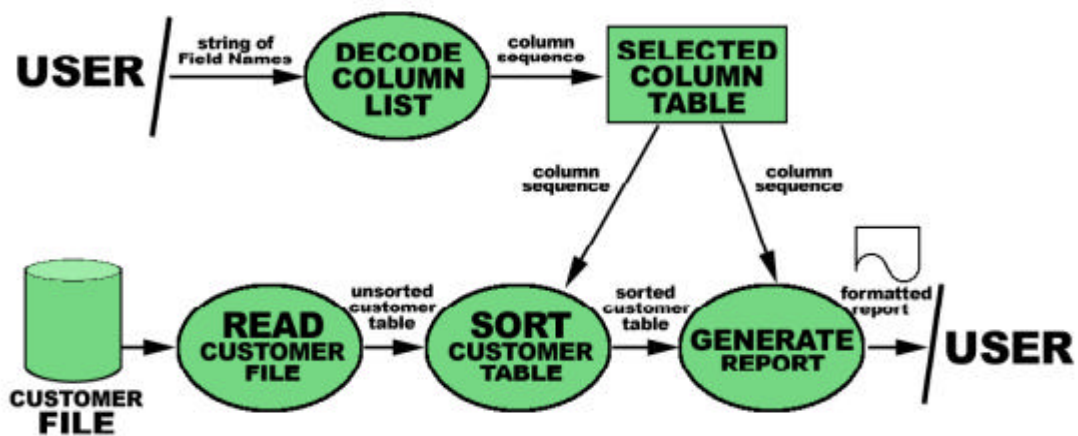


# E-GENTING PROGRAMMING COMPETITION, 2004

## WORKSHOP HANDOUT, WEEK 3, VERSION 1

### 1 BACKGROUND

#### 1.1 Informal Dataflow Diagram



#### 1.2 Useful Definition of a Process

A process is a computational entity that:

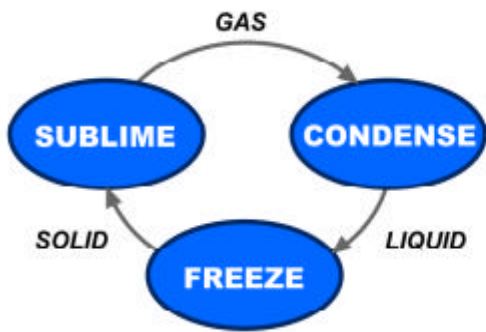
- receives input data;
- processes data;
- emits output data; and/or
- stores state information.

#### 1.3 Flowcharts

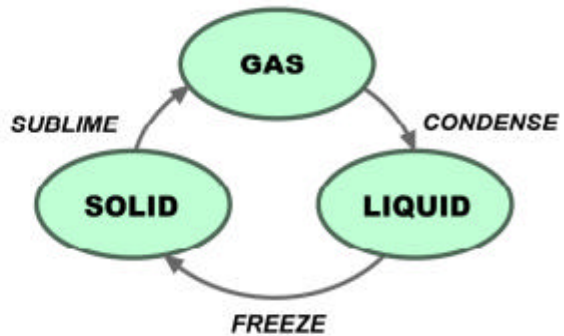
A flowchart is a type of dataflow diagram in which the only datum that passes between the processes is the control token.

It is of no help in system design.

### 1.4 State Transition Diagrams

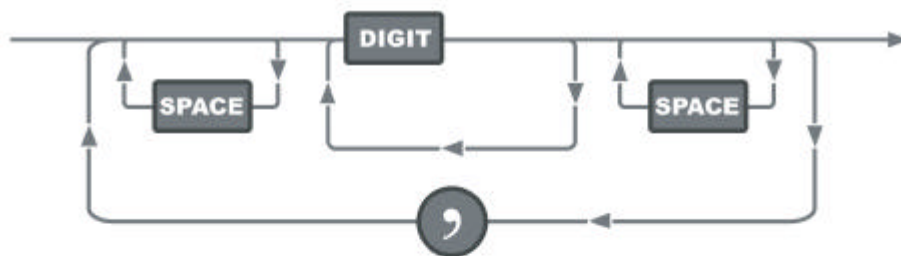


**DATA FLOW**

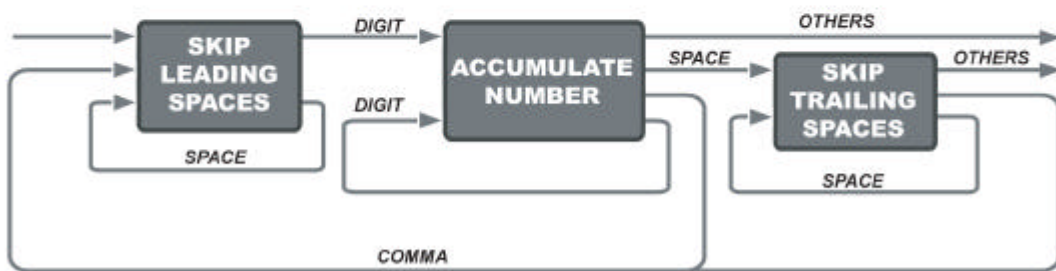


**STATE TRANSITION**

### 1.5 Syntax Diagrams

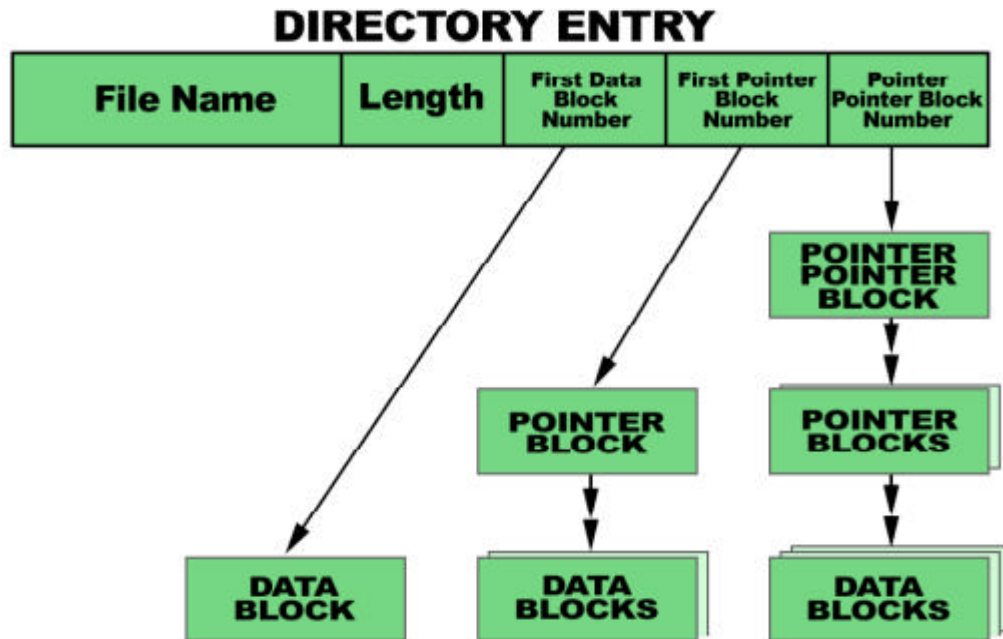


**SYNTAX DIAGRAM**



**STATE TRANSITION DIAGRAM**

## 1.6 An Informal Entity Relationship Diagram



## 2 EXERCISES

### 2.1 Timetrapp

Draw an entity relationship diagram of the database described in the Timetrapp problem and draw a dataflow diagram that helps divide the Timetrapp problem into manageable sub-processes.

This is the Timetrapp problem:

A company called Timetrapp operates a number of arcades that offer video games to teenagers and young adults.

Timetrapp purchases video games from suppliers. It would like to use a more scientific approach for buying new machines.

Timetrapp maintains a database of the turnover on the various types of machines already installed in its arcades. This is the schema of the database:

```

// Sites Table

create table sites (
    siteId      integer not null,      // Site identifier
    siteName    char(20) not null     // Site name
);

// Machine Types Table

create table types (
    typeId      integer not null,      // Machine type id
    typeName    char(20) not null     // Machine name
);

// Machine Table

create table machines (
    machId      integer not null,      // Machine id
    machTypeId  integer not null      // Machine type id
);

// Machine Activity Table

create table activity (
    actSiteId   integer not null,      // Site identifier
    actMachId   integer not null,      // Machine id
    actDate     integer not null,      // Date
    actTurnover integer not null      // Turnover (cents)
);

```

To assist Timetrap to decide whether or not to purchase more machines of a particular type, Timetrap would like a report, that can be requested for a given date range, with the following contents:

1. for each machine type in descending order of the performance rating of the machine type:
  - a. machine type identifier,
  - b. machine type name,
  - c. number of machines of the type,
  - d. turnover of the machines of the type,
  - e. performance ranking of the machine type;
2. total number of machines;
3. total turnover.

Dates are stored on the database in YYYYMMDD format, but the date range submitted to the reporting program must be submitted in DDMMYY format. Two-digit years from 70 to 99 are in the twentieth century. Other years are in the twenty-first century.

A machine type's performance ranking is calculated in the following way.

1. For each site, calculate the total number of machines and turnover at the site at which the machine is installed ( $N_{\text{site}}$  and  $T_{\text{site}}$  respectively).
2. For each machine-type calculate the number and turnover of the machines of that type at each site ( $N_{\text{type,site}}$  and  $T_{\text{type,site}}$  respectively).
3. The average turnover of a machine at a site ( $A_{\text{site}}$ ) is given by:

$$A_{\text{site}} = T_{\text{site}} / N_{\text{site}}$$

4. The unadjusted performance ranking of a machine type at a site ( $R_{\text{type,site}}$ ) is the average turnover of the machine divided by the average turnover of the site. I.e.:

$$R_{\text{type,site}} = (T_{\text{type,site}} / N_{\text{type,site}}) / A_{\text{site}}$$

5. The performance ranking of a machine type ( $R_{\text{type}}$ ) to be displayed in the report is:

$$R_{\text{type}} = \text{SiteSum} (R_{\text{type,site}} * N_{\text{type,site}} * A_{\text{site}}) / \text{SiteSum} (N_{\text{type,site}} * A_{\text{site}})$$

Where the SiteSum function totals the value of its operand over all sites.

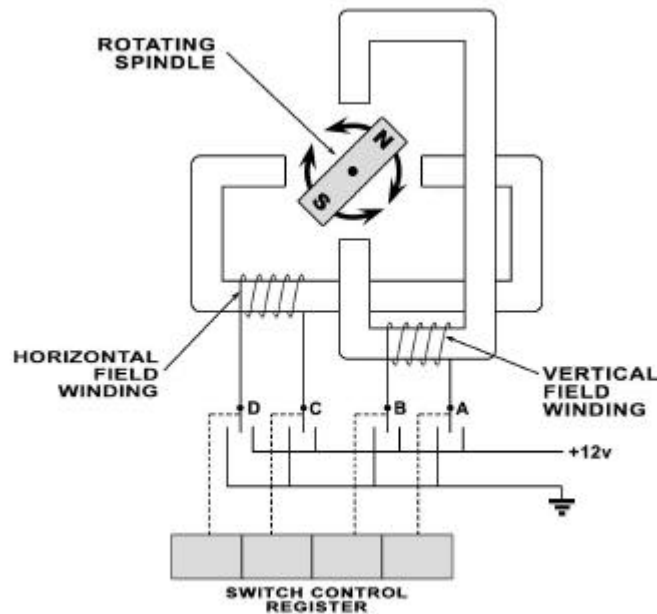
## **2.2 Windscreen Wipers**

Draw a dataflow diagram that helps solve the concurrency problem in the Windscreen Wiper problem in the 2003 competition.

This is the Windscreen Wiper problem:

In an effort to reduce the number of mechanical parts in a windscreen wiper mechanism, an automobile manufacturer intends to replace its current DC motor mechanism with a brush-less stepping motor mechanism.

Conceptually, the stepping motor consists of a permanent magnet mounted on a spindle with two field windings that can be energised in sequence to drive the spindle in one direction or the other. The diagram below represents the mechanism of the stepping motor:



When switch A is switched to +12V and switch B is switched to ground (i.e. 0V), the conventional current will flow from the common of switch A to the common of switch B. This will cause the core of the A-B winding to be magnetised so that its top is a north pole and its bottom is a south pole. This, in turn will cause the spindle to rotate clockwise until the north pole of the spindle is at the bottom. By operating switches A to D in the correct order, the spindle can be made to rotate in either direction.

Switches A to D are controlled via a 4-bit switch control register (SCR) that can be set by software. The least significant bit of the SCR operates switch A and the most significant bit operates switch D. When a SCR bit is '0', the common of the switch is connected to ground; when the bit is '1', the common of the switch is connected to +12V.

The SCR can be set from C or C++ by calling the following function:

```
void SetSCR (int scrVal)
```

The parameter `scrVal` is the value to be loaded into the SCR. Only the four least significant bits of `scrVal` are loaded into the SCR. The higher order bits are ignored.

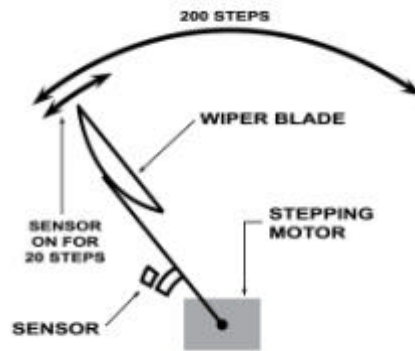
To turn the spindle, the windings must be energised in either the ascending or descending sequence of the rows in the following table. If the windings are energised in the descending sequence, the motor will rotate clockwise. If the windings are energised in the ascending sequence, the motor will rotate anticlockwise.

For clockwise rotation	SCR VALUE				For anti-clockwise rotation
	D	C	B	A	
↓	0	0	0	1	↑
	0	1	0	0	
	0	0	1	0	

	1	0	0	0	
--	---	---	---	---	--

Each row in the above table is referred to as a 'step'. In the windscreen wiper mechanism, the stepping motor is geared down so that there are 200 steps from the left-most position of the wiper blade to the right-most position of the wiper blade. The gearing mechanism does not reverse the direction of the stepping motor. If the stepping motor is rotating clockwise, then the blades will move from left to right and vice versa.

There is a sensor that detects the position of the wiper blades when they are close to the left-most limit of their travel. The following diagram shows how the sensor works.



As the wiper blade swings from the right to the left, the sensor will switch from its off state to its on state. The wiper blade must stop and reverse its direction 20 steps after this transition.

The state of the sensor can be sensed from C or C++ by calling the following function:

```
int GetSensorState()
return value
    is zero if the sensor is off, or non-zero if the sensor is on.
```

The normal operating speed of the motor is 200 steps per second, but if it is switched from stopped directly to 200 steps per second it will almost certainly skip a few steps and the accelerations involved, in time, will damage the reduction gear and wiper blade mechanism.

The maximum acceptable acceleration is 1,250 steps per second squared in either direction. This acceleration is achieved, within the limits required by the mechanism, by using the sequence of inter-step times listed in the table below when starting, stopping and reversing the direction of the wiper blade.

Inter-Step Time (milliseconds)															
42	17	13	11	10	9	8	8	7	7	6	6	6	6	6	5

Your task is to write a computer program to control the windscreen wiper stepping motor.

If you write in C or C++, your program must contain the following functions that will be called by the operating system of the microprocessor.

```
void Initialise ();
void StartWiping ();
void StopWiping ();
void Tick ();
```

The `Initialise` function must first switch off all the field windings by setting the SCR to zero. It must then initialise any internal data structures and then test the state of the sensor. If the sensor is off, it must initiate a process to return the wiper blades to the left-most starting position at the maximum non-progressive speed of 42ms per step. When the blades are in the left-most starting position, the control program must switch off the field windings by setting the SCR to zero. If the sensor is on when `Initialise` is called, the wiper blades must stay in their initial position.

The `StartWiping` function must start the wiper blades moving back and forth. The `StartWiping` function may be called at any time after `Initialise` is called. If the `StartWiping` function is called while the blades are being returned to the left most starting position at 42ms per step, the return process must be completed before the normal back and forth cycle is initiated. Once the wiper blades are in the left most starting position, the program must accelerate the blades to their maximum speed in the left-to-right direction in accordance with the acceleration table. The program must then decelerate the blades so that they stop 180 steps after the sensor switches off. It must then reverse the direction of the motor, accelerate the blades, adjust the theoretical location of the blades when the sensor switches on, decelerate and reverse the direction of the motor and so on until the `StopWiping` function is called.

The `StopWiping` function may be called at any time after `Initialise` is called, even when the blades are already stationary. In all cases, the wiper blades must be allowed to return to the left-most starting position in the normal course of the cycle that was in progress when `StopWiping` was called. The motor windings must then be switched off by setting the SCR to zero.

The `Tick` function is called by the operating system once every millisecond.

The control program must not lock control in any of the application functions. If control is locked in a loop in the `StartWiping` function, the `StopWiping` function will never be called.

### **2.3 M44**

Draw an informal entity relationship diagram of the bucket data structure described in the M44 problem in the 2003 competition.

This is the M44 problem:

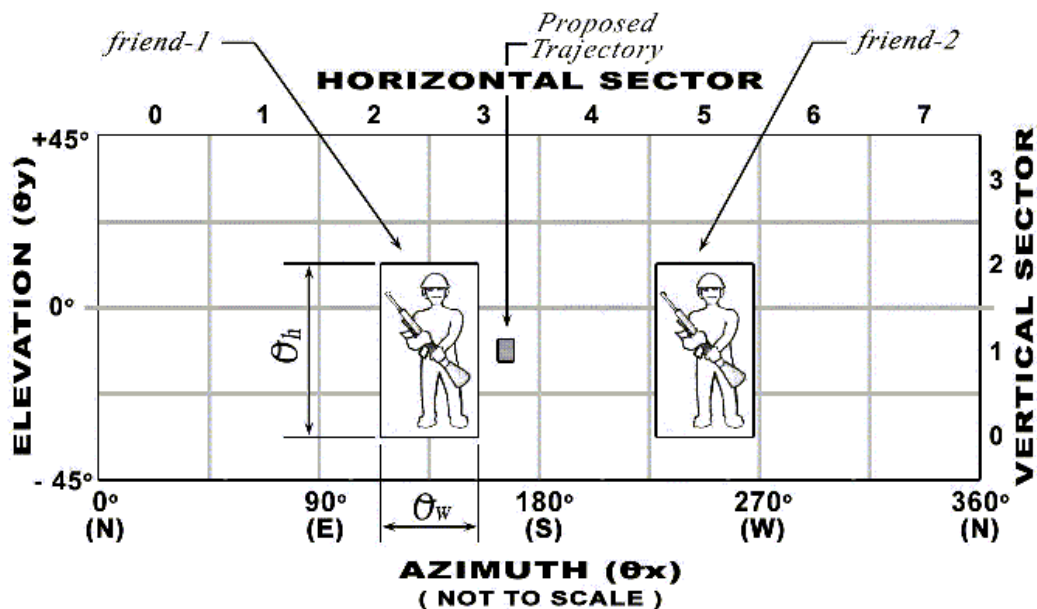
In a recent conflict, the soldiers of Nanopia, a technologically advanced first-world democracy, turned out to be much better at killing each other than the enemy was at killing them. As a consequence of that unfortunate statistic, the Congress of Nanopia passed a bill that restricted its soldiers to a new type of rifle known as the M44. The M44

incorporates a transponder that emits a signal that is received by all the other M44s within a 25km radius. When the trigger of the M44 is pulled, the weapon senses its orientation and checks whether the bullet is likely to hit any of the holders of the other M44s in the vicinity. The firing mechanism is not released until this check is satisfactorily completed.

The soldiers of Nanopia are now engaged in an urban peacekeeping operation. Nanopia has between 60,000 and 100,000 soldiers spread among the population of an urban centre approximately 50km in diameter. This has resulted in a large number of potential 'friends' being stored in each M44. The M44s are not firing a round until 1 to 3 seconds after the trigger is pulled. This is because of the large number of potential friends that the M44s must check before releasing the firing mechanism. The urban guerrillas that the Nanopian army was supposed to bring under control are taking advantage of the delay and the Nanopian casualties are mounting.

In a near-Newtonian experience with a high-velocity sniper round, a member of the Nanopian Engineers came up with the idea that it might be possible to divide the M44's field of fire into sectors. The location of each friend could then be recorded in a separate table for each sector. If this were done it would only be necessary to search the sectors that the proposed trajectory intersected in order to determine if the trajectory was likely to hit a friend. This might reduce the search time from 1 or 3 seconds down to an acceptable period of 10 or 20 milliseconds.

Consider the field of fire of the M44 in polar co-ordinates represented by un-projected azimuth and elevation:



Azimuth is the direction the gun is pointing in, in degrees east of North. Elevation is the angle between the barrel of the gun and the horizon. If the gun is horizontal, its elevation is  $0^\circ$ ; if it is pointing up in the air at an angle of  $30^\circ$  to the horizontal, its elevation is  $+30^\circ$ .

The existing M44 program represents each friend by a rectangle in the azimuth-elevation co-ordinate system. The trajectory of the bullet is also represented by a rectangle. The rectangle of a friend or trajectory is referred to as the 'profile' of the friend or trajectory.

The Nanopian Engineer proposes that a hash-type algorithm be used to dramatically reduce the search time needed to determine whether or not any particular trajectory is likely to hit a friend. He suggests that a hash-bucket be created for each combination of horizontal and vertical sectors.

For the purposes of describing the algorithm, each sector is identified by an expression of the form '(x,y)', where 'x' is the horizontal sector number and 'y' is the vertical sector number. Each hash-bucket is represented by the term  $H(x,y)$ , where '(x,y)' is the sector associated with the hash-bucket.

The engineer proposes that a reference to each friend be put in the hash bucket of each sector that the profile of the friend intersects. For example, in the above diagram, there would be a reference to friend-1 in hash buckets,  $H(2,0)$ ,  $H(2,1)$ ,  $H(2,2)$ ,  $H(3,0)$ ,  $H(3,1)$  and  $H(3,2)$  and a reference for friend-2 in hash buckets  $H(5,0)$ ,  $H(5,1)$  and  $H(5,2)$ .

It is possible by reasonably obvious mathematical means to determine the sectors that a proposed trajectory intersects. In the example, the trajectory only intersects sector (3,1), though it is certainly feasible for a trajectory to intersect more than one sector. To determine whether or not a proposed trajectory is likely to hit a friend, it is only necessary to check the friends with references in the hash-buckets corresponding to the sectors that the trajectory intersects. In the example, it is only necessary to check the friends with references in hash-bucket  $H(3,1)$ .

Just because a trajectory and a friend both intersect sector (3,1), it is not necessarily the case that the trajectory will hit the friend. It is still necessary to check whether the profiles of the friends referenced in the hash-bucket intersect the profile of the trajectory.

Your task is to create a prototype of the engineer's algorithm to test its performance. If the prototype works satisfactorily, your company will win a multi-million dollar contract to provide field upgrades for the M44s and your stock options will transmogrify into a retirement annuity. You will also save the lives of hundreds of soldiers.

The prototype must construct a rectangular theatre of operations 50km by 50km. The subject M44 will be located in the centre of the rectangle. 100,000 friends must then be pseudo-randomly positioned in a uniform distribution over the rectangular theatre, but no closer to the subject than 5m and no further from the subject than 25km. The elevation of each friend is to be pseudo-randomly generated in a uniform distribution, with the centre of each friend having an elevation of between  $-30^\circ$  and  $+30^\circ$  with respect to the subject. The dimensions of the profile of each friend can then be calculated by assuming that each friend is 1.8m high and 0.5m wide.

The prototype must transform the dimensions of each friend into polar co-ordinates and load the hash buckets. Care must be taken in dealing with friends to the north of the subject because such friends may appear in both high-numbered and low-numbered sectors.

The prototype must then record a starting time and generate 1,000 pseudo-randomly selected firing trajectories. The trajectories may have any azimuth between  $0^\circ$  and  $360^\circ$ ,

but the elevations are to be restricted to  $-40^\circ$  to  $+40^\circ$ . The profile of each trajectory may be taken to be  $0.2^\circ$  wide by  $1^\circ$  high. The prototype must determine whether or not each trajectory intersects a friend and store the profile of the trajectory and the identities of the friends, if any, that it intersects. After performing the 1,000 bucket-based intersection determinations, the prototype must record the ending time and then list the profile of each trajectory and those of the friends, if any, that it intersects. The listing must contain the polar co-ordinates (i.e.  $\theta_x$  and  $\theta_y$ ) of the centre of each profile and its width ( $\theta_w$ ) and height ( $\theta_h$ ) in degrees. It must then display the average computation time per trajectory.

The prototype must use symbolic constants to define the horizontal and vertical sector densities. If the sector size is too small, the profile of the trajectory will intersect a large number of sectors and thereby increase the search time. If the sector size is too big each sector will contain a large number of friends and the efficiency of the algorithm will, once again, be compromised. It will probably be necessary to adjust the horizontal and vertical sector densities to determine optimum values.

The Nanopian Engineer presented the following pseudo-code of the prototype to the Joint Chiefs to convince them to provide funding for the project. You may use it as a base for developing the prototype.

1. Initialise the bucket data structures.
2. Repeat the following until the profiles of 100,000 friends have been loaded into the bucket data structures:
  - a. Generate a pseudo-random easterly component (e) of the distance from the subject to the friend in the range  $-25$  to  $+25$ km.
  - b. Generate a pseudo-random northerly component (n) of the of the distance from the subject to the friend in the range  $-25$  to  $+25$ km.
  - c. Calculate the horizontal distance to the friend (r) using the formula  $r = \text{sqrt}(e^2 + n^2)$ .
  - d. If the horizontal distance to the friend is outside the range 5m to 25km, discard the location and generate another pair of co-ordinates.
  - e. Calculate the azimuth of the centre of the friend ( $\theta_x$ ), in principle by the use of the formula  $\theta_x = \text{atan}(e / n)$ , but adjusting for the correct quadrant when n is close to zero and/or when one or both of the location components are negative and, if necessary, converting radians into degrees.
  - f. Generate the pseudo-random elevation of the friend ( $\theta_y$ ) in the range  $-30^\circ$  to  $+30^\circ$ .
  - g. Calculate the angular width ( $\theta_w$ ) and height ( $\theta_h$ ) of the friend using the following formulas:
 
$$\theta_w = (W * 180) / (r * p)$$

$$\theta_h = (H * 180) / (r * p)$$
 Where W is the physical width of the friend (assumed to be 0.5m) and H is the physical height of the friend (assumed to be 1.8m).
  - h. Load the profile of the friend represented by  $\theta_x$ ,  $\theta_y$ ,  $\theta_h$  and  $\theta_w$  into the bucket data structures.
3. Record the starting time ( $t_s$ ).
4. Repeat 1,000 times:

- a. Generate a pseudo-random firing azimuth in the range  $0^\circ$  to  $360^\circ$ .
- b. Generate a pseudo-random firing elevation in the range  $-40^\circ$  to  $+40^\circ$ .
- c. Determine the profile of the trajectory given that it is  $0.2^\circ$  wide by  $1^\circ$  high.
- d. Store the profile of the trajectory.
- e. Use the bucket data structure to determine which, if any, of the friends the proposed firing direction is likely to hit.
- f. For each friend the proposed trajectory is likely to hit:
  - i. Store the identity of the friend.
5. Record the ending time ( $t_e$ ).
6. For each generated trajectory:
  - a. Display the profile of the trajectory.
  - b. For each friend the trajectory is likely to hit:
    - i. Display the profile of the friend.
7. Calculate and display the average time per trajectory  $(t_e - t_s)/1000$ .

Although the profile of the friend is generated in terms of  $?x$ ,  $?y$ ,  $?h$  and  $?w$ , these may not be the most convenient values to use for the purposes of determining profile intersections. The prototype is not required to store  $?x$ ,  $?y$ ,  $?h$  and  $?w$ . It may store the profile in a form more suited to determining profile intersections.

## 2.4 *Braindead*

Draw a dataflow diagram of a solution to the Braindead problem.

The Braindead problem is:

A research company called Gammaswitch has invented a new ultra high-speed computing device that operates by switching high-energy photons instead of electrical charge. The Gammaswitch computer is over a thousand times faster than the fastest semiconductor devices.

An entrepreneurial young hacker by the name of Bob Yates has created the only commercially viable operating system for the Gammaswitch computer. It is an architecturally challenged, barely functional monster called Braindead.

An Internet information service called Regurgital operates a worldwide free-to-wire information service. Regurgital's service has become so popular that its message server has become overloaded and is causing severe delays and inconvenience to Regurgital's customers.

Regurgital now wants to replace the overloaded computer with one of Gammaswitch's new ultra-high speed machines. Your job is to program the Gammaswitch computer in C or C++, but in any case, under the frightful Braindead operating system, to copy the information messages from disk files on the Gammaswitch computer and deliver them to Braindead's HTTP driver.

Your program must be able to handle up to 20 concurrent HTTP downloads.

If you write in C or C++, Braindead provides the following HTTP server support functions.

GetRequest	initiates reception of an HTTP request.
PutData	initiates transmission of a block of output text.
RequestDone	tells Braindead that the request has been completely serviced.
Yield	releases control to Braindead.

GetRequest has the following declaration.

```
int GetRequest ();
```

GetRequest returns an integer known as a ‘channel identifier’. The channel identifier is used to relate subsequent network operations to the invitation to receive an HTTP connection created by the GetRequest call. Multiple calls to GetRequest may be made one after another. Each open channel will be assigned a different channel number by GetRequest.

The GetRequest function merely establishes an invitation to receive an HTTP connection. It does not actually create a connection to a client computer. When, in due course, a client computer attempts to connect to the HTTP server, Braindead will call an application function called ‘RequestReady’ to notify the application that a connection has been made. The application must declare a function called RequestReady or it will fail to link. RequestReady must have the following declaration.

```
void RequestReady (int chanId,
                  const char *fileName);
```

The argument ‘chanId’ will contain the channel identifier returned by the call to GetRequest that created the communications channel.

The argument ‘fileName’ is a pointer to the name of the file to be returned to the client computer. The memory addressed by fileName will only contain the file name during the execution of RequestReady. When RequestReady returns, Braindead may overwrite the text of the file name.

The application can use the normal C fopen, fread and fclose functions to read the file from the mass storage attached to the Gammaswitch computer.

To return the file contents to the requesting client, the application must call Braindead’s PutData function. PutData has the following declaration.

```
void PutData (int chanId
              const char *data, int len);
```

The argument ‘chanId’ is the channel identifier returned by the original call to GetRequest.

The argument ‘data’ is a pointer to a data block that contains the data to be written.

The argument ‘len’ is the length of the data block.

The call to PutData merely initiates a data transfer. PutData will usually return before transmission of the data is finished. When transmission of the data is completed, Braindead calls an application function called ‘PutDone’ to notify the application that the transmission process is completed.

The declaration of PutDone is:

```
void PutDone (int chanId);
```

The argument 'chanId' is the channel identifier returned by the original call to GetRequest.

Braindead does not copy the contents of the data block passed to PutData into an internal buffer. Instead it stores a pointer to the data block in the application's memory segment. Consequently, it is important that the application does not modify the data block or release the data block memory by returning from a function in which it is declared as a local variable before Braindead calls the PutDone function.

When the file has been completely sent by repeated calls to PutData, the application must call the RequestDone function to inform Braindead that the request has been completely serviced. RequestDone has the following declaration.

```
void RequestDone (int chanId);
```

The argument 'chanId' is the channel identifier originally returned by GetRequest.

RequestDone closes the communications channel created by GetRequest and releases the channel identifier. To receive another request, the application must make a further call to GetRequest. Once RequestDone has released a channel identifier, it is possible that a subsequent call to GetRequest may return that same channel identifier.

In order to give Braindead the opportunity to perform its own internal processing and call the GetReady and PutDone functions, the application must release control to Braindead by calling the Yield function. The declaration of Yield is:

```
void Yield ();
```

When the application calls Yield, Braindead scans the devices that perform the physical input and output and checks whether an input or output process has been completed. If such a process has finished, Braindead calls either GetReady or PutDone depending on the nature of the original request. If an input or output process is outstanding, Yield will return after Braindead has called either GetReady or PutDone. If no input or output processes are outstanding, Yield will return immediately.