Proceedings of the 1986 Winter Simulation Conference
J. Wilson, J. Henriksen, S. Roberts (eds.)

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
London, SE1. OAA, England.

ABSTRACT

Pascal_SIM is a collection of Pascal constants, types, variables, functions and procedures for developing event, activity, three-phase or process orientated discrete-event simulation models. Facilities are provided for queue processing, time advance and event list maintainence, control of entities and resources, random number generation and streams, sampling 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 PR1ME 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 ineffiecient.

To facilitate simulation on micrcomputers, 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 (Uyenso 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 (*duration*) are both arguments. The user must code the events and the procedures *initialize* and *report*; for a visual display *display* and *picture* must also be coded. *initialize* and *picture* are called once before *run*; they initialize the simulation and the static picture respectively. *Report* should be called after *run*, and should contain any end of run reporting, for example, final statistics prints. *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 independant 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 *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
        avail:boolean;
        class:class_num;
        col:colour;
        attr,next_B:cardinal;
        time:real;
        end;
```

where the fields of the record represent :-

avail: 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 that 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, *avail* is false, and *next_B* and *time* will be set to appropriate values. Thus there are no explicit event notices in Pascal_SIM because an entity contains all relevant event information. Entities are generated with the function *new_entity*, and can be disposed of with the procedure *dis_entity*.

```
{ 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

Access to entities is acheived through the global variable *current*, which always points to the entity that has caused the present event, or else by searching queues of entities.

The attribute number *attr* uniquely indentifies 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. resoruces are collected into a *bin* - in effect a *bin* is identical to the STORAGE of GPSS; a *bin* with only one resource is identical to a 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 :-

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, ie. 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 patient 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.

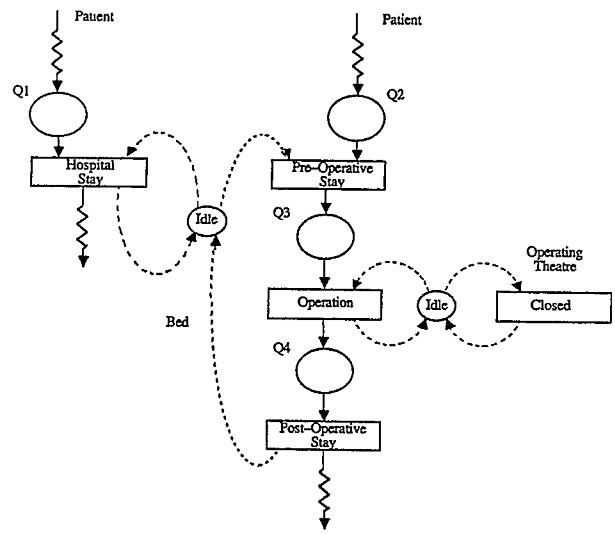

Figure 2: An activity diagram for the hospital simulation.

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 initalized (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

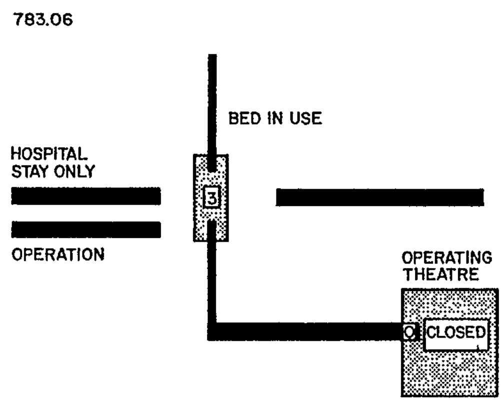

Figure 3: An example visual display. The display shows a patient, represented by the letter *o*, queueing for the operating theatre to open. Three beds are in use; the present clock time is 783.06 time units.

The procedures *move_h* (move horizontal) and *move_v* (move vertical) are the main methods for moving entities across the display. The letter representing the entity is moved on the specified background colour either horizontally or vertically. Note that when ready for an operation, in procedure *end_pre_operative_stay*, operation patients have their colour set to yellow, and will thus appear yellow on the screen from that point on.

5. A PROCESS DESCRIPTION VERSION

Process orientated simulations in Pascal_SIM are written by splitting a procedure into a number of different blocks using a case statement. Entities advance from block to block; a procedure can be 'reactivated' by being called and a block (ie. a branch of the case statement) being attempted. A procedure called *branch* is used to immediately cause an entity to attempt a new block. The first parameter of a *cause* statement now specifies a block number rather than an event, and an entities class number identifies its process. The need for the programmer to split a procedure up with a case statement, to explicitly specify when a process must be suspended (this is done here using a repeat until loop with a flag called *finished*), and to explicitly test if entry to a block is possible, makes the programming fairly ungainly and error prone compared to true process interaction languages. It does, however, allow for process description modelling in Pascal.

In the process description executive, which is modelled on the GPSS executive, an entity is either entered in the calendar or on a chain of suspended entities. Queue priority is implicit, dependent upon an entity's position in the suspended chain. 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 PR1ME 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 unuseable. 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 acheived 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

$$Z_{i+1} := (Z_i * 16807) mod 2147483647$$

and

$$Z_{i+1} := (Z_i * 3993 + 1) mod 32767.$$

The 16-bit version is an implementation of a generator suggested and tested by Thesen, Sun and Wang (1984), which assumes that detection of integer overflow has been disabled. (Incidently, 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 resonable 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 Hurrion, 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 guarenteed ! 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

Bell, P.C. (1985). Visual Interactive Modelling in Operational Research: successes and opportunities. *J. Opl. Res. Soc.* **36**, 975-982.

Bryant, R.M. (1980). SIMPAS: a simulation language based on Pascal. In: *Proceedings of the 1980 Winter Simulation Conference* (Oren, Shub and Roth, eds.), The Society for Computer Simulation.

Christy, D.P. and Watson, H.J. (1983). The application of simulation: a survey of industry practice. *Interfaces* **13**, 47-52.

Fiddy, E., Bright, J.G. and Hurrion, R.D. (1981). SEE-WHY: interactive simulation on the screen. In: *Proceedings of the Institute of Mechanical Engineers* **c293/81**, 167-172.

Nance, R.E. (1984). Model development revisited. In: *Proceedings of the 1984 Winter Simulation Conference* (Sheppard, Pooch and Pegden, eds.), The Society for Computer Simulation.

O'Keefe, R.M. (1983). A system for discrete event simulation in UCSD Pascal. Working paper no. 93, Faculty of Mathematical Studies, University of Southampton, England.

O'Keefe, R.M. (1986). Expert systems and simulation - a taxonomy and some examples. *Simulation* **46**, 10-16.

O'Keefe, R.M. and Davies, R.M. (1986a). A microcomputer system for simulation modelling. *Eur. J. Op. Res.* **24**, 23-29.

O'Keefe, R.M. and Davies, R.M. (1986b). Discrete event simulation with Pascal. Forthcoming in *Journal of Pascal, Ada and Modula-2.*

Thensen, A. Sun, Z. and Wang, T. (1984). Some effecient random number generators for micro-computers. In: *Proceedings of the 1984 Winter Simulation Conference* (Sheppard, Pooch and Pegden, eds.), The Society for Computer Simulation.

Uyenso, D. and Vaessen, W. (1980). PASSIM: a discrete-event simulation package for Pascal. *Simulation* **35**, 183-190.

APPENDIX A: Pascal_SIM FACILITIES

```
const max_cell_num=16;
      max_stream_num=32;
      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=(nul,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,sosq,min,max:real;
              end;

      lookup_table=array [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 q:queue);
procedure give(q:queue;t:link;i:entity);
function  take(q:queue;t:link):entity;
procedure give_top(q:queue;i:entity);
procedure give_tail(q:queue;i:entity);
function  take_top(q:queue):entity;
function  take_tail(q:queue):entity;
function  empty(q:queue):boolean;

{ entities and classes }
function  new_entity(c:class_num;a:cardinal):entity;
procedure dis_entity(e:entity);
procedure make_class(var c:queue;n,size:cardinal);
function  count(var q:queue):cardinal;

{ timing and the executive }
procedure make_sim;
procedure cause(nb:cardinal;e:entity;t:real);
procedure calendar_top;

{ facilities for process executive }
procedure branch(next:cardinal);
procedure remove_entity;

{ resources }
procedure make_bin(var from:bin;n:cardinal);
procedure acquire(var from:bin;n:cardinal);
procedure return(var from:bin;n:cardinal);

{ error messages }
procedure sim_error(s:a_string);

{ random number generator and streams }
procedure make_streams;
procedure rnd(s:stream_num):real;

{ sampling distributions }
function normal(m,sd:real;s:stream_num):real;
function log_normal(m,sd:real;s:stream_num):real;
function poisson(m:real;s:stream_num):cardinal;
function negexp(m:real;s:stream_num):real;
function uniform(l,h:real;s:stream_num):real;
procedure make_sample(var sample_file:text;
                              var table:lookup_table);
function sample(table:lookup_table;s: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;len: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:class_num;l:char;c:colour);
procedure write_entity(x,y:cardinal;e:entity);
procedure write_queue(x,y:cardinal;
                b:colour;q:queue;max_length:cardinal);
procedure write_block(x1,y1,x2,y2:cardinal;b:colour);
procedure move_v(x,y1,y2:cardinal;e:entity;b:colour);
procedure move_h(y,x1,x2:cardinal;e:entity;b:colour);
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,current);
  move_h(12,2,10,current,white);
  write_queue(22,12,white,q1,20);
  cause(1,new_entity(1,1),uniform(60,140,1));
  end;

procedure patient2_arrives; { operation } { B2 }
  begin
  give_tail(q2,current);
  move_h(14,2,10,current,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,current,white);
  dis_entity(current);
  end;

procedure end_pre_operative_stay; { B4 }
  begin
  current^.col:=yellow;
  give_tail(q3,current);
  move_v(30,14,20,current,white);
  move_h(20,30,50,current,white);
  write_queue(60,20,white,q3,30);
  end;

procedure end_operation; { B5 }
  begin
  theatre_available:=true;
  gotoxy(63,21);write('       ');
  move_v(30,4,10,current,white);
  give_tail(q4,current);
  end;

procedure end_post_operative_stay; { B6 }
  begin
  return(bed,1);
  move_h(12,40,70,current,white);
  dis_entity(current);
  end;

procedure open_theatre; { B7 }
  begin
  theatre_open:=true;
  gotoxy(63,20);write('OPEN  ');
  cause(8,current,8);
  end;

procedure close_theatre; { B8 }
  begin
  theatre_open:=false;
  gotoxy(63,20);write('CLOSED');
  cause(7,current,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(5,15,4));
    write_queue(22,14,white,q2,20);
    end;
  end;

procedure start_operation; { C3 }
  begin
  while theatre_open and theatre_available
```

```
    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;
  gotoxy(1,1);writeln(tim:7:2);
  gotoxy(1,1);
  end { display };

procedure run(duration:real;max_C:cardinal);
  var c: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
                calendar_top;
                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
                case c of
                  1:start_hospital_stay;
                  2:start_pre_operative_stay;
                  3:start_operation;
                  4:start_post_operative_stay;
                end;
              end;
         end
  until not running;
  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(8,theatre,8);
  end { initialize };

procedure picture;
  var i:cardinal;
  begin
  enter_class(1,'s',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);
  gotoxy(4,11);write('Hospital stay only');
  gotoxy(4,15);write('Operation');
  gotoxy(32,8);write('Beds in use');
  gotoxy(60,15);write('Operating');
  gotoxy(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
London, SE1 OAA, England.