DISCRETE VISUAL SIMULATION WITH
Pascal.SIM

Robert M. O'Keefe
Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061, U.S.A.

Ruth M. Davies
Department of Mathematical Sciences and Computing
South Bank Polytechnic

ABSTRACT

Pascal.SIM is a collection of Pascal constants, types, variables, functions and procedures for developing event, activity, three-phase or process oriented discrete-event simulation models. Facilities are provided for queue processing, time advance and event list maintenance, control of entities and resources, random number generation and streams, and samppling from parametric and empirical distributions, statistics collection, and visual displays. Pascal.SIM has been designed as a minimal simulation tool. It includes less than 50 functions and procedures, and totals less than 800 lines of code. It is a basis for programming simulations in Pascal, where users can alter or extend the facilities provided, rather than a simulation programming language. The majority of Pascal.SIM conforms to the ISO Pascal standard, enabling high portability to be achieved. It can be used immediately with any Pascal that uses the string type descended from UCSD Pascal, for instance Pro Pascal, Turbo Pascal or Sheffield PRIME Pascal. Alteration of a few lines allows for use with any Pascal that provides a different string type, for instance VAX/VMS Pascal. This paper gives a tutorial presentation of Pascal.SIM, with emphasis on the facilities for visual displays.

1. INTRODUCTION

Recent years have seen some resurgence in interest in the use of general purpose programming languages as vehicles for simulation programs. In part this has been due to the increasing availability of good implementations for strongly-typed block structured languages such as Pascal, Ada, and Modula-2. Further, the use of microcomputers has accelerated such interest, since many Simulation Programming Languages and packages are too large for use on microcomputers, or when available on microcomputers, are highly inefficient.

To facilitate simulation on microcomputers, in 1982 the authors produced a system for programming discrete simulations in Pascal on the Apple II. Called AIMS (O'Keefe, 1983; O'Keefe and Davies, 1986a), it was composed of seven UCSD Pascal library units and a number of associated utilities which the model developer used to construct a simulation as a UCSD Pascal program. At about this time, other Pascal based systems were in development, including PASSIM (Uyenzo and Vaessen, 1980), based on GPSS, and SIMPAS (Bryant, 1980), a SIMSCRIPT-like language which is pre-processed into Pascal.

AIMS had a number of useful and sophisticated features, in addition to the more basic ones such as queue processing and sampling from parametric distributions. One of the units provided facilities for iconic visual displays (this was programmed in assembler, and made direct use of the Apple II's graphic functions in ROM), and another provided for continuous display of time series. Associated utilities included an editor for distribution data, which allowed for the formation of empirical distributions, and a shape editor, where an icon could be defined for future use in visual displays.

AIMS died with the relative demise of the Apple II and the shift to MS-DOS. However, much of it was reprogrammed entirely in Pascal to provide a highly portable discrete simulation system, and was rechristened Pascal.SIM. Various versions of Pascal.SIM have been in use in both education and industry for over 3 years.

2. THE DESIGN OF Pascal.SIM

The philosophy of Pascal.SIM is to provide the basics of a Simulation Programming Language, and little more. Those using Pascal.SIM can then add the facilities they need and change the underlying structure if they wish. A further aim is to provide a means whereby students could learn to write simulations in a familiar language using facilities that are well documented and easy to understand. Visual Interactive Simulation and animation has been very successful (Bell, 1985); therefore some facilities for iconic visual displays have been included.

AIMS enforced the three-phase world view, as first proposed by Tocher, where a simulation is perceived as a number of bound or scheduled events, plus a number of conditional events which are scanned. Pascal.SIM can be used to program a three-phase or a two-phase simulation (ie. pure event scheduling or activity scanning), or using an additional version of the executive, a simple process description simulation.

3. THE STRUCTURE OF A Pascal.SIM PROGRAM

3.1. The Three-Phase Method

The recommended structure of a three-phase orientated Pascal.SIM program is shown in Figure 1. At the heart of the simulation is the executive, the procedure run, which contains the time flow mechanism. Although the structure of this is provided, the user must enter the names of all events into a
case statement in this procedure. The number of conditional events \( \max.C \) and the time duration (\( \text{duration} \)) are both arguments. The user must code the events and the procedures \( \text{initialize} \) and \( \text{report} \); for a visual display \( \text{display} \) and \( \text{picture} \) must also be coded. \( \text{initialize} \) and \( \text{picture} \) are called once before \( \text{run} \); they initialize the simulation and the static picture respectively. \( \text{Report} \) should be called after \( \text{run} \), and should contain any end of run reporting, for example, final statistics prints. \( \text{Display} \) is called after every advance of the clock; it should be used to update the visual display as necessary.

### 3.2. The Two Phase Methods

The three-phase executive can be used for the two-phase event scheduling approach, by setting \( \max.C \) to zero and incorporating all the conditional event logic into the scheduled events. Similarly, a two-phase activity scanning approach can be used by incorporating all scheduled events into the model as conditional events.

### 3.3. The Process View

A separate executive has been written for this approach. Whilst not process interaction, in that processes can not signal each other, descriptions of independent processes are possible. This is sometimes referred to as process description. Further, both servers and transactions can have process descriptions - thus the approach is conceptually closer to process interaction than GPSS. Each process is written as a separate procedure; the user must enter the names of all processes into the executive \( \text{run} \).

### 3.4. The Entity

The basis of Pascal.SIM is an entity type, which is a Pascal record thus:

```
entity = "an_entity;"
  an_entity = packed record
    aval := boolean;
    class := class.num;
    col := colour;
    attr, next.B := cardinal;
    time := real;
  end;
```

where the fields of the record represent:

- **aval:** The availability of the entity
- **class:** The number of the entities class
- **col:** The colour of the entity
- **attr:** The entities attribute number
- **next.B:** The next bound event or block that the entity will enter
- **time:** The time at which this will occur

An entity is always either available, entered in the calendar of future events, or is being used by another entity. If entered in the calendar, \( \text{aval} \) is false, and \( \text{next.B} \) and \( \text{time} \) will be set to appropriate values. Thus there are no explicit event notices in Pascal.SIM because an entity contains all relevant event inform-

```
{ Bound events }

procedure B1;
procedure B2;
  :
  :

{ Conditional events }

procedure C1;
procedure C2;
  :
  :

procedure display;
procedure run(duration: real; \( \max.C \) : cardinal);
procedure initialize;
procedure picture;
procedure report;

begin
  initialize;
  picture;
  :
  run(..., ...);
  :
  report;
  :
end.
```

Figure 1: The structure of a three-phase Pascal.SIM program

mation. Entities are generated with the function \( \text{new.entity} \), and can be disposed of with the procedure \( \text{dis.entity} \).

Access to entities is achieved through the global variable \( \text{current} \), which always points to the entity that has caused the present event, or else by searching queues of entities.

The attribute number \( \text{attr} \) uniquely identifies each entity. If further attributes are required they can either be added to entities by using the attribute number to access another data structure, or else by adding in new fields to the entity record and recompiling Pascal.SIM.

In complex models, the developer would establish classes, where a class is a list of entities, and both the class and each entity may have attributes. For visual displays, the developer must enter classes into a class table, which holds information on the letter and colour used to represent an entity in the display.

### 3.5. Resources

A resource type, with associated routines, is provided to model passive entities which only serve. Resources are collected into a bin - in effect a bin is identical to the \( \text{STORAGE} \) of GPSS; a bin with only one resource is identical to a \( \text{FACILITY} \). Resources are said to be acquired and released by entities.

### 3.6. Functions and Procedures

The provided functions and procedures of Pascal.SIM are grouped into 11 groups, respectively:

```
518
queue processing
entities and classes
timing and the executive
facilities for process description
resources
error messages
random number generation and streams
sampling distributions
histograms
screen control
visual displays

The interface of Pascal.SIM, i.e. all constants, types, global variables and function and procedure heads, is shown in the appendix A. A certain excess redundancy is present in the four routines (give_top, give_tail, take_top, take_tail) which allow for giving and removing entities to the top and tail of a queue. Use of these allow students to develop First In First Out queueing models without having to explicitly dereference pointers.

4. AN EXAMPLE - ADMISSION TO HOSPITAL

The example to demonstrate how to program using Pascal.SIM is a hospital simulation, shown as an activity diagram in Figure 2. Two types of patients are admitted to hospital. Those not admitted for an operation undergo a short stay, and then return home. Patients admitted for operation undergo a pre-operative stay, an operation (which requires an open and available operating theatre), followed by a post-operative stay and discharge. Such a simulation is somewhat simplistic, but might be used to investigate various policies regarding bed and operating theatre provision.

Appendix B shows a Pascal.SIM three-phase orientated program for a visual simulation. An example visual display is shown in Figure 3. In the initialize procedure, the simulation and random number streams are initialized (via make_sim and make_streams), a bin called bed with 4 resources is created using make.bin, and the queues q1,q2,q3 and q4 are initialized using make_queue. The operating theatre is created, and scheduled to close in 8 hours. Note that cause is the scheduling procedure; the first parameter indicates the bound event that will be entered. This has to be specified as an integer, since Pascal does not allow procedure names to be passed as parameters and stored for future calling. Case statements in the executive relate these numbers to procedures calls.

4.1. Programming the Visual Display using the Three-Phase Approach

The visual display is composed of two parts - a static background picture which is written to the console once prior to the simulation run, and a dynamic display which moves over the static picture. The dynamic display can be updated either within an event, bound or conditional, or following a time beat (where one or more events will have been executed) in the procedure display.

Even with using text to program simple iconic visual displays, a minimum amount of screen control is essential. Cursor addressing must be possible; for colour displays the ability to set both foreground and background colour is necessary. Many terminals provide both of these, and thus the visual display routines are highly portable.

A static picture is created in the procedure picture. Entity classes 1 and 2 (respectively hospital stay only and operation patients) are entered in the class_table with letters 's' and 'o'. Both will appear blue, unless the field col in the entity record has been set to a colour - this overrides the class_table entry. To provide a background, blocks coloured magenta are entered in the display, and some simple annotation is provided using the gotoxy procedure in Pascal.SIM and the standard Pascal procedure write.

The procedure display provides for updating of the dynamic display after a time beat. The number of beds in use (bed.number-bed.num_avail) is written. At this point, the display is completely up to date. The simulation is then delayed relative to the time before the new time beat (tim-old.tim). If this is not done, the display advances too quickly for comfortable viewing. The new clock time tim is then written to the display, and the simulation (and thus the part of the picture generated within events) can continue.

Most of the visual display statements are embedded in the events. For instance, when a hospital stay only patient arrives, the following occurs (see procedure patient1.arrives) :-

- put patient on a queue for a bed
- show the arrival of the patient by horizontal movement display the hospital stay only queue for beds cause the arrival of a new hospital stay only patient
This means that the developer of a visual simulation must introduce dummy queues at various points in a process so that the queues can be written to the picture. (This is analogous to having to use dummy queues to collect statistics in GPSS.) Those interested in process description models in Pascal.SIM should refer to O’Keefe and Davies (1986b), which includes a process version of the hospital example.

6. PORTABILITY AND IMPLEMENTATION

Considerable portability is achieved by close adherence to the ISO Pascal standard. Only two non-standard Pascal facilities are used - the use of an underscore in names (which can easily be edited out), and the use of a string type. However, most Pascal implementations provide a string type, and Pascal.SIM can be implemented without change under any Pascal that uses the string type and associated functions descended from UCSD Pascal. Examples include Turbo Pascal, Pro Pascal, and the Pascal compiler for PRIME systems produced at the University of Sheffield in England. If a string type is defined differently, or different functions are provided, a few alterations are necessary. For instance, in VAX/VMS Pascal, the string type is varying array of char rather than string, and strings are concatenated directly using the addition operator rather than a concat function. If strict ISO Pascal is followed, and a packed array of char has to be used, then only one procedure is unusable. This is print.histogram, which prints histograms to a text file.

Pascal.SIM is normally implemented by some method of prior compilation. Methods include adding the functions and procedures to a library and the variables to a common area (this is the method of implementation in Pro Pascal), production of a unit or module, containing all of Pascal.SIM, that is then put in a library (for instance, UCSD Pascal), or by a similar method (for instance, implementation in VAX/VMS Pascal is achieved by production of an environment file for the constants, types and variables, and an associated module for the functions and procedures). Thus the facilities are available to any Pascal program by simple reference to the library, unit or whatever. Pascal.SIM has been used extensively with Turbo Pascal, which provides no facilities for separate compilation. Here the programmer must recompile Pascal.SIM with the simulation.

To implement Pascal.SIM, it is necessary to set up the screen control codes within a number of procedures. Many terminals can be made to accept ANSI screen control codes (for instance, IBM-PC monitors) or use an extension of ANSI (for instance, DEC VT100 and VT240). Thus ANSI screen control (and the extended ANSI descended from Textronix for colour text) is frequently sufficient. Additional copies of some screen control and visual display routines are provided for use with Turbo Pascal, which call the screen control routines built into Turbo Pascal.

Two random number generators are provided - one for 32-bit integer machines, one for 16-bit integer machines. These are respectively the linear congruence generators

\[ Z_{i+1} := (Z_i \times 16807) \mod 2147483647 \]
Discrete Visual Simulation with Pascal SIM

and 
\[ Z_{i+1} := (Z_i + 3993 + 1) \mod 32767. \]

The 16-bit version is an implementation of a generator suggested and tested by Teshen, Sun and Wang (1984), which assumes that detection of integer overflow has been disabled. (Incidentally, the authors have found the built in mathematical functions of Turbo Pascal, for instance, exp and sin, to be poor. Hence distribution sampling methods that employ these, for instance the Box-Muller method for normal variates, provide relatively poor sets of samples, with too few samples from the tail of the distribution.)

7. CONCLUSIONS

The authors have mainly used Pascal SIM with Turbo Pascal and VAX/VMS Pascal. Pascal SIM, Turbo Pascal, a colour monitor, and an IBM-PC/XT or AT allow visual simulations of reasonable display quality to be developed and run. For statistical experimentation, the model can then be ported to a VAX, and the Pascal SIM statements relating to the visual display replaced by statements for statistics collection using histograms (an area of Pascal SIM that has not been covered in this paper).

Programming visual simulations can be time consuming, and typically in the hospital example there are more programmed statements relating to the display than to the logic of the simulation. This is true for other programming language orientated visual simulation systems, for example SEE-WHY (Fiddy, Bright and Hurriion, 1981).

The authors have found the three-phase world view the best for visual simulation. The three-phase method allows the picture to be updated after time dependent changes (bound events), state changes (conditional events), or time beats as appropriate. However, having the range of world views in one package, including two-phase, three-phase and process description views, is very useful for teaching. Students can program models using a number of views, and thus obtain a better understanding of frameworks for simulation model building than when using one approach.

The value of producing a Pascal based simulation tool may be considered questionable, given the recent emphasis on the entire process of model development (Nance, 1984), and the promise of Artificial Intelligence (O'Keefe, 1986). Many simulations are, however, still programmed in FORTRAN (Christy and Watson, 1983). Increasingly students of science and engineering subjects are learning Pascal as their main programming language. They will undoubtedly want to write simulations in Pascal. Pascal SIM provides a structure and the facilities to do this.

Acknowledgements

Many of the ideas in Pascal SIM can be traced back to a Pascal based system produced by John Crookes at the University of Lancaster, England.

This paper was completed whilst the first author was on leave from the Board of Studies in Management Science, University of Kent at Canterbury, England.

Pascal SIM is available on an IBM-PC disc for a nominal fee. It can be obtained from either of the authors or Decision Computing, 1 Worthgate Place, Canterbury, England. However, swift response to any request for Pascal SIM is not guaranteed! Please write - do not phone.

The following are trademarks:

- Pro Pascal: Prospero Software Limited
- Turbo Pascal: Borland International
- VAX/VMS: DEC
- IBM-PC: IBM
- UCSD Pascal: Regents of the University of California
- MS-DOS: Microsoft
- Ada: United States Department of Defence

REFERENCES


APPENDIX A: Pascal SIM FACILITIES

```plaintext
cost max.cell.num=16;
max.stream.num=52;
max.class.num=256;
max.sample.num=20;
```
max_string_length=80;
delay_num=2000;

type a.string=string[max_string_length];
cardinal=0..maxint;
colour=(null,black,red,green,yellow,blue,magenta,cyan,white);

stream_num=1..max_stream_num;
cell_num=0..max.cell_num;
class_num=1..max.class_num;
sample_num=1..max.sample_num;
string_length=1..max_string_length;

entity="an.entity;"
link="a.link;"
a.link=record
   next,pre:link;
   item:entity;
end;
queue=link;

an.entity=packed record
   avail:boolean;
   class:class_num;
   col:colour;
   attr,next,B:cardinal;
   time:real;
end;

bin=record
   number, num.avail:cardinal;
end;

histogram=record
   cell:array[cell_num] of real;
   count, width, base, total, eosq, min, max:real;
end;

lookup_table=table[1..max.sample_num,1..2] of real;

var
tim:real;
current:entity;
calendar:queue;
on_calendar:boolean;
suspended.chain:queue;
running:boolean;
original.seeds:seeds:array [stream_num] of cardinal;
class.table:array [class_num] of
record
   let:char;col:colour;
end;

{ queue processing }
procedure make.queue(var qqueue);
procedure give(qqueue,link:entity);
function take(qqueue:link:entity);
procedure give.top(qqueue:entity);
procedure give.tail(qqueue:entity);
function take.top(qqueue:entity);

function take.tail(qqueue:entity);
function empty(qqueue:boolean;

{ entities and classes }
function new.entity(class.num:cardinal:entity;
procedure del.entity:entity;
procedure make.class(var cqueue,p:size:cardinal);
function count(var qqueue:cardinal;

{ timing and the executive }
procedure make.sim;
procedure cause(abeced,cardinal:entity:real;
procedure calendar.top;

{ facilities for process executive }
procedure branch(next:cardinal;
procedure remove.entity;

{ resources }
procedure make.bin(var from:bin:cardinal;
procedure acquire(var from:bin:cardinal;
procedure return(var from:bin:cardinal;

{ error messages }
procedure sim.error(a:cardinal;

{ random number generator and streams }
procedure make.streams;
procedure md(ess:stream_num:real;

{ sampling distributions }
function normal(u:stream_num:real;
function log.normal(u:stream_num:real;
function poisson(u:stream_num:real;
function negexp(u:stream_num:real;
function uniform(l:real:stream_num:real;
function make.sample(var sample.file:text;
   var table:lookup_table;
function sample(sample:table:lookup_table:stream_num:real;

{ histograms }
procedure reset.histogram(var h:histogram;
procedure make.histogram(var h:histogram;
   cell.base,cell.width:real;
procedure print.histogram(var pr:text;h:histogram;
   state:boolean,plen:cardinal;
procedure log.histogram(var h:histogram;where,what:real;

{ screen control }
procedure make.screen;
procedure gotoxy(x,y:cardinal;
procedure clear.screen;
procedure set.foreground(c:colour;
procedure set.background(c:colour;
procedure reset.colours;

{ visual displays }
procedure delay;
procedure make.class_table;  
procedure enter.class(n:cardinal;char:;colour:;);  
procedure write.entity(x,y:cardinal;entity:);  
procedure write.queue(x,y:cardinal;  
  bcolou:;queue:;max:;length:cardinal);  
procedure write.block(x,y:cardinal;bcolou:;);  
procedure move.v(x,y:cardinal;entity:;bcolou:;);  
procedure move.h(y:cardinal;entity:;bcolou:;);  
procedure write.time;  

{ user written routines }  
procedure display;  
procedure initialize;  
procedure picture;  
procedure report;  

{ simulation executive }  
procedure run(duration:real;max.C:cardinal);  

APPENDIX B: THE HOSPITAL EXAMPLE  

program example;  
var  
  bed:bin;  
  q1,q2,q3,q4:queue;  
  theatre:entity;  
  theatre.open, theatre.available:boolean;  
  { true if theatre is open and available }  
  old.tim:real;  

procedure patient1.arrives: { stay } { B1 }  
begin  
give.tail(q1,curr);  
move.h(12,2,10,curr,white);  
write.queue(22,12,white,q1,10);  
cause(1,new.entity(1,1),uniform(60,140,1));  
end;  

procedure patient2.arrives: { operation } { B2 }  
begin  
give.tail(q2,curr);  
move.h(14,2,10,curr,white);  
write.queue(22,14,white,q2,20);  
cause(2,new.entity(2,1),uniform(24,48,2));  
end;  

procedure end.hospital.stay: { B3 }  
begin  
return(bed,1);  
move.h(12,40,70,curr,white);  
dis.entity(curr);  
end;  

procedure end.pre-operative.stay: { B4 }  
begin  
curr.col:=yellow;  
give.tail(q3,curr);  
move.v(30,14,20,curr,white);  
move.h(20,30,50,curr,white);  
write.queue(60,20,white,q3,30);  
end;  

procedure end.operation: { B5 }  
begin  
thear:available:=true;  
gotoxy(63,21);write(‘.’);  
mov.e(30,4,10,curr,white);  
give.tail(q4,curr);  
end;  

procedure end.post-operative.stay: { B6 }  
begin  
return(bed,1);  
move.h(12,40,70,curr,white);  
dis.entity(curr);  
end;  

procedure open.theatre: { B7 }  
begin  
thear:open:=true;  
gotoxy(63,20);write(‘OPEN ’);  
cause(8,curr,8);  
end;  

procedure close.theatre: { B8 }  
begin  
thear:open:=false;  
gotoxy(63,20);write(‘CLOSED’);  
cause(7,curr,40);  
end;  

procedure start.hospital.stay: { C1 }  
begin  
while (bed.num:avail>0)  
  and (not empty(q1)) do  
begin  
acquire(bed,1);  
cause(3, take.top(q1),uniform(20,40,3));  
write.queue(22,12,white,q1,20);  
end;  
end;  

procedure start.pre-operative.stay: { C2 }  
begin  
while (bed.num:avail>0)  
  and (not empty(q2)) do  
begin  
acquire(bed,1);  
cause(4, take.top(q2),uniform(8,15,4));  
write.queue(22,14,white,q2,20);  
end;  
end;  

procedure start.operation: { C3 }  
begin  
while theatre.open and theatre.available  

523
and (not empty(q2)) do
begin
theatre.available:=false;
cause(5, take.top(q3), 1);
gotoxy(63, 21); write('IN USE');
write.queue(60, 20, white, q3, 30);
end;
end;

procedure start_post_operative_stay; { C4 }
begin
while not empty(q2) do
begin
  cause(6, take.top(q4), uniform(5, 10, 6));
end;
end;

procedure display;
begin
  gotoxy(30, 12); write(bed.number-bed.num.avail:1);
delay:delay;
for i:=1 to trunc((tim-old.tim)/2) do delay;
old.tim:=tim;
goxy(1, 1); writeln(tim:7:2);
goxy(1, 1);
end { display };

procedure run(duration:real; max.C:cardinal);
var i:cardinal;
begin
  running:=true;
repeat
    if calendar=calendar-.next then running:=false
    else begin
display;
tim:=calendar-.next-.item-.time;
if duration*tim then running:=false
else begin
  while (calendar<calendar-.next-and (tim<calendar-.next-.item-.time)) do
begin
  case current-.next.B of 0::
1:patient1.arrives;
2:patient2.arrives;
3:end.hospital.stay;
4:end.pre.operative.stay;
5:end.operation;
6:end.post.operative.stay;
7:open.theatre;
8:close.theatre;
end;
end;
for c:=1 to max.C do
begin
  start.hospital.stay;
end;
end;
end { run };

procedure initialize;
begin
make.sim;
make.streams;
make.bin(bed, 4);
make.queue(q1); make.queue(q2);
make.queue(q3); make.queue(q4);
{ create theatre }
theatre:=new.entity(3, 1);
theatre.open:=true;
theatre.available:=true;
cause(5, theatre, 6);
end { initialize };

procedure picture;
var i:cardinal;
begin
  enter.class(1, 'm', 'blue');
  enter.class(2, 'o', 'blue');
clear.screen;
write.block(28, 10, 32, 14, 'magenta');
write.block(60, 18, 70, 23, 'magenta');
set.foreground('yellow');
goxy(4, 11); write('Hospital only');
goxy(4, 18); write('Operation');
goxy(32, 8); write('Bed in use');
goxy(60, 15); write('Operating');
goxy(60, 16); write('Theatre');
reset.colours;
end { picture };

procedure report;
begin
end { report };

begin
initialize;
picture;
cause(1, new.entity(1, 1, 0));
cause(2, new.entity(2, 1, 0));
old.tim:=0;
run(24*30+12, 4);
report;
reset.colours;
end.
AUTHOR'S BIOGRAPHIES

ROBERT M. O'KEEFE is a visiting assistant professor in the Department of Computer Science at Virginia Tech, on leave from the Board of Studies in Management Science at the University of Kent at Canterbury, England. He received a B.Sc. in Computer Studies and Operational Research from the University of Lancaster in 1979, and a Ph.D. in Operational Research from the University of Southampton in 1984. Major research interests include Artificial Intelligence and simulation, Visual Interactive Simulation, and the application of expert systems. He is a member of SCS, TIMS, ORS, AAAI and BCS, and a Director of Decision Computing Limited.

Robert M. O'Keefe
Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061, U.S.A.
(703) 961-6075

Permanent address: Rutherford College
University of Kent at Canterbury
Canterbury, Kent CT2 7NX, England.

RUTH M. DAVIES has been working on the application of statistics, Operational Research and computing to problems in Health Care for a number of years. A continuing major research interest is the provision of care to patients with end-stage renal failure. She received a B.Sc. in Mathematics from the University of Warwick, and a Ph.D. in Operational Research from the University of Southampton in 1984. Presently a lecturer in Operational Research in the Department of Mathematical Sciences and Computing at the South Bank Polytechnic, London, England, she has also held research positions at the Universities of Reading and Southampton.

Ruth M. Davies
Department of Mathematical Sciences and Computing
South Bank Polytechnic
Borough Road