EM - An Environment Manager For Building Networked Virtual Environments

Qunjie Wang, Mark Green and Chris Shaw
Department of Computing Science
University of Alberta
Edmonton, Alberta, T6G 2H1, CANADA
email: {qunjie,mark,cdshaw}@cs.ualberta.ca

October 14, 1994

Abstract

The Environment Manager (EM) is a high-level tool for constructing both single user and multi-user virtual environments. A script file is used to initialize and run virtual worlds. Independent applications can share information and cooperate with each other across the Internet. EM reduces the effort required to produce a networked virtual world by providing high-level support for application replication, network configuration, communication management and concurrency control. This paper describes the architecture and implementation of EM.

1 Introduction

With many single-user Virtual Reality(VR) applications being successfully implemented [3, 12, 13, 10, 17], Networked Virtual Reality is now becoming a hot topic. *Networked Virtual Reality* refers to virtual environments where multiple users connected by a network can share information with each other. The production of good VR worlds, whether single user or networked, requires a considerable amount of design and programming time. Expertise is required in device handling, user interface design, network programming, graphics programming, and interaction techniques.

We have been building software tools that reduce the amount of development work for VR, allowing the VR implementer to tackle a wider range of applications. These tools are briefly summarized:

The MR Toolkit [15] provides standard software facilities required by VR user interfaces. It provides support for common VR devices such as 3D trackers, Head Mounted Displays (HMDs), gloves, and 3D mice, and supports distribution of the user interface and data over several workstations. A single-user MR application consists of one or more UNIX-style processes, with one designated as the master process, and the others as slave or computation processes. Slaves are used to perform output tasks on non-master machines, such as rendering the other eye's image for a HMD, and computation processes perform CPU-intensive tasks on computation server machines. MR applications are written in C or FORTRAN, and the graphics programming is done using the machine's native graphics library such as GL, Phigs or Starbase.

The MR Toolkit Peer Package is an extension to the MR Toolkit that provides the connection level facilities to allow multiple independent MR applications to exchange data with one another across the Internet [14]. The master process (the **peer**) can transmit device data to other remote master processes and receive device data from them. Application-specific data can also be shared between peers. Any MR Toolkit program may start up the peer package at any time, and may initiate and quit communications with other processes at will. Peers are connected pairwise and one peer may send a message to any or all other peers using procedures.

JDCAD+ [9, 11] is a solid modeling and animation computer-aided design system. It uses a Polhemus or Ascension 3D tracker to sweep out 3D canonical shapes such as boxes cylinders, cones and the like. These shapes can be reshaped, joined together and connected in kinematic chains. JDCAD+ has a keyframe animation facility that can be used to animate various motions of an object. JDCAD+ automatically generates OML animation code, and most animations can be created without the user having to write an OML program.

OML (the Object Modeling Language) is a procedural programming language we have designed, with fundamental data types and operations for geometry, object-oriented programming and behavior specifications [8]. It is used to describe the geometries and behaviors of 3D objects used in virtual worlds. The geometry processor within OML supplies efficient collision detection between objects selected by the world designer, and performs efficient object culling to maximize visual update rate. OML is designed to be portable to any platform, so its geometric modeling aspects are independent of any particular graphics package.

An OML *object* corresponds to a C++ Class, and contains code to generate the geometry of the 3D object, to control how the 3D object is to appear (color, texture, etc), and behavior code. An OML *instance* corresponds to a C++ object instance. An OML *behavior* is a procedure (method) that reacts to an incoming event or combination of events, and typically generates some sort of change in the state and appearance of the 3D object. Behaviors trigger other behaviors via the event mechanism. A built-in *tick* event triggers ongoing behaviors like walking and so on, and an internal time value can be used to interpolate between keyframes.

The OML compiler produces an interpretable version of the object specification (called the *object prototype*) and the OML interpreter is linked with the application program at run-time. One can create an arbitrary number of instances at run-time and have high level control over their behaviors. Different instances for the same object can have different geometries and behaviors, since object specification can be parameterized.

To create a Virtual Environment using OML, an MR Toolkit program (written in C) loads compiled OML code for each of the objects, sets up the VR devices that are to be used, dispatches device-related events to the interpreter as appropriate, and calls the OML interpreter every graphical update. The interpreter evaluates the behaviors of each instance, and draws each instance. Writing a new C program for each new OML-based Virtual Environment is a rather error-prone and tedious process, so a high-level tool called the **Environment Manager** (EM) was developed on top of our existing tools to eliminate the programming effort of initializing and running OML-based virtual environments. EM can run both single user and networked virtual environments, as specified on the command line. EM constructs a virtual environments using only a script file, without any lower level programming. The architecture of our VR tool package and the relationship between its components is illustrated in Figure 1.

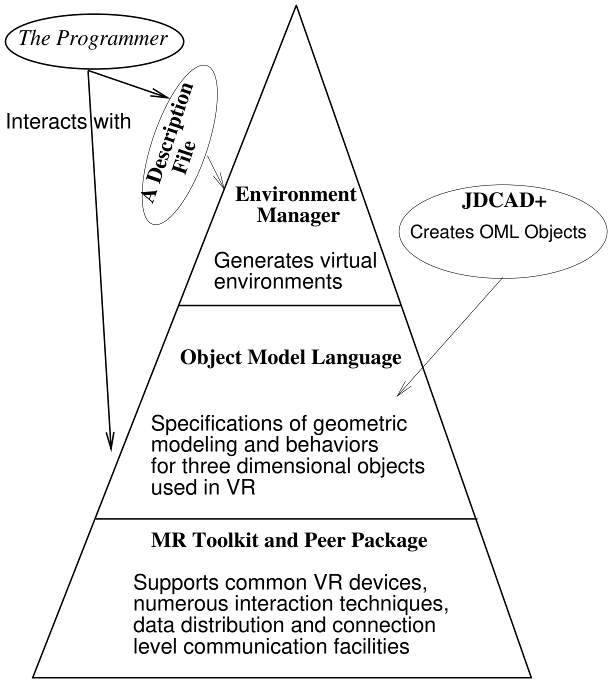

Figure 1: A VR Tool Architecture And Its Component Relationships

2 Related Work

In this section, we focus on existing networked VR systems and tools. All the newly developed tools and systems use the *application replication architecture* – each participating process has a copy of a replicated application database and changes are propagated to the other processes.

The most famous example of multi-user virtual environment is SIMNET [1, 4], which is a distributed interactive virtual world for battlefield simulation and training. In SIMNET, an object broadcasts an event to all objects without calculating which other objects might be interested in the event, or how the receiving objects might be affected by it. The receiving object decides what it is going to do about the received event. Objects transmit information only about changes in their state (position, orientation), and the *dead reckoning* algorithm is used to extrapolate state for objects. NPSNET [17] uses SIMNET's DIS protocol to perform military simulations.

The Distributed Interactive Virtual Environment(DIVE), developed at the Swedish Institute of Computer Science, is a platform for heterogeneous multi-user virtual environments [5, 6]. A process group in DIVE is a set of processes which can be addressed as one entity: atomic multi-cast protocols can be used to relay the messages addressed to the group as one, so each process in the group can receive exactly the same updates with reduced network traffic. In DIVE, there are three mechanisms to ensure consistency in the replicated database: mutual exclusive locks, reliable source ordered multi-cast and distributed locks.

BrickNet is a networked virtual reality toolkit developed at the National University of Singapore [16]. Different from DIVE and the MR Peer Package, which have a peer-to-peer communication scheme, BrickNet has a client/server communication configuration. A virtual world developed using BrickNet is a *client*, which connects to a *server* to request objects of interest and communicate with other clients. BrickNet virtual worlds are not restricted to having identical local databases (set of objects), as is the case with SIMNET and DIVE (they are multi-user-same-content applications). Similar to MR Peers, BrickNet uses UDP to transmit messages.

3 EM Architecture

EM binds together multiple OML objects into a virtual world. It sets up the VR devices, dispatches events, and calls the OML interpreter every update. In the networked (multi-user) case, EM implements the replicated architecture, allows identical or different content network configuration, performs shared information communication management, concurrency control and network bandwidth reduction.

EM is in charge of the initialization and management of the local application as well as communications between the other applications. Each user in a multi-user virtual environment runs a local EM

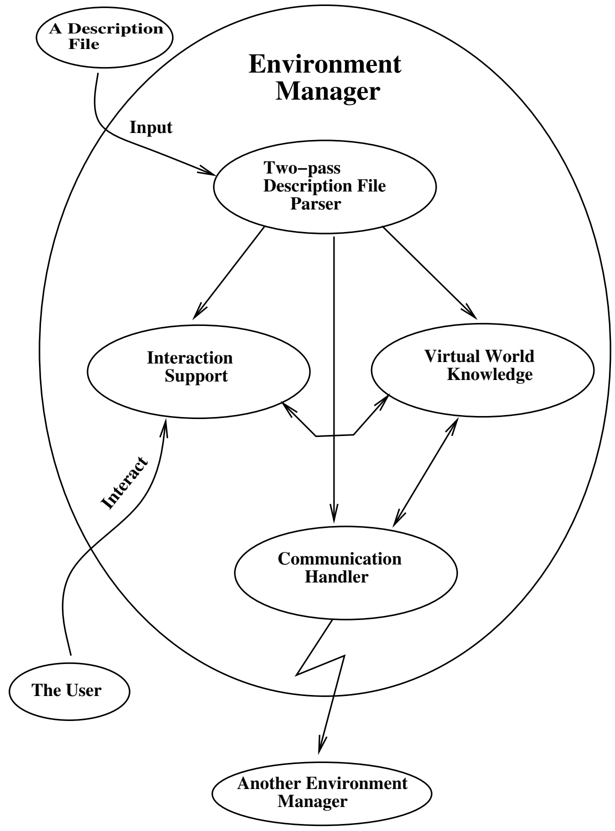

Figure 2: Architecture Of The Environment Manager

which communicates with other EMs across the network. The OML objects managed by EM need not be aware whether the world is single-user or multi-user, since EM handles all the event dispatching that triggers OML behaviors. This allows an object designer to create OML objects without having to worry about what context they may be used in.

The virtual world built by EM (called an *EM world*) is comprised of OML objects and instances that embody their graphical, behavioral and network properties. Unlike SIMNET, EM worlds are not restricted to sharing an identical set of objects. An EM world manages its own set of objects and instances, some or all of which may be shared with other EM worlds on the network. *Local* objects and instances are those which reside on the user's home machine. *Shared* objects or instances may be loaded by other EM worlds on the network. The remote versions of local objects are called *ghosts*.

EM consists of four subparts: (see Figure 2) the description file parser, interaction support, the VR world and the communication handler.

3.1 The Description File Parser

EM starts by reading an EM world description file that states the configuration if this EM world. A two-pass parser reports syntax errors in the EM file, and collects information about device configuration, object files, Instances, and behaviors. This information is processed by the remaining EM subparts.

The following information is specified in an EM description file:

1. Device configuration.

   This part specifies the local devices to use.

2. Object prototype files

   The local OML object prototype files are listed here. EM loads these files into the OML interpreter.

3. Instances

   Each instance of a local object is specified here. Each entry consists of the name of the object, the name of the instance, the parameter values used to create the instance, the mapping from events to behaviors, and the shared variables for the instance, if there are any.

4. Object behaviors

   If an instance does not specify an event for a behavior, it will inherit the event for that behavior from its object. These object behaviors will be inherited by remote instances of this object if this object is shared. This inheritance eliminates the tedium of having to specify the same event for each instance of an object.

5. Networked EM world description.

   This part specifies the shared objects and instances.

For a single-user EM world, networked descriptions (item 5) are not included. The networked EM world description specifies the shared objects and instances that can be exported to or imported from other EM worlds. This specification has four parts:

1. Concurrency control scheme.

   The EM programmer may choose one of several concurrency schemes, as outlined in section 4.

2. Shared object information.

   This is a list of the names of the objects that are required by this EM world. The local EM will solicit remote EMs for these objects. Hopefully the remote EMs which have them and are willing to *export* them.

3. Shared instance information.

   This information is provided when an EM world wants to send shared instances to remote EM worlds. Remote EMs should have the objects that the instances are created from. The information is a subpart of the EM description file containing instance entries, identical to item 3 in the previous list.

4. Expected object information.

   This is a list of objects that the local EM world expects to receive from remote EMs. *Expected* differs from *required* in that required objects are solicited by the local EM while expected objects are not. If the local EM receives an object that is neither required nor expected, it is discarded.

3.2 Interaction Support

The MR Toolkit manages devices such as 3D position and orientation trackers, hand digitizers, and sound I/O using the client-server model. The EM interaction support simply states what devices are to be used, and automatically performs MR's data collection. EM also creates an instance of a special object called "body", which provides instance variables for the user's eye position and orientation, the hand position, orientation and finger shape, and tracker button state.

3.3 The VR World

This subpart of EM manages objects and instances, either *local* or *shared*, to form the content of the virtual world. EM loads in all the OML object prototypes, creates, positions and orients the instances within the environment, and builds the event table for all instance behaviors. The mapping from events in the environment to behaviors is handled at run-time. At each time step, EM checks the events in the event table to see if any behavior needs to be activated. If an event has occurred, EM activates the corresponding behavior, puts it on the active behavior list, and runs it at the next time step.

An EM world can receive shared objects in the form of OML code during the simulation. These objects are loaded into the interpreter by EM as they are received. Due to the interpreted nature of OML, an object can be loaded multiple times during the simulation, allowing incremental updates. An EM world can also receive shared instances in the form of an EM description file, which is parsed by the EM parser to create the ghosts of shared instances.

3.4 The Communication Handler

In BrickNet, the communication is based on the client/server architecture, whereas in EM the communication is peer-to-peer. We put the "server box" of BrickNet's client/server architecture down in each peer. From the command line, EM gets the information about the remote peer that the local application wants to connect to. The local process does not block while starting communications with a remote peer. All connection solicitation and other peer traffic takes place asynchronously, so any EM application may start up the peer package at any time.

EM parses shared data items from the description file, and allocates shared data storage for each item. EM handles application data updates based on the concurrency control scheme stated in the description file. The peer shared data facility also allows a callback to be defined whenever a shared data item arrives from a remote peer. The callbacks are used by EM to update the actual application variables and to activate the EM description file parser if a piece of EM description file code is received.

The peer package and EM maintain a complete graph connection topology by default, which means that each peer is connected to all other peers explicitly. Each peer has a list containing all the active remote peers, and it informs the connected peers when it receives a new remote connection. Any EM world may start up the peer package at any time and may initiate and quit communications with other EM worlds at will. A *quit* command is sent out when an EM world wants to quit, causing the recipient to delete this remote peer from its connected-peer list. EM defines a callback for the quit message, which deletes the shared instances and objects owned by the remote peer sending out the quit message.

3.5 Unique Instances

One of the main features of EM networked worlds is the ability to create multiple new instances while the simulation is running. It should be possible to communicate between instances regardless of the locations of the sender and the receiver. Each peer must be able to determine the instances associated with shared messages. Therefore EM assigns a unique identifier to an instance which can then be used to communicate with it.

3.6 Reducing Bandwidth

The main limitation on maximum performance for distributed VR systems is the bandwidth of the connections between processors in the system. It is necessary to reduce the communication between processors as much as possible. In EM, instances transmit only shared variables whose states have been changed. Messages are sent only to relevant EM worlds. Like SIMNET's dead reckoning, EM supports local simulation of the behavior of shared instances. EM also provides *quenching* and *unquenching* messages to eliminate unnecessary communications, if an EM world decides to stop collaborative work for a period of time and does not wish to be informed of any updates for all imported objects during the break.

4 Concurrency Control

The replicated architecture needs concurrency control to resolve conflicts between participants' simultaneous operations. Concurrency control algorithms used in distributed database systems, such as explicit locking and transaction processing, are not appropriate for networked virtual environment under certain circumstances [7]. Networked VR systems and distributed systems are similar in that they are distributed over a network and they are shared by multiple users. However, a networked VR system is required to be *responsive* [2]. Under certain circumstances, responsiveness can be lost when *locking* (applied in DIVE) or *centralized controller* (applied in BrickNet) are used.

We have found that different applications may need different concurrency control schemes. It is not necessary to find a generic control scheme for every type of application; instead, we can implement several schemes, and leave the choice of scheme to the users:

4.1 Simulation Ownership Token Passing

Some networked VR applications (e.g. the multi-player handball game [14]) are intended for situations where only one participant at a time owns the simulation and is active, while the other participants watch the simulation, and wait for their turns. We maintain consistency of a distributed virtual world database by restricting manipulation in such a way that only one site can perform operations to alter the status of the virtual world.

4.2 Instance Or Variable Ownership And Access Permissions

Instance or instance variable ownership indicates that only one EM world is permitted to have control over that value. The ownership may be fixed at inception, or it may shift between EM worlds as the application demands. For example, the ownership of a tank in SIMNET is never transferred, while the handball game transfers ball ownership.

As mentioned in section 3, an EM world may have local objects and instances which have no relation to other EM worlds, or it may export objects or instances to the network and share them with other EM worlds. Using ownership, we can restrict sharing in various ways. For example, an instance might be visible to other EM worlds, but updatable only by its current owner.

EM assigns access permissions to each instance shared variable, similar to file access permissions. EM defines two kinds of permission for a shared variable:

- Writable Permission

  Other interested EM worlds have permission to write to this shared variable. Every EM world is permitted to write its local shared variables.

- Readable Permission

  Other interested EM worlds have permission to read this shared variable, and the variable owner will send out the current value if necessary.

These concurrency control algorithms are managed and enforced by EM, and do not require special OML coding to implement. For example, if a local instance variable is writable, the OML interpreter does not know whether this local variable has been written by a remote EM. This allows for reuse of objects in both single and multiple user virtual environments without the need for rewriting the OML code for an object. Because instance variables are updated by EM before it calls the OML interpreter, all behaviors will run with the new values. External changes to instance variables will not occur in the midst of OML behavior execution.

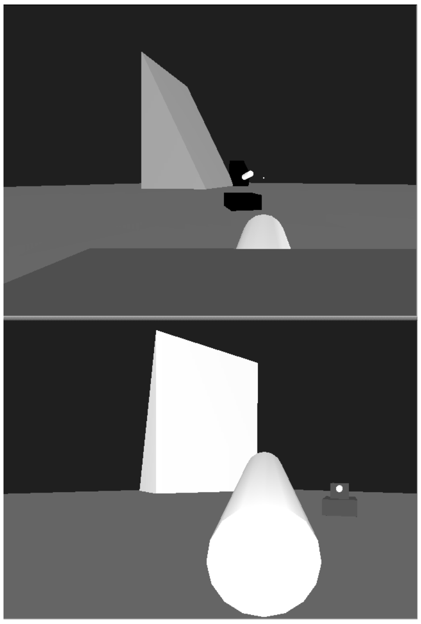

Figure 3: Simplified tank battle. In the upper picture, the black tank has just blown up. The lower picture is the black tank's simultaneous view of this.

5 Examples

We introduce three networked demonstrations build using EM: (1) a dynamic target shooting game, (2) a simple tank battle, and (3) the East Edmonton Mall design environment.

The shooting game allows networked players to shoot dynamic targets selected by the opposing players. One player shoots at a time, while being watched by the remaining players. The *simulation token passing* concurrency control is used in this demonstration.

In the Tank Battle demonstration (figure 3), each EM world represents a soldier owning a tank and its view-scope. The object space is the same across all the participants. Every soldier can enter or leave the battle simulation at any time. Each EM world broadcasts a tank instance (a ghost instance) representing the local tank to all the other EM worlds, so that each soldier can see all of the tanks which are currently in the battle. The instance ownership and access permission are used in this demonstration. OML's collision detection facility generates the collisions between tanks and enemy bullets, which in turn allows a bullet to update the tank's writable hit variable. The appendix shows the EM script for this demonstration, and figure 3 shows two simultaneous screenshots from two users.

The East Edmonton Mall Design Environment is representative of networked design systems that can be supported by EM. Unlike the previous two "same-content" networked VR worlds, this environment consists of three EM worlds, each of which has different objects and content.

The Airplane Museum is an EM world that contains a number of airplane objects on display to its user. The second EM world is the Sculpture Studio, which contains a collection of dynamic sculpture. The Mall Design Studio allows its user to design a building in which various sculptures and airplane models are to be placed. Because the Mall does not have its own sculptures or airplanes, it sends out a request to the network for the specified plane models and sculptures. The Plane Museum and the Sculpture Studio export their objects, along with the object behaviors onto the network upon receiving the request. The Mall Design Studio takes the received models, places them in the desired spots and shows their dynamic features (behaviors). Updates for these shared pieces are sent out by their owners whenever any new design idea has been applied.

6 Conclusions

The Environment Manager (EM), Object Modeling Language (OML), JDCAD+, and the Minimal Reality (MR) Toolkit provide a rich set of functionalities geared towards expediting the creation of both single-user and networked, multi-user virtual worlds. They eliminate the need to learn about low level graphics, device handling and network programming. This is achieved by providing higher level support for graphical, behavioral and network modeling of virtual worlds.

The MR Toolkit provides support for common VR devices and numerous interaction techniques. OML provides geometry and behavioral modeling for three dimensional objects used in virtual worlds. JDCAD+ is used to interactively create and animate dynamic 3D objects, automatically creating OML code. EM sits on top of OML and MR, allowing for the easy construction of virtual environments. Instead of asking the developer to start from scratch, EM generates both single-user and networked virtual worlds using a simple script file where the device configuration, object and instance information (including *parameters*, *behaviors* and *network-shared information*) are specified.

EM provides higher level support for the creation of networked virtual environments. It reduces the effort required to design a networked VR by providing the standard facilities required by a wide range of networked virtual worlds.

References

[1] Earl A Alluisi. The Development of Technology for Collective Training: SIMNET, a Case History. *Human Factors*, 33(3):343–362, June 1991.

[2] Perry A. Appino, J. Bryan Lewis, Lawrence Koved, Daniel T. Ling, and Christopher F. Codella. An Architecture for Virtual Worlds. *PRESENCE: Teleoperators and Virtual Environments*, 1(1):1–17, Winter 1991.

[3] Frederick P. Brooks Jr. Walkthrough - A Dynamic Graphics System for Simulating Virtual Buildings. In *Proceedings 1986 Workshop on Interactive 3D Graphics*, pages 9–21, Chapel Hill, North Carolina, 1986. ACM SIGGRAPH.

[4] James Calvin, Alan Dickens, Bob Gaines, Paul Metzger, Dale Miller, and Dan Owen. The SIMNET Virtual World Architecture. In *IEEE Virtual Reality Annual International Symposium (VRAIS 93)*, pages 450–455, Seattle, Washington, September 18-22 1993. IEEE.

[5] Christer Carlsson and Olof Hagsand. DIVE - a Multi-User Virtual Reality System. *IEEE Virtual Reality Annual International Symposium (VRAIS 93)*, pages 394–400, 1993.

[6] Christer Carlsson and Olof Hagsand. DIVE - a Platform For Multi-user Virtual Environment. *Computers and Graphics*, pages 663 – 669, 1993.

[7] C. A. Ellis, Simon J. Gibbs, and G. L. Rein. Groupware: Some Issues and Experiences. *Communications of the ACM*, 34(1):39–58, 1991.

[8] Mark Green. *Object Modeling Language (OML), Version 1.1.* Department of Computing Science, University of Alberta, 1994.

[9] Sean Halliday and Mark Green. A Geometric Modeling and Animation System for Virtual Reality. In Gurminder Singh, Steven K Feiner, and Daniel Thalmann, editors, *Virtual Reality Software and Technology (VRST 94)*, pages 71–84, Singapore, August 23-26 1994. World Scientific.

[10] Larry F Hodges, Jay Bolter, Elizabeth Mynatt, William Ribarsky, and Ron van Teylingen. Virtual Environments Research at the Georgia Tech GVU Center. *PRESENCE: Teleoperators and Virtual Environments*, 2(3):234–243, 1994.

[11] Jiandong Liang and Mark Green. Geometric Modeling Using Six Degrees of Freedom Input Devices. *Computers and Graphics*, 18(4), 1994.

[12] Michael W McGreevy. The Presence of Field Geologists in Mars-Like Terrain. *PRESENCE: Teleoperators and Virtual Environments*, 1(4):375–403, 1992.

[13] Warren Robinett and Michael Naimark. Artists Explore Virtual Reality: The Bioapparatus Residency at the Banff Centre for the Arts. *PRESENCE: Teleoperators and Virtual Environments*, 1(2):248–250, 1992.

[14] Chris Shaw and Mark Green. The MR Toolkit Peers Package and Experiment. In *IEEE Virtual Reality Annual International Symposium (VRAIS 93)*, pages 463–469, Seattle, Washington, September 18-22 1993. IEEE.

[15] Chris Shaw, Mark Green, Jiandong Liang, and Yunqi Sun. Decoupled Simulation in Virtual Reality with The MR Toolkit. *ACM Transactions on Information Systems*, 11(3):287–317, July 1993.

[16] Gurminder Singh, Luis Serra, Willie Png, and Hern Ng. BrickNet: A Software Toolkit for Network-Based Virtual Worlds. *PRESENCE: Teleoperators and Virtual Environments*, 3(1):19–34, Winter 1994.

[17] Michael Zyda, David Pratt, John Falby, Paul Barham, and Kristen Kelleher. NPSNET and the Naval Postgraduate School Graphics and Video Laboratory. *PRESENCE: Teleoperators and Virtual Environments*, 2(3):244–258, 1994.

7 Appendix

An EM Description File For The Tank Battle Demonstration

```
world tank_simulator

simulation shared information
   round_robin off

   broadcast_instances
     instance Tank tank1 (10, 10, 0.5, (1, 0, 0), 0)
     actions
       collision ( hill_OBJ ) explosion
       collision ( Bomb ) got_it
       collision ( Tank ) explosion
       need_to_explode explosion
       tick marching
     end
     shared information
       heading double writable
       tx double writable dead_reckoning
       ty double writable dead_reckoning
       hit integer readable writable
       start_marching integer writable
     end
   end

   objects expected
     Tank
   end
end

object_files
     world.obj tank.oml.obj body.obj
     hill.obj bomb.obj
end

instance WORLD terrain (-99, 99, -99, 99, 30, 30)

instance test_body body
actions
          tick navigation
end

instance Tank tank1 (10, 10, 0.5, (1, 0, 0), 1)
actions
          tick signal_marching
          begin_marching marching
          tick turning_with_hand
          tick signal_stop
          collision ( hill_OBJ ) explosion
```

```
        collision ( Bomb ) got_it
        collision ( Tank ) explosion
        need_to_explode explosion
end
shared information
        tx double readable
        ty double readable
        hit integer readable writable
        heading double readable
        start_marching integer readable
end

instance Bomb bomb1
actions
        tick stay
        tick start_to_shoot
        need_to_bomb bombing
        collision ( hill_OBJ ) stop_shoot
        collision ( Tank ) stop_to_shoot
        need_to_stop stop_to_shoot
end

instance hill_OBJ hill_1 (50, 50, (0.9, 0.3, 0.8) )

instance hill_OBJ hill_2 (40, -30, (0, 1, 0.5) )

instance hill_OBJ hill_3 (-80, 70, (0.8, 1, 0.5) )

instance hill_OBJ hill_4 (-65, -75, (0.3, 0.4, 1) )


end
```