
Keith & Steven Brain — ISBN 0 946408 03 3 — **£5.95**

The Working Dragon
David Lawrence — ISBN 0 946408 01 7 — **£5.95**

The Dragon Trainer
Brian Lloyd — ISBN 0 946408 09 2 — **£5.95**

ATARI BOOKS

Atari Adventures
Tony Bridge — ISBN 0 946408 18 1 — **£5.95**

Writing Strategy Games on your Atari Computer
John White — ISBN 0 946408 22 X — **£5.95**

SINCLAIR QL BOOKS

Introduction to Simulation Techniques on the Sinclair QL
John Cochrane — ISBN 0 946408 45 9 — **£6.95**

Quill, Easel, Archive & Abacus on the Sinclair QL
Alison McCallum-Varey — ISBN 0 946408 55 6 — **£6.95**

GENERAL BOOKS

Home Applications on your Micro
Mike Grace — ISBN 0 946408 50 5 — **£6.95**

Sunshine also publishes

POPULAR COMPUTING WEEKLY

The first weekly magazine for home computer users. Each copy contains Top 10 charts of the best-selling software and books and up-to-the-minute details of the latest games. Other features in the magazine include regular hardware and software reviews, programming hints, computer swap, adventure corner and pages of listings for the Spectrum, Dragon, BBC, VIC 20 and 64, ZX 81 and other popular micros. Only 40p a week, a year's subscription costs £19.95 (£9.98 for six months) in the UK and £37.40 (£18.70 for six months) overseas.

DRAGON USER

The monthly magazine for all users of Dragon microcomputers. Each issue contains reviews of software and peripherals, programming advice for beginners and advanced users, program listings, a technical advisory service and all the latest news related to the Dragon. A year's subscription (12 issues) costs £10 in the UK and £16 overseas.

MICRO ADVENTURER

The monthly magazine for everyone interested in Adventure games, war gaming and simulation/role-playing games. Includes reviews of all the latest software, lists of all the software available and programming advice. A year's subscription (12 issues) costs £10 in the UK and £16 overseas.

COMMODORE HORIZONS

The monthly magazine for all users of Commodore computers. Each issue contains reviews of software and peripherals, programming advice for beginners and advanced users, program listings, a technical advisory service and all the latest news. A year's subscription costs £10 in the UK and £16 overseas.

For further information contact:
Sunshine
12–13 Little Newport Street
London WC2R 3LD
01-437 4343

Telex: 296275

Printed in England by Commercial Colour Press, London E7.

Here is a book written for people who know what they want from their Commodore 64 but do not know how to achieve it. John Cochrane has produced an eminently practical introduction to the world of computer simulation, progressing from the basics to the development of powerful and useful techniques.

The beginnings of computer simulation are described and a down-to-earth explanation of the various types of simulation and their use is given. Simple examples are used throughout the book to give the reader 'hands on' experience and to develop a thorough understanding of the principles employed.

The book will give you the ability to look at any problem and apply a method of analysis that allows you to develop your own simulations. Practical examples include keeping time, making music, predicting the weather and using a spreadsheet. Working from BASIC, the examples show you how to make the most of your Commodore 64.

John Cochrane is a consultant engineer. He is an experienced programmer and an expert in computer simulations. His experience stretches back for over fifteen years and includes work on many major simulations.



SUNSHINE

ISBN 0 946408 58 0



This was brought to you

from the archives of

http://retro-commodore.eu