DREGS: A DISTRIBUTED RUNTIME ENVIRONMENT FOR GAME SUPPORT

by

Allan Bricker
Tad Lebeck
Barton P. Miller

Computer Sciences Technical Report #657

August 1986

DREGS: A Distributed Runtime Environment for Game Support

Allan Bricker
Tad Lebeck
Barton P. Miller

Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton Street
Madison, Wisconsin 53706

ABSTRACT

DREGS, a distributed environment for game support, simplifies the task of implementing multi-player games. DREGS solves the problems of concurrency control, synchronization, and communication as they apply to distributed games. DREGS provides support for the update and control of replicated objects and uses a central arbitration scheme to enforce a strict ordering of events. DREGS has been implemented to run under any 4.3BSD Unix compatible system and operates across a network of heterogeneous architectures.

A game description language, GDL, provides a framework for programming multi-player distributed games. GDL is a simple language that generates complex distributed programs. It is an object-based language, where objects are defined in terms of their input events and their corresponding actions. The programmer writes the game as if it were a serial program, without concern for concurrent activities. GDL also frees the programmer from the details of device and network interfaces.

The combination of the DREGS runtime and GDL frees a game designer from the distributed aspects of multi-player games. This freedom allows designers to concentrate their efforts on better, more interesting games. DREGS has been used to implement an N-way talk program, a tank game, and a flying saucer space game.

1. Introduction

Powerful, low-priced personal workstations with bit-mapped displays are becoming an integral part of a modern computing environment. This trend, combined with computer scientists' natural appetency for computer games, results in the need for more sophisticated game technology. The goal of **DREGS** is to provide a distributed runtime environment for game support that will allow multi-player games to be easily implemented for any 4.3BSD Unix system. **DREGS** is not in itself a game; it provides a framework on which to build multi-player games. This paper describes the **DREGS** system and the manner in which it simplifies a game designer's task.

One may question the validity of devoting resources to the development of a game writing system. Game research, however, is important for several reasons. First, games are a primary tool for introducing people to the use of computers. Second, distributed games represent a specific area within distributed computing systems. Finally, games are fun.

Historically, computer games have been limited to single-player games; improvements to

Research supported in part by the National Science Foundation grant MCS-8105904 and a Digital Equipment Corporation External Research Grant.

NUMBER of PLAYERS	DISPLAY TECHNOLOGY		
	Text	ASCII Graphics	High Resolution Graphics
1	Star Trek Adventure	Star Trek Rogue	Star Trek Pit-Fall
2			VS Tennis
3			
4			Gauntlet
⋮			
N	Perquacky	Unix MazeWar	Xerox MazeWar

Figure 1: Game Space

these games generally concentrated on display techniques. Early games depended primarily on textual descriptions. Later games attempted to provide a graphical interface on ASCII terminals, representing various objects with normal characters. More recent games were designed to use high resolution graphic displays, such as those used for video arcade games or those attached to personal workstations, to provide a more realistic image of the objects. *Adventure*, *Rogue*, and the arcade game *Pit Fall*, games in which players seek treasures in a dangerous environment, exemplify the trend towards improving display techniques. Many versions of *Star Trek* have also been produced, culminating in a vector graphics video arcade version of this classic computer game.

A recent direction taken in the development of computer games is to allow multiple players to compete against one another. For example, *Perquacky*®, a word game from the company Lakeside, has been implemented as a multi-player game that uses a textual interface. *MazeWar*, a game in which players hunt each other in a two-dimensional maze, has been implemented for several systems. A Unix implementation uses ASCII characters to display the game board, while implementations for various Xerox operating systems use bit-mapped displays. Multi-player computer games, though generally very popular, are notably scarce because they are more difficult to implement than single-player games.

The *Game Space* concept can be used when classifying computer games. Game Space is defined as a two-dimensional coordinate system with display technology as one axis and the number of players allowed as the other. Figure 1 lists several existing games and shows their positions in Game Space. The list of single-player games is hardly exhaustive; the list of multi-player games, however, is fairly comprehensive.

DREGS provides a framework that frees the multi-player game designer from the details of implementing a distributed system. This freedom is available because **DREGS** provides the necessary communication and synchronization mechanisms, allowing game designers to concentrate their efforts on creating a wide variety of interesting multi-player games. The use of **DREGS** in conjunction with high resolution graphic display techniques allows games to be created that effectively span Game Space.

The remainder of this paper discusses the problems involved with distributed games and our

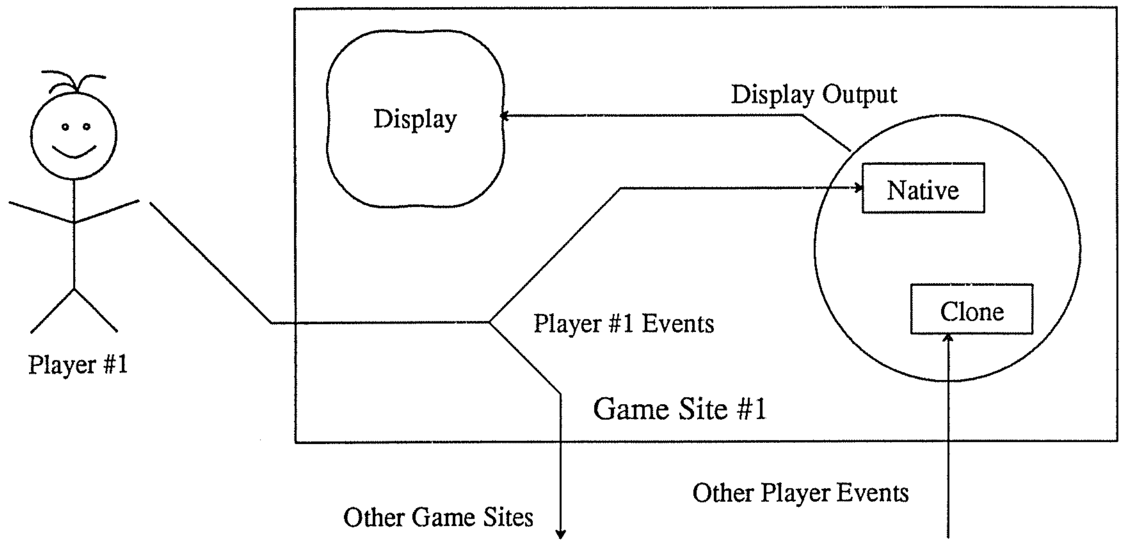

Figure 2: Abstract Distributed Game Model

solutions to these problems. Section 2 briefly presents some problems in distributed programming and indicates how they affect distributed games. Section 3 develops an abstraction for describing games, and uses this abstraction to present the **DREGS** model of a game. The section then discusses how this model solves the problems presented in section 2. Section 4 describes the Game Description Language (GDL), and shows how using the language simplifies the task of the game designer. Section 5 presents some performance considerations and explains certain limitations of **DREGS**. Finally, section 6 presents the current state of **DREGS** and GDL, and discusses our experiences using them.

2. Problems

A common belief is that writing distributed programs is more difficult than writing sequential programs. The difficulties arise because the programmer has to contend with concurrency, synchronization, communication protocols, and fault tolerance (e.g., see [1]): issues not encountered when writing sequential programs. A distributed game has the added complexity of requiring the capability of updating replicated data structures concurrently. Tight timing constraints result because the game must be responsive to player input. **DREGS** addresses the issues of concurrency, synchronization, and communication protocols while attempting to provide real time response. Currently, **DREGS** makes no attempt to provide fault tolerance.

3. The Game Model

We first present an abstract model to describe a distributed game. We then present the **DREGS** model of a game using this abstraction. We go on to discuss how the **DREGS** model solves the problems associated with concurrency, synchronization, communication protocols, concurrent updates, and timing constraints.

3.1. The Abstract Model

A distributed game consists of a collection of game sites, one for each player. Within each game site is an input facility, a display facility, and a set of game objects for each player (see Figure 2). Each object represents a distinct thread of control at the game site. There are two types of objects: those created at the local site and those created at remote sites. Objects created locally are termed *natives* while objects created remotely, and replicated locally, are termed *clones*. All objects, native or clone, accept *events* as input and perform corresponding *actions*. As events are generated by a user, they are delivered to the local native objects and to its remote clones. These events are delivered to each object at the same time, and in the same order, to ensure that the corresponding actions are performed simultaneously at each site.

3.2. The DREGS Model

In the **DREGS** model, each game site includes an *Input Manager*, a *Game Manager*, and an *Output Manager*. In addition to the collection of game sites, a **DREGS** game includes

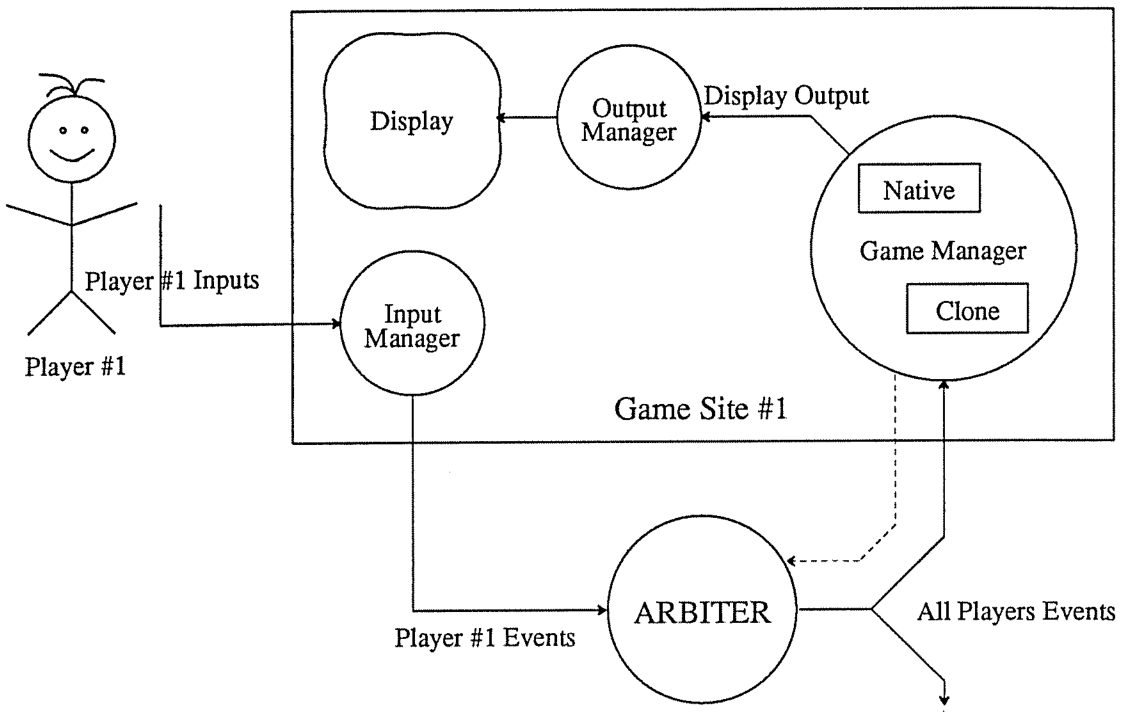

Figure 3: DREGS Model, with Communication Paths

one central *Arbiter* (see Figure 3). The following subsections discuss the functions of the Input, Game, and Output Managers, and the function of the Arbiter. Subsection 3.2.3 describes the communication paths of a **DREGS** game.

3.2.1. Input, Game, and Output Managers

The Input Manager takes input from the player and translates it into an event. This event will eventually be delivered to all Game Managers.

The Game Manager receives all events generated by each Input Manager, and dispatches them to the appropriate objects. The objects then perform the corresponding actions. Next, the native objects are queried about interactions with other objects, clone or native, possibly producing new events. These new events are delivered to all Game Managers, including the Game Manager that produced the new events, as though they had come from the Input Manager. Only the native objects are queried to avoid concurrent updates and to reduce the amount of work to be done at each site. Finally, the display output is sent to the Output Manager.

The Output Manager displays the output produced by the objects. No Output Manager is provided by **DREGS**, but X [2], Andrew [3], and Curses [4] are examples of available Output Managers.

3.2.2. Arbiter

The Arbiter accepts events from all Input Managers and distributes them to all Game Managers. In this way, the Arbiter simplifies the communication machinery of the Input Manager and Game Manager by allowing them to perform their respective input and output of events on a single data stream. The alternative is to have each Input Manager connected to each Game Manager, forcing them to manage a data stream for every active player. The simplified process interconnect structure also drastically reduces the number of messages transmitted.

By quantizing time, the Arbiter guarantees a temporally consistent execution of all actions. The Arbiter buffers all events it receives during a given time-quantum. At the end of each time-quantum the Arbiter sends the events as one message to each Game Manager. Sending a message at the end of each time-quantum also serves to synchronize the Game Managers.

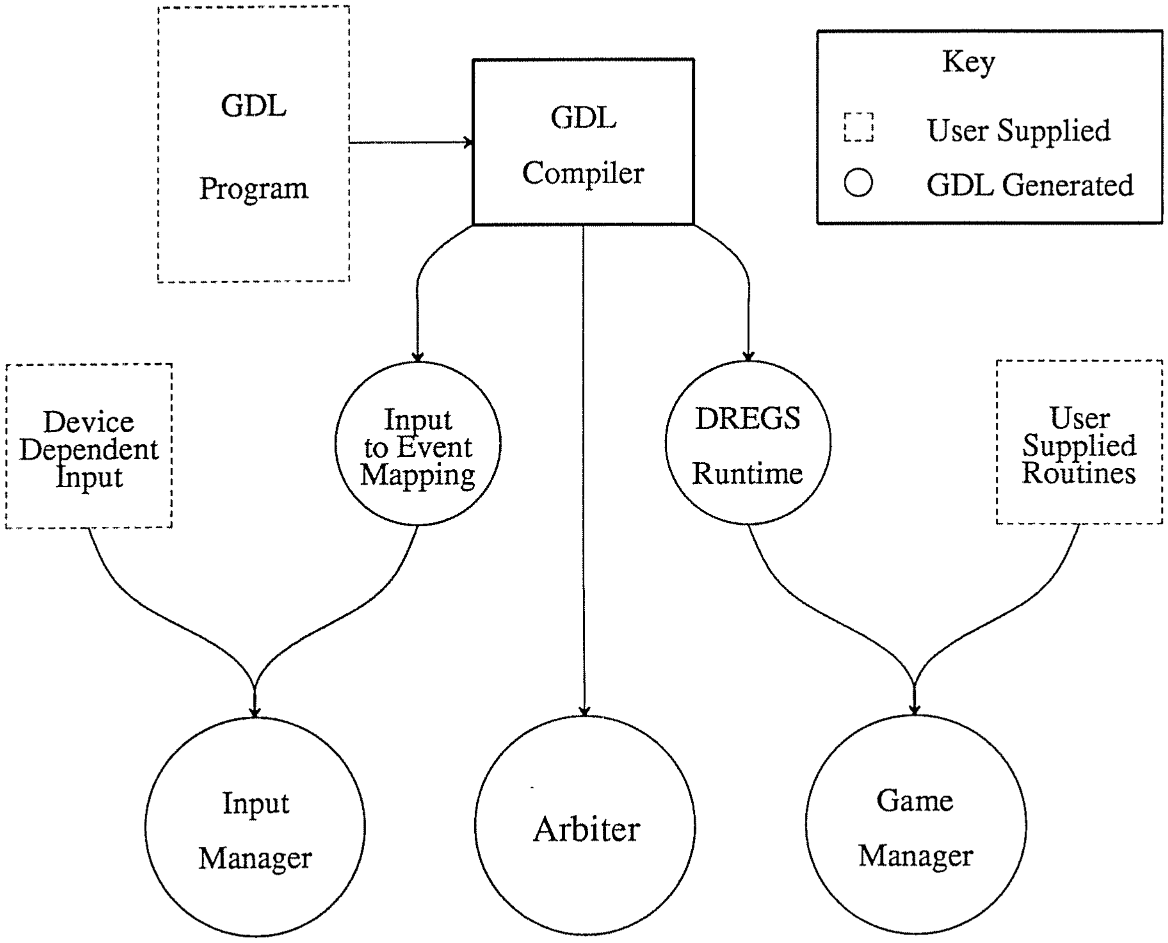

Figure 4: Game Generation From a GDL File

3.2.3. Dregs Communication

The Input Manager has one data path to the Arbiter on which it sends events. The Game Manager has two data paths: one on which it receives events from the Arbiter, and the other on which it sends events to the Arbiter. The Game Manager needs this second path to generate new events as a result of an object's action. Figure 3 shows the communication paths between the Input Manager, Arbiter, Game Manager, and Output Manager. To support heterogeneous architectures, messages sent between the Input Manager, Arbiter, and Game Manager are encoded using *XDR*, Sun Microsystems®, Inc.'s external data representation protocol [5].

4. GDL

We have created *GDL*, a game description language, to facilitate automatic generation of **DREGS** games. A game is specified by defining all objects within the game. An object consists of state information and input events for that object. Event types fall into four categories: generic object events generated by **DREGS**, user-defined events generated by the Input Manager, events generated by a Game Manager calling a replicated procedure, and events which occur due to an interval timer expiring. For each specified input event, the game designer defines the corresponding action routine. Given a game defined in GDL, a GDL compiler will automatically generate the appropriate Input and Game Managers, as well as an Arbiter. Figure 4 shows what a GDL compiler generates based on game designer's GDL program. An example of a GDL program to implement a simple game is given in the Appendix. The following subsections describe the four event types.

4.1. Generic Object Events

There are several generic events defined for all objects. These events deal with object creation, destruction, update, and display. The game designer can optionally specify code for any of the following generic events:

STARTUP	This event will cause code to be executed once when the Game Manager is initially invoked. This code is used to initialize the game-specific state as well as an Output Manager.
INITIALIZE	This event will cause code to be executed after an object is created. For each *object*, a procedure *object*_Create is automatically generated. This procedure creates an instance of the object.
TERMINATE	This event will cause code to be executed before the **DREGS** run-time removes an object after a player quits.
UPDATE	This event will cause code to be executed once per time-quantum even if the object has not received new input events. This code typically will be used to update the positions of movable objects.
CONTACT	This event will cause code to be executed on native objects to allow them to check for intersections with other objects.
DISPLAY	This event will cause code to be executed once per time-quantum to give an object the opportunity to display itself.

4.2. Input Manager Events

The game designer defines Input Manager events by specifying the mapping of user inputs, such as mouse or keyboard input, to object-specific events. The game designer is also responsible for specifying the corresponding code to be executed when an event occurs.

4.3. Replicated Procedure Events

Replicated procedures are used by **DREGS** for performing remote procedure calls at multiple sites. They are not used to ensure reliability as in Circus [6]. When a replicated procedure is called, an event is generated by the Game Manager and sent to the Arbiter in the same way that the Input Manager sends an event to the Arbiter. Thus the event will be distributed to all Game Managers during the following time-quantum and the body of the replicated procedure will be executed at all game sites.

4.4. Interval Timer Events

The game designer may specify an arbitrary number of interval timer events. An interval timer is specified by giving a time-quantum, Q, and action code to execute every Q ticks. The timer is automatically reset to Q after expiring.

5. Performance Considerations and Constraints

One of the performance considerations of **DREGS** is to allow a game designer to provide real-time animation. Animation in television or in movies is obtained by displaying a sequence of still images quickly enough so that the eye will blend them into smooth motion. The number of frames shown per second ranges from 30 frames per second for television, to 24 frames per second for a 32 millimeter film, to 18 frames per second for an 8 millimeter film. The goal for the current version of **DREGS** is to allow a minimum of 10 screen updates per second.

Smooth animation in **DREGS** is limited by several factors: the inter-machine message delay, the number of players, the desired time-quantum between updates, and the screen update time. We have derived a simple relationship that holds among the first three factors. Since **DREGS** does not provide an Output Manager, the screen update time was not taken into consideration.

We have observed the inter-machine message delay, D, to be about 10ms for our implementation, which uses XDR Record Streams [5] over a 10Mb Ethernet between DEC Vaxstation IIs running 4.3BSD Unix. The number of players, N, and the time-quantum, Q, are bound by the relationship:

$$N \times D \leq Q .$$

Intuitively this means that the delay for sending messages to all players ($N \times D$) must be less than the time-quantum between updates. By fixing the smallest acceptable time-quantum, Q, you can derive the maximum number of players, N. For example, in our current configuration D is 10ms and we have set Q to 1/10th of a second (100ms), limiting the number of players to 10.

6. Concluding Remarks

We have implemented several games using **DREGS**. From our experiences, we have found that the **DREGS** game model is appropriate, workable, and easy to use. The central Arbiter plays a crucial role in this model by simplifying the operation of the Input and Game Managers and by providing synchronization. Further, by using XDR to encode messages, we are able to operate in a heterogeneous environment with little overhead.

Although we have not yet completed a GDL compiler, hand-compiling several GDL programs has shown us that such a compiler is feasible. Our experiences have shown that implementing a game using GDL is straightforward. Currently, the most time consuming part of implementing a **DREGS** game is hand-compiling the GDL program. Once the GDL compiler has been completed, the task of implementing a **DREGS** game will be greatly simplified.

To date, we have written three programs using **DREGS** and GDL. *PLine*, an N-way talk program, was the first. We wrote PLine because it is simple and does not require an extensive graphical interface. The program demonstrated the workability of **DREGS** and showed that adequate performance could be obtained.

To further test **DREGS** and GDL, we wrote *Tank*, a game in which the player manipulates a two-dimensional tank. Tank is a more complicated game, consisting of two types of graphical objects (tanks and missiles) which can move vertically or horizontally. Implementing Tank showed us that GDL is a versatile, and therefore powerful, tool for specifying games.

We modified and improved Tank to create our most recent game, *SpaceWar*. The premise of SpaceWar is the same as that of Tank; however, in SpaceWar the tanks have been replaced by flying saucers that can move at arbitrary angles. Again, **DREGS** and GDL made this otherwise challenging task relatively simple and straightforward.

Finally, from the experiences of implementing PLine, Tank, and SpaceWar we learned that games really *are* fun!

7. Acknowledgements

We would like to thank our wives, Sharon and Sue, for their patience while we slaved over the design and implementation of this project. Their careful reading and editing eliminated numerous errors and greatly improved the readability of this paper. The work that Robert Hagens has done on the GDL compiler will undoubtedly make future game generation much easier.

We would also like to thank Andy, Beau, Brian, David, Derek, Dick, Joel, Julie, Kishore, Nancy, Prasun, Scott, Toby, and all the others that helped us "debug" our games.

8. References

1. R. Finkel and U. Manber, ''DIB - A Distributed Implementation of Backtracking,'' *Proceedings of the 5th International Conference on Distributed Computing Systems*, pp. 446-452 (May 1985).
2. J. Gettys, R. Newman, and T. D. Fera, ''Xlib - C Language X Interface,'' X Reference Manual (November 1985).
3. J. Morris, M. Satyanarayanan, M. Conner, J. Howard, D. Rosenthal, and F. Smith, ''Andrew: A Distributed Personal Computing Environment,'' *Communications of the ACM* 29(3), pp. 184-201 (March 1986).
4. K. Arnold, ''Screen Updating and Cursor Movement Optimization: A Library Package,'' in *UNIX Programmer's Manual.*
5. B. Lyon, ''Sun External Data Representation Protocol Specification,'' Sun Microsystems, Inc. Technical Report (April 1985).
6. Eric C. Cooper, ''Replicated Distributed Programs,'' *Proceedings of the 10th Symposium on Operating Systems Principles*, pp. 63-78 (December 1985).

9. Appendix

This Appendix contains an example of a GDL program for a simple game in which balls move on the screen. When a ball collides with another, it explodes and a new ball is created. The player can control the speed at which the ball moves by inputing a single digit ranging from 0 (stopped) through 9 (very fast).

```
STARTUP BEGIN                           /* Define initialization code */
    struct Ball b;
    InitOutputManager();                /* Initialize the Output Manager */
    InitGame();                         /* Initialize the Game State */
    InitBall(&b);                       /* Fill in initial values for b */
    Ball_Create(&b);                    /* object_Create routines take pointers to objects */
END

OBJECT Ball                             /* Define an object called 'Ball' */
BEGIN
    STATE BEGIN                         /* Define Ball state information */
        int     b_x, b_y;               /* X and Y coordinates of the ball */
        int     b_dir;                  /* Direction of travel */
        int     b_speed;                /* Speed at which the ball is moving */
    END

    /*
    **  For each object the following variables are
    **  automatically defined and set by the runtime environment:
    **          object_Ptr          a pointer to the object
    **          object_Input        a structure containing the input
    **                              that caused the event.
    */
    CONTACT BEGIN                       /* Define code to check for collisions */
        if( HitBall(Ball_Ptr) ) {       /* HitBall is externally defined */
            Ball_Explode(Ball_Ptr);     /* Ball_Explode is defined below */
        } else if( HitWall(Ball_Ptr) ) {  /* HitWall is externally defined */
            Ball_Bounce(Ball_Ptr);      /* Ball_Bounce is defined below */
        }
    END

    UPDATE BEGIN                        /* Define code to update a ball's position */
        /*
        **  Xincrement and Yincrement are externally defined
        */
        Ball_Ptr->b_x += Xincrement[Ball_Ptr->b_dir] * Ball_Ptr->b_speed;
        Ball_Ptr->b_y += Yincrement[Ball_Ptr->b_dir] * Ball_Ptr->b_speed;
    END

    DISPLAY BEGIN                       /* Define code to display a ball */
        DisplayBall(Ball_Ptr);
    END
    REPLICATED Ball_Explode                   /* Define a replicated procedure Ball_Explode */
    BEGIN
        struct Ball b;
        Explode(Ball_Ptr);                    /* Call a local procedure to explode the ball */
        InitBall(&b);                         /* Initialize another ball */
        Ball_Create(&b);                      /* Create the new ball */
    END

    REPLICATED Ball_Bounce                    /* Define a replicated procedure Ball_Bounce */
    BEGIN
        Ball_Ptr->b_dir = Reflect(Ball_Ptr);  /* Calculate new direction */
    END

    INPUT '0-9'                               /* Define an input event */
    BEGIN
        /*
        **    Ball_Input.i_char will be set to the input character
        */
        Ball_Ptr->b_speed = (int) (Ball_Input.i_char - '0');
    END
END                                           /* Object Ball */

TIMER 60 SECONDS                              /* Define code to be executed every 60 seconds */
BEGIN
    CheckForMail();
END
```