

# DeepMatrix – An Open Technology Based Virtual Environment System

Gerhard Reitmayr

Vienna University of Technology  
Vienna, Austria

Shane Carroll

Geometrek, Nyon, Switzerland

Andrew Reitemeyer

Geometrek, Nyon, Switzerland

Michael G. Wagner\*

Arizona State University  
Tempe, AZ, USA

October 30, 1998

## Abstract

In this paper we present DeepMatrix, a virtual environment system based on two open technologies, Java and the Virtual Reality Modeling Language (VRML). The system is designed for use on current consumer hardware and requires only a standard Web browser with VRML plug-in. Due to a lean client – server implementation, system performance is superior to comparable approaches and easily extensible. This paper also introduces authoring for DeepMatrix environments and discusses results drawn from a large experimental implementation of widely varying and interconnected virtual worlds.

**Keywords:** VRML, Java, World Wide Web, Virtual Environment System

## 1 Introduction

Computer-generated simulations of real world environments where different users can interact with each other as well as with the environment itself are generally referred to as shared virtual environment systems. Recent years have seen a significant increase in the development of such systems [1, 2, 3, 4, 10, 13] with applications ranging from distance learning [11] to robotics [7] and medicine [8].

At the same time developments in the area of Multimedia and Information Technology resulted in open standards supporting 3D content storage and transmission. These standards include, for example, the Virtual Reality Modeling Language (VRML) ISO standard [19] which allows users to view three-dimensional content in a standard Web Browser equipped with a VRML plug-in. Unfortunately, however, the current VRML standard does not address networking issues required for interaction within a shared virtual world. As a result, a significant amount of proprietary multi-user VRML software has been released during past two years. This caused an inoperable mix of various multi-user VRML systems and hindered the development of broad consumer applications.

Nevertheless, due to the scalability of VRML, generic multi-user systems are possible by utilizing the External Authoring Interface (EAI) of VRML [19]. This interface allows to address complex tasks by connecting the VRML Web Browser plug-in with a Java applet within the same Web page. Consequently, it is possible to employ the powerful Java networking interface for communication purposes within a shared VRML world. A first distributed virtual environment system based on these technologies was recently proposed by Wray and Hawkes in [13]. Their architecture uses the Keryx Notification System for communication purposes resulting in a bandwidth efficient implementation.

---

\*email: wagner@asu.edu

In this paper we propose a system conceptually similar to the system by Wray and Hawkes. However, in contrast to their approach we decided to employ generally accepted communication standards such as TCP and UDP. The resulting client – server architecture is stable, robust, scalable, interoperable, open, extensible and easy to use. It minimizes server workload and leads to a lean, easy to maintain system implementation. In addition, the system is compliant with the LivingWorlds specifications as proposed in [18].



Figure 1: DeepMatrix entry world.

The main aim of this paper is to give a comprehensive overview of the DeepMatrix system including its client – server architecture and an introduction to DeepMatrix virtual environment authoring. In order to prove our concept we have implemented and tested an extensive number of shared virtual worlds ranging from interactive games to artistic virtual installations. We believe that these experiments clearly show the applicability of the DeepMatrix system for general purpose virtual environments.

The paper is organized as follows. In the second section we will first identify the technical requirements for realizing shared virtual environment on the Internet. We then will discuss which technologies fulfill these requirements and how they are connected. The third section deals with the server architecture of the system. After discussing the client architecture of the system we explain authoring issues, DeepMatrix authoring process and the creation of avatars for DeepMatrix. Experimental results and a brief outlook on future work conclude the paper.

## 2 Background

Any implementation of a shared virtual environment system has to address the following three basic issues.

1. *Data sharing.* In order to allow multiple users in a single environment, information has to be shared between different clients. Since these clients may be distributed over a heterogeneous network, any implementation has to platform independent.
2. *3D and multimedia rendering.* Virtual environments try to mimic the real world. Although we will mainly refer to the term virtual as a visual representation of a synthetic environment, acoustic and other sensual simulation is equally important. A proper multimedia rendering engine, that has to be platform independent as well, is therefore essential to guarantee smooth visualization of the environment.
3. *Interaction.* In a shared virtual environment, users have to be able to interact with each other as well as with objects located within the environment. Consequently, we have to provide functionality which permits the manipulation of multimedia information and the way it is rendered.

The last years have seen significant improvements in the development of standards that allow us to address the above issues exclusively based on open technologies. These technologies include Java, the Virtual Reality Modeling Language (VRML) and the External Authoring Interface (EAI) of VRML. In the following we will briefly describe the basic properties of each technology and their importance for the DeepMatrix system.

### 2.1 Java

The importance of Java to the DeepMatrix system has two aspects. Firstly, Java is platform independent and allows the server program to run on any operating system which supports a Java virtual machine. Secondly, Java code can be executed in HTML browsers and used as scripting language in VRML as well. Java applets can open network connections to servers and present a graphical user interface. Furthermore with the External Authoring Interface, it has access to the VRML plug-in functionality to display and control the visual simulation.

## 2.2 VRML

VRML is a file format to describe three dimensional geometry. Moreover it can describe simple animations and supports scripting to provide more complex functionality. It uses a prototype definition to realize extensibility. With this mechanism new objects can be defined and given new data fields and functionality. It has become the most widely accepted format to display three dimensional graphics in a web based environment. This was possible because of the development of HTML browser plug-ins supporting this format.

The extension mechanism is very important to the realization of the DeepMatrix System. It uses special defined prototypes to filter its relevant information out of the VRML files used to define the multi-user worlds. It receives and distributes events to these special nodes that form an interface to the VRML content itself.

## 2.3 EAI

The External Authoring Interface connects the Java Virtual Machine running in a web browser to execute applets and the plug-in used to display VRML content as shown figure 2. It is accessed with a set of Java classes defined in the EAI specification [16]. The DeepMatrix client uses the EAI to control the visual simulation in the VRML plug-in.

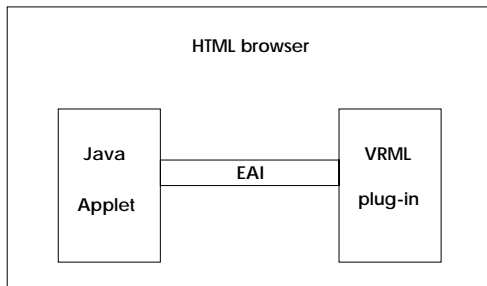


Figure 2: External Authoring Interface.

This interface allows a two way communication between the two components. The applet loads VRML content into the plug-in and adds avatar geometry to the existing world representation. Navigation is possible by using the plug-in controls, therefore the plug-in has to update the

applet about the users current position and orientation in the world and pass events to it from the VRML content.

## 3 DeepMatrix client – server architecture

Deep Matrix uses a client – server network architecture. There are two main reasons for this decision. Firstly the less involved implementation of a client – server architecture as opposed to a multicast architecture is expected to yield a more stable and robust system. The server guarantees that all messages are delivered in the same order to all clients. In a multicast approach keeping a consistent state across all participating processes requires a more complicated design. Secondly the clients are Java applets running in a HTML browser. The security policy of Java restricts applets to network connections with only the server they were downloaded from which actually forces us to employ a client – server architecture.

### 3.1 World model

The server constructs a simulation of the relation of real objects by supporting two notions. First there are *rooms* which provide a background geometry, any further content such as game boards and the VRML logic to drive the content. These rooms are provided by the server and cannot change their content at runtime. The second notion is *client objects* that are introduced into a room. These client objects are controlled and managed by the clients, and added or removed from rooms at runtime.

The relationship between rooms and client objects can be compared to a stage and actors playing their part on the stage. The stage provides the background and the stage props, whereas the actors move in the scene the stage displays, say lines and work with gestures to play their part. We will call this the stage – actor paradigm.

The room is used in a similar way to locals described in [1]. There is a subset of all clients related to each room. Only clients related to the same room can receive messages from each other and are updated on state changes in the VRML representing this room.

Each room holds an URL to a VRML file defining its visual representation. To provide more than static geometry

the DeepMatrix systems defines a set of VRML prototypes that allow the distribution of VRML events occurring at one client to other clients. The room again defines which clients receive these events.

These prototypes are implementations of the *network state* nodes defined by the Living World specification [18]. These nodes encapsulate a single VRML field whose value is shared across all clients. To use this functionality the world designer routes the VRML event he or she wants to be shared into a network state node and furthermore from this node to the destination node. Upon receiving an event the network state node passes it to the client which in turn transmits the event to the server for distribution to all clients. While we are focusing on the technical aspects here, section 6 will describe the concept of network state nodes from the authoring point of view.

A client can introduce client objects into a room with a VRML file giving the visual definition to use in other clients. The client can also control the position and orientation of such an client object and distribute signals to it that trigger VRML events in the VRML code of the client object (for example animations).

The client objects a client can introduce into a room are used as avatars for the clients. They may also be used as avatars for a robot program that may even control several objects. These client objects are identified by unique names.

The world model used by the DeepMatrix system implements the stage – actor paradigm very well. The rooms form a scene on a stage with a background and stage props realized with shared VRML events implemented by network state nodes. The client objects are actors set into that scene with exactly the possibilities described in the stage – actor paradigm.

### 3.2 Network Protocols

There are two different connections between the server and the client. First a TCP stream connection between the two programs is opened. This is used for general control messages that handle login and logout, managing of Avatar objects and passing of chat messages.

Furthermore the server distributes state information about the states in rooms and the client objects location and signals via UDP packets. It receives updates from a client and passes them on to the other clients. The set of

receiving clients is the set of clients in one room. This architecture is shown in figure 3.

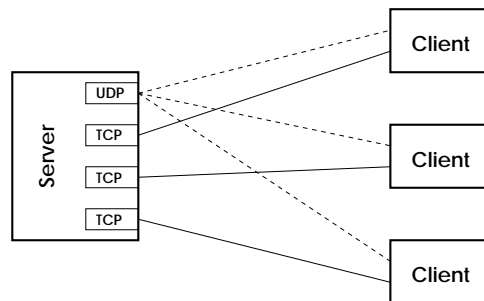


Figure 3: DeepMatrix client – server architecture.

This splitting of the data streams into a pure client – server related one and a client – client related one (with the server acting as a intermediary) circumvents certain problems appearing in the approach using only one stream to transport control and state messages. Processing state messages can use considerable amount of CPU time because the VRML plug-in renders a new frame if a state is updated. If there are a lot of state changes important control messages may be processed very late and the systems response time to user interaction suffers.

VRML code is transported using the HTTP protocol. This is not handled by the DeepMatrix system itself but the client uses the plug-in’s functionality to load VRML code specified by an URL. The system itself only passes information about the location of VRML code for the rooms or client objects by transmitting URLs pointing to VRML files.

## 4 DeepMatrix server

We will now give a detailed description of the architecture of the DeepMatrix server and the protocols being used.

### 4.1 Overview

The DeepMatrix server is a multi threaded server program managing two types of connection, TCP streams and a UDP packet distribution. It listens to a TCP port and spawns a new thread for each connection it receives. This

thread is then responsible for any further communication with this client. In addition to that a dedicated thread listens for UDP packets on a UDP port. These packets are relayed to the set of clients they are designated for. Three Java classes implement these features.

In addition to that the main program also handles the task of reading and parsing a configuration file and writing a log file. The configuration file is used to set the port number the server listens to, the path of the log file and the rooms it offers for its clients. This information consists of name - value pairs, where name is the name of the room and value is an URL of the VRML file that describes that room.

## 4.2 ServerThread

The Java class `ServerThread` implements the communication with a client over a TCP stream. It is created immediately after the main program receives a new request for connection and controls this connection from that moment on. It uses the TCP stream to implement an asynchronous message passing protocol to communicate with the client.

The messages are encoded as text strings and one line is exactly one message. This allows us to easily parse the messages and implement the protocol in any language. Furthermore it is easy to debug the server program, because a simple telnet session can be used to test it for the desired behavior. The structure of such a message is very simple. It starts with a three digit number to encode the command, then follows the name of the receiver or sender and it is ended with an argument. The name and the argument depend on the command the message encodes.

This protocol is used to implement the following functionality :

- login information such as name and UDP port the client uses
- logout of the server
- adding and deleting avatar objects
- changing between different rooms
- obtaining a list of the clients in the current room
- sending chat messages to other clients

When the main class listening to the TCP server socket receives a new connection it spawns a new `ServerThread` to handle that connection. First it enters the login sequence where information about the clients name and UDP port is obtained and the client is updated on the servers UDP port and information about the room it enters. Then the `ServerThread` object enters a loop to process subsequent messages from the client. It reads a messages (blocking if no message is in the TCP stream) and handles the clients request. Its core implementation is shown in appendix A.

## 4.3 Room

The class `Room` keeps track of all clients in one room. It holds a table of all clients in one particular room and another one of the objects added by them. If a new client changes to this room all other clients are updated on its existence. The new client obtains a list of all objects in the room to load them into its instance of the world. The `ServerThread` uses methods of this class to distribute messages such as chat messages to all other clients in one room.

It also distributes real time messages via UDP to its clients. For that the `RealTimeServer` class, which will be explained in the next section, maps the client from which the real time message originated to its room and forwards the message to this room for delivery to all other clients.

The `Room` is instantiated at startup when information about the rooms is read from the configuration file. For each name – URL pair an object of the class `Room` is created with this name and URL as data.

## 4.4 RealTimeServer

The class `RealTimeServer` is the third main part of the server program. Its core implementation is described in appendix B. The class receives the UDP packets used to update state information about the VRML world. Using the right room it distributes these real time messages to the right clients.

A real time message is a bit more complicated build than a control message. It holds the following information :

- the sending client's ID, a 32 bit integer number, that is given to a client at login and is unique for each client
- a string denoting the network state node it updates. This is the tag value of the node
- a VRML field value with the latest value of the network state node
- a boolean value `echo`, that tells the server if it should copy back the message to the sender itself

When the `RealTimeServer` receives a message it finds the client from which the message originated by the message's client ID. If found the client's room is used to relay the message to all clients in the same room. The room also evaluates the boolean value `echo` to check whether it should copy back the value to the sending client.

## 5 DeepMatrix client

We will explain the architecture of the client applet and describe how the VRML plug-in is controlled to display the visual simulation.

### 5.1 Overview

The client is implemented as a Java applet. It can be broken down into three independent main parts, that are build from several classes each. One part handles the network communication with the server. A second manages the graphical user interface of the applet itself. Graphical display of the world in the VRML plug-in is controlled by a third part that uses the External Authoring Interface (EAI).

The main class derived from the class `Applet` implements the application features by using these three parts. These parts can be seen as three sets of APIs, each dealing with a special part of the program.

### 5.2 Graphical User Interface

The graphical user interface of the client applet is simple and uses only standard Java AWT 1.0 features. We are limited here to using version 1.0 because of the various status of Java implementation in different HTML browsers. Again it can be broken down into two main parts that each reflect a special state of the application. The

first we call the *Login Interface* which displays controls to enter login information and connect to the server. The second is called *Runtime Interface* and displays controls used during a DeepMatrix session. A more detailed explanation of these interfaces follows.

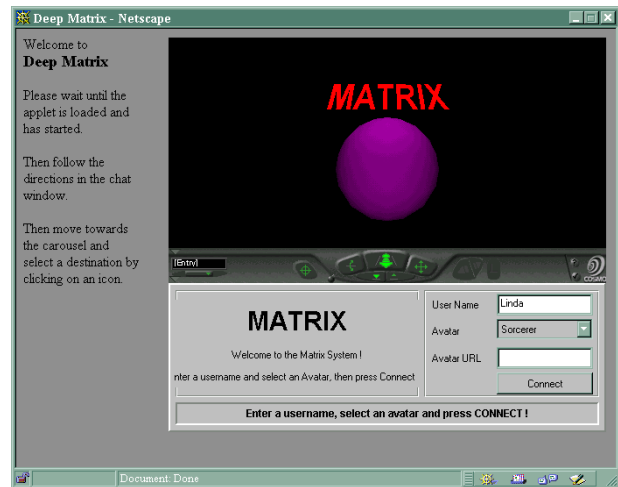


Figure 4: DeepMatrix login interface.

The *Login Interface* (see figure 4) displays a greeting text to the left of the applets window. The right holds three input controls and one button. Both parts are bordered at the bottom by a status line used to display various information. The first input control takes the users login name. Below this there is a choice control offering several different avatars to be used. The last entry in this choice control enables the user to supply his or her own avatar by entering the URL of this avatar in the last input control. The URL should point to a VRML file holding the visual definition of the avatar. The button below these inputs connects to the server and starts a DeepMatrix session.

The *Runtime Interface* (see figure 5) consists of a chat output area and a chat input line. Furthermore it has a list box displaying all users in the current room and a choice box to activate any avatar behaviors your avatar might have. It is bordered at the bottom by a button to disconnect from the server and the DeepMatrix session and a button to switch into *Ghost* mode, causing you to leave your avatar and roam the world without it. You can see your own avatar at its last position and other users are not notified of this change. They experience your avatar as stand-

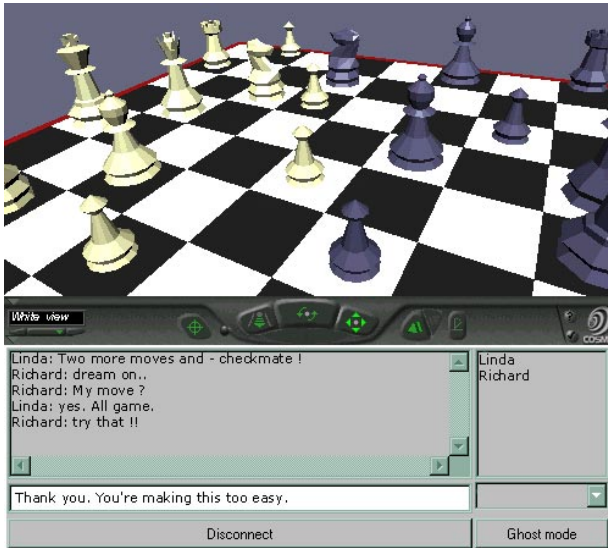


Figure 5: Discussion about a game of chess in the runtime interface.

ing still.

### 5.3 Network communication

The communication with the server is maintained by the class `Backend`. It implements the basic services to deal with both the TCP connection with the server and the real time messages send with UDP. This class opens and manages the TCP connection to the server acting as a counterpart to the `ServerThread` class in the server. Its implementation is very similar to the class `ServerThread`. It marshals messages into the string form used for communication and parses the strings received from the server. Furthermore it handles the login protocol and errors in the communication. Messages received by the server are passed to any observer that registered with the `Backend`.

To deal with the real time messages it uses a second class and an interface. The `RealTimeHandler` class listens to a UDP socket and receives the UDP packets from the server. It also marshals the real time messages and sends them to the server. Moreover it manages a list of objects implementing the `RealTimeConsumer` interface to receive or send real time messages. Upon receiving a message the `RealTimeHandler` tries to find the

`RealTimeConsumer` the message is addressed to and passes it to this object to deal with it. Being the counterpart to the `RealTimeServer` class of the server, its implementation is also very similar to the `RealTimeServer` class. The definition of the `RealTimeConsumer` interface is shown in appendix C.

### 5.4 VRML control

Controlling the VRML plug-in with the External Authoring Interface is implemented in a less centralized way. There are different classes that represent different aspects of the VRML simulation of the shared world. These use the EAI to accomplish their tasks.

Loading and setting up of the VRML file that represents a certain room is implemented in the applet class itself. To load a new world it creates an empty `Transform` node and replaces the current world with this node. Then it loads the new VRML file into the `Transform` node's children field.

When loading is completed it parses the new nodes and searches for the first `Group` node. It assumes that this group node holds all information necessary to link the world to the system. The group node children are parsed and their information extracted and observers are setup for the different nodes.

At the end of the loading process, the applet loads the users avatar and any other shared objects.

The class `Gate` controls a gate in the VRML world. This is a node that sends a string with the name of another world to the applet if it is triggered by a `SFBool` event. The `Gate` class registers an `EventOutObserver` with the corresponding event of the node and notifies the applet class if the node was triggered.

The network state nodes are controlled by a class called `Route` that implements the `RealTimeConsumer` interface. It also registers an observer for the `value_tonet` event of the node. If an event is caught, its value is sent as a real time message to the server. If the `RealTimeHandler` object receives a new value for this network state node it passes the value to the corresponding `Route` object using a method of the `RealTimeConsumer` interface. The `Route` object in turn transmits the VRML event to the node.

The class `Avatar` controls a shared object that was introduced by another client. It loads the objects geome-

try and adds a name sign to it. It also moves the object in the world according to the position and orientation updates it receives from the `RealTimeHandler` via the `RealTimeConsumer` interface it implements.

`UserAvatar` is a class to control the user's own avatar. It loads the user's avatar geometry into the world and builds the necessary node tree to support the ghost mode. By parsing the avatar geometry it also finds any behaviors the avatar exposes and exports these in the user interface. Moreover it registers an `eventOutObserver` for the `ProximitySensor` node's position and orientation events, respectively. These events are then transmitted to the server to update the position and orientation of the user's avatar at the other clients. It also implements the `RealTimeConsumer` interface.

## 6 DeepMatrix authoring

In the following we describe the general authoring concept in the DeepMatrix system which involves authoring in VRML. For a detailed introduction to VRML, we refer the reader to [9].

### 6.1 Worlds

Authoring DeepMatrix multi-user worlds involves three steps. First we have to create an HTML web page containing the client applet and an embedded VRML world. Second we have to implement some basic DeepMatrix functionality in the VRML world which enables the applet to access the world through the EAI. Finally, we have to edit the configuration file of the DeepMatrix server which tells the server to instantiate a room for the new multi-user world.

#### 6.1.1 HTML

The following is an example HTML file which will load a multi-user world called matrix.

```
<HTML>
  <HEAD>
    <TITLE>DeepMatrix Test</TITLE>
  </HEAD>
  <BODY BGCOLOR="#909090">
    <embed src="world/logo.wrl" border=1
```

```
      width="100%" align=center
      height="70%">
  <applet code="matrix.deck.Beta.class"
  mayscript width="100%" height="150">
    <param name=Port value="6666">
    <param name=Room value="matrix">
    <param name=Avatar1 value="Sphere">
    <param name=AvatarUrl1
      value="sphere.wrl">
    <param name=Background
      value="ff00ff">
    <param name=Foreground
      value="00ff00">
  </applet>
</BODY>
</HTML>
```

The client applet parameters have the following effects. The parameter `Port` tells it to which port to connect and has to be the same as the server is listening to. `Room` is the name of the world the user starts in. This has to be one of the names defined in the `matrixrc` file described later. `Avatar1` is the name of a default Avatar and `AvatarUrl1` is an `Url` of a VRML file holding the geometry of that avatar. It is either absolute, or relative to the codebase of the applet. In this example the codebase is the location of the HTML file. Additional avatars can be defined by adding `Avatar#`/`AvatarUrl#` parameter pairs. The parameters `Background` and `Foreground` set the background and foreground colors used in the applets user interface.

#### 6.1.2 VRML

DeepMatrix needs information about the VRML world. It has to track the users movements with a `Proximity` node and bind the initial `Viewpoint`, `Background`, `Fog` and `NavigationInfo` nodes. It also needs to know what jumps points, called gates, to other VRML worlds exist and which events are distributed to other instances of the same world.

The client applet waits for the world to be loaded and then searches the root nodes for the first group node it finds. Then it assumes that this group node holds all information it needs. It traverses the group node children and handles them according to their type. For each type of bindable nodes it encounters, the first one is bound. The first `ProximitySensor` node found is used to track the

users movements. The following code is an example of such a group node.

```
DEF MATRIX_CORE Group {
  children [
    DEF MATRIX_TRACKER ProximitySensor {
      size 100000 100000 100000
    }
    Viewpoint { position 0 1 10 }
    NavigationInfo { speed 5 }
    Background {
      skyColor [ 0.9 0.9 0.9 ]
      groundColor [ 0.1 0.1 0.6 ]
    }
    DEF color NetworkSFColor {
      tag "color"
    }
    DEF time NetworkSFTime {
      tag "time"
    }
    DEF door MatrixGate {
      target "cone"
    }
  ]
}
```

There are two different types of nodes that can be used in the MATRIX\_CORE group, namely gates and network state nodes.

**Gates.** Gates are nodes with tell the client to jump to another VRML world. They are triggered by an SFBool event. Their definition reads as follows.

```
PROTO MatrixGate [
  eventIn SFBool activate
  eventOut SFString isActive
  field SFString target ""
]
{
  Script {
    eventIn SFBool activateSc
    IS activate
    eventOut SFString isActiveSc
    IS isActive
    field SFString targetSc IS target

    directOutput TRUE
    mustEvaluate TRUE
```

```
url "javascript:
  function activateSc(value) {
    if(value==true) {
      isActiveSc=targetSc;
    }
  }
"
```

The script in this definition is necessary because the EAI allows only the monitoring of outgoing events. The SFString field target tells the client which world to jump to. The value of this field is the name of a certain world as in the server configuration file. The jump is triggered if the node receives an SFBool event via its eventIn activate.

**Network state nodes.** Network state nodes are implemented as proposed in the Living Worlds specifications [18]. As mentioned earlier they are a set of prototyped nodes which implement the distribution of events to other instances of the VRML world. There is one net node type for each VRML event type. The following PROTO defines a NetworkSFBool node for SFBool events. Other net nodes are built in exactly the same way.

```
PROTO NetworkSFBool [
  eventIn SFBool set_value
  eventOut SFBool value_changed
  eventIn SFBool value_fromnet
  eventOut SFBool value_tonet
  exposedField SFString tag ""
  exposedField SFBool pilotOnly TRUE
  field SFBool localCopy TRUE
  exposedField SFBool echo TRUE
  exposedField SFBool cont FALSE
]
{
  Script {
    eventIn SFBool InSc IS set_value
    eventOut SFBool OutSc
    IS value_changed
    eventIn SFBool netInSc
    IS value_fromnet
    eventOut SFBool netOutSc
    IS value_tonet
    field local IS localCopy

    directOutput TRUE
```

```

mustEvaluate TRUE

url "javascript:
  function InSc(value) {
    netOutSc=value;
    if(local==true)
      OutSc=value;
  }
  function netInSc(value) {
    OutSc=value;
  }
  "
}

```

Network state nodes have two pairs of events. The first labeled `set_value/value_changed` are connections to the VRML file. Any ROUTE can connect to these events. The other pair is labeled `value_fromnet/value_tonet`. The event `Out value_tonet` is monitored by the client applet and the event `In value_fromnet` is used to insert a distributed event into the VRML world. The `tag` field sets the name of the net node used in the messages sent across the network. It has to be unique in this world. The `echo` flag tells the server to send back an event from the network. Thus the local world will receive a copy from the event it sends out to the other instances of the same world.

The `cont` flag is a DeepMatrix extension to the Living Worlds specification. It tells the system that this event represents a continuously changing value. For example position of some object or interpolator events. In this case DeepMatrix doesn't propagate each change over the network in order to avoid delays due to event overflows.

### 6.1.3 Server configuration file

Finally, the user has to update the server configuration file `matrixrc` which contains the following information.

```

# sample matrixrc file
Loglevel 2    # 0 - 3
port 1009
logfile /usr/tmp/matrix.log

#here we start
start world/Start.wrl
cone world/cone.wrl

```

```
matrix matrix.wrl
```

The structure of such a file is very simple. If a line starts with the word `Loglevel` the server expects an integer between 0 and 3 to follow. This sets the detail of logging information to print to the standard output. 0 suppresses any information. 1 shows only errors, log ins and log outs. 2 adds room and avatar changes and 3 displays all information running over the server. A line starting with `logfile` gives the name of the log file the server should write to. If a line starts with `port`, the following integer number gives the port number the server listens to. Any other line describes the name and physical location of a room supported by the server. In the current setup the server has to be restarted after editing the `matrixrc` file in order to activate the changes.

## 7 Avatars

An avatar is a geometry that users choose to represent themselves in a multi-user environment. Restrictions to avatar design are mainly technological and these are being overcome as the web and VRML develops. These must be kept in mind as there is not much point designing an avatar today that could only be used at a future date. On the other hand, keeping in mind possible future developments will help the designer build into his design the capacity to update the avatar when appropriate.



Figure 6: Group of avatars in forest world.

### 7.1 Designing avatars today

The main concern today, in designing Avatars for multi-user systems such as DeepMatrix, is to meet the needs

of the avatar user while taking care not to make something that will seriously affect the smooth functioning of the server. The main areas of concern are:

**Badly coded VRML.** One has to keep in mind that VRML code may contain errors which can cause the client machine to crash. Syntax checking tools such as Chisel(TM) should therefore be employed to check the VRML code for such errors.

**Too large file size.** Excessive file size causes delay in initial loading and is generally handled by g-zipping the file. However there are limits to the amount of code a server can process, restricting avatar complexity and the amount of canned animation that can be built into an avatar file. This is especially true when `CoordinateInterpolator` nodes, as used in facial animation, cause scenes to render slower and thus limits the number of avatars that can be present simultaneously and may cause navigation problems for users on slower machines.

**Overuse of textures.** Excessive use of textures causes slower rendering of scenes, especially if a texture dominates the screen at some point or is animated. On the other hand, selective use of textures can help reduce the number of polygons needed to produce the effect required.

**Overuse of animation and scripting.** Excessive use of scripts uses up processing power that might be better used elsewhere. It is preferable to rely on interpolator nodes where possible and to avoid mesh deformations. Another way to reduce complexity is to tie animation scripts to `VisibilitySensor` nodes.

The advantage of designing for the DeepMatrix system is that the concept is based on custom design of worlds and avatars. This means the avatar's design parameters and the multi-user environment are known in advance and the above factors taken into account and balanced to give a specific solution. This is easier than trying to create an avatar that will have to operate in many different and often unknown multi-user environments. For example, a role playing game would require a set of avatars that are limited in number and have appearances and gestures that are

specific to the game's rules. This means resources do not have to be wasted on superfluous characteristics and can be used to make a more interesting design or spared to improve overall performance.



Figure 7: Avatar girls from forest world.

Customization does not mean that interoperability between systems is not possible. If a user wishes to use an avatar designed for one world in another, be it DeepMatrix or another multi-user system they should be able to as long as it is welcome and does not adversely interfere with the operation of the guest world. Issues that govern interoperability are:

**Avatar size.** Avatars should be built with respect to normal human dimensions, the most important of these being height. An avatar made as an insect should be scaled up accordingly and larger geometry such as a space ship should be scaled down.

**Avatar position.** Avatars made to humanoid animation (h-anim) specification [17] have the point of origin between the feet and the avatar is standing in a neutral position facing along the positive  $z$ -axis with its right side on the negative side of the  $x$ -axis.

**Avatar control.** DeepMatrix avatars are controlled by sending an event to the `Avatar` node which contains a suitable `eventIn` field and a `MFString` description with which to label the button on the control console. This is similar to other systems, however, differences in syntax and event type mean writing a control mechanism for each multi-user system. This problem will be resolved with the

advent of Core Living Worlds specification [18] which defines the general avatar layout roughly as follows<sup>1</sup>:

```
PROTO Avatar [
  exposedField MFString gestures
  [ "myGesture" ]
  eventIn SFTime myGesture ]
{
  # the geometry of the avatar is
  # placed here
  DEF Timer TimeSensor {
    cycleInterval 2
    startTime IS myGesture
  }
  # Interpolators, Scripts and ROUTES are
  # placed here
}
```

Designing and construction a functional humanoid avatar is expensive in terms of time and effort and it is therefore recommended to follow the h-anim specification [17]. This ensures that the avatar can be used in other h-anim compliant applications and will respond correctly to various types of animation data, such as motion capture data.

## 7.2 Future developments in avatar design

Currently used avatars are very restricted in their functionality. The following issues have to be addressed in order to improve practicability of avatars in systems such as DeepMatrix. Although there exist solutions to all of these problems, none of them have yet been sufficiently addressed in an VRML browser implementation.

- *Data streaming.* Streaming animation data to an avatar on demand will reduce the need for canned animation and reduce the file size and thus the initial load.
- *Audio Streaming.* Streamed audio would enable sound driven behaviors and audio chat.
- *Synchronization.* Synchronizing audio and one or more data streams would for example enable a waltzing couple to move independently and yet in time with each other and the music.

<sup>1</sup>At the date of this writing, DeepMatrix utilized boolean events instead of time events as an implementation bug in the CosmoPlayer VRML plug-in prevented the use of time events.

- *Natural Language Programming (NLP).* NLP is not essential, but desirable in producing intelligent autonomous avatars or bots.

## 8 Experimental results

Designing for multi-user environments starts with identification of constraints and deciding on acceptable performance parameters. For any widely deployed multi-user system, the constraints are severe and fall into two general categories, network latency and local rendering speed. We have designed, tested and analyzed a large system of interconnected worlds [14] using DeepMatrix (see figures 1, 8 – 14). The system is meant to be accessible over a low bandwidth network connection, but not limited to it. It also takes into account the lowest cpu speed which still allows the use of VRML - usually 166 mhz without hardware graphics acceleration. In the following we describe some of the experiences and observations we made during the test phase.



Figure 8: View over DeepMatrix entry world.

### 8.1 DeepMatrix performance

The system was tested on a variety of platforms but proved to be most stable on Netscape Communicator 4.05 or Internet Explorer 4 in combination with the CosmoPlayer 2.1 VRML plug-in on Windows 95/NT. Considering the fact that this combination in itself was known to be rather unreliable at the date of this writing the stability of the DeepMatrix implementation proved to be more than satisfactory.

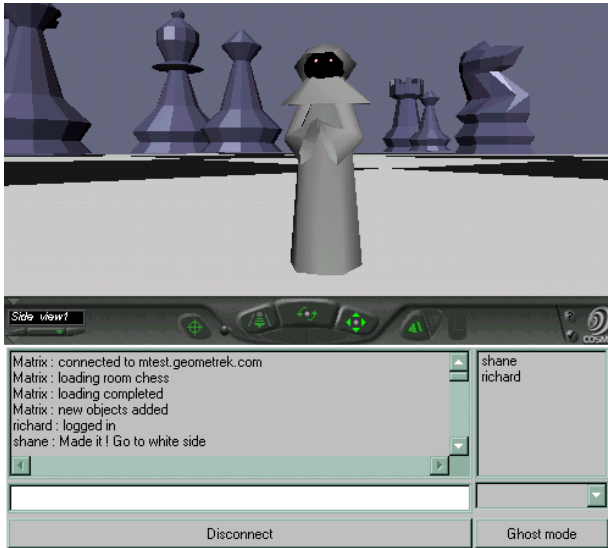


Figure 9: Sorcerer visiting the chess world.

Interaction within the DeepMatrix worlds is sometimes handicapped by state update delays introduced by the current VRML browser implementations. Unfortunately, VRML browsers tend to allocate all available cpu for rendering purposes and do not allow the DeepMatrix client to process incoming data as necessary. Although these delays are hardly recognizable in a standard chat world, they can disturb gameplay in one of the action game worlds such as the car race world. This problem, however, can only be solved at the browser level since there exists no



Figure 10: Virtual band playing in virtual disco.



Figure 11: Virtual dancer.



Figure 12: View on racing car in race world.

possibility to influence cpu time distribution from the outside.

## 8.2 General design observations

The purpose of a multiuser world must inform all aspects of its design. The design should accept the constraints of the system. While keeping an eye on the whole, experimentation with different design approaches may be enhanced by the acceptance of temporary and artificial parameters based on narrow definitions of purpose or focus. In the following, we describe three overlapping areas of focus in designing desktop multiuser worlds.

### 8.2.1 Social focus

Most current multiuser content has a social focus. Frame rates are very low, so the primary activity is chat. All



Figure 13: Passing a car in race world.

available cpu cycles should be spent on avatar behaviors, rendering, and network updates, but instead are often squandered on elaborate 'sets'.

Presentation of real world structures emphasizes the current limitations of desktop multiuser without taking advantage of the opportunities. In social worlds, the avatars take precedence over the environment. Real world architectural constructs are unnecessary and a hindrance. They soak up precious cpu cycles. They set up an expectation of mobility that is unavailable at the moment due to current navigation devices and low frame rates. Emphasizing the difficulty of movement does a disservice to the user. Frustration sets in, avatars become fancy location indicators, and users resign themselves to text chat.

The environment can be composed of a very low polygon count and no animation. Sensors can be limited to gates for inter-world movement. Aside from situations where volume clipping is an issue, i.e. dynamically generated spaces based on the number of users, there is no absolute need for walls. Transparent containers can keep users within the range of required proximity sensors while allowing for creative use of backgrounds. Except for architectural descriptions, there is no need for ceilings, load bearing members, or doors.

Current system constraints present an opportunity to set aside convention. The technology is capable of providing the setting for a major shift in social interaction; why limit the design to the tenets of physical architecture or Euclidean space? Traditional environments, while initially comforting, perpetuate the status quo. Exploration of different forms of user interaction is bound to be attenuated



Figure 14: View on racetrack.

if the users are in a recognizable setting, a setting in which behavioral relationships are well established.

Most environments described in other media are too detailed for efficient use in current VRML multiuser systems. One of the social worlds in the Geometrek system is an exception that proves the rule - BlackSun, the bar for cyberspace hackers described in the novel *Snow Crash* [12]. The world is monochrome. (see figure 15) The only design feature is mood created by very low light. The reference to the book telegraphs attitude. The light level concentrates the cpu and visual focus on the avatars. The default VRML headlight for each user is left on, but there is little environmental geometry to reflect it. There is no interaction with the world, other than collision detection, terrain following (gravity), and a gate providing transport back to the HUB or central world.

Visitors to BlackSun can select one of eight avatars or import their own. Allowing importation of custom avatars creates additional motive for participation and increases user awareness of the potential of multiuser spaces. It also requires some sort of regulation to prevent system wide problems due to wide varieties in polygon count, dimensions and behaviors. Still, a combination of off-the-rack avatars and custom avatars is necessary in social worlds.

Anything facilitating smooth communication between



Figure 15: BlackSun world.

avatars will increase the likelihood of movement in avatar design beyond humanoid or anthropomorphic representation and towards metaphorical and/or metadata presentation. Experimentation with such interaction will also be of value in other multiuser areas, such as data visualization, by increasing the available lexicon.

### 8.2.2 Idea focus

In the short term, the most productive use of multiuser technology may be in environments devoted to the exchange and development of ideas. Occupancy will be limited to a handful of predetermined users. Avatars are not requisite. The productivity of the cpu cycles and network events would be increased by the absence of avatars – an option seldom exercised. Scant resources should be focused on behaviors and objects for use in communication and experimentation with ideas. Share the objects and behaviors, make them indistinguishable. The computer game *Pong* illustrates this idea. The paddles are avatars, the ball a shared object. All you must see is your paddle and the effect of both paddles on the ball. Seeing the other paddle is instructive re technique and telegraphs intent, but is not essential. Seeing an avatar manipulate the opposing paddle is destructive if it robs the ball of a smooth animation path.

### 8.2.3 Event focus

Event simulation has the broadest potential for attracting large crowds and major advertising because it can provide many features impossible to implement on-site or via television due to the corporeal nature of the audience.

To fully take advantage of the unique aspects of multiuser events, avatars must be limited to the *actors*. The actors could be representations of people or of the objects they control, i.e. race cars. Insisting on the provision of audience avatars introduces all the real world logistics of event presentation; a series of mundane tasks and associated problems such as seat assignment and sight lines. The denial of audience avatars means users can share the best seat in the house and move freely without interfering with the event or each other. The audience can move *on-stage* in an unobtrusive manner and experience the perspectives, both visual and aural, of the actors. The stage – audience relationship dissolves and everything is truly in the round. There are no curtains, no need for backstage passes (given human desire for information regarding social status it's likely that such needs will be manufactured).

Even though the audience is without location indicators, some mechanism to allow real time feedback is necessary. This feedback could be visual and/or aural and be created by low priority boolean events. Instead of being part of the VRML data, audience feedback could be registered as metadata on a console display, but that would relegate them to strictly numerical data. The audience will want to see the sensual effect of their opinion in the world. Visual feedback can be registered by giving each audience member control over a single pixel. The pixels would reside in an ordered collection, static or animated as a whole. Aural feedback can be manufactured by composite and incremental shifts in a global audience sound loop.

While large scale multiuser events are currently unfeasible, an intermediate step seems well within reach – captured-in-cyberspace events, simulations of simulations created for the single user playback market and distributed via downloads or discs. The content could present a multiuser feel through the inclusion of an invited and highly controlled audience. This approach gains all the advantages of rehearsals, pre/post production, and event capture, while avoiding the logistical problems of multiuser casting. The creation of such media will reveal many of the problems and solutions required to make large scale multiuser applications possible, and will prepare consumers for the ultimate experience.

## 9 Conclusion

In this paper we have presented a virtual environment system completely based on open technologies. Due to its lean implementation the system proves to be more stable, robust, scalable, and interoperable than comperable systems. Furthermore the system is accessible with todays consumer hardware through a standard VRML enhanced Web browser without additional software. We have also shown the virtual environment authoring process and included a detailed discussion on practical experiences with our system.

At the dawn of 3D multimedia technologies such as Java3D, Chromeffects and MPEG 4, there are still a lot of open questions that have to be answered in order to make virtual environments more realistic. In particular, any practical consumer application will have to provide possibilities for geometry streaming. While progressive loading of static 3D mesh geometry is well investigated and will be included in the upcoming MPEG4 standard, streaming of animated geometry or more complex geometry such as Constructive Solid Geometry (CSG) trees has barely been considered yet.

Due to its openness and extensibility, solutions to these and similar problems can be implemented with the current setup of the DeepMatrix system. This way DeepMatrix is capable of serving as a platform for developing future technologies relevant to distributed virtual environment systems.

### A ServerThread class

The following code shows the core of of the ServerThread class. It contains a loop that reads new messages from the input stream of the TCP port, parses these messages and acts accordingly. The variable daemon refers to the main class that holds the tables of clients and rooms.

```
public void run() {
    Message msg=null;

    if(login()==false) {
        close();
        daemon.log("Login failed",1);
        return;
    }
}
```

```
try {
    while(Thread.currentThread()==runner) {
        msg=getMessage();

        if(msg.command==Message.ERROR) {
            daemon.log(msg.toString(),1);
            sendMessage(msg);
            continue;
        }
        else
            daemon.log(msg.toString(),3);
        switch(msg.command) {
            case Message.OBJADD:
                ...
            case Message.OBJDEL:
                ...
            case Message.UPDATE:
                ...
            case Message.MSG:
                ...
            case Message.WHOIS:
                ...
            case Message.ROOM:
                ...
            case Message.LOGOUT:
                ...
        }
    }
}
catch(Exception e) {
    daemon.log("Error "+e,1);
}
finally {
    ... closing down
}
}
```

### B RealTimeServer class

The following code constitutes the main loop of the RealTimeServer class. Again, the variable daemon refers to the main class that holds the tables of clients and rooms.

```
public void run(){
    ServerThread th;
    byte[] data=new byte[MAXSIZE]
    RealTimeMessage rtMessage=
```

```

        new RealTimeMessage();
while(true){
    try {
        packet=new DatagramPacket(data,
            MAXSIZE);
        rtSocket.receive(packet);
        rtMessage.parse(packet);
        th=(ServerThread)daemon.Threads.get(
            new Integer(
                rtMessage.getClientID()));
        if(th==null) {
            daemon.log(
                "RTMsg from unknown "+
                "user with ID "+
                rtMessage.getClientID()+
                " received",1);
            continue;
        }
        th.room.postRTMessage(rtMessage);
    }
    catch(IOException e){
        daemon.log(
            "Error sending RTMsg from "+
            rtMessage.getClientID(),1);
    }
}
}

```

## C RealTimeConsumer interface

The `RealTimeConsumer` interface has the following definition.

```

public interface RealTimeConsumer {
    public String getName();

    public void handleMessage(
        RealTimeMessage msg)
        throws ConsumerException;

    public void delete();
}

```

The method `getName` returns the name of the `RealTimeConsumer` that is used to identify the messages addressed to it. This name has to be unique. `handleMessage` is called with a message received from the server that the object should handle. When the

object is deleted the method `delete` is called, so that the object can free any resources it holds.

## Acknowledgements

The authors would like to thank the following members of the Geometrek VRML development group for their input especially in setting up the test implementation of DeepMatrix: R. Armitage, A. Brown, D. Brown, K. Dabkowski, H. Kaufmann, M. Müller, and N. Olofsson. The Geometrek project is sponsored by Spacecubes, Inc., and Polynomial Solutions, Inc.

## References

- [1] J.W. Barrus, R.C. Waters and D.B. Anderson, *Locales: Supporting Large Multiuser Virtual Environments*, IEEE Computer Graphics and Applications, **16** (1996) 50–57.
- [2] J. Calvin, A. Dickens, B. Gaines, P. Metzger, M. Miller, and D. Owen, *The SIMNET virtual world architecture*, in: Proceedings of the IEEE VRAIS'93 Conference, 1993, 450–455.
- [3] C. Carlsson and O. Hagsand, *DIVE — a multi-user virtual reality system*, in: Proceedings of the IEEE VRAIS'93 Conference, 1993, 394–400.
- [4] R. Hawkes, *A software architecture for modeling and distributing virtual environments*, Ph.D. Thesis, University of Edinburgh, 1996.
- [5] C. Low and L. Babarit, *Distributed 3D Audio Rendering*, Computer Networks and ISDN Systems **30** (1998) 407–415.
- [6] M.R. Macedonia and M.J. Zyda, *A taxonomy for networked virtual environments*, IEEE Multimedia (1997) 48–56.
- [7] R. Oboe and P. Fiorini, *A Design and Control Environment for Internet-Based Telerobotics*, International Journal of Robotics Research **17** (1998) 433–449.

- [8] B. Richards, A.W. Colman, and R.A. Hollingsworth, *The Current and Future Role of the Internet in Patient Education*, International Journal of Medical Informatics **50** (1998) 279–285.
- [9] B. Roehl, J. Couch, C. Reed-Ballreich, T. Rohaly, and G. Brown, *Late Night VRML with Java*, Ziff-Davis, 1997, ISBN: 1562765043.
- [10] D.N. Snowdon, *AVIARY: a model for a general purpose virtual environment*, Ph.D. Thesis, Department of Computer Science, University of Manchester, 1995.
- [11] K. Stefanov, S. Stoyanov, and R. Nikolov, *Design Issues of a Distance Learning Course on Business on the Internet*, Journal of Computer Assisted Learning **14** (1998) 83–90.
- [12] N. Stephenson, *Snow Crash*, Bantam Spectra, 1993, ISBN: 0553562614.
- [13] M. Wray and R. Hawkes, *Disrtibuted Virtual Environments and VRML — an Event-Based Architecture*, Computer Networks and ISDN Systems **30** (1998) 43–51.
- [14] *DeepMatrix Homepage*, <http://deepmatrix.com>
- [15] *Trapezium Homepage*, <http://www.trapezium.com>
- [16] *VRML External Authoring Interface specifications*, 1997, <http://vrml.org/WorkingGroups/vrml-eai/>
- [17] *VRML Humanoid Animation Working Group proposal draft 1.1*, 1998, <http://ece.uwaterloo.ca/~h-anim/>
- [18] *VRML Living Worlds Working Group proposal draft 2*, 1998, <http://vrml.org/WorkingGroups/living-worlds/>
- [19] *VRML Standard Version 2.0, ISO/IEC CD 14772*, 1996, <http://vrml.org/VRML2.0/>