COMPOSING NETWORKED VIRTUAL ENVIRONMENTS

BY

DAVID E. PAPE
B.S., Computer Science, Rensselaer Polytechnic Institute, 1988
M.S., Computer Science, Rensselaer Polytechnic Institute, 1990

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering and Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2001

Chicago, Illinois

ACKNOWLEDGMENTS

I would like to thank all of the countless people who have helped me and made this work possible. Unfortunately, in singling people out for acknowledgment by name, I'm almost bound to forget one, or many, of them, especially after all the years it's taken to get to this point. But, I'll give it a shot, and hope not to offend anyone.

Thanks go to:

- the members of my committees – Tom DeFanti, Dan Sandin, Andy Johnson, Tom Moher, Bob Kenyon, and Steve Jones – for their support.
- Josephine Anstey, Tomoko Imai, and Maria Roussou, for insisting, against my better judgment, on continuing to use 'the MMB code' in new projects, thus leading to this dissertation.
- Carolina Cruz-Neira, Tom, and Dan, for creating the CAVE and then passing on to me the combined blessing and curse of responsibility for some of it.
- Maxine Brown, Dana Plepys, and Maggie Rawlings, for actually keeping EVL running and successful.
- Jim Costigan, Greg Dawe, Gary Lindahl, Alan Verlo, and other support crew over the years, for making everything work in the lab and on the road.
- Horst Hörtner and the folks at Ars Electronica, for supporting many of the projects described here.
- All the students of EVL, present and past, for always making it an exciting place to work.

DEP

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES (continued)

LIST OF ABBREVIATIONS

6DOF	Six Degrees-Of-Freedom
AEC	Ars Electronica Center
AIFF	Audio Interchange File Format
AOI	Area of Interest
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
ATM	Asynchronous Transfer Mode
CAVE	CAVE Automatic Virtual Environment ™
CAVERN	CAVE Automatic Virtual Environment Research Network
COVEN	Collaborative Virtual Environments project
DIS	Distributed Interactive Simulation
DIVE	Distributed Interactive Virtual Environment
DSO	Dynamic Shared Object
EC	European Commission
EVL	Electronic Visualization Laboratory
FHW	Foundation of the Hellenic World
GMD	German National Research Center for Information Technology
GUI	Graphical User Interface
IEEE	Institute of Electrical and Electronics Engineers
IPv4	Internet Protocol, Version 4

LIST OF ABBREVIATIONS (continued)

IPv6	Internet Protocol, Version 6
IRB	Information Resource Broker
NPS	Naval Postgraduate School
NSF	National Science Foundation
NVE	Networked Virtual Environment
SGI	SGI (formerly Silicon Graphics, Incorporated)
SIGGRAPH	ACM Special Interest Group in Computer Graphics
STAR TAP	Science, Technology, And Research Transit Access Point
TCL	Tool Command Language
TCP/IP	Transmission Control Protocol / Internet Protocol
UDP/IP	User Datagram Protocol / Internet Protocol
UIC	University of Illinois at Chicago
URL	Uniform Resource Locator
VE	Virtual Environment
VR	Virtual Reality
VRML	Virtual Reality Modeling Language
VSS	Vanilla Sound Server
WTK	WorldToolKit
XP	[acronym with no meaning]

SUMMARY

This dissertation describes the design of a framework for constructing shared virtual environments. Shared virtual environments are of interest in many different application domains. This particular framework concentrates on applications such as art and education, where the focus is on worlds planned out by an author, with objects or characters that have their own autonomous behaviors and that can interact with users.

The framework, Ygdrasil, is built on two primary elements, which are intended to simplify the creation of parts of these shared environments and to make it possible to quickly compose them from those parts. These elements are a shared scene graph structure, with automatic data sharing and discovery, and a script-like method to define a virtual world. As part of this, a standardized structure for world components is defined. Furthermore, a collection of basic tools is provided to handle many common tasks.

Following the description of Ygdrasil itself, its use in two testbed applications is presented. The performance of the framework in these, and in some more basic tests, is evaluated.

1. INTRODUCTION

In their book *Networked Virtual Environments*, Singhal and Zyda describe composability as one of the chief problems to be solved in creating shared virtual worlds (Singhal and Zyda, 1999). Composability refers to the ability to dynamically bring objects and their behaviors into a virtual world, even when these objects were originally created as part of a completely different virtual world; the objects would be automatically able to interact in the new environment without any coding modifications. This is akin to the goal of general re-usability in software engineering. A fully composable system would speed the creation of significant shared worlds. It would aid the development of very large scale environments distributed in a massively parallel manner, and allow many different environments to be seamlessly interconnected across the Internet, such that users could easily travel from one world to another.

This thesis describes a framework for building networked virtual environments (NVEs). The framework, nicknamed Ygdrasil, takes elements of existing systems for VR programming, but focuses on enabling rapid and easy development of NVEs via a scripting language and a shared scene graph. It allows world creators to re-use existing work and to combine pieces of virtual worlds at will.

1.1. Networked Virtual Environments

A networked virtual environment is a shared, computer-simulated world. In other words, it is first of all a VR world – a real-time simulation of a 3D environment, usually providing a sense of realism through viewer-centered, 3D computer graphics and audio, with which a human user can directly interact. A networked VR world involves multiple client and/or server computers, using the network to share data about the common world that they are simulating. Some of the hosts may simply simulate the VR world, without direct user involvement, while other hosts may provide users with interfaces into this world. An

NVE that includes multiple users should provide those users with a shared sense of place and of presence; that is, the users should all believe that they are in the same virtual world, and they should be aware of each other and able to communicate. Typically, this involves avatar representations of the users, which are a part of the NVE along with simulated virtual objects that make up the shared world.

1.2. Scene Graphs

Most current VR development systems use some form of scene graph representation for their database. At its most basic, a scene graph is simply a hierarchical organization of objects in the world; the hierarchy usually encodes the nesting of 3D transformations, as described by Robinett and Holloway (Robinett and Holloway, 1992). The scene graph is usually either a tree or a directed acyclic graph (DAG); a DAG is valuable when simple re-use (a.k.a. multiple instancing) of models or other elements is desired. In many systems, such as OpenInventor or OpenGL Performer, the internal nodes of a scene graph can also represent things such as turning a subgraph on or off, or level-of-detail selection (Strauss and Carey, 1992; Rohlf and Helman, 1994).

1.3. Scripting

Scripting languages are an alternative to systems programming languages for developing applications. They are considered "higher level" than systems languages, and are typically interpreted rather than compiled. Scripting languages are often used as a sort of glue, to combine powerful tools; shell scripts are an example of this. They have been used in some existing VR development systems, but not to a wide degree – most VR programming is still done in C and C++. Ousterhout states that for many tasks (not specifically VR), programmers are moving from systems programming languages, such as C or Java, to scripting languages, such as Perl or TCL (Ousterhout, 1998). He argues that scripting languages can

provide an order of magnitude improvement in programmer productivity, and that they are the wave of the future for much of software development.

1.4. Framework Features

The important features of the Ygdrasil framework are a scene graph system with an interface defined for plugging together virtual world elements, automated networking underlying all elements of the world database, and a abstracted user representation. The scene graph interface makes it possible for application developers to independently build modules that can later be combined in a virtual world. Ideally, they will be able to create a character and then drop it into a world without the world having been specifically designed for it, or vice-versa. The automated networking makes it possible to run applications distributed over multiple machines, or have multiple networked users in an environment, without application builders having to do any added work. The user representation is necessary in order to make virtual tools and interactive objects more modular and re-usable.

1.5. Applications

Ygdrasil is not intended to be an all-encompassing solution for any possible networked virtual world. Rather, I have targeted it toward a specific type of application, although this type can apply to many different uses of VR, including art, entertainment, education, and cultural heritage. The relevant applications are ones that focus on the behaviors of virtual objects and on interactions between users and these objects; they primarily involve pre-modeled objects and sounds. These applications are also often meant for the general public, or otherwise non-technical users. In many cases, they could also be described as "plotted" applications. By plotted, I mean that they have some sort of story or script. Although an application as a whole might not have a narrative storyline, the virtual world's creator often

has some at least partial plan for how the action in the environment should unfold. At a more micro level, there are pre-defined chains of events that will happen, normally in response to user actions. Ygdrasil is not directed at applications such as visualizing large datasets or steering supercomputer simulations, or applications with complex user interfaces such as 3D menuing systems. Some projects which serve as examples of the sort of applications that the framework is meant to support are the Siemens Mobile Workshop, The Thing Growing, the Round Earth Project, and Virtual Harlem.

The Mobile Workshop is a demo created for the Ars Electronica Futures Lab, to be shown at the Siemens 150th anniversary expo. It was not intended as a technical or educational environment, but rather as a flashy, high-tech demo for the general public; it was to introduce Siemens' new model of mobile (cell phone), as well as to show off their ATM networking hardware. The application ran on an ImmersaDesk at the expo in Berlin, and in the CAVE at the Ars Electronica Center in Linz; the two VR systems were connected over the ATM network. Users would start in a virtual "office of tomorrow" – each user in a different such office. From there, they traveled to a common workshop space. In the workshop, each would see an avatar of the other, remote user, and together they could assemble and customize a new Siemens mobile. The mobile started in four separate parts, which the users assembled with a special "welding" tool. They could also choose among different colors and textures, either for the mobile as a whole or for the individual parts, by dipping the mobile into cubes of the colors and texture maps. Once the new mobile was completed, it was automatically cloned so that each user had one. They then returned to their respective offices, and could communicate through the power of their new phones. This communication was actually done via a live video link over the ATM network, with the video displayed on a large screen in the virtual office. Some of the noteworthy features of this application are that it was

intended for the general public, it featured relatively simple interaction which could nonetheless produce impressive results, and the networked users spent part of the time interacting with shared objects and part of the time with separate, unshared environments.

The Thing Growing is an interactive narrative for VR (Anstey et al., 2000). In it, the user finds himself as the protagonist in a story, and encounters and interacts with a virtual character, the Thing. The interaction in the story consists of navigating through the world, dancing (the Thing observes the user's movements via several CAVE trackers), and picking up and applying objects (a key and a gun). The majority of the development effort in this project was to define the Thing's behavior at different levels. At a lower, physical level, most of the Thing's actions consist of playing back recorded motion-tracking data and sound files. At a higher level, the Thing had to respond to the user's actions in ways that were defined by the general plot of the story. The application is intended to run standalone, by a single user; however, during development we also created a networked version. In the networked version, a remote user stood in for some of the Thing's intelligence, which had not yet been automated. This person would be in the scene invisibly, watching the Thing and the user's avatar, and using a virtual menu, which only he had, to direct parts of the action by sending commands to the Thing.

The Round Earth Project is part of research into using virtual environments for educating young children; it involves the collaboration of computer science, education, and psychology researchers (Johnson et al., 1999a; 1999b). The educational VEs being studied are networked in order to promote learning, by encouraging the students to collaborate and converse about the subject matter. In this specific application, children are meant to acquire the concept of the Earth being spherical. Two approaches are taken to presenting this concept; in one, the students explore a small, spherical asteroid,

distinct from the Earth; in the other, they begin on the Earth's surface, where it may appear flat, and then launch upwards into space to explore the Earth from orbit. In both cases, two students, using two separate VR devices, were involved at a time. One student had the asteroid surface or low-earth orbit view, while the second had a distant, mission-control view of the whole planetary body, with a simple avatar representation of the first student on its surface. The two were required to work together in accomplishing a task of finding 10 objects scattered about the planet's surface. The environments consisted of models of the planet, spaceship, avatar, and other objects, which were constructed in Alias; recorded sound effects and narration from the environment were mixed with live audio of the two students communicating; pre-defined animated sequences were used to introduce the environment and to end the experience. One additional detail, relatively unique to this environment, was the need to create a navigation system that worked on a spherical surface, rather than the more or less flat surface of typical virtual worlds. Future work in connection to the Round Earth Project is expected to involve the rapid creation of other VEs to teach children other concepts; these worlds would ideally be created under the direction of school teachers, and not require large teams of VR experts.

Virtual Harlem is a cultural heritage project by the University of Missouri-Columbia to reconstruct Harlem of the 1920's and 1930's in VR (Carter, 1999). It was developed for an African American Literature course, to allow students to become engaged in an interactive literature course, one where they can see and better understand the environment which produced some of the works they are studying. The original project was written with Paradigm's Vega, for the Virtual Environment Instruction Lab's curved screen and stereo displays. Researchers from UIC are now also contributing to the project, using it in a similar UIC literature course, porting it to the CAVE, and adding a networked component so that an

instructor in Missouri can lead students in Chicago and elsewhere through the reconstruction. In Virtual Harlem, visitors are able to navigate the city streets, examine buildings and people, and hear the sounds of the city. As the environment grows, they will be able to enter some of the important buildings, see film clips or 3D re-enactments of historical performances and events, and possibly interact with historical characters inhabiting the virtual space. Furthermore, the students themselves will be able to contribute to the environment. As part of their course, they will research some of the background of the Harlem Renaissance, and obtain images, texts, and recordings that can then be incorporated into the reconstruction.

A number of common, important features can be found in these applications. The user's interaction consists primarily of navigating around in the environment, directly manipulating objects, and communicating with other users. The direct manipulation takes the form of picking up objects, applying virtual tools, or otherwise activating dynamic objects. The user is also often an important part of the virtual world. In contrast, in applications such as scientific visualization, although networked users may have avatars that appear in the same space as the visualized data, the users normally control what happens in the world, but are not themselves affected by it or otherwise really a part of it. Finally, no matter how many tools a particular framework may provide, there will always be applications which need to extend it, such as the live video in the Mobile Workshop, the use of many sensors in The Thing Growing, or the unusual navigation in the Round Earth Project. Ideally, for a system to be truly useful, every aspect should be extendible.

1.6. Test Cases

I will use cultural heritage applications as a particular test case for Ygdrasil. A recent joint EC/NSF workshop identified cultural heritage as one of the key application domains to drive future research in virtual environments and user-centered computing (Brown et al., 1999a). An important question then is what things do these VEs really need, and how can they make use of the features of the framework. The typical cultural heritage VE currently consists of large, detailed, static models of ancient cities, buildings, or artifacts that users navigate about in and look at, usually with a trained guide to explain everything to the user. With Ygdrasil, I hope to expand this model, and explore other possibilities for the use of VR in historical reconstructions.

Networked environments would allow the models involved to be stored, and more importantly updated, on a host museum's server, and visited remotely, very much like web pages; remote users would use a standard, basic application, equivalent to a web browser, that connects to the server and receives all the 3D model and sound data, as well as behavioral information. Networking of virtual environments usually also implies that many users can share the space. This, however, might not seem so useful in a cultural heritage application – for example, when visiting the Acropolis virtually, the crowds of other tourists there are probably one of the last things that you'd want to reproduce. On the other hand, providing expert human guides via the network would be valuable. An expert guide can remain at a host site, and enter the virtual world to assist remote visitors. Besides the public use, networked environments would allow remotely distributed researchers to meet and examine, discuss, or work on a model.

In addition to networked human guides, we would like to be able to provide automated guides, a.k.a. computer agents. The guides can be implemented as recorded avatars, as in the V-Mail Virtual Trainer

(Imai et al., 1999), or they can be given programmed behaviors. Automated guides could be capable of supporting different languages and different levels of expertise among visitors; part of the information needed to implement this would have to be carried by the user's representation, similar to the idea of visitors using smart cards in real world museums. Pre-recorded and programmed agents could also be used as actors, in addition to tour guides. Current cultural heritage applications typically consist of just static buildings and objects. A more complete historical re-creation should be dynamic – there should be people inhabiting the buildings, and objects should be functional.

Adding interaction is also important. Beyond just exploring a space and looking at buildings or artifacts, visitors should be able to actually try out things. For example, an exhibit of early scientific instruments could allow people to use the instruments and learn how they worked – something that's not likely to be permitted in a physical exhibit with real, several hundred year old artifacts. Besides making it easier for people to understand more complex objects, interaction would engage users more directly in the exhibit. This would hopefully increase their immersion in and enjoyment of it, and lead them to get more out of the experience.

A composable NVE framework such as Ygdrasil can contribute to developing these applications in a few ways. Most of the projects envisioned or underway are large and involve groups of contributors, and the elements they create individually will need to work together in a common environment. Also, the framework is intended not simply to make composing worlds possible, but easy, so that contributors without a computer science background can use it for significant work. A number of common features will be found repeatedly in these different applications, ranging from simple keyframe-animated objects to

intelligent guides. If the framework is successful, it will be possible to create such features once and quickly bring them into any number of shared environments.

2. PREVIOUS WORK

Many NVE research systems have been created over the years. These systems have been designed for a wide range of application domains, from military simulation to 3D graphics education. This chapter will review some of the significant projects and their contributions.

Work in NVE systems has generally focused on implementing different methods of sharing state information between distributed participants. Meanwhile, other researchers have explored alternative, scripting-based approaches to make developing VR applications easier. Ygdrasil draws on both of these areas of work, while deriving its design from a very scene-graph-centric view of application construction.

2.1. SIMNET / NPSNET

One of the earliest systems for networked virtual environments is SIMNET, the US Department of Defense's networked battlefield simulation program (Calvin et al., 1993; Pope). SIMNET was created to improve military training capabilities, by supporting simulation exercises that involve hundreds or (theoretically) thousands of heterogeneous units, which are distributed between many distant sites, but share a common virtual world. The units may be controlled by humans in tank or airplane simulators, or they may be computer-controlled semi-automated forces. Each simulator involved has a full copy of the virtual world database, including information on the location and state of all other objects. The database is maintained by broadcast events – each object will inform all other units whenever its state changes. Broadcast traffic is reduced by using dead reckoning algorithms; the locations of remote objects are continually recalculated from their past locations and velocities; update events only need to be sent when an object determines that the dead reckoned position others have for it is significantly incorrect. DIS, a formalization of the SIMNET protocol for broadcasting object data, has been accepted as an IEEE

standard for distributed simulation (IEEE, 1993). SIMNET and DIS have proven very successful within their application domain. They have been used in large simulations, with up to 300 participating simulators; limited experiments have been run involving up to 5,000 entities.

NPSNET is an implementation of DIS by the Naval Postgraduate School (Macedonia et al., 1994). The NPSNET project has experimented with using multicast networking protocols for object broadcasts. Multicast groups are used to reduce network traffic, in order to further increase the number of units which can take part in a simulation (Macedonia et al, 1995). In this method, the virtual world is divided into geographic cells, and a separate multicast group address is assigned to each cell. A unit subscribes only to the group for the cell that it is currently in, and nearby cells. As a unit moves around in the virtual world, it will unsubscribe from old groups that it is no longer near, and subscribe to new ones corresponding to its new location. This restricts communications to be only between objects that are potentially interested in each other.

2.2. BrickNet

BrickNet is a client/server networked VR toolkit developed by the Institute of Systems Science at the National University of Singapore (Singh et al., 1994; 1995). In a BrickNet application, servers maintain databases of objects; clients request the objects that they're interested in from a server, and deposit their own new, shared objects with a server. Objects can include behaviors, which are written in the interpreted, frame-based language Starship. The behaviors are downloaded along with the objects, and are run on each client. The owner of an object can send a synchronization message to the server, which will cause all the clients to receive the latest state of the object. Unlike in most other toolkits, the BrickNet

database is not necessarily identical on all clients; an application can have private objects, which are not sent to the server to share with others.

2.3. VR-DECK

VR-DECK is a module-based framework for shared virtual worlds from IBM (Codella et al., 1993). It is an object-oriented system built around event and message passing. An application is constructed from a collection of modules. Modules represent objects, as well as trackers, renderers, etc., and are written in a rule-based system derived from C++. They run in a distributed fashion, produce events, and accept events from other modules; the distribution and event routing are automated by the system. Modules are linked together in a graphical editor to form an application; connections define which modules share events. Multi-user applications are built by simply adding several user modules (however, this limits the ability of people to enter and leave a virtual world dynamically, after it has been started).

2.4. DIVE

DIVE, the Distributed Interactive Virtual Environment, is a research platform for multi-user VR, developed by the Swedish Institute of Computer Science (Carlsson and Hagsand, 1993a; 1993b; Benford et al., 1995), which runs on a wide range of platforms. DIVE is intended as an open platform to allow experimentation with many database and user-related abstractions in the design of virtual environments. It has been used as the basis for a number of research projects, such as COVEN (Normand, 1999), which have extended the system for their own particular needs. A DIVE world is a shared database of objects; the database is fully replicated among all participants, and changes are propagated by reliable multicast network messages. Any participant may modify the database; distributed object locks are used for concurrency control. Objects are organized in a transformation hierarchy; they

can have multiple views (e.g. polygons, pixmaps, and text strings), that a rendering process may select among dynamically; they can have simple behaviors attached to them, in the form of finite state machines which are triggered by messages. The behaviors can be written in plain C, or in DIVE/TCL, a superset of the TCL scripting language; any node that has a copy of an object may execute the object's TCL script. Users see and interact with the virtual world through a visualizer, an application that forms an interface to the world database. The interface also includes a model of the user – a "person"– to abstract aspects of the interaction with the database. Finally, DIVE uses the concept of auras to control network traffic and object interactions; an object's aura represents its area of interest; two objects will only need to interact and exchange data if their auras overlap (Hagsand et al., 1997). A hierarchy of Aura Managers monitors all the objects in the database and determines when auras intersect.

2.5. dVS / dVISE

dVS and dVISE are commercial VR development packages created by Division Ltd. (Ghee and Naughton-Green, 1995; Division Ltd., 1995). dVS uses a client/server architecture – the world is stored in a networked database, accessed by client Actors that form the user's interface to the virtual world. Individual Actors provide basic VR services such as rendering the virtual world, playing spatialized audio, tracking the user, or collision detection. The Actors may run in parallel on a single machine, or be distributed over a local area network. Networking among distributed Actors is done using TCP/IP, with direct connections made between every machine involved.

In the shared dVS database, every object is assigned a unique instance number, which clients use to get copies of its data. New data are passed between clients and servers via events, which describe what data in an object has been changed; events can also provide notification of the creation or deletion of

objects. Actors must register interest in specific data to receive events associated with it. The objects in the database are organized in a hierarchy; objects include both visuals and audio, and the database hierarchy can also contain movement constraints, bounding volumes, and collision detection options.

dVISE is a dVS Application Actor – it allows one to create applications without direct programming. It builds and uses a script file that represents the virtual environment. The script file contains a description of the world's object hierarchy, and defines behaviors via events that can be generated or received by objects. The dVISE environment may also be extended for an application by code in C or C++. A dVISE world can be constructed using a GUI, or directly within the virtual environment using a special virtual toolbox. The virtual toolbox is a 3D menu system that the user can carry around in the VE; it includes tools for adding objects and lights, setting movement constraints and parenting of objects, and recording simple keyframe animations of object motions.

2.6. WorldToolKit / WorldUp / World2World

WorldToolKit, WorldUp, and World2World are a suite of commercial VR systems from the Sense8 Corporation (Sense8 Corp., 1998). WorldToolKit is a collection of C functions for driving VR input and output devices and manipulating the world database. The database is a hierarchical scene graph of objects; application code can manipulate the objects, or simple behaviors such as path-following can be used. WorldUp is an object-oriented authoring tool that provides a higher-level framework for applications. A world database can be assembled with a GUI, similar to dVISE. Behaviors are programmed as Visual Basic scripts, which can be attached to objects in the world to continuously update the object. Events, which are changes in an object property's value, can cause scripts to run, or be routed

to other objects' properties. User and Sensor objects provide interfaces for a user to interact with objects.

WorldToolKit and WorldUp in themselves are not networked systems. World2World is a client/server networked object toolkit which can be used with WTK or WorldUp (or other systems) to build distributed environments. World2World servers manage the sharing of object properties. A client joins a world by connecting to a server manager, and then receives data from one or more simulation servers. Properties can be assigned different update rates, to control the amount of network bandwidth used. Portions of the database can be locked to control updates to the scene.

2.7. Alice

Alice is a programming environment for interactive virtual environments, from the Stage 3 Research Group at Carnegie Mellon University (formerly the User Interface Group at the University of Virginia) (UVa User Interface Group, 1995). The objective of Alice is to provide an easy-to-use rapid prototyping environment for 3D applications. It uses Python, an object-oriented scripting language, for programming object behaviors. The scripts are interpreted, and may be modified while the system is running, making it easy to experiment and build worlds piece by piece. Objects are stored in a transformation hierarchy, and standard functions exist for manipulating objects relative to other coordinate systems. The publicly distributed version of Alice only functions as a desktop tool and web browser plug-in. The internal version supports head-mounted displays, and has been used to create multi-user environments, although this has not been well documented in any publications.

2.8. Avango

Avango, formerly known as Avocado, is an object-oriented, shared scene-graph framework developed at the German National Research Center for Information Technology (GMD) (Tramberend, 1999). It was created to provide a transparent method for building networked VR applications, and to allow rapid-prototyping of applications. It is based on IRIS Performer, extending the Performer nodes with field classes to automate access to node data; the field system supports a streaming interface that can be used to save and restore objects as well as to share their data over a multicast network connection. Fields can be connected between nodes in a data-flow graph, where new data in one node's field will be automatically sent to the linked node's field; this programming method is borrowed from Open Inventor, and is also found in the VRML2 format (Strauss and Carey, 1992; ISO, 1997). When running a networked application, nodes are added to the shared scene graph by first creating them locally on a host, and then migrating them to a distribution group, which will cause all hosts sharing that group to automatically create a copy of that node and receive any new data for the node's fields. If a new host joins an already running world, one of other hosts will take responsibility for atomically transferring the complete, current state of the shared scene graph to the joining host; the new host then receives field updates normally. In addition to scene graph nodes, Avango defines sensor classes that handle input devices, such as a wand or 6DOF trackers worn by a user. These sensors, however, are not part of the scene graph, and not shared among networked hosts; they are only used on the local host to affect the shared scene.

The second goal for Avango, besides simplifying development of networked applications, is to be a rapid prototyping system, where developers can quickly create and modify applications. It uses the

interpreted language Scheme for this purpose. A Scheme interface to Avango exists so that any high-level object can be created and manipulated by a Scheme script. Developers create applications by implementing performance critical features in C++ as new nodes, and then creating objects, connecting them, and forming a scene interactively in Scheme.

2.9. Bamboo

Bamboo is a portable system for networked virtual environments, being developed by the Naval Postgraduate School (Watsen and Zyda, 1998). Bamboo provides a plug-in style architecture for building applications. Individual elements are programmed into modules, which are compiled into dynamically loadable libraries. The Bamboo kernel loads modules as they are requested, or based on the dependency requirements of other modules. The modules can be shared over the Internet via HTTP. The use of dynamically loaded modules is intended to promote re-use of code; an application can, in theory, be built by simply bringing together a set of already existing modules. Bamboo also provides a hierarchical, multithreaded callback framework for structuring code execution. New modules can insert themselves into an environment by attaching their callbacks to other, existing callback loops. Bamboo itself does not include any graphical or database features; instead, it is meant to build on such systems as X Windows, OpenGL, and Cosmo3D.

Bamboo is an extremely flexible system, running on a wide range of platforms and languages. On the other hand, its flexibility makes it very complex to learn and to program. As a result, its use for teaching and current development has been dropped (for the time being) even within NPS (Capps et al., 2000).

2.10. Comparison of features

The following are comparisons of some of the features of the above toolkits that specifically relate to the design of Ygdrasil.

Table I summarizes each toolkit's method of organizing an application's database. Early VR toolkits either did not specify any organization, or used a flat database. A flat database is simply a collection of objects (such as tanks and terrain in SIMNET), with no hierarchy or other connections among them. Most modern systems have moved to the use of a scene graph. The core form of this in 3D graphics is a hierarchy of transformations, as in DIVE. Later systems tend to build on Performer or similar toolkits, and thus also include concepts such as switching in their scene graphs. For truly large scale, widely distributed, shared virtual environments, some form of hierarchy is important to organize the world in a useful way. That is, a scene graph hierarchy is invaluable to adding new elements to an existing world by grouping and localizing information; a sub-graph of a scene forms a self-contained entity that can be moved about and reused easily.

TABLE I

DATABASE ORGANIZATION

System	Database
DIS	flat
BrickNet	flat
VR-Deck	linked modules
DIVE	transformation hierarchy
dVS	scene graph
WTK / WorldUp	scene graph
Alice	scene graph
Avango	scene graph
Bamboo	none

Generally, we say that immersive VR requires high performance, in order to maintain high frame rates and quick interaction response time, both of which are vital to the believability of a system. Consequently, most VR work is done using system programming languages such as C and C++. However, as seen in Table II, many toolkits also provide scripting language interfaces; in fact, Alice and WorldUp are solely programmed in scripting languages. Interpreted scripts, which might normally be rejected as too slow for the requirements of VR, are in fact quite suitable for the high-level definition of actions and behaviors in VR. This is because the truly computationally intensive activities, such as rendering or intersection testing, can be implemented in a system programming language, while activities such as changing an object's position in response to a user's button-click are actually fairly lightweight. Alice has further shown that interpreted scripts make rapid prototyping of environments possible, as object behaviors can be quickly

tested and modified in a running application; this has also made it easier for novice VR programmers to create applications. Even for expert programmers, a powerful scripting language can greatly increase their productivity.

TABLE II

PROGRAMMING METHODS

	System programming	Scripting	Graphical
DIS	arbitrary		
BrickNet	?	Starship	
VR-DECK	C++ based		GUI
DIVE	C	DIVE/TCL	
dVS	C		GUI, virtual toolbox
WTK	C		
WorldUp		Visual Basic	GUI
Alice		Python	GUI
Avango	C++	Scheme	
Bamboo	C++, Java		

Finally, Table III summarizes some of the significant ideas from the different systems – specifically those that have influenced or been used in the design of Ygdrasil. DIS has been probably the most successful so far as a composable system – many different simulators from different vendors can be linked up into a single shared battle exercise. In part this is due to the restricted application domain, but it is also due to the simplified protocol; because each entity controls its own data and simply receives information

about other entities in the world through PDUs, the ways that object state can change are well defined and developers have less to worry about when making a client to work with other, unknown clients. VR-DECK demonstrated the use of object-oriented design, and programming interaction in VEs by events and messages. Bamboo focuses on the sharing and re-use of code modules through dynamically loaded plug-ins and the distribution of these plug-ins over the Internet, rather than requiring monolithic, pre-built applications to share an environment.

At its core, Ygdrasil attempts to merge the simplicity of Alice's interpreted scripting with the power of Avango's shared scene graph for networking. It explores new territory by binding these two more closely - the scripting is not a traditional language, but a description of the scene and connections between objects. Also, it borrows Bamboo's plug-in method for dynamic expandability, but defines some common structure for world elements, so they can more easily interact. Finally, rather than taking a 'loose,' globally shared approach to data as in Avango, DIVE, etc., it defines ownership of objects, as in DIS. However, the scene graph data is more general and extendible that DIS's domain-specific protocol.

TABLE III

SIGNIFICANT IDEAS

DIS	Data owned by entity
BrickNet	Private as well as shared objects
VR-DECK	Re-usable modules; event-based programming
Alice	Interpreted, real-time-modifiable scripts
Avango	Shared scene graph
Bamboo	Plug-ins

3. TOOLS

Ygdrasil is built on top of several existing tools. The nature of these tools in some cases directly affects the design of Ygdrasil. In other cases, the tools are used because of the specific features that they contribute. This chapter reviews the various tools that have been used and their salient features.

3.1. Virtual Reality Hardware

Applications created in Ygdrasil may be expected to be run on a wide range of display hardware. However, the primary platforms are CAVEs, ImmersaDesks, and related projection-based VR displays (Cruz-Neira et al., 1993; Czernuszenko et al., 1997).

A standard CAVE is a 3 meter by 3 meter "room" consisting of four large projection screens – three walls and the projected floor. A few CAVEs have five or six screens to more fully surround the users, but these are significantly more expensive to build and hence quite rare. Interleaved, active stereoscopic images are displayed on the screens, and must be viewed with LCD shutter glasses. Sounds are played from loudspeakers around the edges of the CAVE. An electro-magnetic (or, in a few cases, inertial/acoustic) six degree-of-freedom tracking system provides position and orientation data for a single user's head and for a wand. In some cases, multiple wands, or the user's hands, legs, and/or torso may also be tracked. The most common wand, which is referred to as the "EVL wand", has three buttons and a small joystick that can be used in interacting with programs; however, a number of CAVE sites have experimented with using different types of wands, data gloves, or other control devices. Because there are multiple, active-stereo screens, which must all be driven precisely in synch to provide an illusion of a seamless display, a CAVE's graphics are normally generated by a large SGI rack Onyx computer with multiple graphics pipes.

An ImmersaDesk is similar to a CAVE in general, with the primary difference being that it has just a single, smaller screen, sloped at an angle and resembling a drafting table. With only a single display, an ImmersaDesk often uses a deskside Onyx or Octane workstation rather than a rack Onyx.

There are several ways that the CAVE hardware influences the design of applications and of application-building toolkits. Because of the size of a CAVE, users are encouraged to move about when working with virtual objects in an environment; the 10' x 10' area is sufficient for many applications involving small to medium sized objects. However, it is not big enough to explore spaces such as virtual buildings or landscapes. In addition, whereas with a head-mounted display a user can simply turn around to look in any direction, because most CAVEs have no back wall display, there is a whole region of the virtual world that users cannot normally see or reach. As a result, some sort of navigation mechanism is often necessary to move the user large distances or to turn him around; the typical form that this takes is to conceptually move the CAVE through the virtual environment like a vehicle, while the user is still able to move about physically within the confines of the CAVE. The only input that a CAVE application can expect from a user is the 6DOF tracking data and the state of the wand buttons and joystick. Hence, interaction must normally be based entirely on the user's position and on the wand. This includes the controls for the navigation, as well as interactions with virtual objects. The data from the electromagnetic trackers are subject to a great deal of noise and distortion; although techniques exist to reduce these errors, they cannot be entirely eliminated, and they are sometimes still very large (Ghazisaedy et al., 1995; Kindratenko, 1999). Applications that involve direct manipulation of virtual objects must compensate for this, as it can often be difficult for users to precisely locate small objects. Methods that help include visual

or audio feedback that indicates when the wand is touching an object, and increasing the effective size of objects such that the user can grab them without having to make direct contact.

3.2. CAVE Library

The CAVE library (a.k.a. CAVElib) is the core software toolkit for developing applications that use CAVE hardware. It was originally developed to support the particular hardware used in the first CAVE, but has grown to provide a transparent interface to many different systems, including CAVEs, ImmersaDesks, and HMDs, and the different types of components that they may use (Cruz-Neira, 1995; Pape et al., 1999). The major tasks of the CAVElib are to read data from the input devices (trackers and wand or other controller), configure the graphics output, and manage the multiple, parallel processes in an application. It also uses a simple startup file that allows users to configure these features at run-time.

The library and its associated software contain all the device-specific code necessary to handle several different types of input devices, such as the Ascension Flock of Birds, Intersense IS-900, and EVL wand. Most devices are now handled by tracker daemon software, separate programs that communicate with the library via shared memory, allowing new device support to be added without changing the library itself. The data from the devices is stored in generic (i.e. not device-specific) data structures that are then read by application code. The user selects which devices will be used in the configuration file; hence, an application does not need to hard-code any specific tracker or controller choices and can transparently adapt to changing hardware. However, although the exact hardware used does not matter, most CAVE applications tend to expect the default arrangement of two tracked sensors, three buttons, and two-dimensional (X/Y) joystick.

The basic graphics setup is also controlled by options in the configuration file. The user can configure the library to render one or many views of the virtual environment (e.g. the four screens of a CAVE or multiple subsections of a single, tiled wall). The exact physical geometry of the screens for these views and the layout of the corresponding windows on the graphics workstation can be changed, to allow for different system arrangements. The configuration file can also be used for details such as setting up different styles of stereo (active, passive, or anaglyphic), and choosing whether to render a head-tracked view or a view from a fixed location. As with the input, all of these output options are transparent to an application; the application merely has to provide code for drawing the virtual world, and the CAVElib will take care of arranging the necessary windows and applying the correct perspective.

Most CAVE systems are based on multi-processor, multi-pipe graphics workstations. Applications must run multiple, parallel processes to use such workstations optimally. Hence, multi-processing is a core feature of the CAVElib. The library will automatically start a separate process for each graphics pipe that is in use, as well as distinct processes for application computations and for reading the tracking hardware. Separating the graphics from the computation process allows the graphics processes to perform their rendering as fast as possible, which is important to maintaining the high frame rates needed for interactivity. The CAVElib stores its data, such as the tracker positions and wand button states, in shared memory so that it can be accessed from all of the processes; it also provides functions for applications to store their own data in shared memory. The library takes care of synchronizing the processes when necessary; for example, it causes the rendering processes to wait until all of them are finished drawing the current frame before swapping the graphics buffers, so that all of the display screens will change together, maintaining the illusion that the CAVE is a single, seamless display.

In addition to being an interface to the VR hardware, the CAVElib provides a "simulator" feature for desktop development of applications (Pape, 1996). The CAVE simulator replaces the tracking and wand hardware with controls based on the keyboard and mouse. Instead of rendering for an immersive display, it creates a traditional desktop window with options for viewing the virtual world from the user's simulated position or viewing the user in the virtual world from an outside vantage point. These simulated inputs and outputs are treated the same as regular, physical VR devices within the CAVElib, and can be selected similarly in the configuration file. As a result, their use is again transparent to the application. This makes it easier to test and develop applications either at an ordinary workstation or in an actual VR system, and to quickly switch between the two.

3.3. OpenGL Performer

OpenGL Performer (formerly known as IRIS Performer), is a toolkit from SGI for high-performance, real-time 3D graphics (Rohlf and Helman, 1994). It was designed for visual simulation applications, and has become one of the dominant systems for high-end VR development. Performer is divided into two major library layers, called libpr and libpf.

The libpr library is a collection of classes and functions intended to provide a foundation for fast rendering of geometric primitives. The primitives are stored in standardized data structures, the GeoSet for geometry, and the GeoState for drawing state information (e.g. materials and texture images). Libpr uses the OpenGL graphics library, and is designed to efficiently manage the geometry and state. It squeezes as much performance as possible from the graphics pipeline, based in part on the designers' intimate knowledge of the SGI graphics hardware.

The libpf library is built on top of libpr, and provides a scene graph API for building graphics programs, as well as other higher level tools such as automatic multi-processing and 3D model loaders. It contains a collection of node classes that are used to store the world data, including geometry, lights, transformations, and special grouping nodes. The nodes are arranged in a hierarchical scene graph – a directed acyclic graph; the leaf nodes are the geometry and lights that form the visible scene, while the internal nodes provide organization and nested coordinate transformations for the objects.

Because of the widespread use of Performer in current VR application development, and the tools it provides, it was chosen to handle the graphics rendering side of Ygdrasil. It provides a well-known, common starting point for developers to work from. In addition, its scene graph layer forms the basis of Ygdrasil's structure of a virtual world and the approach to programming worlds.

3.4. Vanilla Sound Server

The Vanilla Sound Server (VSS) is a library and server program for playing audio in CAVE environments (Das et al., 1994). VSS is implemented in a client-server model because older high-end SGI workstations did not have their own sound hardware; as a result, the sound and graphics had to be generated by separate machines. CAVE applications run on the graphics machine and send commands to the VSS server on the audio machine, telling it what sort of sounds to play and how to mix them. As low latency is important when sound and images must be closely synchronized, all communication between the client and server is via UDP/IP sockets, rather than TCP/IP. In the case of newer systems that support both sound and graphics on a single machine, this same model and communication protocol is used, although the UDP connection can use Unix's "loopback" network interface, thus avoiding the overhead of going out over an actual network.

VSS provides a number of powerful sound synthesis tools. The most basic way to use it is to play back pre-recorded sound samples (i.e. AIFF files). It can also generate sounds algorithmically, such as by frequency modulation or additive synthesis. The client can start and stop sounds at any time, and can vary each individual sound's amplitude or pitch in real time. Sound "envelopes" can be used for additional precise control of the sounds' playback. Later versions of VSS have also included spatialization features. With this, it is possible to make the a sound seem like it is coming from a particular direction or 3D position.

VSS follows a roughly object-oriented programming approach. Each individual sound that is played is called as a "note" object; the notes are controlled by a set of "actors". The client application controls the sounds by sending messages to the actors and notes. However, the programming interface for the VSS library is a plain C API; this was important because most CAVE applications developed at the time that VSS was designed were written in C.

3.5. Bergen

Bergen is a sound server and library that was created to deal with certain limitations in the VSS software; however, it was also intended to be a much simpler system. Overall, Bergen provides far fewer capabilities than VSS. Its primary use is to play pre-recorded audio sample files, and to control their amplitude. As it happens, this is all that perhaps the large majority of CAVE applications actually use for their sound; hence, Bergen is much simpler to learn and apply in these cases. It follows the same client-server model, with UDP/IP communications, as VSS.

Audio is an important element for fully immersive virtual worlds, and so Ygdrasil must include it at its core, along with support for 3D visuals. Since Performer does not provide any audio features itself, Bergen is used for this side of things.

3.6. CAVERNsoft

CAVERNsoft is a toolkit for building tele-immersive VR applications; the current version is called CAVERNsoft G2 (Leigh et al., 1997; Park et al., 2000). Tele-immersion is defined as the combination of collaborative VR with audio and video conferencing, supercomputer simulations, and massive, remote data-stores, all connected over high-speed, wide area networks. It enables people at distant locations to work together in a common virtual space, particularly on problems in highly compute-intensive areas such as scientific visualization, computational steering, and design engineering. CAVERNsoft's purpose is to enable rapid generation of tele-immersive applications, without the application authors needing to worry about network protocols and architectures.

CAVERNsoft is a C++ library that provides a wide range of tools at different levels of complexity. It includes low-level network classes that form interfaces to TCP, UDP, and multicast socket functions, and other classes for threading and cross-platform data conversion. Built on top of these are middle-level modules for such things as remote transfer of very large files, HTTP communications, and remote procedure calls. Above these are database modules that can be used to emulate a distributed shared memory system.

The CAVERNsoft database module provides a simple two-field database, associating arbitrary chunks of binary data with character string keys[1]. The keys are treated like Unix directory paths, so that a hierarchical arrangement of data is possible. When a client connects to a CAVERNsoft database, it can make asynchronous requests to fetch particular keys' values, and it can store new values for keys. Stored data is automatically reflected to all other clients by the database server. The database client class can also be used without a server, in which case it operates in a standalone mode, making it transparent to the application whether the database is network-shared or not. An additional feature of the database is that data may be stored using either a reliable or an unreliable network connection, under control of the application. This allows one to store state changes, such as a switch being turned on or off, reliably, so that all clients will be sure to receive the change, while storing data that may be a continuous stream, such as avatar positions, unreliably, so that it can be delivered to other clients more quickly.

[1] It's really more of an associative array or dictionary than a full-blown database.

4. XP

4.1. Objectives

Ygdrasil is based in part on the XP system and experiences from that system (Pape et al., 1998). XP is a framework for creating CAVE applications, based on Performer, Bergen, and the CAVE library; it was not designed for networked virtual worlds, although it has been extended for some basic networked uses (The Thing Growing and the Siemens Mobile Workshop applications). It was first developed for the "Multi-MegaBook in the CAVE" project (Fischnaller and Singh, 1997), and later refined for other applications. Ygdrasil later evolved from XP; it addresses some of the problems that were found in XP's design after using it in many applications, as well as making some important revisions to this design that are necessary in order to build networked virtual worlds with multiple users. These issues will be discussed, along with an example of the use of XP, in Chapter 5. The important concepts for Ygdrasil that were introduced in XP are the construction of virtual worlds by assembling a scene graph of behavioral nodes, and the use of a script-like interface to quickly define a scene.

The goal for XP was to provide a system that makes it easy for teams of computer artists and engineers to build large-scale interactive virtual environments. These environments typically consist of pre-modeled worlds containing dynamic, sometimes autonomous objects; users are expected to navigate through these environments and interact with the objects in them. They can involve hundreds of megabytes of models, texture maps, and sound clips; they can cover large virtual spaces, and include multiple scenes. Most of the artists involved are experienced with tools such as Softimage, Alias, and Photoshop, but are not expert computer graphics programmers. The XP framework contains many features common to virtual art environments, allows experienced VR programmers to build tools needed

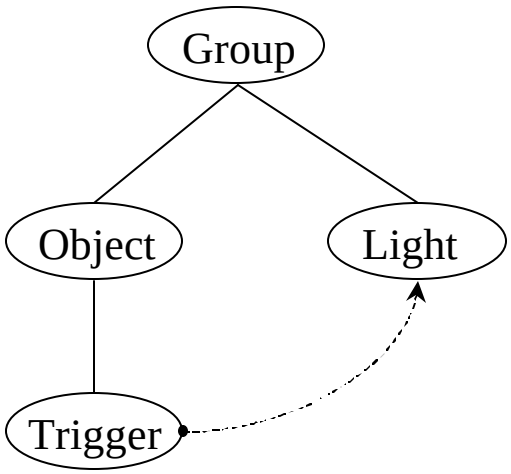
for features unique to a specific application, and allows the artists to create the final environments by assembling the appropriate pieces.

4.2. Design

XP has two major aspects – a text file (scene file) that defines an application as a collection of nodes and their attributes, and a set of lower-level C++ classes that implement the nodes. The division into these two parts makes sharing the work of world-creation between programmers and non-programmers possible. The programmers create application-specific nodes, adding them to the core classes, while other team members build the virtual world itself by plugging together nodes in the text file. With this system, it is also easier to re-use code between applications, because the code is all in modular XP nodes with standardized interfaces.

The scene file is a high-level description of a world's database. This file originated as simply a compact way of representing the Performer scene graph, an alternative to highly repetitive C++ code that would otherwise be used to create all of the nodes that make up a world. Encapsulating the scene creation in a text file made it simpler to add and remove objects, and to rearrange them, without recompiling the program. Basic attributes, such as model files, transformation data, and colors of lights, could be specified with the individual nodes. In developing applications, we had taken to a model of subclassing Performer nodes to encode all behaviors and interaction tools into nodes in the scene graph, rather than having "stand-alone" code that would be called from the program's main loop to manipulate the database. These nodes, and relevant attributes, were therefore also specified in the scene file. In this way, the file evolved from merely a description of objects in a world to what is effectively a scripting language that could describe both the composition of a world and the behaviors in it.

Figure 1 shows a simple example of an XP scene file, containing an object and a trigger that turns a light source on or off. The scene graph hierarchy is defined in a manner similar to that of Inventor or VRML files. User interactions, and behaviors involving multiple nodes, are defined by events and messages. In XP, an event is loosely defined as simply something that happens "within" a node; that is, something that might happen that a particular node type is interested in. The C++ code implementing a node will check during each frame's update whether any events have occurred, and signal when they do. Messages can then be sent from the node to other nodes in response to the event; messages are also loosely defined, being merely text strings that are parsed and reacted to by nodes. In Figure 1's example, a trigger node detects when a wand button-press event has occurred, and sends the message "toggle" to the light source, to turn it on or off. The association of messages with events is done in the scene file, so that the actual C++/Performer implementations of nodes may be left fairly general, and the nodes then adapted to different uses in different applications.




```
group ()
   {
   object (file=LightSwitch.iv)
     {
     wandTrigger (eventMessage=
          "button1,Lamp,toggle")
     }
   light (name=Lamp)
   }
```

Figure 1. Example scene graph and its XP scene file – a trigger turns a light on or off

One added feature of the event/message definition that has proven very useful is delays. For any event/message combination in the scene file, the message can be given a delay; the message will then be sent the given number of seconds after the event occurs, rather than instantaneously. In many applications, especially a narrative, the author will often have a set of actions, among multiple objects, which should occur in a particular, scripted order. In The Thing Growing, for instance, when the user clicks on a key, it animates and opens a box, a sound plays, rocks fly out and land at various places on the plain, and finally the Thing emerges and introduces itself to the user. This is implemented by a single trigger that detects when the user clicks the key; a short sequence of messages then queues up all the succeeding actions with their pre-planned timing.

The standard, core classes that are part of XP include *transform*, *switch*, *object*, *light*, *sound*, and a set of *trigger* classes that respond to user actions.

Transform nodes are used to translate, rotate, and scale the parts of the world that are under them in the scene graph. By default, they are static, but subclasses are often defined to create dynamic transformations, such as playing back key-framed animations.

A *switch* node is used to turn parts of the virtual world on or off at run-time, such as in a transition between scenes.

Object nodes encapsulate 3D object models, which can be in any modeling format supported by Performer; Performer provides database-loaders for a number of common formats, and new custom-built ones can be easily added. *Object* nodes have a number of options, which include being grabbable (i.e. a user can pick up and drop the object), being used for collision detection, or being used for terrain

following. They can also be marked as undrawn, in the case of objects that are solely intended to control the user's movement.

Sound nodes contain audio clips that can be played in response to messages; being a part of the scene graph, they have a 3 dimensional position, and their amplitude can be varied based on the user's distance from the sound source. Many systems that implement 3D sounds, such as VRML, attempt to model such sounds realistically. That is, they define sounds as point sources, whose amplitude decays in a spherical or ellipsoid pattern around the point. In XP, rather than focusing on strict realism, we added features that are useful to artists in creating their environments – a sound can occupy a volume (a sphere or box) within which its amplitude is constant; outside the volume, the amplitude decays normally. This makes it simpler to create such things as a sound that is emitted uniformly by a large object, or a background sound that fills an entire room.

The *trigger* classes detect user actions, and are used for much of the basic interaction in environments. They detect events such as the user entering or exiting a region, the wand entering or exiting a region, a button being pressed while the wand is within a region, or the user pointing at an object.

When a programmer wants to add a new node class to implement special behavior for an application, he does this by extending one of the existing classes. For example, to play back keyframed animation data that may have been created in a traditional 3D animation package, he would create a new sub-class of the standard xpTransform. All XP nodes have a few common virtual functions that implement the scene file parsing, message processing, and per-frame updates. The keyframe animation node could define these functions to parse its new options, such as the name of the data file and duration of the animation, to accept start and stop messages, and to calculate new transformation values each frame

based on its keyframe data. The programmer would then add this new node class to the list of nodes that the main XP parser knows of, and then a user could place instances of the keyframe node in a scene file.

Many nodes also have a debugging state, which is used during development and testing. When debugging is enabled, additional elements are drawn, showing normally invisible aspects of the scene. For example, triggers will draw their bounding volumes, so that the developer can check their size and placement in the scene; their state changes are indicated by changing colors. Events and significant messages are printed to the terminal, so that the flow of the application can be monitored. Figure 2 shows a view of the Multi-MegaBook environment in debugging mode.

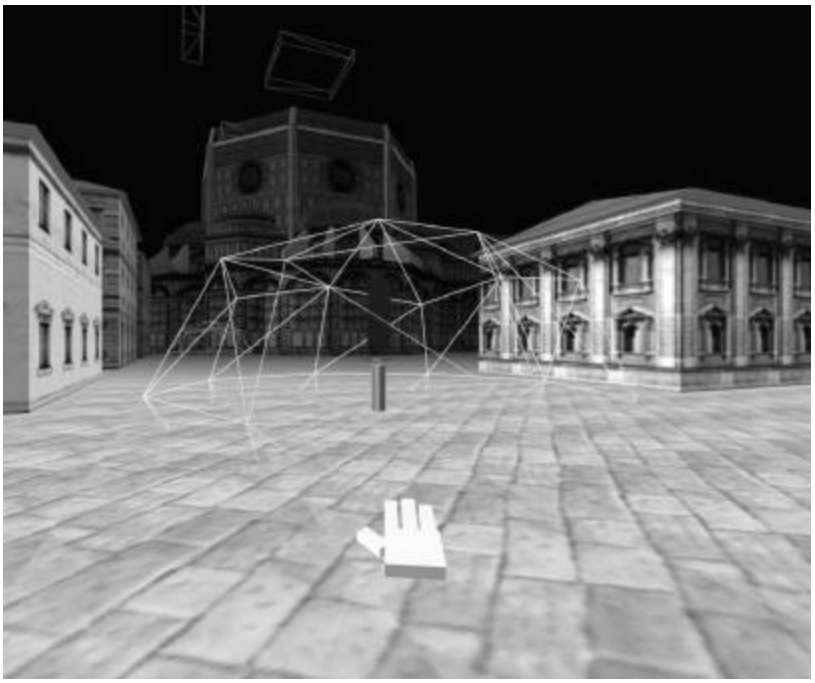


Figure 2. Debugging XP – the wireframe sphere shows the area of a normally invisible trigger

In addition to the main scene graph defined by the scene text file, other major elements of the XP framework include the *navigator*, *world*, and *user* nodes, which are automatically created and added to the scene graph for any environment.

The *navigator* is used to move the CAVE through the virtual world. It provides tools for both user-controlled and application-controlled navigation. Typically, the user travels through the world under his own volition, via the wand – he points the wand in the direction to move, and uses the joystick to control the speed of movement or to turn left or right. In many cases, however, an application needs to take control of the user's movement. Pre-defined features for this include teleporting to a specific location, following a spline path, or attaching the CAVE to an object in the scene (e.g. a boat to carry the user somewhere). The navigator node also provides optional collision-detection, to prevent users from passing through walls, and terrain following, to keep users walking on the ground of the environment. Further application-specific features can be added by sub-classing the standard navigator node class.

The *world* node serves as the root of the scene graph, and provides an interface to some global attributes. It can be used to change the background sky color, enable or disable fog, and vary the clipping plane distances. It also encapsulates all of the text file parsing code, and controls the scene graph traversals.

Basic user information is represented by *user* and *wand* nodes. These are available for C++ code to get access to things such as the user's head position or the state of the wand buttons. They encapsulate calls to CAVE library functions so that, in theory, one could replace these classes by alternate implementations that were for non-CAVE interfaces.

4.3. Implementation Details

The following is a more in depth explanation of how XP is implemented.

A scene file is a plain ASCII text file describing the world as a hierarchical collection of nodes. A Backus-Naur form definition for the format of this file is given in Figure 3.

<scene> ⇒ <tree> **

<tree> ⇒ <singleNode> | (<singleNode> "{" <tree> ** "}")

<singleNode> ⇒ <className> "(" [<optionList>] ")"

<optionList> ⇒ <tag> "=" <value> ["," <optionList>]

<className> and <tag> are any valid names
<value> is any string, possibly in quotation marks

Figure 3. BNF grammar for an XP scene file

The code for the core XP system consists of 25 C++ classes. The majority of these classes are derived from the xpNode class, which is the basic implementation of a scene graph node. The remainder are small utility classes. The node classes are listed in Table IV.

TABLE IV

XP NODE CLASSES

Class Name	Parent Class	Purpose
xpNode		grouping node; ancestor of all other node classes
xpLight	xpNode	light source
xpNavigator	xpNode	allows user to travel through the environment
xpPath	xpNode	defines a path that the user or an object can be moved along
xpPoint	xpNode	an X/Y/Z position; useful for attaching a name to a position
xpTransform	xpNode	a static or dynamic 3D transformation – translation, rotation, and scaling
xpGrabber	xpTransform	a transformation that can be grabbed by the user's wand
xpObject	xpGrabber	loads an object model; can be grabbable or not grabbable; can be used for collision detection and terrain following
xpScript	xpNode	reads a separate file containing collections of messages that are to be sent as a group
xpSelector	xpNode	makes only one of its child nodes active at any time
xpSound	xpNode	plays a single audio sample, occupying a particular volume
xpSoundSource	xpNode	similar to xpSound, but plays any audio sample requested
xpSwitch	xpNode	makes all of its child nodes active (on) or inactive (off)
xpScene	xpSwitch	loads a separate scene file, that can be turned on or off
xpTrigger	xpNode	generic parent of trigger classes; handles defining a trigger's volume and associating messages with events
xpPointAtTrigger	xpTrigger	detects when the user's wand points at objects below the trigger in the scene graph
xpUserTrigger	xpTrigger	detects when the user's head enters or leaves a volume
xpWandTrigger	xpTrigger	detects when the wand enters or leaves a volume, or a button is pressed while inside the volume
xpUser	xpNode	encapsulates the tracker data for the user's head
xpWand	xpNode	encapsulates the tracker and controller data for the wand
xpWorld	xpNode	root of the scene graph; parses the scene file, manages message passing and scene-graph traversal

The classes xpNode and xpWorld form the heart of the whole system. xpNode defines the basic interface of member functions that implement the parsing of nodes in the scene file, run-time traversal updating, and message passing. xpWorld manages all of this, providing the overall parser that reads scene files, keeping track of and traversing the full scene graph, and routing messages.

The basic interface of xpNode, which other node classes inherit and extend to define their special behaviors, is shown in Figure 4.

```
class xpNode
        {
         xpNode(void);
         virtual void parseOption(char * tag, char * value);
         virtual void postInit(void);
         virtual void message(char * msg);
         virtual void app(void);
         virtual void reset(void);
         virtual void switchOff(void);
         }
```

Figure 4. xpNode class interface

The rough execution flow of an XP application, as managed by the main program and the xpWorld class, is shown in Figure 5.

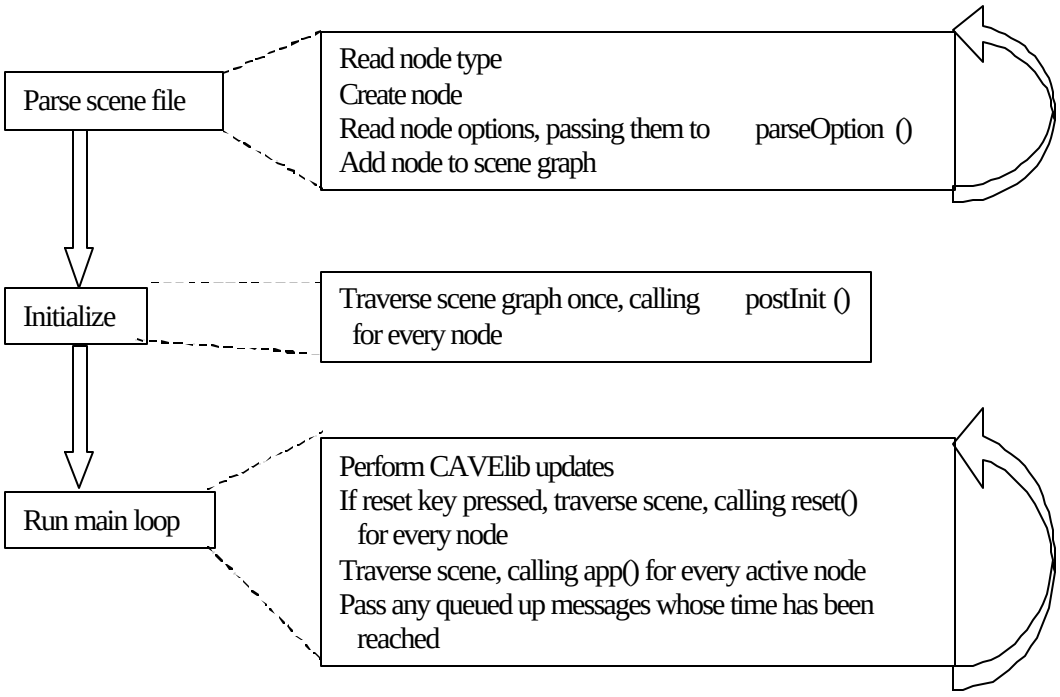


Figure 5. Rough flow chart of XP's main program

For each node in the scene file, the parsing loop looks up the node's class name in a table of known classes, and calls the corresponding constructor function given in the table. Each option given for the node (in the form of strings, the tag and value, e.g. 'name = theTrigger') is passed to the member function parseOption() for interpretation. Once the entire scene graph has been created, it is recursively traversed and the function postInit() is called for each node. This step is required for certain classes that cannot complete their initialization until all of their scene file options have been received, or until other nodes in the scene graph have been created. For example, an xpTransform uses the postInit() to save a copy of its

initial transformation matrix, with any initial translation, rotation, or scaling options; trigger nodes use the postInit() to locate any other nodes in the scene to which they will be sending messages.

While the program executes, each active node's app() function is called once per frame. An inactive node is one that is below (in the scene graph hierarchy) a switch that is turned off or a selector that is selecting some other child node. The app() function performs any simulation or other time-based updates to the node; for example, this is where a trigger will determine if any of its triggering events are true for the current frame. Inter-node communication can be done with the message() function (in practice this is used mostly by triggers when responding to events), which passes a character string to a node; the node parses the string and reacts to it. Nodes do not actually call message() directly; instead, they make a request for the xpWorld to add their message to a global queue. This queue is necessary to handle messages with a time delay, rather than having the individual nodes (which might become inactive before the delay expires) queue their own messages.

The reset() and switchOff() functions are more special-purpose. The reset() restores a node to its initial state; it exists because of the nature of the applications for which XP was developed. CAVE art applications are often run in shows where they need to be started afresh for new groups of visitors. Having a standard, quick reset command that can do this without having to exit and reload the program is useful.

The switchOff() function is called when a node is below a switch that has just been turned off (or below a selector that changes state). It is used by nodes that need to perform special actions when they become inactive. The initial, motivating case of this was xpSound, which must send a message to the

external sound server to stop playing an audio sample, for example when the sound is in a scene that the user is leaving.

In addition to creating the main scene graph based on the input scene file, the xpWorld also automatically creates the user interface nodes, that is, the xpUser, xpWand, and xpNavigator nodes. The xpUser contains the user's head tracking data; it reads the data from the CAVE library, and provides functions for other nodes to obtain this data, either in world coordinates or relative to a particular node's local coordinate system. The xpWand similarly contains the wand's tracking data, and the state of the buttons and joystick. The xpNavigator performs all navigation operations, moving the user based on the wand's joystick state and the direction the wand is pointed, and performing tests against the scene database for collisions and terrain following. It also responds to messages to change the navigation speed or to perform special actions, such as teleporting to a particular location, moving the user along a pre-defined path, or enabling flying (i.e. turning off terrain following).

At run time, the major jobs for the xpWorld are to perform the app() traversal of the scene graph, manage the queue of inter-node messages (described previously), and to control global viewing parameters – the background color, fog state, and near and far clipping planes. The viewing parameters can be changed by any node, by sending a message to the xpWorld. The xpWorld thus hides the interface to the actual Performer and CAVE library functions that control these state elements.

4.4. Networked XP

As previously mentioned, the Mobile Workshop and The Thing Growing managed to extend the XP system to support networked use. We accomplished this by first creating a set of network classes, such as *netTransform*, which extended corresponding core classes, sharing their data via the CAVE library's

simple network communication functions. For example, a netTransform, derived from xpTransform, would check the state of its transformation matrix on each frame. Any time this matrix changed, it would broadcast the new values on the network. Remote instances of this same netTransform would receive the new data and update their matrices with it. All dynamic classes that needed to share data were then derived from these network classes, and the data sharing was effectively transparent. In general, we wanted the intelligence for any particular node to run on only one host, so on that host we would run the XP program with a scene file containing the "intelligent" node, while on all other hosts the scene file contained simply a netTransform that served as a proxy, only receiving updates, not broadcasting new data. User avatars were implemented similarly – an avatar tracker node would broadcast the head and wand data to remote netTransforms, which had the avatar models as children. However, this method was severely limited by the fact that the basic XP scene graph is static; that is, all of the nodes and objects which will be in the world are defined in the scene file that is read at startup. Hence, when running a networked application, one would have to know in advanced that it would be networked and how many users would be involved; special scene files would need to be created for each host, already containing the avatars for the remote users. Nonetheless, this approach was perfectly adequate for the two applications where we used it, as their networking arrangements were fixed in advance by their story design.

5. USE AND EVALUATION OF XP

XP has been used to develop many different CAVE applications. These include the artistic environments "The Multi-MegaBook in the CAVE" (Fischnaller and Singh, 1997), "Mitologies" (Roussos and Bizri, 1998), "Blue Window Pane" (Dolinsky, 1998), and "CAVE" (Kogler and Pomassl, 1999), as well as several industrial demonstrations created at the Ars Electronica Center Futures Lab, such as the "Virtual Heavy Plate Mill"[1] and "Continuous Casting Simulator Environment" (Hörtner et al., 2001). It has also been used at Indiana University in teaching a course on building CAVE environments; students with no previous VR experience created small interactive environments using XP, and showed them at a public, end-of-the-semester exhibition[2].

The following sections describe the application "The Thing Growing" in detail, and explain how XP was used to create it. After that is a discussion of the general problems that have been encountered in XP, as well as specific issues related to adapting its design to building more general, networked applications.

5.1. The Thing Growing

"The Thing Growing" is a virtual reality Interactive Fiction (Anstey et al., 2000). Our goal was to create a story in which the user is the main protagonist in an emotional journey. Our focus was the construction of the "Thing", a virtual character. The user engages at an emotional level with the Thing and its world.

---

[1] Virtual Heavy Plate Mill. http://kultur.aec.at/lab/show_pro.asp?PID=144

[2] CAVE Art Student Exhibition 2000. http://dolinsky.fa.indiana.edu/caveart/index00.html

The impetus for "The Thing Growing" was a short story by Anstey. The story described a relationship that was cloying and claustrophobic but emotionally hard to escape. An immersive, interactive VR environment seemed an ideal medium to recreate the tensions and emotions of such a relationship. Someone reading a book or viewing a film or video may identify with the protagonist but in VR the relationship is more direct – the user is the protagonist.

5.1.1. The Storyline

In the first act of "The Thing Growing", the user finds herself on a large plain. A voice-over prompts her to go to a shed. Inside is a box. If the user opens the box, it bursts open and the Thing leaps out. It dances around and shouts, "I'm free! You freed me! I love you!" The two protagonists, the user and the Thing are introduced and the Thing declares its interest in the user.

In the second act the Thing tells the user that it is going to teach her a special dance; a dance for the two of them. In this act the interaction is designed to be so natural as to be invisible, and involves the user's whole body rather than any interface device. The Thing demonstrates a dance step and asks the user to copy it. The Thing praises or criticizes the dance. If the user gets fed up and navigates away the Thing runs after her and coaxes, whines or threatens her into continuing to dance.

Figure 6. A user dances with the Thing in the CAVE

Act One introduces the user to the environment and familiarizes her with the wand. It also introduces the Thing. Because the user frees the Thing and is loved for doing so, the ideal user also feels warmth and a sense of satisfaction for doing good. However, in Act Two the Thing progressively reveals that it is dominating and controlling. At first it praises the user's dancing, later it begins to nit-pick and complain that the user isn't really trying. The user feels increasingly invaded by the Thing, which is always a little too close for comfort, and grows sick of it. When it finally flies into a temper and runs off, the user is relieved.

However, the relief is short-lived. Once the Thing has gone, rocks on the plain come alive and herd and stalk the user. One of them rears up and traps her. Seconds later the Thing arrives to tell the user that it will get her out from under the rock if she is nice to it. If the user shows a willingness to dance, she is released. The Thing brightly announces that now they can begin the whole dance again. Almost universally users groan when they hear this. However, as an added incentive, the Thing will now copy the user's movements and let her create some of the dance steps.

Act Three begins as lightning crackles across the plain and a god-like voice asks what is happening. A bolt of lightning cracks at the user's feet and the earth opens. She and the Thing fall into a new, darker environment and the user is immediately caged. The two are welcomed by the Thing's four cousins, but the Thing is frightened and whispers that the cousins are fanatics, furious that it and the user have been dancing a sacred dance together. Our goal in this Act is to put the Thing and the user on one side against a common enemy.

Figure 7. The cousins welcome the user and the Thing

The Thing is right to be frightened of the cousins. They beat it up and denounce it for engaging in a relationship with a meat object – the user. They toss the Thing into the cage with the user and exit mouthing dark threats. The Thing produces a gun, and it becomes the user's job to blast them out of prison and then to kill the cousins. The user is usually only too willing to run about and shoot at the

cousins. They are evidently "baddies", besides which it is a moment of agency for the user who, up to now, has merely been trapped and bullied.

Finally all the cousins are killed or have escaped. The Thing and the user are alone again. But now the user has a gun. The entire piece is designed for this moment. The Thing suddenly realizes that the user could turn the gun on it. The question for the user is should she kill the Thing or not?

There are two endings, one for each alternative. However, neither allows the user to ultimately escape the trap of this clinging relationship.

5.1.2. Constructing the Story and the Character

The narrative structure was created with the XP script files. Timed sequences were intercut with the interactive episodes. The narrative flow as a whole was structured using triggers based on time, user proximity, or the completion of specific events. The script file serves as production manager for the story, which can therefore be easily edited and changed.

We extended the basic XP system to build the intelligence of the Thing, the main virtual character, and also to program special behavior for objects such as the rocks that chase the user.

The Thing has a body (motor component) and a brain (cognitive/perceptual component). The body is composed of multi-colored translucent pyramids. Arms, head, body and tail are animated with motion tracking. In this case the pyramids do not connect – the life-like movement that results from the motion tracking creates a strong illusion of an autonomous being formed from a collection of primitive shapes – the illusion is not broken by parts of the body joining badly.

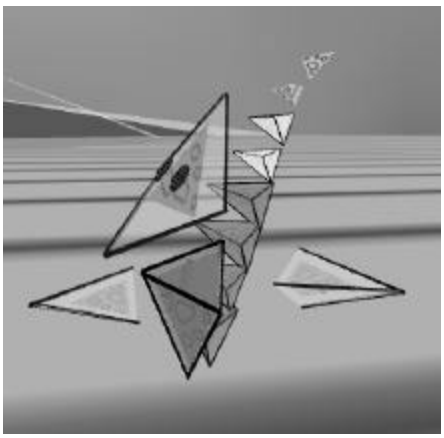


Figure 8. The Thing itself is modeled very simplistically

The Thing's voice is pre-recorded. Based on the storyboard, we recorded hundreds of phrases for its voice. Sometimes its speeches are scripted and do not vary – for example when it is freed from the box. But mostly it speaks in response to the user – for example when it is teaching the user to dance. For this section we recorded different versions of words of praise, encouragement, criticism, explanation. We also recorded the different types of utterance in different moods; happy, manic, sad and angry. Each phrase lasts a few seconds.

Body movements are captured while the phrases play until we build up a library of actions (Action = Movement + Phrase). This motion capture was itself done using a small XP world; the world consisted of a collection of virtual tools that could play attached sounds and record CAVE tracking data into files. In addition to the motion-captured movement for each body part, we also need to determine a movement for the body as a whole. Depending on the circumstances the Thing may move relative to the user or relative to the environment. Therefore each action also contains information about what global body movement

goes with the specific body-part movement and phrase (Action = Body Part Movement + Global Movement + Phrase). All the actions are stored in the scene and can be accessed by the brain. It is very simple to modify, add or remove actions and essentially edit the Thing's behavior.

The brain's main perceptual input is information from the tracking system about the user's body movements and use of the wand. It uses this information in conjunction with information about the state of the world and the passing of time. The main job of the brain is to select an appropriate action from its stores, according to the point in the narrative; the user's actions; and the Thing's internal state. As the program runs, the body interpolates between the end of one action and the beginning of the next, so that the movement between actions is fluid.

In order to be quickly able to respond to changing situations, the brain has several basic strategies. Certain state changes will send it a message to interrupt its current action – the specific state change will also send an additional message to indicate which kind of action should now be picked. Otherwise the brain will complete its action, go through a series of checks on the state of the world and the user, and if none of these trigger alternative actions follow an internal set of rules for selecting the next action.

The narrative becomes a very useful tool for constraining the kind of action the brain can pick, thus simplifying the rule structure. For example when the Thing is attempting to teach the user to dance, it has a basic routine to follow. It demonstrates each part of the dance, then observes or joins the user as she copies the movement. Information on whether the user is dancing correctly is recorded so it can be accessed by the brain's checking system. The Thing may admonish, encourage or praise the user according to her behavior and its own mood. It may repeat a part of the dance that the user is doing incorrectly or it may teach another step. This routine is interrupted if the user tries to run away and

behavior is triggered to make the Thing run after the user and plead with or scold her to continue the dance. Each type of response – "user_danced_well, user_ran_away, new_dance_step" – corresponds to a store filled with possible actions. The brain can pull an action out of the store sequentially – for scripted moments in the story – or randomly, or by mood.

5.1.3. Networked Thing and Autonomous Thing

Our intention had always been to make the Thing entirely autonomous. However, we built the Thing's body and the basic routine to teach the dance before writing the checking system that would use the tracking data from the user to judge how well they were dancing. We were also unsure how to proceed with changing the Thing's moods. Therefore for SIGGRAPH 98, as an interim step, we built a networked version of the project, which effectively gave us a "Wizard of Oz" brain. A networked user was an invisible voyeur on the scene between the Thing and an avatar of the participant. This user had a menu to tell the Thing if the participant was dancing well or not, and also to control its moods (the Thing can be happy, angry, sad, or manic).

In this scenario, although the Thing had its in-built routines of behavior it was also getting help from a much more flexible intelligence system, with a wealth of fine-tuned interactive experience. More importantly, the task of building its intelligence later was greatly simplified by the observations we made during the shows. We observed both the users' reactions to the character, and our own behavior when we played Wizard of Oz.

First, we fell into a fairly standard way of altering the Thing's moods. The dancing interaction lasts for 2-3 minutes. The finale is the Thing running off in a huff. Essentially the mood changes from good to bad over time. If the user is uncooperative – refuses to dance, runs away a lot - the Thing becomes whiny or

angry quicker. Second, users had a fairly standard way of reacting to the Thing. They either tried to obey it, or refused to dance and tried to get way from it. Those that tried to dance, varied widely between people who would copy exactly and those too shy to move very freely – as the Wizard of Oz we tended to treat these alike to encourage the timid.

We built an autonomous dancing Thing based on these observations. Its mood changes based on time and the level of user co-operation. We assume any arm movement that travels more than an arbitrary minimum distance indicates an attempt to dance. We do not bother to check each dance movement separately and precisely to make sure that the user is doing a specific move. Over time the Thing becomes randomly pickier, criticizing and rejecting movements that earlier it would have praised. In response the users watch more carefully and refine their dancing. The completely autonomous dancing Thing has run successfully at the Virtuality and Interactivity show in Florence, May 1999, at SIGGRAPH 99 and at the Ars Electronica Festival in Linz, September 1999.

5.1.4. Implementing the Thing in XP

Table V provides a summary of the size of The Thing Growing, in terms of the amount of code (both C++ and scene files) written, and the models and other data created. Figure 9 and Figure 10 show portions of the scene graph; Figure 9 is the first scene of the story (on the plain); Figure 10 is the character of the Thing itself – both its body and its intelligence nodes.

As can be seen, The Thing Growing is a fairly complex virtual environment. That the large majority of it was implemented by a VR developer who had very little programming experience prior to beginning the project is a testament to XP's ability to manage this complexity. The construction of the Thing's

intelligence sub-graph also demonstrates that it is possible to build advanced behaviors by assembling modular scene nodes.

Furthermore, the summary in Table V gives an idea of the scale of applications that should be expected to be implemented in Ygdrasil. Future applications may grow even larger. These worlds will have large numbers of models, images, and sounds that need to be loaded by clients. Their scene graphs will include hundreds of nodes; these nodes will have to be 'discovered' and replicated locally by each client for a user to join, view, and interact with a world that is already running. The clients will also need to receive updates from possibly many dynamic nodes; some of these nodes will be changing state continuously (e.g. the Thing's body), while others will only change very infrequently.

TABLE V

CONTENTS OF THE THING GROWING

Scene graph	1435 nodes 11 scene files 3588 lines in scene files	
Dynamic nodes	138 transformations 59 switches 10 geometry interpolators 93 sound sources 68 triggers	
Code	38 new classes 11,761 lines of code	
Data	308 models 37 texture images 658 sound files 354 motion capture files 156 motion paths	2.8 Mbytes 3.0 Mbytes 208.0 Mbytes 13.8 Mbytes 0.6 Mbytes total: 228.2 Mbytes

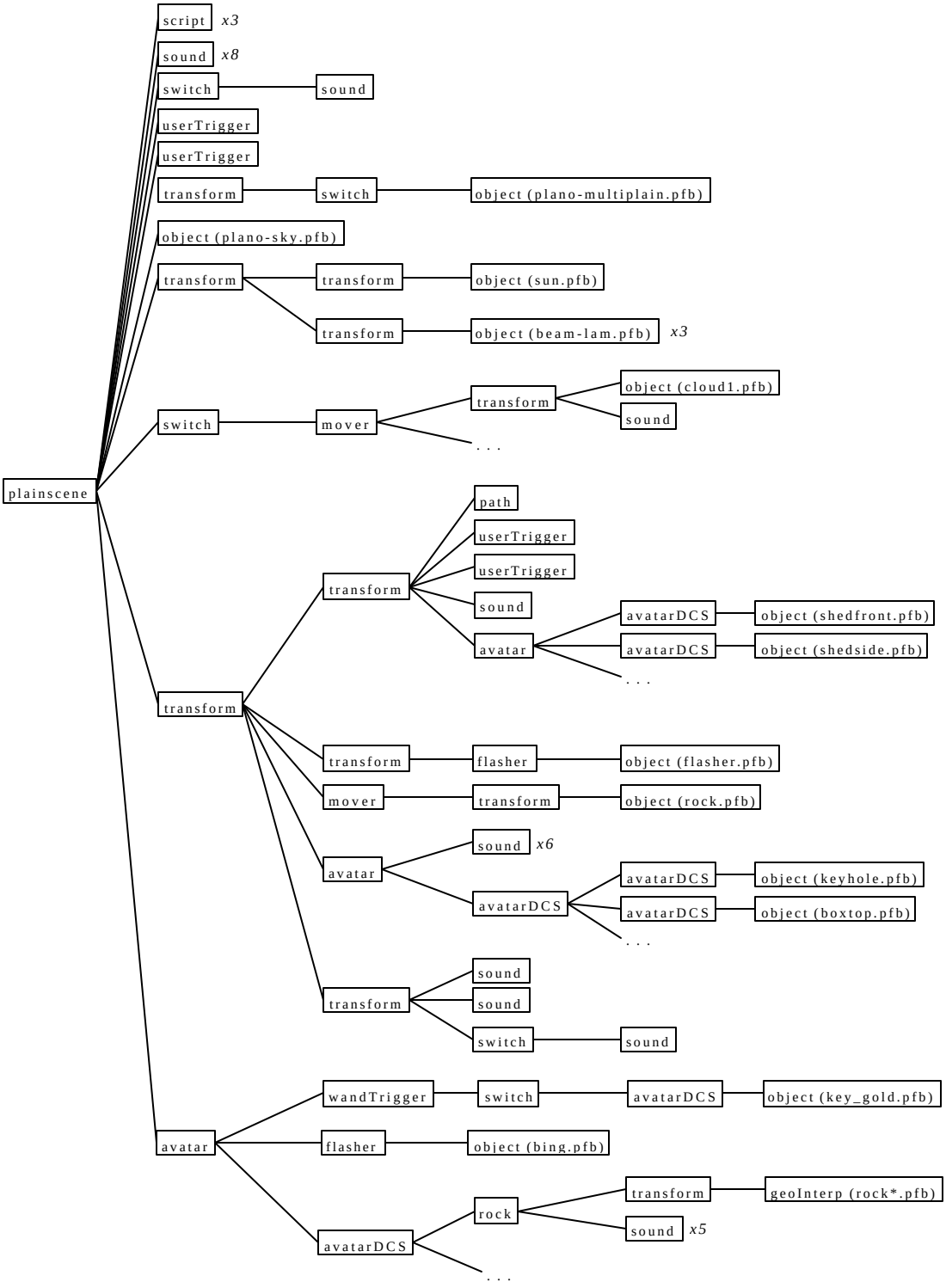


Figure 9. The Thing Growing – scene graph structure of the “on the plain” segment

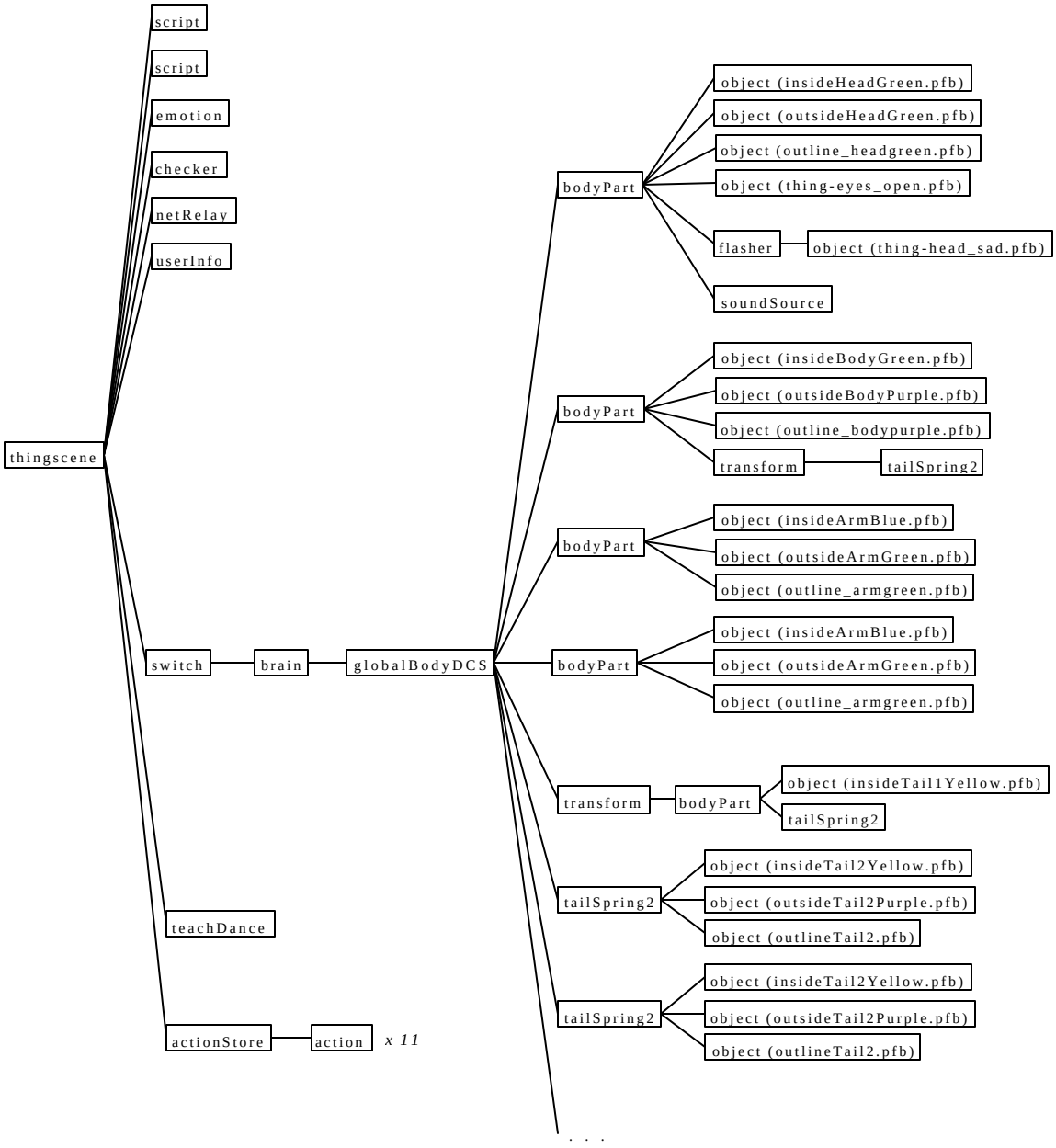


Figure 10. The Thing Growing – scene graph structure of the character of the Thing

5.2. Discussion of Design Problems

Despite the successes of the applications built with XP, there are a number of problems with its design. Some of these problems are limitations that affect any use of XP; others arise when attempting to adapt something that was created for single-user virtual worlds to building shared virtual worlds.

XP is extended by defining new node classes. However, doing this requires recompiling the main XP program to link in the new class, and also modifying part of the main program's code (the xpWorld) to add the new class to the table of known types. Thus, although it is possible to share node classes between applications, it is non-trivial. A dynamic, plug-in architecture, similar to that used by Bamboo, would improve this immensely. The main program's executable would no longer need to change, and code could be re-used by simply copying a plug-in file.

As is true for the underlying CAVE library, and most other non-networked VR development systems, XP assumes that there is only one user. It provides pointers to objects representing a single head, wand, and navigator; these objects are effectively fixed global variables, in that user interaction code always knows that they exist, and the objects are only created at program startup and deleted at program exit. In a networked application, there can be any arbitrary number of users, and the users may come and go over time. Therefore, nodes that a user can interact with will have to be able to find all of the current users whose data must be checked. The local system ought to maintain a dynamic list of users, rather than requiring each node to search the entire world database for users on every update. It might also be useful if this list could be filtered, so that only those which are likely to be important for a particular node are passed to it – i.e. an aura approach, similar to DIVE, could be used to reduce the amount of calculations.

When linking messages to events in a scene file, the single-user assumption also has an effect. A common use of triggers that detect user actions is to send some command to the user in response to his action; for example, a trigger may be set up to teleport the user somewhere else when he enters a specific area. In a multi-user system, the world-builder will need some way to know who generated an event, in order to send a command specifically to that user. One possibility is to add parameters to events; these parameters could then be used in the arguments of messages connected to that event, e.g. "$user.teleport(0 0 0)" could tell the user who set off a trigger to teleport back to the origin.

XP also assumes a specific user model, corresponding to the interface common to most CAVEs and ImmersaDesks. That is, the information available about a user consists of the position and orientation of a head and a single wand, and the state of three buttons and joystick from an EVL wand. A general purpose distributed system should be able to deal with many different types of user input interfaces; i.e. a user should be able to have more than two sensors, and different numbers and types of controls, such as data gloves or different types of wands. Ideally, the system should also be capable of dealing with other control modes, such as speech recognition or keyboard input. Existing XP input nodes are partially tied to the hardware implementation; *wandTrigger* nodes trigger events based on the state of buttons 1, 2, and 3; *grabber* nodes specifically check the wand's second button to grab and release objects. A better, generalized interface could have the user-data nodes generating simple but abstracted commands such as "grab" or "activate" (or even just "command1", "command2", etc.); a user would then be allowed to make arbitrary control mappings to indicate what action (button press, voice command, etc.) will generate a "grab" or other command. Grabbers, triggers, and so forth will then watch for generalized events from user nodes, rather than looking at their underlying state.

Although the individual contents of an XP scene may be very dynamic, the actual structure of the scene in general is static. An XP program reads its scene file or files at startup and creates the scene graph from them; after that, there is no provision for modifying the world. It is possible at the C++ level for existing nodes to create additional nodes, but there is no way to create or remove nodes at the scripting level, or to modify many attributes such as event/message links. The V-Mail application, which was based on XP, needed to dynamically modify its scene in order to add new message objects as they were created. It accomplished this by the awkward method of writing out a new scene file, distributing it to all hosts via CAVERNsoft, and then deleting its existing scene graph and reloading the scene file (Imai et al., 1999). Obviously, a more elegant solution is desirable. In any general, networked application, a dynamic scene graph is called for, at the very least to allow creating and removing avatars as users enter and leave. Beyond that, to meet the goal of a composable system where large environments may be implemented by being distributed over many hosts, the system must be able to handle new portions of the scene graph being added at any time, from potentially any host on the network.

6. YGDRASIL

The system that has resulted from expanding XP to support shared VEs is Ygdrasil The name Ygdrasil comes from Norse mythology; it is the "World Tree", a gigantic ash tree that symbolized the universe (Sturluson, c1220).

6.1. Design

The primary features of Ygdrasil are: a distributed scene graph, a user model, scripting, and dynamic loading of code.

These features are intended to yield a composable system, one where VE creators can assemble a world out of arbitrary existing components and bring new objects into a running world. For components to be able to "link up" and communicate with each other, a clearly defined structure for the data that is shared is necessary. The scene graph approach provides a basic structure, in the form of data associated with graph nodes, and so it the sharing automated as much as possible. A defined user model allows the creation of reusable interactive objects, as the objects will know how to communicate with any user. Keeping this model flexible and somewhat abstract will still allow the system to adapt to the many different VR devices with which it might be used. Given a set of object components which know how to communicate with each other and with users, a scripting layer provides the glue that makes it possible for authors to assemble them quickly and easily. Finally, Ygdrasil also includes support for dynamically loaded code, so that virtual world components (i.e. node classes) can be easily shared and re-used by world authors.

Figure 11. The World-Tree Ygdrasil

Figure 12 illustrates how Ygdrasil is built up on top of existing tools. Ygdrasil is divided into a core system, which is the main executable program, and a collection of modules. The modules include standard classes that are provided along with the core program, and new ones that are added by virtual world developers. The core system uses Performer to store the visual database and render it, Bergen to generate audio, and CAVERNsoft to handle networking. The add-on modules are then built on the core; in some cases they may make direct use of Performer, OpenGL, or Unix features. Modules that provide the CAVE-based interface (e.g. trackers for an avatar's parts) also use the CAVE library.

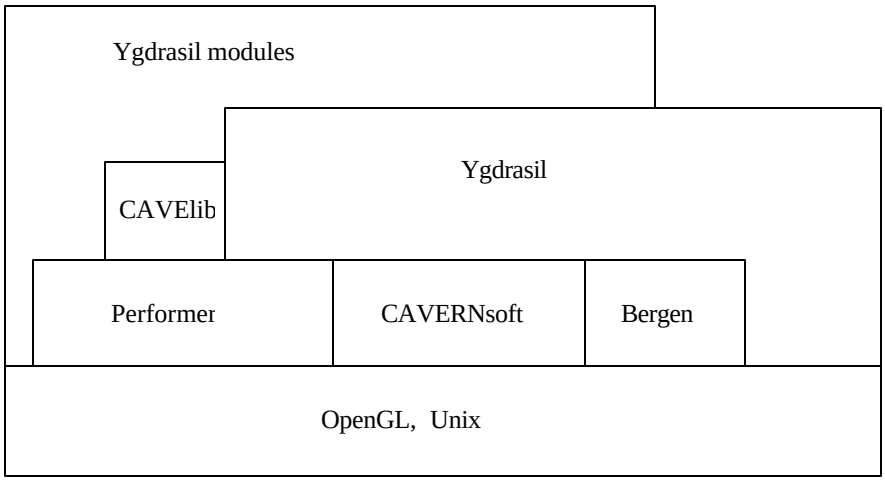


Figure 12. Ygdrasil software layers

6.2. Distributed Scene Graph

The Ygdrasil system uses a scene graph structure for its world database. However, it is a distributed scene graph, which does not require a central server for storage[1]. In most cases, no single machine will have a complete copy of the true "master" scene. Conceptually, different subgraphs of the full scene can exist on different machines, and be linked over the network. Any particular machine will only have the parts of the scene graph that it controls, and proxies for any other parts that it needs for its calculations, rendering, or whatever that machine is doing.

Figure 13 shows an example scene graph for a world that contains two distinct scenes, and that has three users in it. This is the theoretical global scene graph, containing all the objects of the virtual world. Figure 14 shows how the scene graph might actually be broken up among multiple hosts. Each box represents a single host's subset of the entire scene, that is, the portion of the total world that is owned and updated by a particular host. Figures 15 and 16 show the portions of the scene graph of which different hosts might actually have copies. These consist of the subgraph for which the host is responsible, plus the other fragments that it requires (bold type indicates nodes owned by that host, italics indicate proxies that receive data from another host). The data in these additional scene fragments are received from their controlling hosts via the automated networking feature that will be described later.

[1] In the current implementation, a UDP reflector that all clients talk to is treated somewhat like a server, but it does not actually store any data itself.

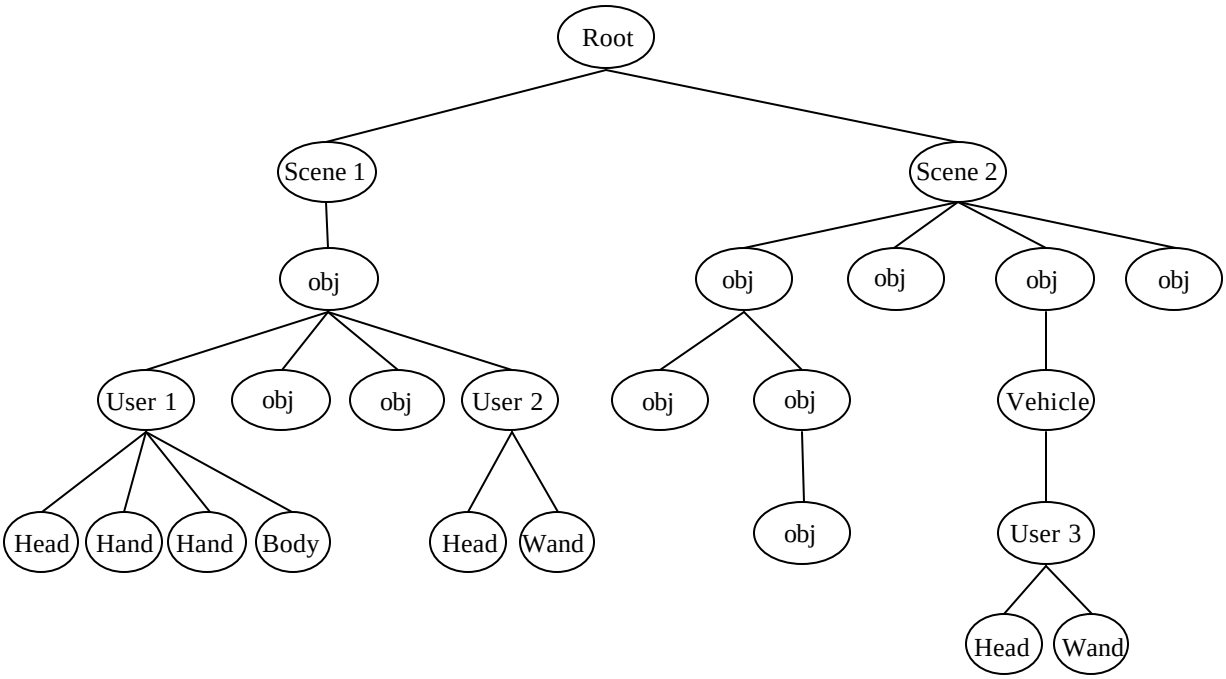


Figure 13. Example of a global scene graph

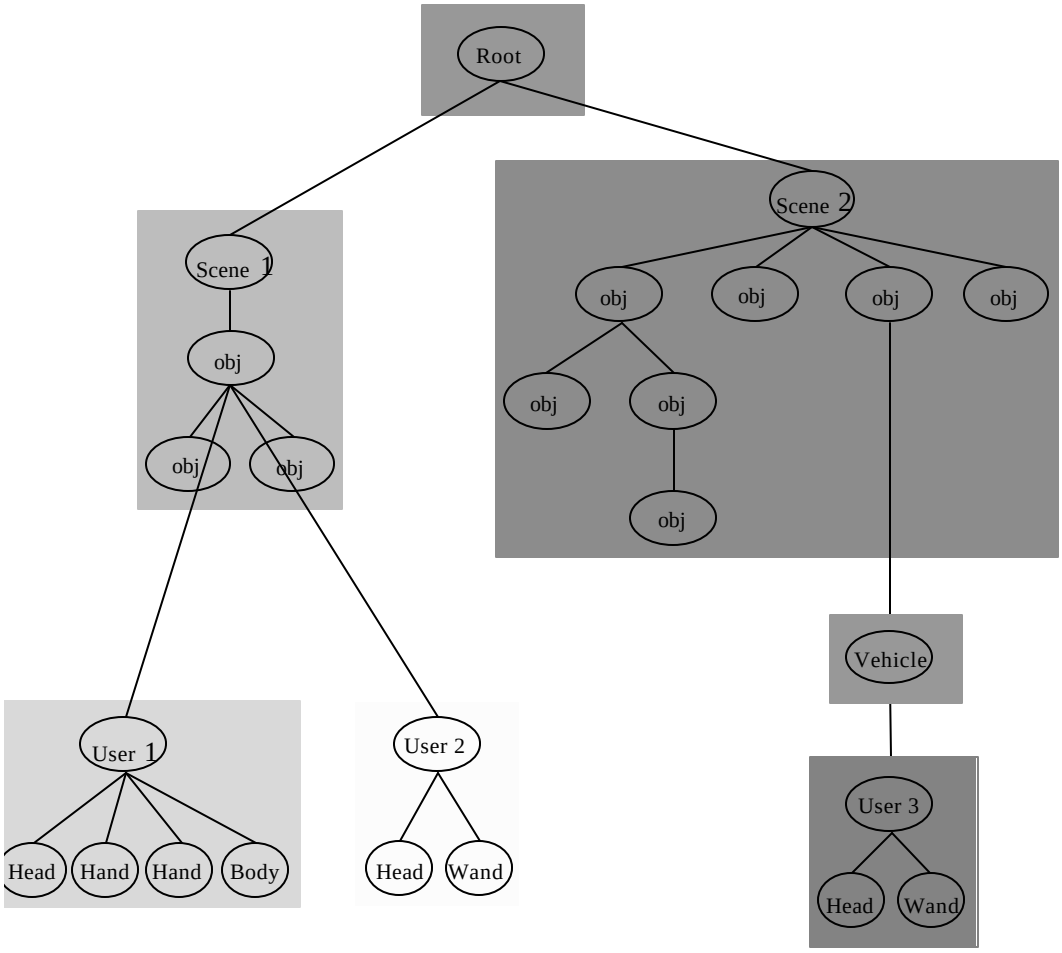


Figure 14. The global scene graph can be broken up and distributed among several computers

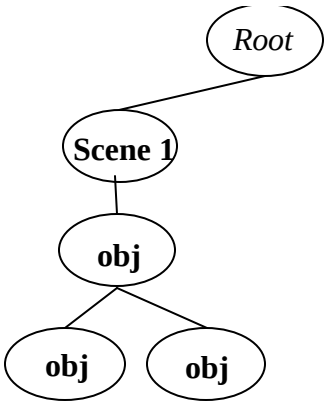


Figure 15. Scene 1's scene graph

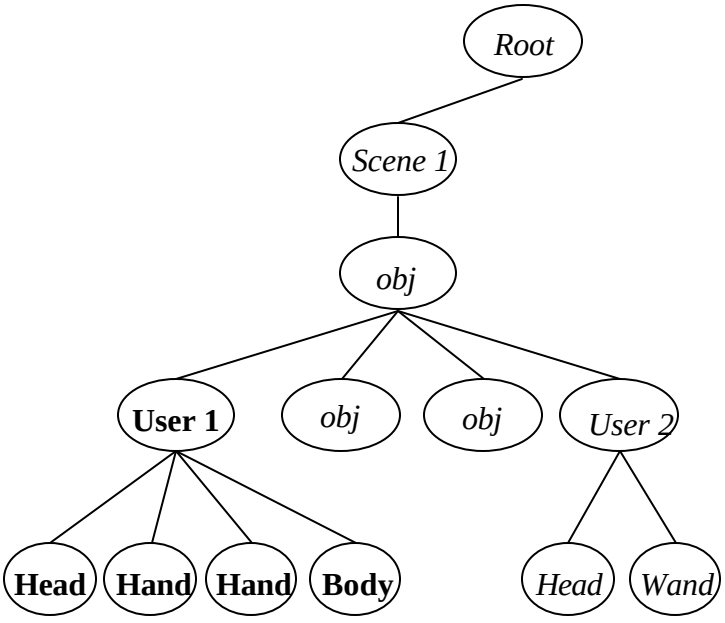


Figure 16. User 1's scene graph

Explicitly including the scene graph structure in the definition of the framework can also be useful for area of interest (AOI) management. Many networked VR systems, such as NPSNET and DIVE, use AOI to manage both the network and rendering load for applications. In NPSNET, the world is broken up into a grid of cells, and users only need to pay attention to other entities that are in the same or nearby cells. DIVE defines auras – areas around users that cover the region that they can see or interact with; a process continuously checks for overlap between the auras of the user and other entities, and then only receives data for those that do overlap. In Ygdrasil, these and other AOI management schemes can be built on top of the scene graph. A cell-based system can be created by defining a top layer of scene nodes that represent the different cells; an aura-based system can simply define aura volumes for each node and test them, similar to visible bounding volumes in the rendering stage. However, given the existence of the scene graph, either of these systems may be implemented in a more hierarchical manner for improved performance and flexibility.

6.2.1. Node Structure and Automated Networking

A scene graph is built of nodes; each node contains data representing one particular element of the scene. In Performer, a single node can contain geometry data, a light source, a transformation, or one of various types of switches, or it can simply be a grouping node. This sort of data is sufficient for the visual aspects of the application, which Performer deals with. An application's audio can similarly be represented by nodes containing basic information such as the name of the sound file to play and the current state (stopped, playing, or paused) of the sound. In Ygdrasil, the overall system deals with behavioral programming as well as the visual and audio representations of the scene. All data required for

these aspects will need to be stored in a node, but they are divided into two parts – private data, which is used strictly by the behavioral functions, and shared data, which is available to other clients for such things as rendering the scene.

As outlined above in the description of the shared scene graph, for any given node there is a host that is considered to own that node; this is the host that performs calculations to update the node (in the case of a user avatar, for example, it reads the tracker information). Other hosts that are interested in the node will also have copies of it. However, only the owning host is able to change the node's data; all others have proxy copies, and are in effect only able to read its data. Hence, a remote host's proxy for a node should only contain the data that is needed to render it or otherwise use the node in calculations performed by the remote host. In a purely visual application, this would be the sort of data that Performer nodes contain. Data that is used internally by the code that controls a node is not shared, and only exists on the owning host. The programming style thus follows an object-oriented model of separating the public interface of an object from its implementation; if another node wishes to change the state of a node, it does not change the data directly, but sends a message to the node, specifically to the master copy of the node on the owning host. This distinction also means that proxy nodes can actually be of a simpler class than the master copy. For example, a "Spinner" node class can be defined, derived from the basic transformation class, with the behavior that it continuously spins around a given axis. Remote hosts that create a proxy for this node can simply use the base class (transformation node) for their proxies, since all that the proxy has to do is receive updates to the transformation matrix. Thus, clients will not need to have copies of or know anything about the behavior code being run by the master version of a world; they will only need the core program, and modules for any new nodes that they will add to the world.

The distribution of nodes' shared data is done using a CAVERNsoft networked database. Each specific piece of data (attribute) in a node is shared separately; each one has its own database key. Figure 17 shows an example of a transformation node's matrix being shared among multiple hosts. Host 1 owns node “xform”, so it initially creates the node, and adds a key to its CAVERNsoft database for the matrix; any time that the matrix data in xform is changed, the new matrix will be written to the database. Host 2 decides that it is interested in xform, and so creates a copy (xform’), which will then reference the database entry for the matrix key from host 1; whenever new data is received by the key, it will be copied to xform'. Other hosts will similarly connect to the matrix key when they need to know about xform. As shown in Figure 18, every node attribute has a separate key; different attributes could be shared in different manners – an array of children pointers could use a reliable connection, while frequently updated matrix data could use an unreliable connection.

When a new client wishes to join and see a shared world, it can get a copy of the complete world scene graph by being given a reference to the root node (i.e. a network address for the database, and the name of the root node). The client will create a local copy of the root node and begin receiving the root's data keys; this will give it a list of the root's children nodes; following these node references recursively will eventually produce a copy of the entire scene.

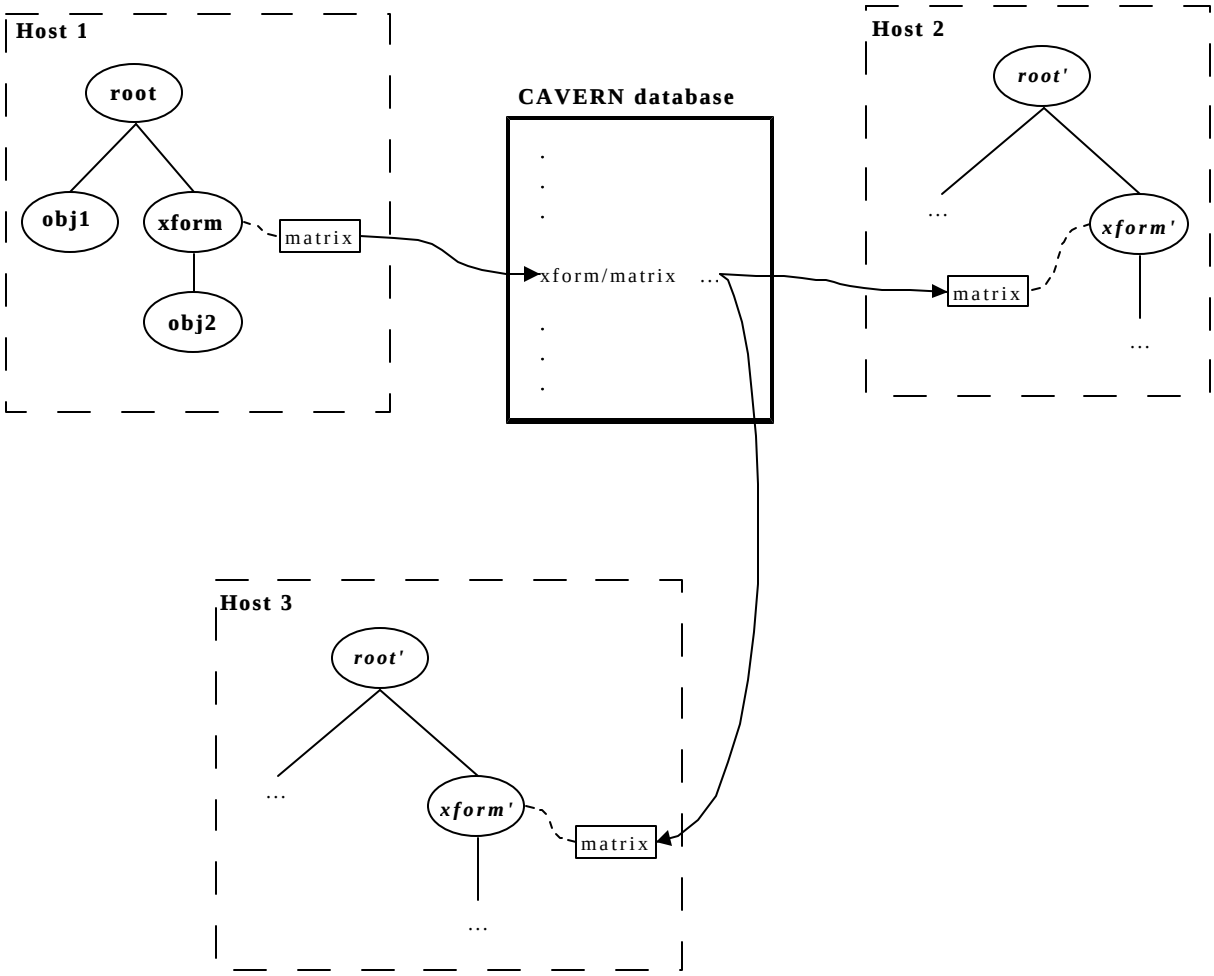


Figure 17. Sharing node data by storing keys in and retrieving them from a CAVERNsoft database

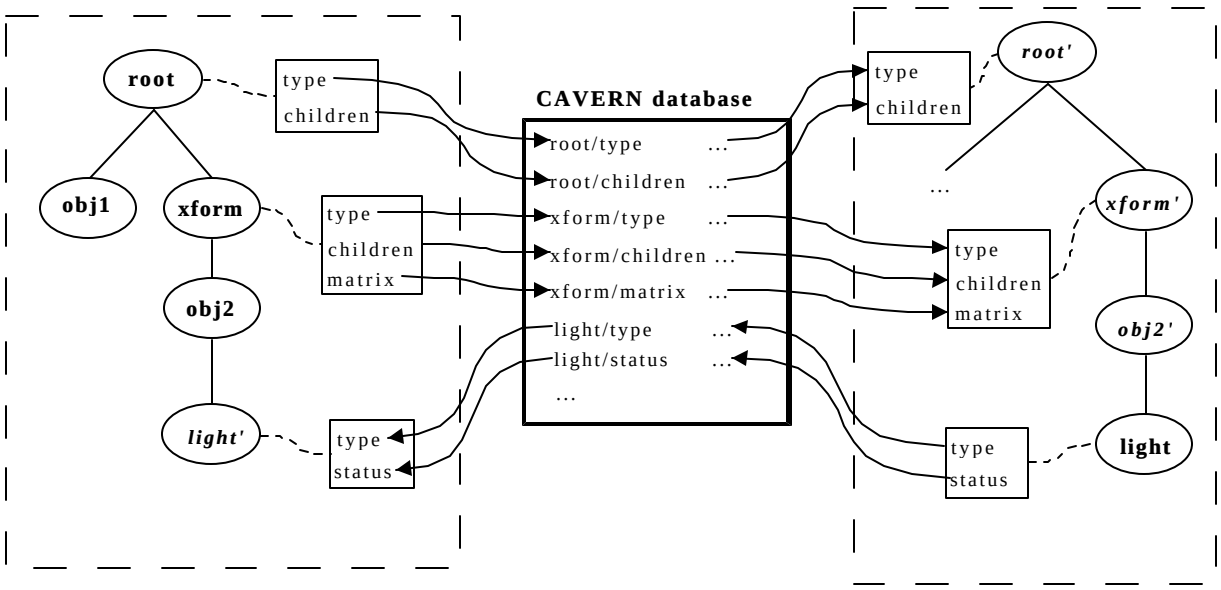


Figure 18. Every attribute of every scene graph node is stored as a separate key

When a new client wishes to add an object to a shared world, it will have to be given a reference to whatever existing node under which the object should be added. The client will create keys for all the data attributes that will be shared, and then will send a message to the parent node telling it to add the new node as a child. The parent node will update its children attribute, and so any other clients that have copies of the parent will see the reference to the new object. Any clients that are interested in the new object will subsequently create a local proxy and request its attribute keys as needed.

Database consistency is often an important issue in networked virtual environments. System designers include specific features to guarantee that two remote users do not end up with different, conflicting data. For instance, Avango uses process groups and total message ordering to make sure everyone receives the

same sequence of state updates. However, database inconsistencies are primarily a problem in systems that follow a distributed shared memory model and allow any client to directly modify any data in the world. In DIS, the only action that is required to guarantee consistency is for an entity's host to send regular updates; no one else is allowed to change the state of the entity. Ygdrasil similarly avoids such problems by having objects owned by the host that creates them. For example, if two users try to grab an object simultaneously, they will not get inconsistent results, as the object itself will decide who grabbed it. However, actions that depend on the state of a remote object can encounter problems; performing collision detection calculations against a moving object might result in a user passing through the object because the local navigator didn't get the most recent data in time. In general, an Ygdrasil world will maintain consistency over the long run, in that everyone will see the same world; the only concern is that there might be logical errors in this world – things happening which should not have happened. In designing XP, we said that we expect things to go wrong occasionally, and so tried to provide tools to get out of likely errors – collision detection can be turned off, the user can be instantly sent back to the starting point, and if all else fails the entire world can be reset (without having to exit and re-run). Similar considerations should probably be taken in Ygdrasil applications, although this may not be as simple. Because XP applications are not networked, they are normally shown in an environment where an expert guide either leads users through the application or is available to step in and help when the user runs into trouble. Thus, "secret handshakes" (special wand button combinations) and keyboard controls could be used for getting out of problems. In the kinds of networked applications that Ygdrasil will hopefully be used for, an expert guide will not necessarily be available locally to notice when the user has a problem and solve it. At least two approaches to resolving this issue are possible. One is to make it possible for a

remotely located expert guide to control another user, for example picking them up and dropping them back in a safe location. Another is to make it easier to provide virtual tools that users carry with them, that can perform these special reset operations in a more "user-friendly" way. Both of these approaches depend on a general, flexible model of the user as an explicit part of the virtual world, as described next.

6.3. User Model

Because user interaction is an essential component of the target applications, a standardized user model is necessary. Furthermore, as described in Chapter 1, users need to be treated as part of the world, rather than distinct from it. Therefore, an Ygdrasil user is represented as a scene subgraph, as in Figure 19. For the physical aspect – the avatar – the "user" node contains the user's navigated position in the world; the "head tracker", "wand tracker", and "body tracker" nodes contain transformations for individual tracked body parts; the model nodes below them contain the geometric models that make up the user's avatar.

The precise contents of a user graph are flexible, to meet the goal of supporting many different interfaces rather than the simple "one head and one EVL wand" interface. When a user starts a client program to join a shared world, he must provide an avatar definition file. This file is an Ygdrasil scene file, the same as one that defines a world; in this case, it describes the set of tracking sensors to be used, the wand(s) or other control devices, and the models for the avatar. In addition to normal, tracked sensors, it is possible to create derived pseudo-sensors; i.e. the body position data in Figure 19 might not come from an actual sensor, but be calculated based on the head tracker data. The definition of the avatar also defines the user's interaction controls – how he can navigate in the VE, and which buttons or other controls will be used for actions such as grabbing objects.

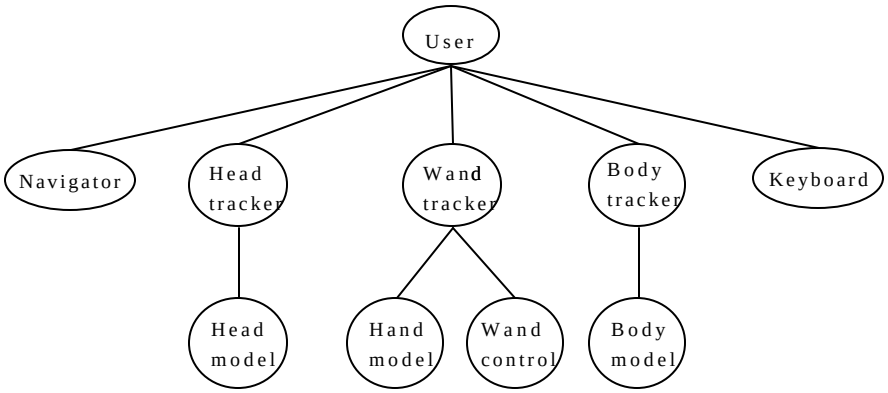


Figure 19. Example of a sub-graph representing a user's controls and avatar

When a user joins a world, his avatar is attached to the shared scene graph, and thus becomes visible to all remote clients. The remote clients will add a reference to the root "user" node to their lists of current users, which are passed to any local behavior nodes that require information about users. These nodes can query this root node to find out further information, such as a pointer to the user's wand. They can also use it to send commands back to the user, such as teleporting him to a new location.

The exact definition of a user's sub-graph is relatively free-form, but certain elements are expected in order to provide a common interface for other nodes that interact with users. All users are assumed to be navigating through the virtual world, so they will have some sort of 'navigator' node as part of their sub-graphs. This navigator node will perform its update calculations, and pass new position and orientation data to the base user node, which contains the actual transformation applied to the user. Relative to this base navigation transformation are the user's body parts – the head, wand, torso, etc. For nodes (such as

triggers) that want to know the current state of the user's parts, the user node keeps a list of its collection of 'UserPart' nodes; these nodes can be queried for their position and orientation. The UserPart nodes are also labelled, to allow distinguishing between a head and a hand, for instance. However, Ygdrasil itself does not attach any special meaning to any body parts other than the head, so this labelling will have to follow some convention that is defined by the applications that use it.

6.4. Scripting

The scripting system of Ygdrasil is based on the scene files of XP, but expands their capabilities and resolve some of their problems. Its objective is to provide a high-level interface for assembling the shared scene, one that makes rapid prototyping of worlds possible. The many users, both novice and expert, of Alice have demonstrated that an interpreted, scripting language is a very valuable tool for experimentation and easy creation of worlds from predefined components. Most current VR development toolkits include a scripting layer. However, a significant difference between them and the approach here is that these other toolkits use traditional scripting languages. Obviously, this has the advantage of leveraging existing technology and programmers' familiarity with it, but none of these languages was defined with VR in mind. The Ygdrasil scripting method is oriented specifically toward the problem of manipulating objects and their behaviors in a virtual world, and thus operates primarily at a simpler level than languages that deal with manipulating data structures and traditional programming control structures.

A scene file defines the structure of a world, as well as setting attributes of individual nodes and defining message-based connections between nodes. This approach helps the idea of a behavior being part of an object, rather than separating the program from the data, and has proved very useful in easily constructing environments. On the other hand, it is more difficult to make programmatic changes to a

running world. It is possible for a programmer to send individual messages, at the command line, to existing nodes, but it is not possible to make general changes to the world structure. This limitation may need to be addressed in future versions of Ygdrasil.

6.5. Implementation

6.5.1. Scene Files

The format of an Ygdrasil scene file is roughly the same as that of an XP scene file. The most significant changes that occurred are in the syntax for node options and messages, the placement of node names, and the addition of "event arguments". The basic syntax for a scene file is described by the BNF grammar in Figure 20. An example scene is shown in Figure 21.

<scene> ⇒ <tree> *****

<tree> ⇒ <singleNode> | (<singleNode> "{" <tree> ***** "}")

<singleNode> ⇒ <className> [<nodeName>] ["(" [<messageList>] ")"]

<messageList> ⇒ <message> ["," <messageList>]

<message> ⇒ <messageName> "(" [<argumentList>] ")" ["+" <delay>]

<argumentList> ⇒ <argument> [[","] <argumentList>]

<className>, <nodeName>, and <messageName> are any valid names
<argument> is any string, possibly in quotation marks
<delay> is a floating point number

Figure 20. Ygdrasil scene file grammar

```
light sun ()
environment (volume(box –1000 -1000 -1000 1000 10 1000),
                    skyColor(.5 .7 1))

transform (position(-5 2 7))
    {
    spinner (axis(0 1 0))
          {
          object (file(carousel.pfb), floor, wall)
          }
    }

userTrigger (volume(sphere 0 0 0 10), when(enter $user.teleport(1000 0 0)))
```

Figure 21. Example Ygdrasil scene file containing a background color, spinning object, and trigger

6.5.2. Core Classes

Most of the important code in Ygdrasil centers around the classes *Object*[1] and *Node*, which implement the basics of a shared scene graph node. The implementation of a node was split into these two classes to reduce some of its complexity. Object represents any shared object, and handles the storing of data in the CAVERNsoft database; Node, which is a subclass of Object, handles the scene graph-related functions such as storing pointers to child nodes, and traversing the graph. Closely tied to Object are the classes *Handle* and *ObjectDB*.

[1] In the actual C++ code, all Ygdrasil class names begin with the prefix *yg* (for example, *ygObject*). However, the prefixes will be omitted in this text, for the sake of readability.

The basic attributes of any object, those taken care of by Object, are its name, its class name, and its set of database keys. Every object must have a unique name so that this name can be used as a path for its database keys. For example, a transform node named "foobar" would store its matrix as "foobar/matrix", to keep it distinct from any other transform nodes' matrices. Within the main Ygdrasil program, the class name of an object is used for two purposes – debugging flags, and networked proxies. Debugging flags are described in section 6.5.8. For networked proxies, a remote client must be able to create a proxy of the appropriate class in order to receive and use all of the node's shared data. Hence, the object records its class name as a string, and stores this string in the database for other clients to request. This, and any other shared database keys, are kept track of from Object in a set of NetKeys; this class is described in section 6.5.7.

The Object class interface also defines the virtual function 'message()', so that all types of shared objects may receive messages through a common interface.

A Handle is a stable reference to an Object. It is necessary because of the potentially dynamic state of the collection of objects that make up a networked application. When an Ygdrasil client first learns about a new, remote object via its name (usually in a list of children of some other node), it requests the object's class name in order to create the correct proxy. However, because the round-trip times on wide-area networks can be significant (on the order of 10s of milliseconds or more), we would prefer not to have the program wait until it actually receives a response before continuing on. So, the client first creates a dummy proxy that can be used until the real proxy is created and replaces the dummy. However, if in the meantime other objects (e.g. a node's parent node) have gotten pointers to the dummy proxy for later use, these pointers will be invalid when the dummy is replaced. Similar problems can arise when a user

exits a world – any pointers to the user's avatar nodes will be bad, assuming the avatar data has been properly deleted. The Handle class takes care of these issues by providing an intermediate reference to an object. Each Object has a pointer to its Handle, and vice-versa. When a dummy proxy for an object is replaced by a different proxy, the new proxy uses the old one's handle, and tells the handle to now point to the new proxy object. Objects that need to retain long-term pointers to other objects, such as a node keeping a list of its children, will use Handles instead of direct pointers, and thus will automatically have references to the correct objects. In the case of an object being deleted, such as when a user exits, is dealt with by having the Handle create a dummy, so-called "void object" to use when it no longer references any other object.

All of the Objects that make up a world are kept track of in the ObjectDB class. Its primary functions are to create new objects and to find existing objects by name. Any code that wishes to create a new object does not simply use the C++ *new* operator, but instead calls ObjectDB::create(). There are two purposes for this. One is to make sure that all objects are recorded, so that they can be found by name later; the second is to provide an interface to the DSO-loading mechanism for new classes (described in section 6.5.6). ObjectDB's find() function is used in a number of different places throughout the Ygdrasil program. The most important uses are to locate objects that are to receive messages as defined in the scene file, and for the automated CAVERNsoft database layer to locate proxy objects that have received new data from their remote, master versions.

Data and functions related to the scene graph are in the class Node. Some of the features of Node are the list of a node's children, the graph traversal, event generation and response, signals for simple node state or commands, and the corresponding Performer node. All Nodes have functions app() and view(),

which are the basis of the scene graph traversal. To update the state of the world, the graph is traversed and the app() function is called for every active node (inactive nodes are those below switches that are turned off, or selectors that are currently selecting a different subgraph). The view() function is called in a separate traversal, and is intended for updates that are related strictly to the viewing of the scene by one user, as opposed to updates that affect the data seen by all users in the shared world. Most nodes do not require a view() function; the distinction is important only for a few special nodes. For example, a special level-of-detail node would need to choose which version of a model to draw based on the location of the user who is viewing it – not all the users should see the same detail level at any given moment. The event generation and response is mostly the same as that in XP; it is described further in section 6.5.3. Signals are a feature that was added for basic inter-node communication such as when a trigger wants to check a user's wand for button-presses. Because the master versions of the wand and the trigger may be executing on different hosts, the button-press information must be shared across the network; this can be done by setting a particular signal, such as "act", which the trigger node knows to check. The signal mechanism deals with the event-like nature of signals by guaranteeing that any given signal will be set for the duration of one frame in each client's proxy copy; this is important to avoid timing errors (such as from clients running at different frame rates) that could cause a client to either miss a signal or believe the signal was set for multiple frames in a row. Finally, the Node class retains a pointer to the Performer node that corresponds to the Ygdrasil node, and provides functions for getting some basic data about it, such as its position and orientation relative to any other node.

The relationships between these and some of the other classes described below are summarized in Figure 22. The diagrams use a simplified Object Modeling Technique style, based on that of *Design Patterns* (Gamma et al., 1995).

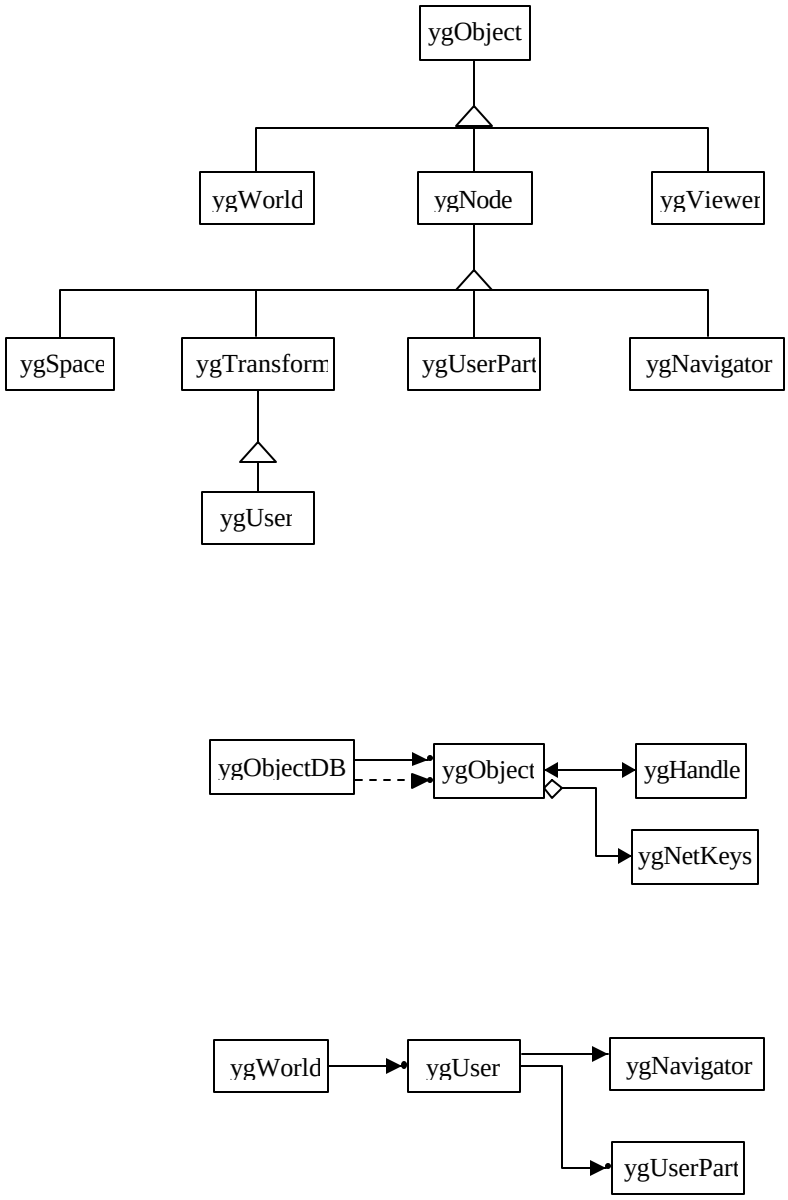


Figure 22. Relationships among the basic Ygdrasil classes

6.5.3. Events and Messages

As in XP, the programming of an Ygdrasil virtual world at the scene file level is done using events and messages. Events are detected and reported by the behavior code for a node class, and messages can be sent to any node to change a value or to start some action. Messages are typically sent as part of the initialization of a scene from the scene file, or in response to events. Initialization messages are handled by the file parser; linking events to messages is handled by the 'when' command of the Node class. When creating a new class, all that an application programmer has to do is to process the new, class-specific messages in the message() function, and to signal events using the function Node::eventOccurred().

Events are represented by the *Event* class; each occurrence generates a separate Event object. At its most basic, and Event is simply an arbitrary string. For example, in the userTrigger node in Figure 21, the string "enter" is an event that indicates that a user just entered the trigger region; the C++ code for userTrigger tests, in its update function, whether any users have entered, and calls eventOccurred("enter") is response. However, as was noted in the evaluation of XP, some events will need to have additional information associated with them. This is implemented as a collection of event arguments, each of which is a string, with a string label. In the case of the userTrigger "enter" event, the name of the user that entered the trigger is passed as an argument labeled "user". The labeling is necessary so that scene file messages have an easy way to make use of these arguments. For example, in Figure 21's userTrigger, "$user" will be replaced by the value of the argument "user", and so the message "$user.teleport(1000 0 0)" is sent to whichever user generated the enter event. These arguments must be attributes of the event, rather than simply of the node that generates the event, because a node might conceivably generate several events, with different arguments, during the course of a single frame.

Messages are very similar to events, in that they have a string name and a collection of string arguments. They are represented by the class *Message*. A message is sent to a node by passing it to the node's function message(). This function parses the message by testing its name, and if the name is recognized, parsing the arguments as necessary and calling the appropriate class member function. The Message class includes a number of utility functions for converting arguments into such things as integers, floating point numbers, boolean values, and vectors. It also includes functions for translating a string, such as "foo.position(1 2 3)", into a Message object, and for the reverse – writing a string that corresponds to a particular Message object; these functions are used by the scene parser, and by any new classes that might want to generate messages on the fly.

The primary structural distinction between an Event and a Message is how the arguments are defined. In an Event, each argument has a mnemonic label, such as "user", whereas in a Message the arguments are simply identified by their order (e.g. in "position(10 0 7)", "10" is argument 0, etc.). These approaches were each chosen for their convenience in writing scene files; for instance, labeling message arguments would result in lengthier commands such as "position(x=10 y=0 z=7)", but for events it was felt that "$user.teleport(1000 0 0)" is preferable to "$1.teleport(1000 0 0)". However, these differences are not necessarily irreconcilable, and it might be interesting, at some point in the future, to consider merging the classes Event and Message and making them interchangeable.

6.5.4. World and View

Two classes – *World* and *Viewer* – are responsible for managing the overall scene. The World class keeps a pointer to the scene graph, and performs the 'app' traversal on the graph in order to update every node each frame. It also takes care of passing messages to nodes; when one node wants to send a

message to another, rather than doing this directly it hands the message to the World's scheduleMessage() function. This is done because messages often have delays associated with them, and so they are placed in a queue maintained by the World; if an individual node tried to take care of its delayed messages, it might potentially be disabled (e.g. by being under a switch) before the message was sent, thus causing the message either to be lost or to be sent much later than requested. In addition to these tasks, the World object keeps pointers to any User nodes that are added to the scene. This is provided as a convenience for the many different nodes (triggers, etc.) that are expected to interact with users; rather than requiring each of these nodes to search the scene graph for Users, they can simply request the list from the World.

While the World object takes care of simulation updates, and thus must always exist, the Viewer object is responsible for view-related updates, and thus only exists when the Ygdrasil program is running a CAVE or other display. Ygdrasil instances that function strictly as servers do not need a Viewer. The Viewer object provides an interface to special Performer functions such as changing the clipping distances or the background color, which can be used by environmental nodes in the scene. It also provides information about the person viewing the scene – that is, the person's position and orientation. This information is needed by nodes that vary their local behavior based specifically on the local viewer. Most nodes' behavior will be affected by any user in the shared world, regardless of whether that user is local or networked. But some nodes do need to know about the local user; for instance, 3D sound nodes will need to know the user's position in order to implement spatialization. The Viewer class is considered an abstract class that may have multiple derived classes to provide different user interfaces. There is a CAVEViewer class that uses the CAVE library to get the tracked user's position and orientation; a

different Viewer class might be created to provide a simple mouse-based desktop interface that doesn't require the CAVE library.

6.5.5. User Classes

A user's representation in the shared scene is built from three classes – the *User* node, the *Navigator*, and the *UserPart*.

The User node is the root of the sub-graph forming the user's avatar. Its purpose is to contain the avatar and form an interface between it and other objects in the scene. The User class includes functions to get pointers to the user's Navigator node, Head node, and any UserPart nodes. The User class itself is derived from the Transform class, which is a basic transformation node, equivalent to Performer's pfDCS (dynamic coordinate system). This particular transformation is the user's base navigated position in the scene; positions of parts such as a tracked head and wand are then local transformations relative to the User node.

Navigator is a base class for different nodes that may control the user's navigation. The basic Navigator class itself does not perform any navigation, but simply defines a standard interface and contains the raw navigation data – the user's position, orientation, and size. A class that implements a particular type of navigation is then derived from Navigator, and extends its app() function to perform the appropriate calculations and update the position/orientation/size data stored in the parent class. The Navigator class's app() function then passes that data to the User transformation itself.

The UserPart class represents any sort of individual body part of the user, such as the head, the wand, the torso, etc. All this class actually does is to provide a common base class for user body parts, and allow them to be labeled. Labeling the parts with strings such as "head" or "wand" will allow other nodes

(e.g. triggers) to find just the specific parts of the user that they need to check. Special classes are derived from UserPart, such as *Head*, to represent the user's head position, and *CAVEWand*, which checks the CAVE wand for button presses and generates corresponding events. UserPart is not a transformation node; instead, it is assumed that UserParts will be attached to other transform nodes that provide the desired avatar movement – e.g. the *CAVETracker* node class, which gets its data from CAVElib tracked sensors.

6.5.6. Adding Node Classes

New node classes can be added to Ygdrasil via DSOs (dynamic shared objects), sometimes referred to as "plug-ins". This is similar to the approach used by Bamboo (Watsen and Zyda, 1998), and makes extending Ygdrasil simpler than extending XP, as the main program itself does not need to be recompiled; DSOs can be easily shared among developers, and new node classes can be added to the environment even as it is running.

The DSOs are managed by the class *ConstructorDB*. This class keeps a list of all DSOs that have been loaded so far, indexed by the name of the node class that they implement. When the ObjectDB wishes to create a new node, it requests a pointer to the appropriate constructor function from the ConstructorDB. If no DSO has been loaded yet for that class name, the ConstructorDB searches its directories for a ".so" file with the same name as the requested class; if it finds the file, it loads it and returns a pointer to the constructor function.

In the case where a new class is derived from another class defined in a another DSO, the parent class must be loaded first for the system to be able to load the derived class and resolve all its function addresses. To address this issue, an Ygdrasil DSO may include a list of dependencies – that is, other

classes on which the DSO depends. If such a list is found, the ConstructorDB first searches for and loads the dependency DSOs. This action is recursive, so it can first load any dependencies of the requested DSO's dependencies, and so on back through the class hierarchy until it reaches dependencies that are already met (i.e. classes that are already loaded).

6.5.7. Networked Database

All networked objects use two classes to handle their shared data – *Net* and *NetKeys*.

The Net class is a set of functions that administer the networking. They initialize CAVERNsoft and the shared database, and start a separate thread for sending and receiving data updates. The function Net::requestKey() queues a request to fetch a particular key's value. The function Net::storeKey() stores a new value for a key. Net's update() function will pass any new key values that have been received from the CAVERNsoft database server to the Object responsible for the key. All of the actual writing and reading of data on the network is done by the separate thread, so that the main application process is not blocked by these operations.

The NetKeys class is an interface between the shared object classes and the Net functions. Objects do not directly call Net functions, but instead store their data in NetKeys objects; NetKeys then call Net::storeKey() for those keys that have changed during the current frame. NetKeys also handles the translation of various common data types into packets that can be sent to the shared database. These types include strings, integers, vectors, and matrices. An object defines a net key to be shared by calling the function Object::addNetKey(), and passing it a name for the key, a pointer to the variable containing the key's data, and the type of the data. In this way, the NetKeys can not only send new data to the database, but it can automatically save newly received data from the database into the appropriate object

variable. The Object function acceptNetKey() is called when this automatic receiving is done, so that a node can check and use the new data if necessary.

6.5.8. Utilities

There are several general utility classes and functions in Ygdrasil. The ones described here are the *String*, *Volume*, *PFDBase*, and *DebugFlags* classes.

Ygdrasil makes heavy use of text strings, for such things as passing messages, events, and names of classes and objects. Hence, it includes a String class to simplify handling them. The class allows easy comparison, assignment, and appending of strings via overloaded C++ operators. It also has functions for splitting a string into tokens; for example, breaking a colon-separated search path into its component directories. One other important feature is that it performs case-insensitive comparisons (e.g. "Transform" is equal to "transform").

Spatial volumes are also common in Ygdrasil applications. They are used to define trigger regions, areas where sounds are audible, and areas that have certain environmental characteristics (fog, etc.). The abstract class Volume, and its derived classes Box, Cylinder, InfiniteVolume, Point, and Sphere, are standard tools for these uses. The volume classes include functions for testing if a point is inside the volume or if a ray intersects the volume, and for computing the distance of a point from the volume. There are also utility functions to parse a string message describing a volume and create the corresponding volume, to create such a string message from an existing volume, and to create Performer geometry that takes the shape of a given volume.

The PFDBase class provides functions for loading Performer models asynchronously, using Performer's DBASE process. Using this separate process is important because of the dynamic nature of

shared worlds. New objects may be added to a scene at any time – at the very least, when users join a world, their avatars must be added. If the model files for these objects were loaded synchronously, within the main app process, the graphics (and simulation) would freeze any time a new object was added. The PFDBase class allows a node to request that a model file be loaded; it is given a PFDBaseRequest object in response. The node can then check the PFDBaseRequest object's state to learn when the model has actually been loaded, in case the node needs to do something with it, such as setting intersection flags.

The debugging features in Ygdrasil are intended to be much more flexible than those in XP. In XP there was just a single flag, indicating whether "debug mode" was on or off. This meant that if many objects provided detailed debugging information, the user could be overwhelmed and not easily find the information he wanted. In Ygdrasil, each node class can have a set of debugging flags to turn on or off multiple debugging options independently. The user can set any of these flags, either by an environment variable or by messages to nodes. With the debugging environment variable, wildcards can be used, and flags can be set for all nodes of a particular class, or for a specific node. For example, setting "UserTrigger.volume" would enable the volume outlines for all UserTrigger nodes, whereas "trigger1.volume" would enable the volume outline just for the node names "trigger1". This is all implemented using the DebugFlags class, which is referenced within the Object class. Node classes can simply call Object::debugFlag() with a string name for the flag and a pointer to a boolean variable, and later check the state of the variable, which will be automatically set when requested by the user.

7. USE AND EVALUATION OF YGDRASIL

As a test of its feasibility for large, shared VEs, Ygdrasil was used to create two cultural heritage applications shown at iGrid 2000. The International Grid (iGrid) is a series of research demonstrations highlighting the value of international high-speed computer networks in science, media communications, and education (Brown et al., 1999b). iGrid 2000 took place at the INET 2000 conference in Yokohama, Japan. It provided a 100 Mbps connection from the conference site to STAR TAP in Chicago; STAR TAP serves as an international connection point for several research networks in America, Europe, and Asia.

The two VEs shown were Shared Miletus and the Virtual Harlem Project; both were based on existing, non-networked applications. The applications were run in a CAVE in the iGrid booth, connected to a second CAVE in Chicago at EVL; the Virtual Harlem Project also connected to the Virtual Environment Instruction Lab at the University of Missouri-Columbia. The following sections describe these applications and the work necessary to build them, and discuss some of the Ygdrasil design issues that arose as a result.

The final sections of this chapter present the results of some benchmark tests that were run to evaluate the scalability of Ygdrasil and the costs of using it.

7.1. Shared Miletus

Shared Miletus is an environment created in collaboration with the Foundation of the Hellenic World (FHW) (Pape et al., 2001). The FHW is a non-profit, privately funded museum and cultural research institution in Athens, Greece. Its mission is to preserve and present Hellenic history and culture; it seeks to use state-of-the-art technology to accomplish these goals. The FHW owns two virtual reality systems,

an ImmersaDesk and a ReaCTor, that are used to present a variety of content created by Foundation staff (Roussou, 1999). Exhibits using the systems have included 3D reconstructions of ancient cities and buildings, as well as educational, interactive environments such as the history of Hellenic costume. Experienced museum guides lead visitors through the exhibits; the guides must have both technical skills to operate the VR displays and museum education skills to explain the history of the city. The guides are an important part of the exhibits, and any networked version of these exhibits must in some way take into account their role in educating visitors.

One of the first applications shown in the VR systems at FHW was a reconstruction of Miletus, an ancient city on the coast of Asia Minor (see Figure 23); the original application was developed in XP. Detailed models of some of the buildings of Miletus were created, and museum visitors can explore the city as it was in antiquity. The objective of the iGrid version of Miletus was to take the content that would normally be shown in the controlled environment of FHW's museum, and let remote, networked people visit it. In particular, we did not want to simply make it something like a VRML model that visitors would download and then play with on their own; instead, it was to be considered a dynamic, shared world, "hosted" by the conference demonstrators (or, in the future, the museum).

Figure 23. Shared Miletus – a view inside the Delfinio

7.1.1. The Demo

In creating Shared Miletus, we focused on two issues – guiding visitors through the city, and providing them with information about what they were seeing. These features needed to work in an internationally distributed environment, where users could come and go from the space at will.

Many museum-based VR exhibits will lead visitors through the virtual world on a pre-selected path, so that users do not have to learn any special controls or know where they should be going. The river

metaphor, described in (Galyean, 1995}, extends this model by allowing the users to stray somewhat from the fixed path, but always guaranteeing that they continue to progress in the right direction. In our case, we wanted to give the visitors freedom to explore Miletus at their own pace. They were given a 3D wand used for simple joystick-driven navigation; a recorded introduction when they entered the space explained how to use the wand. To make it easier to get to places of interest, and to rescue the visitors if they got lost, we also gave them a dynamic, virtual map. This map showed the layout of the city, the user's position in it, and also the positions of any other participants in the shared world. This helped them to drive to particular buildings, or to meet up with other visitors or guides from the museum. In addition, the map could be used as a navigation shortcut – clicking on a particular building would summon a magic carpet that then automatically brought the user to that building's entrance. If a visitor were completely lost, a special reset button would start him back at the entrance to Miletus, and replay the instructions on how to use the wand and map.

The first stage of providing visitors with information about Miletus was to include expert human guides. Guides from the actual, "official" museum could enter the shared world, just like an ordinary visitor. Through their avatars, and streaming network audio connections, the guides could then interact with the visitors, pointing out special details and answering questions.

Given the international scope of the shared space, human guides alone are unlikely to be enough – there could be large numbers of visitors, and they could be exploring the space at any time of day. So, we placed automated information kiosks within the various buildings of Miletus. These kiosks contained pre-recorded audio commentary describing each building and its history. In order to support an international audience, this audio was available in multiple languages; for the iGrid demo we provided English and

Japanese commentaries, but given enough time and translation personnel, any number of languages could be supported. The multi-lingual capability was implemented by having each visitor carry their own virtual audio tool. The tool was effectively a part of the user's avatar, and kept track of his preferred language. When the user approached a kiosk, a trigger detected the presence of an audio tool and sent the tool messages informing it of what recordings the kiosk could provide. If the user chose to listen to one of them, the tool would send a request back to the kiosk, asking for the appropriate sound file for the desired language. Other tools at the entrance to the world could be used to switch languages – clicking on a Japanese flag icon would send a message to the user's audio tool to use Japanese, for example. The audio tool also provided the introduction and navigation instructions in the appropriate language.

7.1.2. Implementation Details

Table VI summarizes the contents, both data and code, that went in to creating Shared Miletus. Several new node classes were programmed for Miletus. These include the *Visibility*, *LODObject*, *PathFollower*, *FlyingCarpet*, *MiletusNavigator*, *NodeTrigger*, *AudioTool*, *SoundNode*, *Mapper*, and *LocalData* classes.

TABLE VI

CONTENTS OF SHARED MILETUS

Scene graph	1025 nodes 23 scene files 2300 lines in scene files	
Code	18 new classes 2712 lines of code	
Data	227 models 291 texture images 139 sound files	17 Mbytes 60 Mbytes 151 Mbytes total: 228 Mbytes

The Visibility and LODObject classes were created to improve the graphics performance of the application. The complete model of Miletus is much too complex to be viewed at acceptable frame rates. Performer's culling and other built-in optimizations help, but were still not enough. Visibility nodes were used to improve culling. They are used around complex models in the scene, and define a region or regions where the models are visible; for example, if the user is inside one building, the other buildings should not be drawn. The LODObject class provides an interface to Performer's level-of-detail feature, and allowed us to include simpler versions of things such as pillars, to be used when the viewer is far away from them.

The PathFollower was created for simple animations of objects. It reads a data file of key frame information; when activated, it moves itself along the path.

The FlyingCarpet and MiletusNavigator implement the automatic carrying of the user to selected places in the scene. Each user has his own FlyingCarpet node that is defined along with the user's avatar. When the user clicks on a destination on the map, a command is sent to the FlyingCarpet; it finds the user's current location, starts there, and then moves in a smooth hop to the destination. At the same time, an 'attach' message is sent to the user's navigator. This message is part of MiletusNavigator, being an extension to the normal CAVENavigator's features; it causes the user's navigation to be tied to an object (in this case, the carpet) and automatically move with it until a 'release' message is received.

The NodeTrigger, AudioTool, and SoundNode classes were created for the information kiosks. A NodeTrigger is similar to a UserTrigger, except that it can detect nodes of any requested type; in the case of Miletus, we used it to detect AudioTool nodes as they enter a kiosk's area. Each user's avatar includes its own AudioTool node, so that each user can listen to the commentaries separately, in his preferred language. The AudioTool receives messages from the kiosks' NodeTriggers, informing it of when there are sounds to play; it receives messages from other triggers, attached to the language flags, telling it what language to use. It also receives messages from a WandTrigger that is attached to it, so that the user can click on the tool to start playing a sound. The SoundNodes are interfaces to the different commentaries in multiple languages. A single SoundNode corresponds to one commentary, and contains a list of AIFF files, one for each supported language. When the AudioTool wants to play a sound, it asks the SoundNode for the name of the correct AIFF file to use.

The Mapper class implements part of the user's dynamic map. The only aspect of the map that needed new code was the display of other users' positions. The Mapper gets the locations of all of the users in the scene from the ygWorld node. It creates a set of small markers, and positions them according

to the users positions. This node is attached to a model of the map itself, so that the markers will appear on it. The control that allows a user to click on the map and be carried to a building simply uses existing trigger classes.

The LocalData class was added after initial tests of Miletus. Because Ygdrasil shares its entire scene graph among all clients, by default users could see each other's maps and audio tools. This was considered unnecessary clutter, so we created the LocalData node to hide these objects from other users. When objects are attached below a LocalData node, their scene graph information is still shared, but on clients that do not own the LocalData, it turns itself off so that everything below it is not rendered.

7.1.3. Issues Encountered

Constructing Shared Miletus brought to light a few potential issues with the design of Ygdrasil.

One of the hypotheses behind the design of Ygdrasil's shared scene graph was that, for any particular node, one host could hold the master copy and execute its behavior code; all other hosts would have simpler proxies that receive new state from the master. It was hoped that most of the state that needs to be shared would be common, Performer and Bergen-related information used to view/hear the nodes. In other words, proxy nodes would normally be of one of the basic, built-in types, such as Transform, Switch, or Model, and clients would then not need any new executable code (DSOs) in order to join a world that is running on another host. This hypothesis holds true for the large percentage of actual nodes in the Miletus scene graph, but it turned out that several classes (Visibility, AudioTool, SoundNode, and LocalData) needed to be 'networked classes'; that is, clients would also have to have copies of their DSOs to even use the proxy versions of the nodes. Other Ygdrasil world-authors continue to come up with new behavioral nodes that similarly must be networked. Hence, in the future, Ygdrasil should be

extended to be able to automatically download new DSOs from a server as needed, similar to the approach taken by Bamboo (Watsen and Zyda, 1998). One important thing that should be part of this feature is security, perhaps in the form of digital signatures, as is currently done in web browsers, to help ensure that clients do not download malicious code from unknown servers (Bamboo itself does not yet include any security, but merely notes it as a future addition.)

The need for the LocalData node showed that not everything in the scene graph should be shared equally. This is similar to the inclusion of 'private' objects in BrickNet (Singh et al., 1995). This feature should probably be incorporated into the core set of classes, but also be expanded to allow sharing objects between restricted groups of users.

Finally, when nodes whose master copies are on different machines interact, network lag can have an effect. This can produce visible artifacts with objects that users will interact with directly. For example, if the flying carpet node had not been included with the user's avatar, and thus run on the same host, when the user was attached to the carpet others might see the user and carpet moving at different rates. This problem will be more important in future applications that contain objects that users can pick up. If a grabbed object updates its own position based on the user's hand position, network latency will cause the object to appear to lag behind the user's hand. A solution to this problem is to simply change the scene graph hierarchy dynamically; when a user attaches himself to a vehicle, his avatar will be re-parented under the vehicle in the scene graph; when a user grabs an object, it will re-parent itself under the user's hand. One issue that then arises, however, is what to do at when the object is dropped. If a user picks something up in one area, travels a long way to some completely unrelated area, and then drops the

object, where should it be re-attached in the scene graph? This will require some experimentation and careful world-design.

7.2. Virtual Harlem

Virtual Harlem is an environment developed in collaboration with the University of Missouri-Columbia and the UIC English Department (Carter, 1999). It is intended to supplement African American literature courses at both universities, as well as potentially many other schools across the country. It is a reconstruction of 10 square blocks of New York's Harlem in the 1920s, the period known as the Harlem Renaissance. The reconstruction is based on photos, maps, films and recordings of the time. The intent is to allow students to visualize the setting and context of the writings that they are studying, hopefully leading them to be more directly engaged with and more deeply understand the material. Students can navigate through the streets and buildings of Harlem, hearing the music and people of the period and visiting some of the sites related to the works that they are studying; they can see performances at the Cotton Club and hear speeches by figures like Marcus Garvey. An instructor leads the students through the environment, explaining things and answering questions.

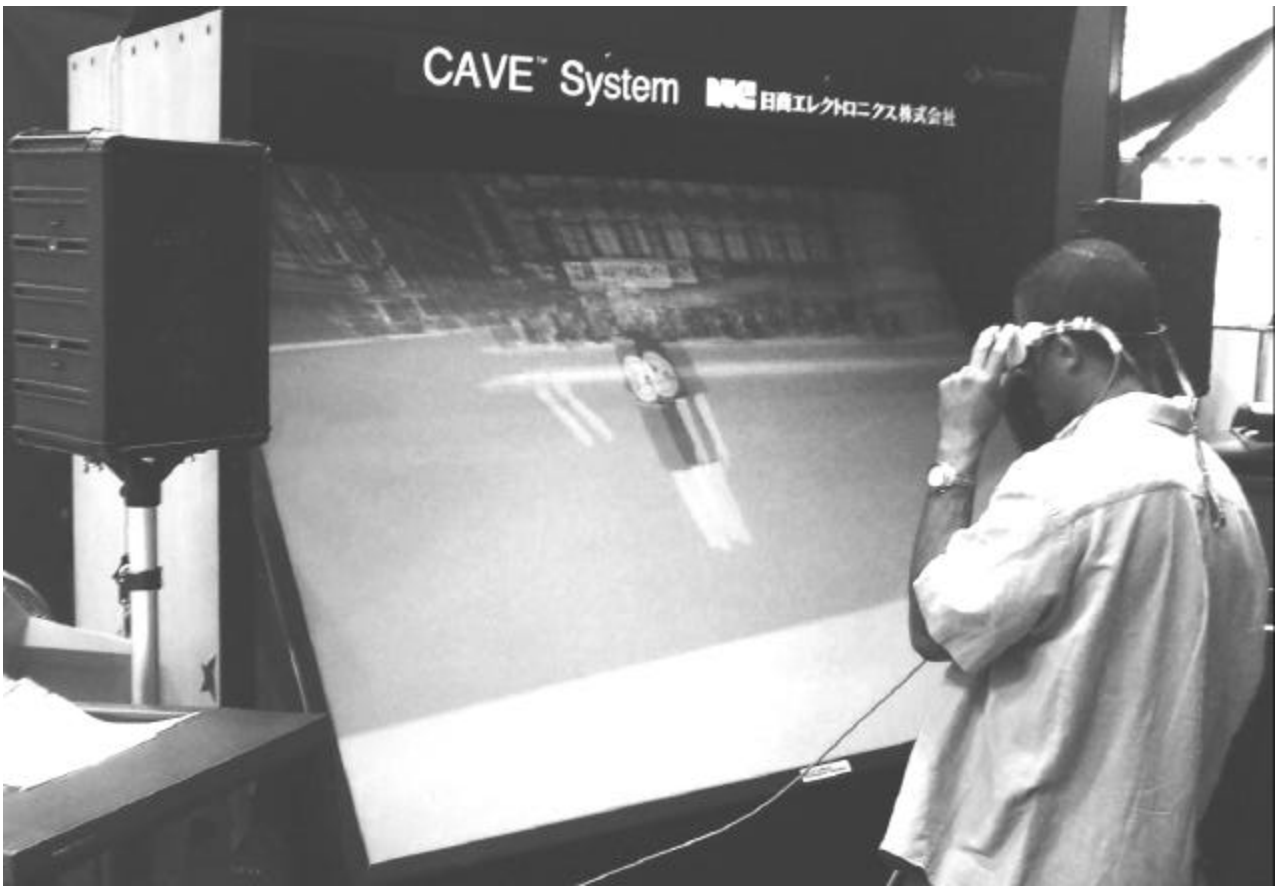


Figure 24. Virtual Harlem running on an ImmersaDesk at iGrid 2000

7.2.1. Contents of Virtual Harlem

Table VII summarizes the amount of data and code that were created for Virtual Harlem.

The environment consists of roughly 60 buildings, some of them significant sites such as the Abyssinian Church, others ordinary apartments and shops. The models of the buildings were constructed by students at the University of Missouri. Distributed throughout the area, in front of certain buildings, are historical characters – Langston Hughes, Marcus Garvey, a group of women headed to a rent party, etc. These characters are billboarded, cutout images of the people from 1920s photographs. When a user

approaches these characters, recorded speeches or conversations are played. A trolley car moves automatically through the streets. Visitors can board the car by entering it, at which point they are 'attached' to it and move with it; they can exit by stepping outside of the car. Visitors can also enter the Cotton Club; inside is a static re-creation of patrons and staff in the main hall, and an interactive movie screen on the stage. By clicking wand buttons, any visitor can play back QuickTime movies of various performances that were at the Cotton Club. Figure 25 and Figure 26 summarize the scene-graph layout of the environment. The complete scene files used in Virtual Harlem are provided in Appendix B.

The main new node class that we coded for Virtual Harlem was the *movieScreen* class. A movieScreen node takes a QuickTime movie file and displays it on a virtual screen. This involves reading individual frames of the movie using SGI's moviefile library, converting them to texture map images, and attaching the texture to a square. The movieScreen node used in the Cotton Club accepts messages from a set of triggers there; the triggers tell it which movie to play and when to start playing, whenever a user presses a wand button.

TABLE VII

CONTENTS OF VIRTUAL HARLEM

Scene graph	220 nodes 5 scene files 384 lines in scene files	
Code	3 new classes 545 lines of code 18 re-used classes	
Data	133 models	24 Mbytes
	453 texture images	53 Mbytes
	25 sound files	110 Mbytes
	5 QuickTime movies	25 Mbytes
		total: 212 Mbytes

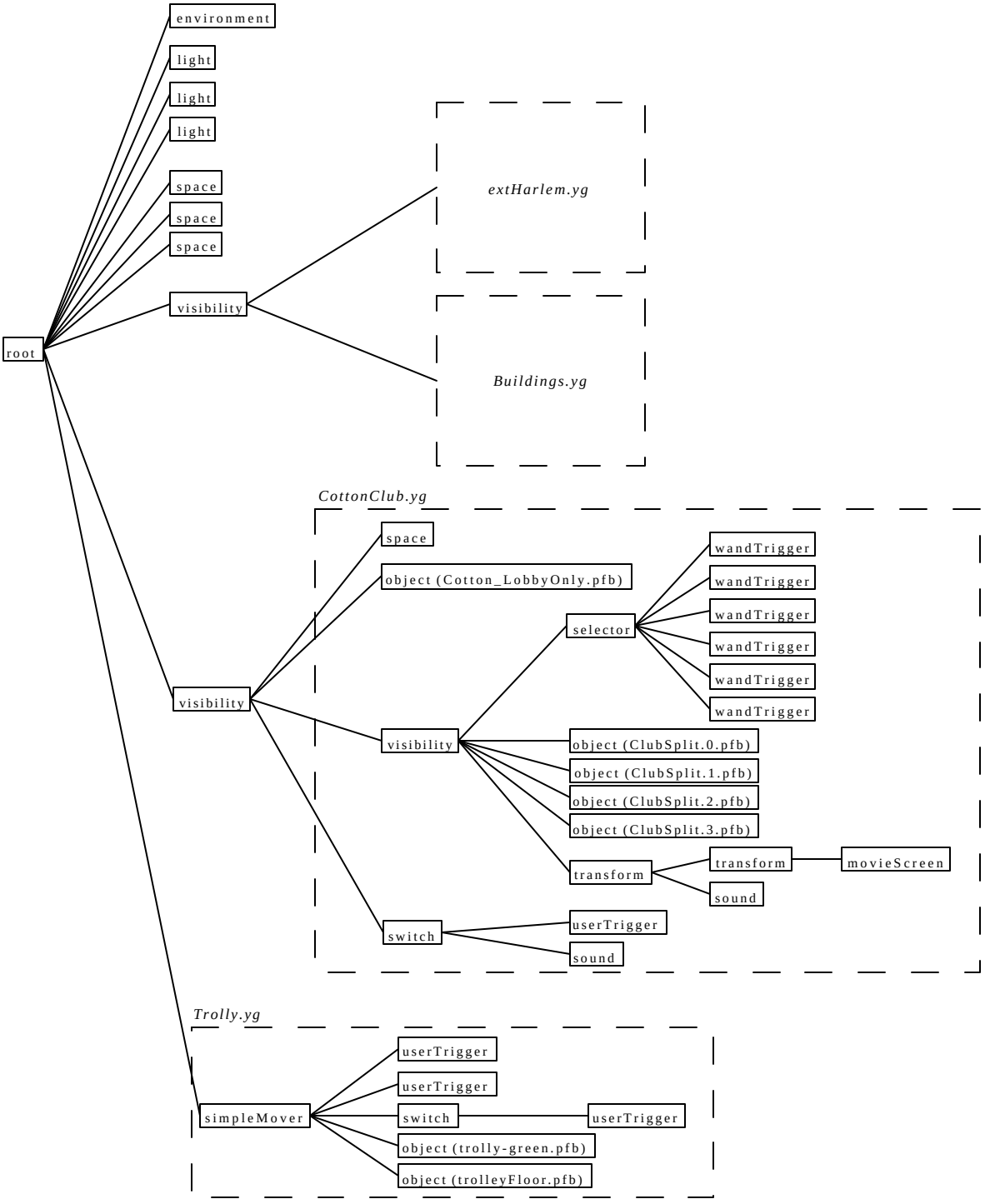


Figure 25. Virtual Harlem scene graph structure of the Cotton Club and trolley

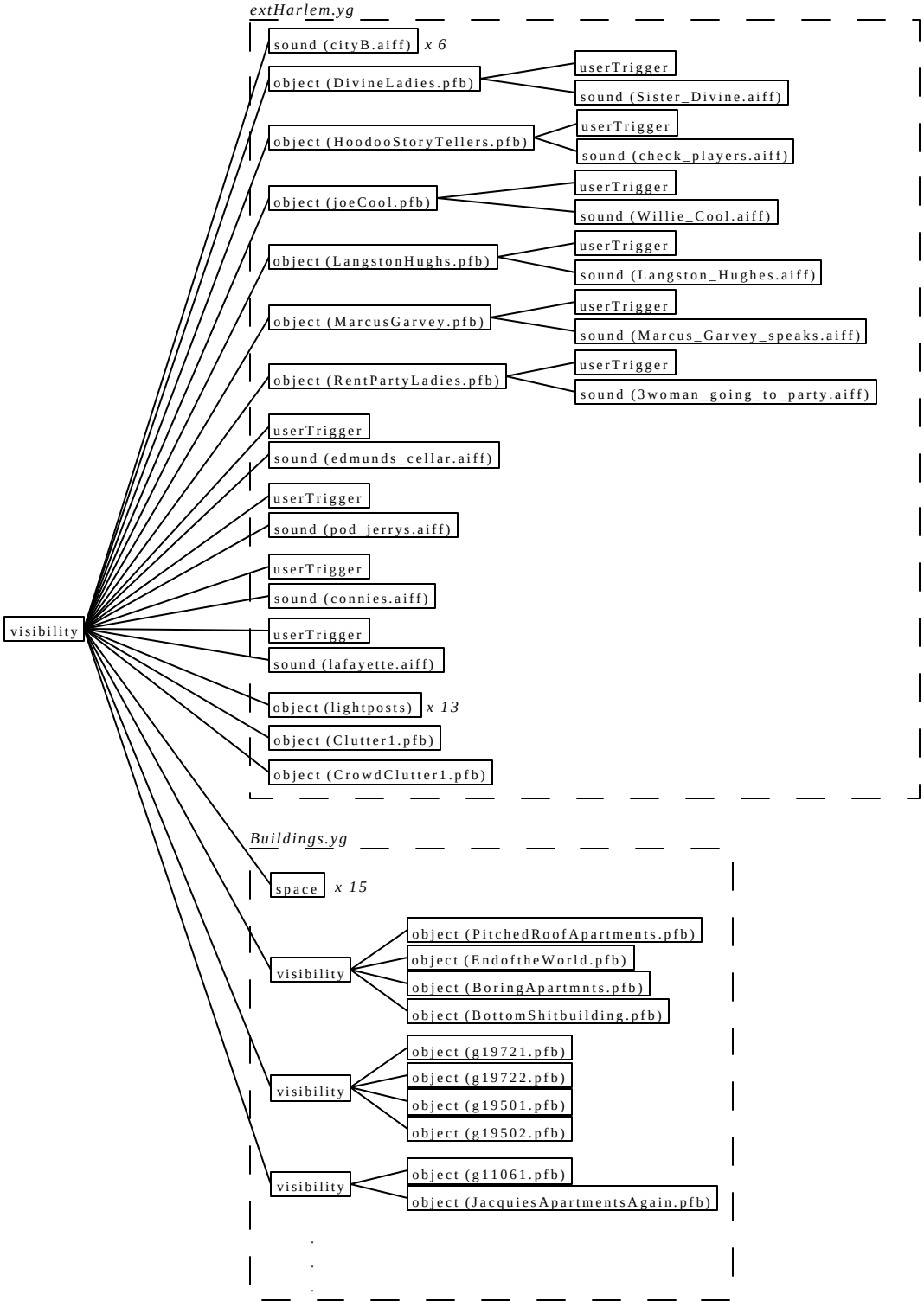


Figure 26. Virtual Harlem scene graph structure of the city

7.3. Composability

The development of Virtual Harlem and Miletus showed the ease of composing new environments in Ygdrasil. In particular, Virtual Harlem was very easily assembled. Starting from the already existing raw materials (the building models, sounds, and movies), most of the environment was created in a few days (an exact measure is impossible, as Virtual Harlem was worked on in parallel with the first draft of Ygdrasil itself.) It was built using some of the special nodes that were created for Shared Miletus. Miletus' extended navigator class was used in the avatars for participants. The Visibility node was also used, to improve the rendering performance. These, along with other, standard, 'added' node classes such as WandTrigger and CAVETracker, were all incorporated into Virtual Harlem via DSOs loaded at run-time, and did not require any special changes. In current work, the application is being expanded by adding nodes that use Miletus' PathFollower and FlyingCarpet classes.

In total, 209 of the 220 nodes in Virtual Harlem were of existing types, either the basic Ygdrasil classes or those taken from Shared Miletus. In the case of Miletus, only 376 of the 1025 nodes were of the pre-defined Ygdrasil types. However, the majority of the other nodes were of the Visibility (24 nodes) and LODobject (483 nodes) classes. These are both fairly general-purpose classes that can be expected to be re-used in many other applications, and should be included in the collection of standard Ygdrasil code. Given that, only about 20% of the Miletus environment required application-specific code. As the collection of node classes expands with further applications, even more re-use should be possible, and the large majority of any new virtual environment will be able to be assembled quickly from existing classes.

As further verification of its value in composing large worlds, Ygdrasil is currently being used as the basis of a networked group show for the 2001 Ars Electronica Festival. The overall VE for this show is an assembly of eleven different environments created by roughly twenty artists and programmers. Similar shows have been created in the past, but they involved a great deal of struggle to get different pieces of code, that were developed separately, to work together without serious conflict. In the new show, composing the total environment has been smooth and involved simply linking together the various pieces in a top-level scene file.

7.4. Networking Problems

In developing and testing the first draft of Ygdrasil for iGrid 2000, two major problems were encountered in the networking implementation.

The first problem was deadlocking that would occur when multiple clients joined a shared world. This appeared to be due to the use of CAVERNsoft's TCP-based shared database classes. In the CAVERN database, a single TCP/IP stream is used for communications in order to guarantee that all clients receive all messages, and thus keep their databases consistent. However, an a Ygdrasil environment, as a new client joins, it starts to request and receive many packets describing the scene graph. At the same time, the other clients, who are broadcasting the data for the existing scene, begin to request the new client's information. In a complex environment (anything more than a couple avatars and a handful of static objects), these many requests and data packets would begin to back up, and the clients ended up deadlocked, waiting for information from each other before they can send out their own new packets.

To resolve this, the CAVERNsoft database was replaced by a similar one based on UDP/IP. With this implementation, if any client's incoming messages get backed up, new ones are lost, but the host that

sent them is still able to continue. For data that really needs to be received, such as the type of a particular node, if it is not received after a certain amount of time, a new request is sent, thus avoiding most problems due to lost packets. The latest version of CAVERNsoft includes unreliable (i.e. UDP) updates in its database class, so this may also solve the problem.

The second problem was similar to the so-called 'long fat pipe' issue in TCP communications (Stevens, 1998). In TCP, this problem occurs when a network has a high latency and a relatively small TCP window. A small window means that a computer can only send a small amount of data (often 64 kilobytes) before it must wait for an acknowledgment that the remote computer has successfully received the data. If it takes a long time for this to come back acknowledgment (tens of milliseconds or more), then only a few windows of data can be transmitted each second.

In the first draft of Ygdrasil, although TCP/IP was not used, the creation of nodes in the scene graph was effectively serialized. Whenever the program first learned of a new node, usually when it appeared in another node's list of children, that node's local proxy had to be created. To do this, Ygdrasil would send a request for the node's class name, and wait for the response before being able to create the proxy. Until the response is received, nothing else could happen. If there were many nodes in the scene graph, and the time to receive a database response was slow, this would add up and result in it taking a long time to build the local version of the scene graph.

At iGrid 2000, the round-trip time between Yokohama and Chicago, as measured by the *ping* utility, was 150 milliseconds. For actual traffic, such as database requests and responses, the round-trip time would be greater than that. As a result, starting up the Shared Miletus application, with its 1025 scene

graph nodes, took roughly 10 minutes. This is much slower than was desired[1]. In the second draft of Ygdrasil, this issue is being addressed by making the shared scene graph creation more asynchronous. Whenever a new node is learned of, its type is requested, but a dummy proxy is created and the program continues on, using the dummy proxy until it can be replaced by one of the correct type. This dummy proxy is still a scene graph node, so it knows to immediately request its list of children, so that these nodes can also start to be created, even before the creation of the first node's real proxy has been completed. This should result in startup times that are proportional to the depth of the scene graph, rather than the number of nodes – i.e. an O(log N) time rather than O(N).

7.5. Performance Tests

Miletus and Virtual Harlem demonstrated the general success (and problems) of using Ygdrasil in creating real, functional shared environments. The following are the results of a series of more restricted experiments intended to identify the actual performance costs of using the system.

The experiments used a virtual world that consisted of between 1 and 500 simulated user avatars (Figure 27). Each simulated user was equivalent to a typical, real user's avatar; it had a top level transformation node for its navigation, three transformations below that, with attached models for the head, hand, and body positions. The four transformations were 'Spinner' nodes, a new node class defined in a DSO that continuously rotates in place around an axis. In this way, each avatar was constantly moving, producing the same amount of database updates and network traffic as an actual user or a highly dynamic, autonomous object in a world. Because of some of the network limitations

[1] In an ideal world, the entire 228 Mbyte database could have been sent over the 100 Mbit APAN connection in 19 seconds. In this case, the 10 minutes was just for the scene graph description, a very small fraction of the full database.

encountered with multiple users (as described above in section 7.4 and below in 7.5.3), it is unlikely that an actual Ygdrasil application will involve hundreds of users in the near future. However, these simulated users can also represent many dynamic transformation nodes for autonomous objects in a scene; as the statistics for The Thing Growing (Table V) show, several hundred dynamic objects can easily occur in current applications.

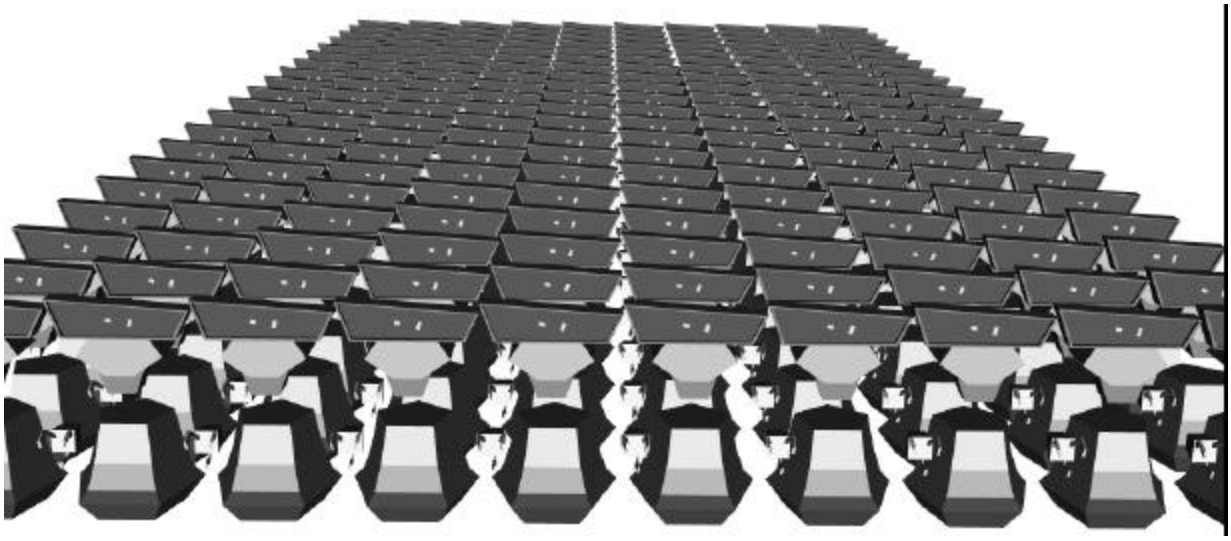


Figure 27. An array of a few hundred simulated users

The standalone tests were run on a SGI Onyx/IR with 4 194 MHz R10000 processors and 512 megabytes of memory, running IRIX 6.5.7 and Performer 2.2.7. The distributed test used the Onyx and 3 SGI O2s (180 MHz R5000 processor, 128 Mbytes RAM) for the clients, and a SGI Indigo2 Impact (195 MHz R10000 processor, 288 Mbytes RAM) for the reflector; all hosts had 10BaseT Ethernet connections.

All of the tests used single, 64x64, monoscopic graphics windows, rather than a full CAVE or ImmersaDesk display, so that the rendering stage of the program took a negligible amount of time and did not affect the overall speed of the tests – my interest here was in measuring how fast the application stage could update the world. The application timing information was collected using Performer's statistics-gathering functions; these functions record high-resolution timestamps at the beginning and ending of each stage's update, so that the time spent actually performing calculations can be measured. Performer's update loop is synchronized with the double-buffered graphics, which are in turn synchronized with the video display; looking at just the frame rate would therefore have given quantized, imprecise timing information. The network bandwidth was measured with the system utility *netstat*, which directly accesses IRIX kernel memory to report the amount of traffic on each interface during a 1 second sampling period.

7.5.1. Non-networked Scene Updates

The first experiment evaluated the basic cost of using Ygdrasil without networking. For comparison, a simple Performer/CAVElib application was written that created the same basic environment as the Ygdrasil test world. This 'straight Performer' program constructed an equivalent scene graph, saved pointers to all of the transformations' pfDCSs in an array, and updated all of these pfDCSs on each frame. It thus represented the theoretical best possible implementation of this particular world. Figure 28 shows the results of this test, with the average time taken for each frame's update, as the number of avatars varied from 5 to 500 in steps of 5.

As can be seen, the original Ygdrasil program took about 3.5 times as long as the Performer program for its update[1]. The additional two lines in the plot (labeled "no db" and "static") show the results of modified versions of the Ygdrasil test, for the purpose of isolating the update costs. The "no db" line comes from running a modified version of the program. In the original program, each Spinner node's application update would compute a new angle of rotation, and pass that to the function Transform::setOrientation; setOrientation would in turn pass the new value to the Transform node's pfDCS, and also store the new matrix value in the shared database (even though the shared database was not connected to any other networked hosts). In the modified program, the call to store the matrix in the database was eliminated. This significantly improved the performance. However, the database cost is still relevant when running a networked application.

The "static" line in Figure 28 shows the performance of a modified scene. The Spinners that made up the user avatar were all replaced with ordinary Transform nodes, which do not perform any per-frame updates. This therefore shows the cost of just traversing the Ygdrasil scene graph and calling the application-update function for every node. The straight Performer program avoided this cost by storing pointers to all its pfDCSs in a simple array. This sort of optimization is not as applicable to a general purpose program for more realistic applications, and so the traversal cost, although it might be reduced, is more or less unavoidable.

---

[1] One positive note is that at least all of the tests show a linear relationship between the number of nodes and the update time.

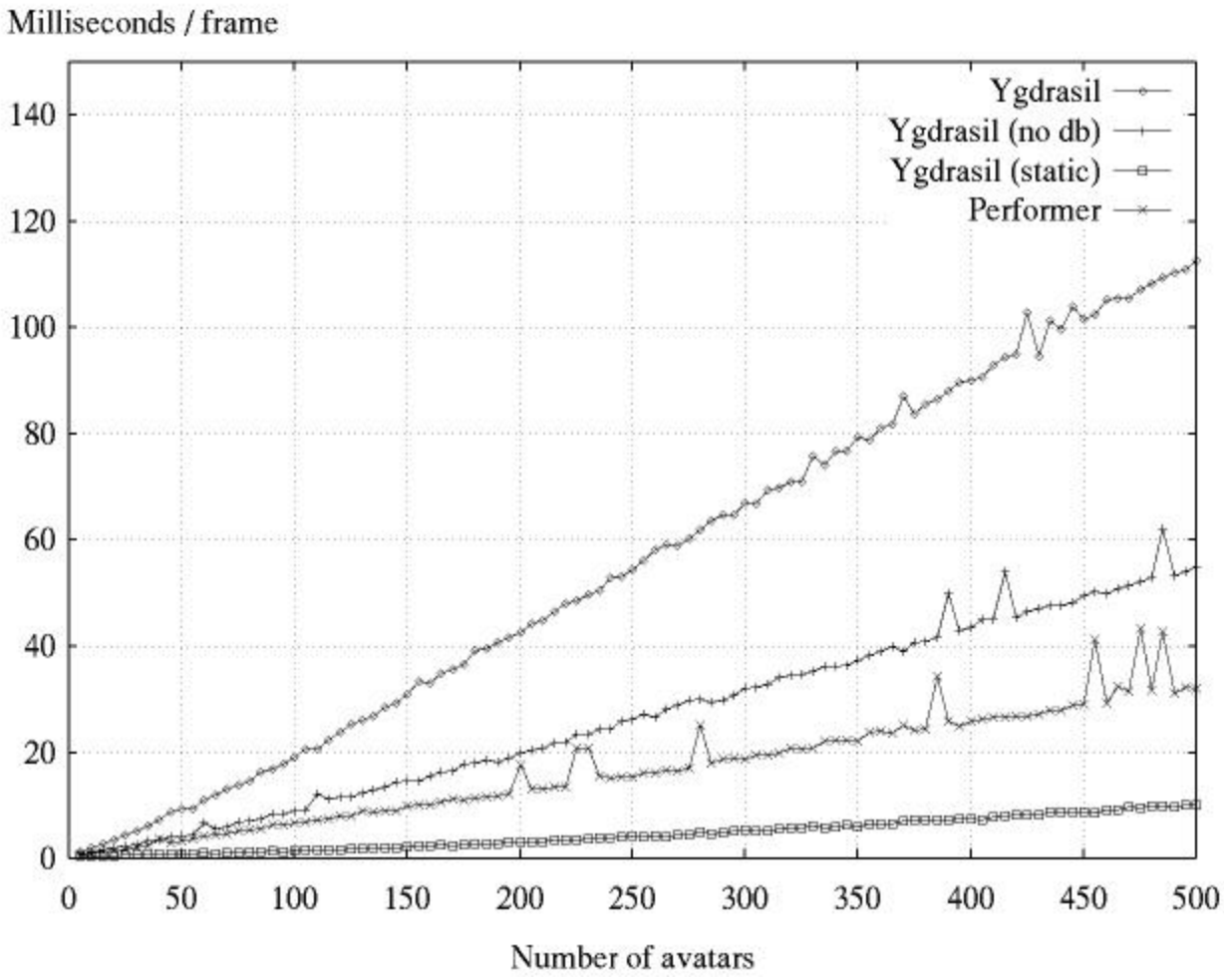


Figure 28. Update speed of Ygdrasil vs. straight Performer, with many dynamic avatars

Figure 29 shows the exact numbers measured for the case of 250 avatars (1000 transformations). From these measurements, we can estimate the time spent in the different steps of the Spinner update – 4.1 milliseconds (i.e. 4.1 microseconds per node) for the application traversal, 21.8 ms for computing the new rotation and passing it to the pfDCS, and 28.4 ms for storing the matrix in the database. The 21.8 ms rotation update is still not as good as the 15.4 ms that the same operation took in the straight Performer program, but it is reasonably close. Figure 30 shows the calculated transformation-update time for all of the test cases; the results are consistent with those for the 250 avatar case.

Measured data	
Test	**Update time**
Performer	15.4 ms
Ygdrasil	54.3 ms
Ygdrasil (no db)	25.9 ms
Ygdrasil (static)	4.1 ms

→

Calculated timing	
Ygdrasil Step	**Time**
App traversal	4.1 ms
Performer setRot	21.8 ms
Database update	28.4 ms

Figure 29.Breakdown of timing data for 250 avatars

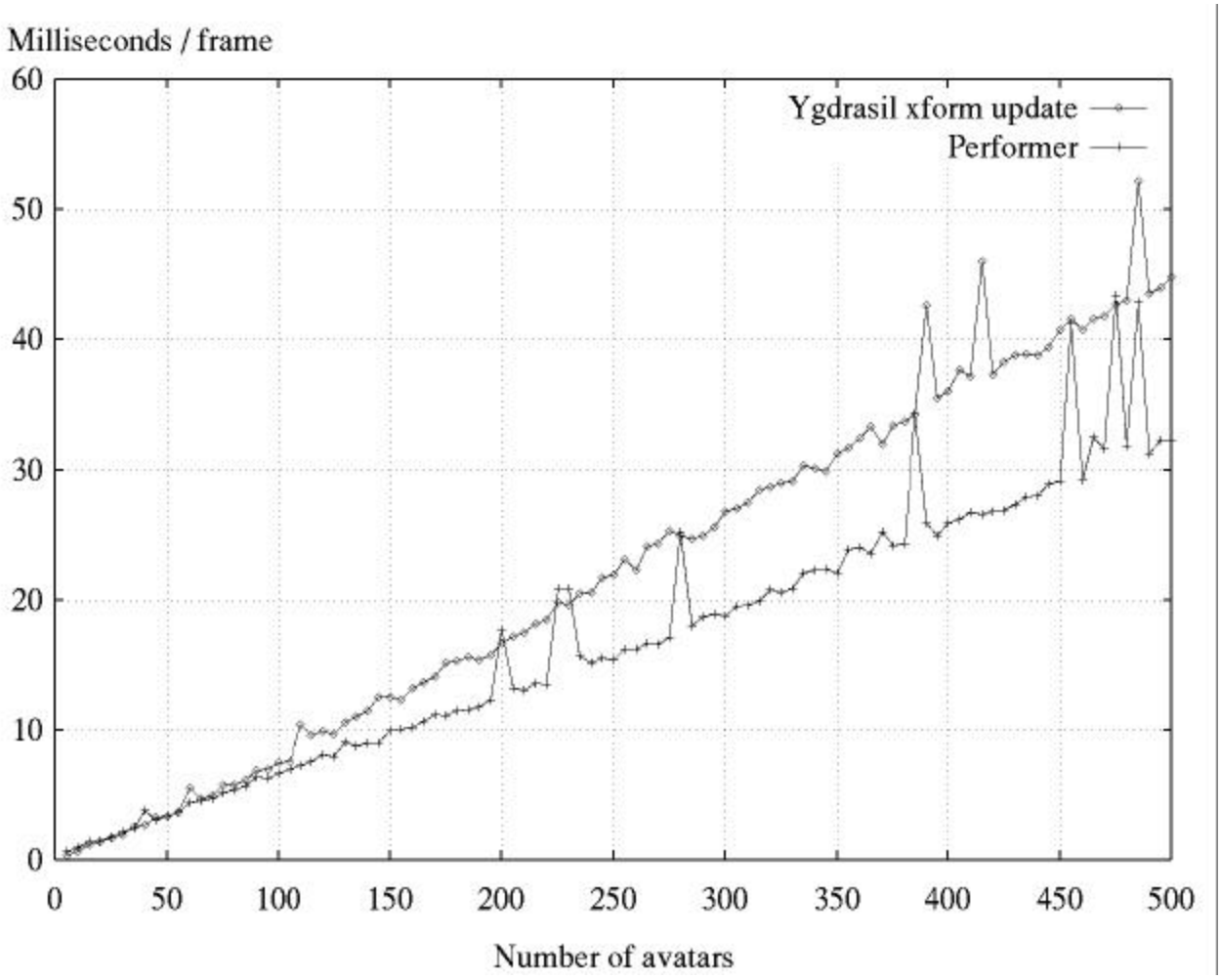


Figure 30. Speed of updates to dynamic transformation nodes

7.5.2. Networked Updates on a Single Host

The second set of experiments measured the performance of the Ygdrasil program when networking was enabled. In this case, there was only one host in the shared world; all of its changing transformation matrices were sent over the network to the reflector machine, but there were no other clients to receive the data or to send other new data to the host. Hence, the basic difference between this test and the first test (with the database enabled) is that each node's update also causes the new matrix data to be packed into a UDP message that is then transmitted to the reflector.

Ideally, the sending and receiving of network packets would be handled by a separate thread, so as to not affect the performance of the main application process, unless traffic became very heavy. Unfortunately, the implementation of pthreads on SGIs is still not completely reliable when used with Performer on multi-processor Onyxes. It can work, but often requires multiple attempts to get an application to start without Performer immediately dying. This is not acceptable for serious use, and so the current version of Ygdrasil runs its networking stage in the main process, along with the scene updates. Hopefully, future IRIX releases will solve this problem.

Figure 31 shows the time taken to perform each frame's update when Ygdrasil was run in networked and standalone modes, with the number of avatars ranging from 10 to 100. However, the validity of these results beyond 20 avatars is uncertain; after that point, the numbers reported by netstat indicated that the 10BaseT network interface was being used at full capacity, and so most of the UDP packets being sent were actually dropped, and would not have been received by any other hosts.

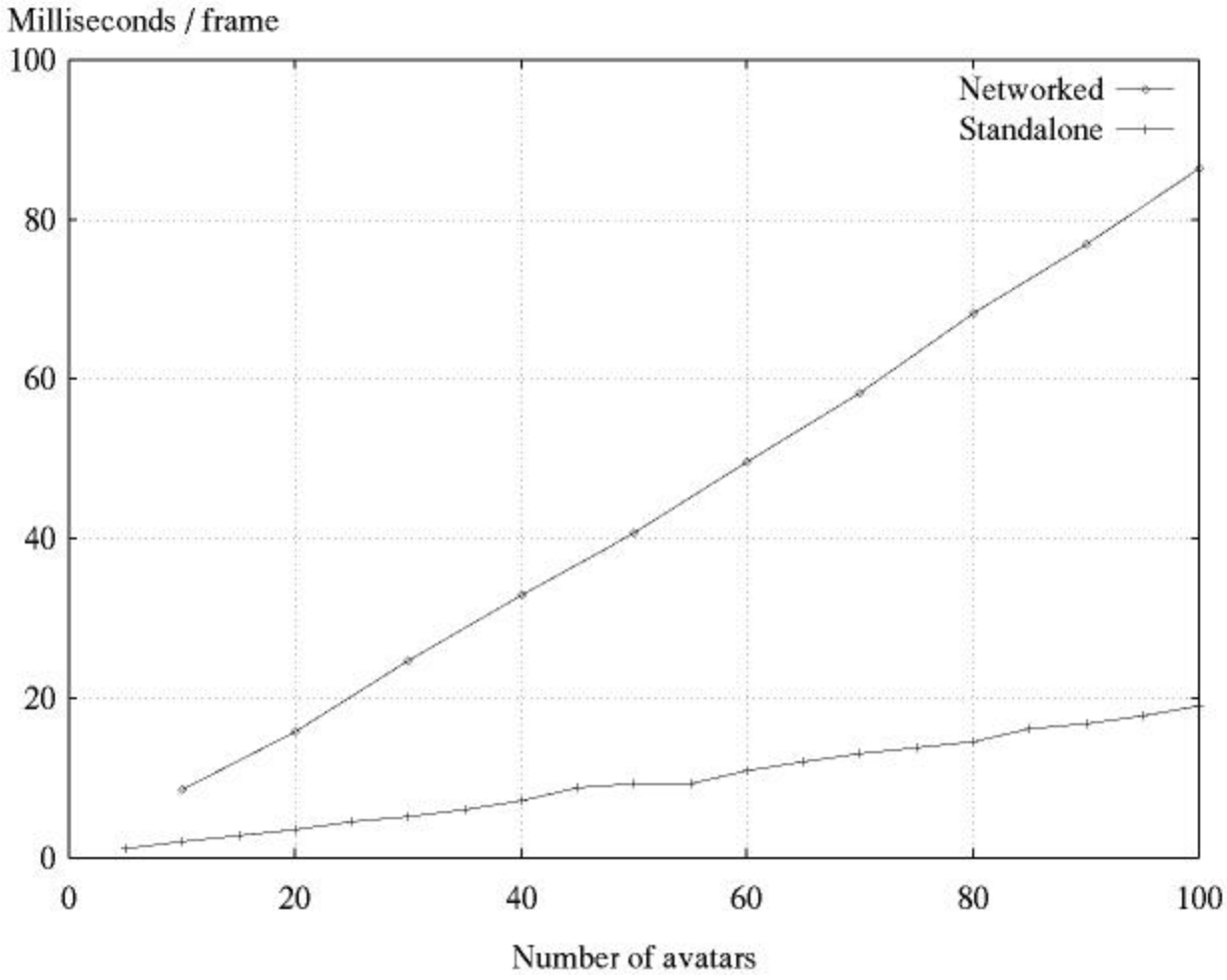


Figure 31. Update speed of networked vs. standalone Ygdrasil

7.5.3. Bandwidth use

The third experiment involved running individual avatars on multiple hosts, and measuring the amount of incoming and outgoing bandwidth on the host running the reflector with netstat. The results for one to four hosts (not counting the workstation running the reflector) are shown in Table VIII; after four hosts, the network was overloaded.

TABLE VIII

NETWORK BANDWIDTH WITH MULTIPLE HOSTS

Number of hosts	Incoming bandwidth	Outgoing bandwidth
1	66,000 bytes/sec	0
2	127,000 bytes/sec	116,000 bytes/sec
3	158,000 bytes/sec	283,000 bytes/sec
4	400,000 bytes/sec	400,000 bytes/sec

These numbers clearly demonstrate that the simplistic approach to the shared database, as currently used, will need to be improved. Although it makes the implementation of the shared scene graph straightforward, it does not scale well at all. The basic problem is that it involves large numbers of small, independent updates, sometimes being sent very frequently.

The client programs in the multi-host test were all running at 60 – 75 frames per second. On each frame, all four of the avatar's transformations were updated, and thus caused new matrices to be stored in

the database. Storing each matrix results in a UDP packet being broadcast with the new information. The size of the matrix itself is 64 bytes (16 floating point numbers). The packet contains header information identifying it as a database store operation and the size of the data; in this implementation the header is 12 bytes. It also must contain the database key to indicate where this data is to be stored; the key is the name of the node, plus the string "matrix"; on average, this can require another 32 bytes. Furthermore, below the CAVERNsoft level there is the overhead of UDP/IP itself; every IPv4 packet includes a header of at least 20 bytes (Stevens, 1998)[1]. This adds up to at least 128 bytes per packet. Each O2 broadcasting 4 matrices at 75 frames per second would then be expected to produce 38,400 bytes/second of traffic or more. The measured traffic was close to twice that, so apparently there is even more overhead hidden somewhere, but even at 38 Kbytes/sec we could not expect to handle more than 6 hosts on a 10 Mbit network. This is because of the unicast nature of the database. For an update, each client sends a single packet to the reflector, which in turn must send a copy of the packet to every other client. Six hosts would thus result in 36 streams of data going in and out of the reflector.

There are several things that can be done to improve the bandwidth requirements. In short, they are compressing, aggregating, and throttling. The actual data that is sent could be compressed in size; this is especially true for transformation nodes, which will rarely make use of a full 4x4 matrix. In a best case scenario, the matrix data could be replaced by a position and Euler angles, and the individual numbers could be reduced to 2 bytes each (this is the typical resolution of tracking data from devices like the Flock of Birds; it's not likely to be sufficient, however, for navigation data). Rather than sending character strings for the name of the node and the specific key ("matrix"), integer ID numbers could be used.

---

[1] In the future, as systems move to IPv6, this overhead will be even larger (at least 40 bytes).

Allowing for very large worlds, then, this integer key could be 8 bytes. Aggregating the data would mean collecting several database updates into a single packet, in order to reduce the overhead due to the network protocol. In the simple avatar test case, we would probably want to limit this to collecting all four matrix updates into one packet, since grouping data from more than one frame would not be useful once the data is received by remote hosts. Finally, the number of updates per second that particularly dynamic nodes, such as avatar transformations, could be throttled back. In order to maintain reasonably smooth animation speeds, as seen by the remote clients, we would not want to reduce the update rate to less than 10 per second. This rate might be reduced even further by using a dead-reckoning approach, as in DIS. One DIS experiment found that the simulators involved averaged 0.17 updates per second for tanks, and 1 update per second for aircraft (Pullen and Wood, 1995). However, it's uncertain how well dead-reckoning would work here; the movements of a person's head or arm are usually not as simple and predictable as the movement of a tank. Theoretically, together the various improvements could reduce the bandwidth needed to about 1 Kbyte/second (from 38.4 Kbytes/second).

The 'unicast explosion' of bandwidth will still be a problem for large-scale shared environments. If each host produced only 1 Kbyte/second of data, 30 hosts would still saturate a 10 Mbit network. This calculation assumes nothing but avatar data is being sent. In many real applications, there is also streaming audio, so that the people can speak to one another. Audio will require at least 8 Kbytes/second, when the person is talking. Given that, about 16 hosts, with perhaps 4 people speaking at any one time, would be the safe limit for a 10 Mbit network. This estimate matches with the largest CAVERNsoft collaboration demo reported thus far (Johnson et al., 1998). In order to enable significantly larger environments in the future, multicast and area-of-interest algorithms will need to be added.

7.5.4. Final Comments

At the root of the overhead in both the application update times and the network use is Ygdrasil's goal of being a generic system. It is intended to flexibly handle many different configurations of virtual worlds and of users, built on a common scene graph API. Although the scene graph mechanism makes it easy to assemble a world out of small pieces, this also means that it becomes easy to assemble very large scene graphs – the Thing Growing and Shared Miletus are both examples of this. In the end, the added overhead of traversing and updating these large scenes is a trade-off, against the work it would otherwise take to program completely optimal versions of the applications.

Most of the driving applications for CAVERNsoft are scientific and engineering applications involving large data. For example, the volume visualizer CIBRview is used to examine multi-gigabyte datasets (Park et al., 2000); besides transmitting the large, monolithic volume slices, it only needs to share avatar data for a handful of users and a very small amount of state information, such as the current frame number for a time-sequence playback. Furthermore, most CAVERNsoft avatars assume a simple, fixed model of a user – a head and one wand, with position and orientation in world coordinates. As a result, the network usage of these CAVERNsoft driving applications is significantly different than that of the typical Ygdrasil application.

8. CONCLUSION

A fully composable networked virtual reality system would make it possible to construct a virtual object with programmed behaviors once, to introduce this object into any existing virtual world, and to have it immediately be able to interact with users and other objects in that world. A good system would also make it simple to define the object's behavior, by describing it at a high level or using a set of powerful tools. The Ygdrasil system introduced in this dissertation is one step toward that ultimate goal.

Ygdrasil, and its predecessor XP, have specifically explored two major concepts for composing virtual worlds – scripting and a shared scene graph. The scripting layer simplifies the quick combining of existing components into new applications. A notable difference between Ygdrasil's scripting and that normally found in VR toolkits is its basis in the scene graph; rather than being a collection of procedures, an Ygdrasil script is thought of as a description of the virtual world's scene. The purpose of the shared scene graph is to provide a standard interface for all components. Further, it provides a programming model where every object is controlled by one participant in the shared environment, and all other participants can interact with the object in a simplified form, without needing to know about the implementation of its behavior.

The use of Ygdrasil in Shared Miletus and Virtual Harlem has shown that it is useful for constructing significant networked worlds. However, it also showed that developers will always come up with new requirements, requirements that sometimes make it not as easy as hoped for remote participants to be unaware of all objects' implementation details; the system will need to remain flexible to deal with this fact. Furthermore, the issue of bandwidth use will have to be addressed before Ygdrasil can be used in widely distributed worlds intended for many simultaneous participants.

Future work that should be considered includes:

- Reducing the amount of network bandwidth used by a client. Prediction algorithms, similar to DIS's dead-reckoning, should be investigated.
- Incorporating area-of-interest and other algorithms to filter what parts of the shared database a client receives. This could eventually take the form of a subscription interest expression for each node or client, similar to that described in Singhal and Zyda.
- Distributing new executable code to remote clients automatically, in a secure manner.
- Enabling the dynamic replacement of node classes in a running application. This would allow a rapid-prototyping style of development, where code can be revised and tested in an otherwise stable virtual world.

The origin of Ygdrasil's name in the sometimes fatalistic Norse mythology reflects some of the attitude that should be taken toward the use and development of the system. Many interesting things will be done with Ygdrasil, but some day it should come to end, to be replaced by something newer and better.

CITED LITERATURE

Anstey, J., Pape, D., and Sandin, D:. The Thing Growing: Autonomous Characters in Virtual Reality Interactive Fiction. In *Proceedings of IEEE Virtual Reality 2000*, New Brunswick, NJ, March 2000.

Benford, S., Bowers, J., Fahlen, L. E., Greenhalgh, C., Mariani, J., and Rodden, T.: Networked Virtual Reality and Cooperative Work. *Presence: Teleoperators and Virtual Environments* 4 (4), Fall 1995, 364-386.

Brown, J. R., van Dam, A., Earnshaw, R., Encarnação, J., Guedj, R., Preece, J., Shneiderman, B., and Vince, J.: Human-Centered Computing, Online Communities, and Virtual Environments. *IEEE Computer Graphics and Applications* 19 (6), Nov/Dec 1999, 70-74.

Brown, M. D., DeFanti, T. A., McRobbie, M. A., Verlo, A., Plepys, D., McMullen, D. F., Adams, K., Leigh, J., Johnson, A. E., Foster, I., Kesselman, C., Schmidt, A., and Goldstein, S. N.: The International Grid (iGrid): Empowering Global Research Community Networking Using High Performance International Internet Services. In *Proceedings of INET '99*, San Jose, CA, June 1999, 3-9.

Calvin, J., Dickens, A., Gaines, B., Metzger, P., Miller, D., and Owen, D.: The SIMNET Virtual World Architecture. In *Proceedings of IEEE Virtual Reality Annual International Symposium (VRAIS '93)*, Seattle, WA, Sept 1993, 450-455.

Capps, M., McGregor, D., Brutzman, D., and Zyda, M.: NPSNET-V: A New Beginning for Dynamically Extensible Virtual Environments. *IEEE Computer Graphics & Applications*, 20 (5), Sep/Oct 2000, 12-15.

Carlsson, C., and Hagsand, O.: DIVE - A Multi-User Virtual Reality System, In *Proceedings IEEE Virtual Reality Annual International Symposium (VRAIS '93)*, Seattle, WA, Sept 1993, 394-400.

Carlsson, C., and Hagsand, O.: DIVE - A Platform for Multi-User Virtual Environments. *Computers and Graphics*, 1993, 663-669.

Carter, B. Virtual Harlem. *SIGGRAPH '99 Conference Abstracts and Applications*, Los Angeles, CA, Aug 1999, 103.

Codella, C. F., Jalili, R., Koved, L., and Lewis, J. B.: A Toolkit for Developing Multi-User, Distributed Virtual Environments. In *Proceedings of IEEE Virtual Reality Annual International Symposium (VRAIS '93)*, Seattle, WA, Sept 1993, 401-407.

Cruz-Neira, C., Sandin, D. J., and DeFanti, T. A.: Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE. In *Proceedings of SIGGRAPH '93 Computer Graphics Conference*, Anaheim, CA, August 1993, 135-142.

Cruz-Neira, C.: Virtual Reality Based on Multiple Projection Screens: The CAVE and its Applications to Computational Science and Engineering. PhD Dissertation, University of Illinois at Chicago, Chicago, 1995.

Czernuszenko, M., Pape, D. Sandin, D., DeFanti, T., Dawe, G., and Brown, M.: The ImmersaDesk and Infinity Wall Projection-Based Virtual Reality Displays, *Computer Graphics*, 31 (2), May 1997, 46-49.

Das, S., DeFanti, T. A., and Sandin, D. J.: An Organization for High-Level Interactive Control of Sound. In *Proceedings of the International Conference on Auditory Display '94*, Santa Fe, NM, February 1994.

Division Ltd.: *dVISE for UNIX Workstations – User Guide*. 1995.

Dolinsky, M.: Virtual Environment as Rebus, In *Proceedings of Consciousness Reframed, International CAiiA Research Conference*, Newport, Wales, July 1998.

Fischnaller, F., and Singh, Y.: Multi Mega Book. *FleshFactor: Ars Electronica Festival 97*, Springer, Vienna, 1997.

Galyean, T.: Guided Navigation of Virtual Environments. In *Proceedings of 1995 Symposium on Interactive 3D Graphics*, Monterey, CA, April 1995, 85-92.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.

Ghazisaedy, M., Adamczyk, D., Sandin, D. J., Kenyon, R., and DeFanti, T. A.: Ultrasonic Calibration of a Magnetic Tracker in a Virtual Reality Space. In *Proceedings of the IEEE Virtual Reality Annual International Symposium (VRAIS '95)*, Research Triangle Park, NC, March 1995.

Ghee, S., and Naughton-Green, J.: Programming Virtual Worlds. *ACM SIGGRAPH '95 Course Notes – Programming Virtual Worlds*, Orlando, FL, August 1995, 6-1 – 6-58.

Hagsand, O., Lea, R., and Stenius, M.: Using Spatial Techniques to Decrease Message Passing in a Distributed VE System. In *Proceedings VRML 97 - Second Symposium on the Virtual Reality Modeling Language*, Monterey, CA, Feb 1997, 7-16.

Hörtner, H., Maresch, P., Lindinger, C., Pomberger, G., and Pinger, H.: 3D Simulation of Industrial Production Processes in a Virtual Continuous Casting Environment. *Laval Virtual 2001 - Virtual Reality International Conference*, Laval, France, May 2001.

Imai, T., Johnson, A. E., Leigh, J., Pape, D. E., and DeFanti, T. A.: The Virtual Mail System, In *Proceedings of IEEE Virtual Reality '99*, Houston, TX, March 1999.

IEEE (Institute of Electrical and Electronics Engineers): International Standard, ANSI/IEEE Std 1278-1993. Standard for Information Technology, Protocols for Distributed Interactive Simulation. 1993.

ISO (International Standards Organization): The Virtual Reality Modeling Language. International Standard ISO/IEC 14772-1:1997.

Johnson, A., Leigh, J., and Costigan, J.: Multiway Tele-Immersion at Supercomputing 97. *IEEE Computer Graphics & Applications*, 18 (4), Jul/Aug 1998, 6-9.

Johnson, A., Moher, T., Ohlsson, S., and Gillingham, M.: The Round Earth Project: Deep Learning in a Collaborative Virtual World, In *Proceedings of IEEE Virtual Reality '99*, Houston, TX, March 1999.

Johnson, A., Moher, T., Ohlsson, S., and Gillingham, M.: The Round Earth Project - Collaborative VR for Conceptual Learning, *IEEE Computer Graphics and Applications* 19 (6), Nov/Dec 1999, 60-69.

Kindratenko, V.: Calibration of Electromagnetic Tracking Devices, *Virtual Reality: Research, Development, and Applications*, Vol. 4, 1999, 139-150.

Kogler, P., and Pomassl, F.: CAVE. *LifeScience: Ars Electronica 99*, Springer, Vienna, 1999, 364-369.

Leigh, J., Johnson, A. E., and DeFanti, T. A.: CAVERN: A Distributed Architecture for Supporting Scalable Persistence and Interoperability in Collaborative Virtual Environments, *Journal of Virtual Reality Research, Development and Applications*, 2 (2), Dec 1997, 217-237.

Macedonia, M. R., Zyda, M. J., Pratt, D. R., Barham, P. T., and Zeswitz, S.: NPSNET: A Network Software Architecture for Large-Scale Virtual Environments, *Presence: Teleoperators and Virtual Environments* 3 (4), Fall 1994, 265-287.

Macedonia, M. R., Zyda, M. J., Pratt, D. R., Brutzman, D. P., and Barham, P. T.: Exploiting Reality with Multicast Groups: A Network Architecture for Large-Scale Virtual Environments, In *Proceedings IEEE Virtual Reality Annual International Symposium (VRAIS '95)*, Research Triangle Park, NC, March 1995, 2-10.

Normand, V.: The COVEN Project: Exploring Applicative, Technical, and Usage Dimensions of Collaborative Virtual Environments, *Presence: Teleoperators and Virtual Environments* 8 (2), April 1999, 218-236.

Ousterhout, J. K.: Scripting: Higher-Level Programming for the 21st Century. *IEEE Computer*, 31 (3), March 1998, 23-30.

Pape, D. A Hardware-Independent Virtual Reality Development System, *IEEE Computer Graphics and Applications*, 16 (4), July 1996, 44-47.

Pape, D., Imai, T., Anstey, J., Roussou, M., and DeFanti, T.: XP: An Authoring System for Immersive Art Exhibitions. In *Proceedings of the International Conference on Virtual Systems and Multimedia (VSMM '98)*, Gifu, Japan, November 1998, 528-533.

Pape, D., Sandin, D., and DeFanti, T.: Transparently Supporting a Wide Range of VR and Stereoscopic Display Devices, In *Proceedings of SPIE Vol. 3639 Stereoscopic Displays and Virtual Reality Systems VI (The Engineering Reality of Virtual Reality 1999)*, San Jose, CA, Jan 1999, 346-353.

Pape, D., D'Souza, S., Anstey, J., DeFanti, T., Roussou, M., and Gaitatzes, A. Shared Miletus: Towards a Networked Virtual History Museum. In *Proceedings of the International Conference on Augmented, Virtual Environments and Three-Dimensional Imaging*, Mykonos, Greece, May 2001.

Park, K., Cho, Y., Krishnaprasad, N., Scharver, C., Lewis, M., Leigh, J., and Johnson, A.: CAVERNsoft G2: A Toolkit for High Performance Tele-Immersive Collaboration, In *Proceedings of the Symposium on Virtual Reality Software and Technology 2000*, Seoul, Korea, Oct 2000.

Pausch, R., Snoddy, J., Taylor, R., Watson, S., and Haseltine, E.: Disney's Aladdin: First Steps Toward Storytelling in Virtual Reality, In *Proceedings of SIGGRAPH '96*, New Orleans, LA, Aug 1996, 193-203.

Pope, A. BBN Report No. 7102. The SIMNET Network and Protocols. BBN Systems and Technologies, Cambridge, Massachusetts.

Pullen, J. M., and Wood, D. C.: Networking Technology and DIS. In *Proceedings of the IEEE*, 83 (8), August 1995, 1156-1167.

Robinett, W., and Holloway, R.: Implementation of Flying, Scaling, and Grabbing in Virtual Worlds. *Computer Graphics* 25 (2), 1992, 189-192.

Rohlf, J., and Helman, J.: IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics, In *Proceedings of SIGGRAPH '94 Computer Graphics Conference*, Orlando, FL, August 1994, 381-395.

Roussos, M., and Bizri, H. Mitologies: Medieval Labyrinth Narratives in Virtual Reality. In *Proceedings of the 1st International Conference on Virtual Worlds*, Paris, France, July 1998.

Roussou, M.: Incorporating Immersive Projection-based Virtual Reality in Public Spaces. In *Proceedings of the 3rd International Immersive Projection Technology Workshop*, Stuttgart, Germany, May 1999, 33-39.

Sense8 Corporation. *WorldToolKit Reference Manual, Release 8*. 1998.

Sense8 Corporation. *WorldUp Users Guide, Release 4*. 1998.

Sense8 Corporation. *World2World Release 1 Technical Overview*. 1998.

Singh, G., Serra, L., Png, W., Wong, A., and Ng, H.: BrickNet: A Software Toolkit for Network-Based Virtual Worlds. *Presence: Teleoperators and Virtual Environments* 3 (1), Winter 1994. 19-34.

Singh, G., Serra, L., Png, W., Wong, A., and Ng, H.: BrickNet: Sharing Object Behaviors on the Net, In *Proceedings IEEE Virtual Reality Annual International Symposium (VRAIS '95)*, Research Triangle Park, NC, March 1995, 19-25

Singhal, S., and Zyda, M.: *Networked Virtual Environments: Design and Implementation*, ACM Press, New York, New York, July 1999.

Stevens, W. R.: *Unix Network Programming, Volume 1: Networking APIs: Sockets and XTI*, Prentice Hall PTR, Upper Saddle River, NJ, 1998.

Strauss, P. S., and Carey, R.: An object-oriented 3D graphics toolkit, In *Proceedings of SIGGRAPH '92 Computer Graphics Conference*, Chicago, IL, August 1992, 341-349.

Sturluson, S.: *Prose Edda*, Iceland, circa 1220.

Tramberend, H.: Avocado: A Distributed Virtual Reality Framework, In *Proceedings of IEEE Virtual Reality '99*, Houston, TX, March 1999.

UVa User Interface Group: Alice: Rapid Prototyping for Virtual Reality. *IEEE Computer Graphics and Applications*, 15 (3), May 1995. 8-11.

Watsen, K., and Zyda, M.: Bamboo - A Portable System for Dynamically Extensible, Real-time, Networked Virtual Environments. In *Proceedings IEEE Virtual Reality Annual International Symposium (VRAIS '98)*, Atlanta, GA, March 1998, 252-259.

APPENDICES

APPENDIX A

Ygdrasil Node Class Reference

CAVEKeyboard

Derived from: Node

Messages	*none*	

Events	keyA … keyZ, key0 … key9	occur whenever the user presses the corresponding key

CAVENavigator

Derived from: Navigator

Messages	teleport (*x y z*)	instantly transports the user to world coordinates (*x y z*)
	speed (*s*)	sets the user's maximum translation speed to *s* feet/sec
	rotspeed (*r*)	sets the user's maximum rotation speed to *r* degrees/sec
	collide ([*boolean*])	turns collision detection on or off. if no argument is given, true (on) is assumed
	collideRadius (*r*)	sets the radius used by the collision-detection algorithm to *r* feet
	toggleCollide	toggles collision detection between on and off
	fly ([*boolean*])	turns flying on or off. if no argument is given, true (on) is assumed
	toggleFly	toggles flying mode between on and off
	printNav	prints the current navigation position and orientation to standard-output

APPENDIX A (continued)

CAVETracker

Derived from: Transform

Messages	sensor (*num*)	tells the node to read CAVElib sensor number *num* for its position and orientation data. sensor 0 is the head, 1 is the wand, 2+ are any other sensors.

CAVEWand

Derived from: UserPart

Messages	*none*	

Events	button1, button2, button3, etc.	generated when the corresponding CAVE wand button is pressed

Environment

Derived from: Space

Messages	clip (*near far*)	sets the near and far clipping distances
	skyColor (*r g b*)	sets the background color. *r*, *g*, and *b* are floating point numbers between 0 and 1
	fog (off)	turns off fog
	fog (*type r g b onset opaque*)	turns on fog. *type* should be either "linear", "exp", or "exp2". *r g b* is the color of the fog (floating point numbers between 0 and 1). *onset* and *opaque* are the fog equation arguments for the starting distance of the fog and distance at which it is fully opaque

Head

Derived from: UserPart

Messages	*none*	

APPENDIX A (continued)

Light

Derived from: Node

Messages	on	turns the light on
	off	turns the light off
	toggle	toggles the light between on and off
	position (*x y z w*)	sets the light source's position / direction. if *w* is 0 (or is omitted), the light is an infinite, directional light; otherwise, it is a local light
	diffuse (*r g b*)	sets the diffuse color of the light
	ambient (*r g b*)	sets the ambient color of the light
	specular (*r g b*)	sets the specular color of the light
	attenuation (*c l q*)	sets the coefficients for the light's attenuation function
	spotDirection (*x y z*)	sets the direction for a spotlight
	spotCone (*exponent spread*)	sets the cone falloff and spread-angle parameters for a spotlight

Events	lightOn	generated when the light is turned on
	lightOff	generated when the light is turned off

Model

Derived from: Transform

Messages	file (*filename*)	loads Performer model *filename*
	wall ([*boolean*])	sets 'wall' flag (for collision detection) true or false. if no argument is given, true is assumed
	floor ([*boolean*])	sets 'floor' flag (for ground-following) true or false. if no argument is given, true is assumed
	draw ([*boolean*])	sets 'drawable' flag true or false. if no argument is given, true is assumed

Navigator

Derived from: Node

Messages	*none*	

APPENDIX A (continued)

Node

Messages	reset	resets node to its initial state
	resetTree	resets node and everything below it
	when (*event message [message2 …]*)	tells node to send *message* whenever *event* occurs
	addChild (*node*)	makes *node* a child of the calling node
	removeChild (*node*)	removes *node* from calling node's children
	debug (*flag*)	turns on debugging flag *flag*
	print (*string*)	prints *string* to standard output
	event (*eventName*)	tells node to generate event *eventName*
	signal (*signalName*)	tells node to set signal *signalName*

PointAtTrigger

Derived from: Node

Messages	distance (*d*)	sets the maximum distance from a wand to the trigger that will generate an event

Events	start part=*wandName* user=*userName*	generated when a user's wand begins pointing at the trigger. $part is the name of the wand node that generated the event, $user is the name of the user node that owns the wand
	act part=*wandName* user=*userName*	generated when a user's wand is pointing at the trigger and the wand's 'act' signal is set
	stop part=*wandName* user=*userName*	generated when a user's wand stops pointing at the trigger

Selector

Derived from: Node

Messages	select (*name*)	selects the child node named *name*
	selectNum (*num*)	selects child node number *num*. the first child of a node is number 0, the second is 1, etc.

APPENDIX A (continued)

SimpleBodyTracker

Derived from: Transform

Messages	doRotation ([*boolean*])	tells the body tracker to set its rotation around Z to the same as the user's head; if false, the rotation will be 0. if no argument is given, 'true' is assumed

Sound

Derived from: Space

Messages	file (*filename*)	tells the sound to use the sample file *filename*
	falloffDistance (*d*)	sets the falloff distance to *d*
	amplitude (*a*)	sets the maximum amplitude to *a*
	loop ([*boolean*])	sets whether the sound sample will loop infinitely when it is played. if no argument is given, 'true' is assumed
	play ([*filename* [*amplitude*]])	starts playing the sound. if *filename* is given, that sample file will be used. if *amplitude* is given, the maximum amplitude is set to that value
	stop	stops playing the sound

Events	startPlaying	generated when the sound starts playing, in response to a 'play' message
	stopPlaying	generated when the sound stops playing, either in response to a 'stop' message, or because the end of the sample file is reached

APPENDIX A (continued)

Space

Derived from: Node

Messages	volume (infinite)	makes the space an infinite volume
	volume (point *x y z*)	makes the space a point at position (*x y z*)
	volume (box *minX minY minZ maxX maxY maxZ*)	makes the space an axis-aligned box, ranging from the positions (*minX minY minZ*) to (*maxX maxY maxZ*)
	volume (sphere *x y z r*)	makes the space a sphere, centered at (*x y z*), with radius *r*
	volume (cylinder *bottomX bottomY bottomZ topX topY topZ r*)	makes the space a cylinder, with a center axis running from (*bottomX bottomY bottomZ*) to (*topX topY topZ*), and radius *r*

StaticModel

Derived from: Node

Messages	file (*filename*)	loads Performer model *filename*
	wall ([*boolean*])	sets 'wall' flag (for collision detection) true or false. if no argument is given, 'true' is assumed
	floor ([*boolean*])	sets 'floor' flag (for ground-following) true or false. if no argument is given, 'true' is assumed
	draw ([*boolean*])	sets 'drawable' flag true or false. if no argument is given, 'true' is assumed

Switch

Derived from: Node

Messages	on	turns the switch on
	off	turns the switch off
	toggle	toggles the switch between off and on

Events	switchOn	generated when the switch is turned on
	switchOff	generated when the switch is turned off

APPENDIX A (continued)

Transform

Derived from: Node

Messages	position (*x y z*)	sets translation to *x y z*
	orientation (*x y z*)	sets rotation around x-, y-, and z-axes to *x y z*
	size (*size*)	sets uniform scale factor to *size*
	size (*x y z*)	sets non-uniform scale factor to *x y z*

User

Derived from: Transform

Messages	hideLocal ([*boolean*])	if true, everything below the node (such as the user's avatar) will not be drawn by the local client (the one that owns the User node). if no argument is given, 'true' is assumed
NB:	all other messages will be passed to the User's Navigator node	

UserPart

Derived from: Node

Messages	*none*	

APPENDIX A (continued)

UserPartTrigger

Derived from: Space

Messages	checkSignal (*signal* [*event*])	tells the trigger to check user parts for the signal *signal*. if *event* is given, this event name is generated in response to the signal; otherwise, the event generated uses the same name as the signal
	checkPart (*label* [*label2* …])	tells the trigger to only check user parts labelled *label* (and *label2*, etc, if given)

Events	enter part=*partName* user=*userName*	generated when a user part (with a label given to checkPart) enters the trigger
	exit part=*partName* user=*userName*	generated when a user part exits the trigger
	event part=*partName* user=*userName*	generated when a user part is inside the trigger and sets a signal that was given to checkSignal. the event name will be the one given to checkSignal for this particular signal

UserTrigger

Derived from: Space

Messages	*none*	

Events	enter user=*userName*	generated when a User node enters the trigger
	exit user=*userName*	generated when a User exits the trigger
	firstEnter user=*userName*	generated when a User enters the trigger and the trigger space was previously empty
	empty	generated when a User exits the trigger and there are no more User's in the space

APPENDIX A (continued)

WandTrigger

Derived from: Space

Messages	*none*	

Events	enter part=*partName* user=*userName*	generated when a user's wand enters the trigger
	exit part=*partName* user=*userName*	generated when a user's wand exits the trigger
	act part=*partName* user=*userName*	generated when a user's wand is inside the trigger and sets its 'act' signal

APPENDIX A (continued)

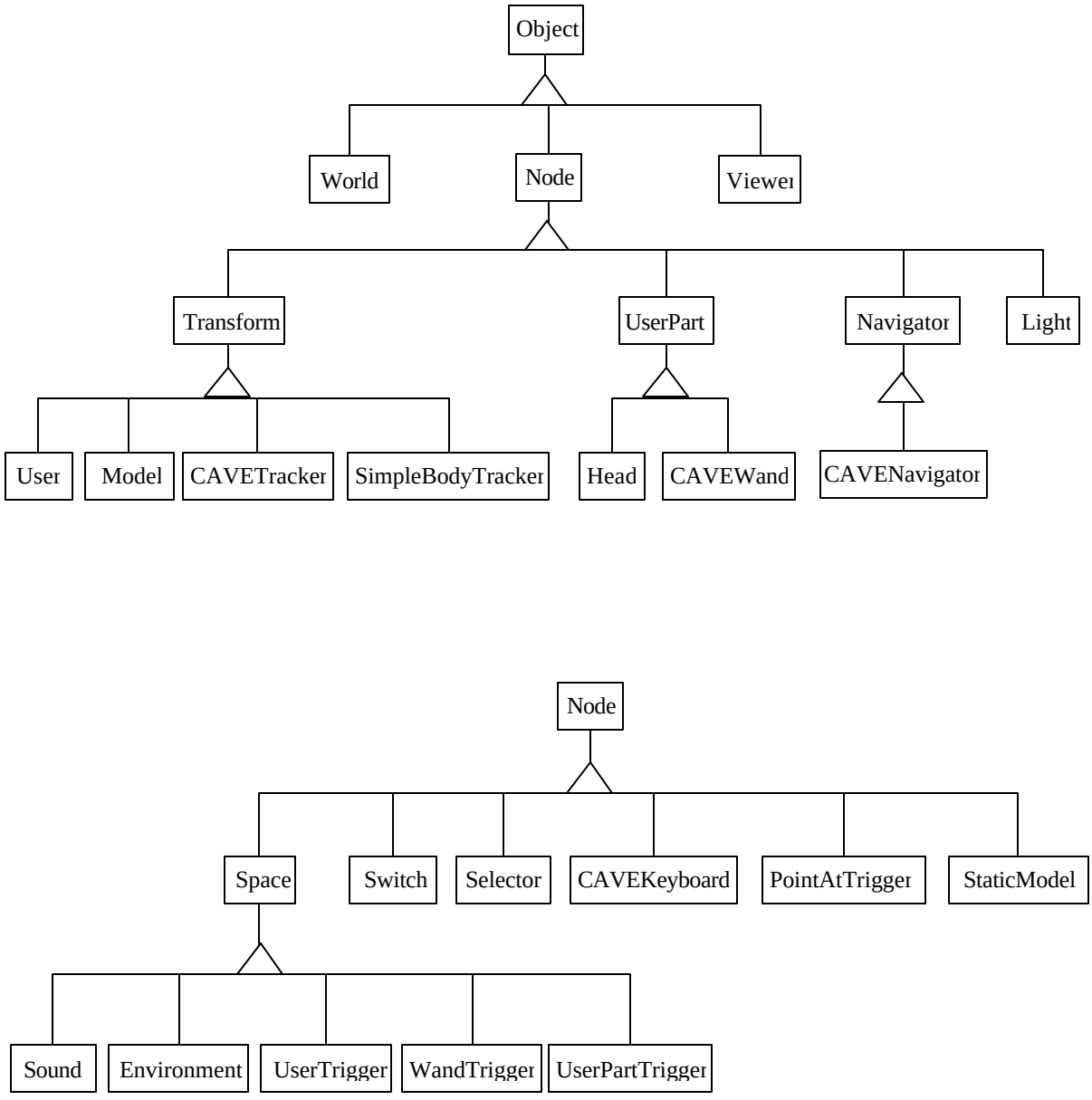


Figure 32. Ygdrasil class hierarchy

APPENDIX B

Virtual Harlem Scene Files

All.yg

```
Environment (skyColor(.3 .3 .3), clip(1 10000))
light (color(1 1 1), position(-1 0 1))
light (color(1 1 1), position(0 -1 .5))
light (color(1 1 1), position(1 .5 .5))

space CottonClubSpace (volume(box 167 55 -20  240 120 48))
space CottonClubLobbySpace (volume(box 195 35 -20  223 55 48))
space CottonClubEntrySpace (volume(box 165 -35 -20  240 35 48))

Visibility (outside,space(CottonClubSpace))
      {
      #include "extHarlem.yg"
      #include "Buildings.yg"
      }


Visibility (inside, space(CottonClubLobbySpace), space(CottonClubSpace),
            space(CottonClubEntrySpace))
      {
       #include "CottonClub.yg"
      }

#include "Trolly.yg"
#include "User.yg"
```

APPENDIX B (continued)

extHarlem.yg

```
//***************************** ambient sound ********************************
userTrigger (volume(box -1070 -561 0 1070 -483 20),
             when(enter,122stresound.play))
sound 122stresound (file(cityB.aiff), volume(box -1070 -561 0 1070 -483 20),
                  loop(1))
userTrigger (volume(box -1070 -300 0 1070 -225 20),
            when(enter,123streetsound.play))
sound 123streetsound (file(cityB.aiff), volume(box -1070 -300 0 1070 -225 20),
            loop(1))
userTrigger (volume(box -1070 -40 0 1070 40 20),
            when(enter,124streetsound.play))
sound 124streetsound (file(cityB.aiff), volume(box -1070 -35 0 1070 35 20),
      loop(1))
userTrigger (volume(box -1070 229 0 1070 301 20),
            when(enter,125streetsound.play))
sound 125streetsound (file(cityD.aiff), volume(box -1070 224 0 1070 296 20),
            loop(1))
userTrigger (volume(box -1070 485 0 1070 564 20),
            when(enter,126streetsound.play))
sound 126streetsound (file(cityB.aiff), volume(box -1070 483 0 1070 564 20),
            loop(1))
userTrigger (volume(box -110 -750 0 100 760 20),
            when(enter,lennoxAvesound.play))
sound lennoxAvesound (file(cityB.aiff), volume(box -110 -750 0 90 760 20),
            loop(1))

//************************* specific sounds ******************************
object DivineLadies (file(DivineLadies1.pfb))
      {
      userTrigger (volume(sphere -66 290.5 0 20),
                  when(enter,DivineLadiessound.play))
      sound DivineLadiessound (file(Sister_Divine.aiff), amplitude(2),
                  volume(sphere -66 290.5 0 15), falloffDistance(30))
      }

object Hoodoo (file(HoodooStoryTellers1.pfb))
      {
      userTrigger (volume(sphere 66.5 127.85 0 20),
                  when(enter,Hoodoosound.play))
      sound Hoodoosound (file(checker_players2.aiff), amplitude(2),
                  volume(sphere 66.5 127.85 0 15), falloffDistance(30))
      }

object cool (file(joeCool1_0.pfb))
      {
      userTrigger (volume(sphere 214.019 -29.766 0 20),
                  when(enter,coolsound.play))
```

APPENDIX B (continued)

```
      sound coolsound (file(Willie_Cool.aiff), amplitude(2), volume(sphere
                  214.019 -29.766 0 15), falloffDistance(30))
      }

object langston (file(LangstonHughs1.pfb))
      {
      userTrigger (volume(sphere 59.118 -440.54 0 30),
                  when(enter,langstonsound.play))
      sound langstonsound (file(Langston_Hughes.aiff), amplitude(1.5),
                  volume(sphere 59.118 -440.54 0 15), falloffDistance(30))
      }

object marcus (file(MarcusGarvey1.pfb))
      {
      userTrigger (volume(sphere 61.85 -167.84 0 20),
                  when(enter,marcussound.play))
      sound marcussound (file(Marcus_Garvey_speaks.aiff), amplitude(1.8),
                  volume(sphere 61.85 -167.84 0 15), falloffDistance(30))
      }

object rentladies (file(RentPartyLadies1.pfb))
      {
      userTrigger (volume(sphere -61.113 -389 0 20),
                  when(enter,rentsound.play))
      sound rentsound (file(3woman_going_to_party.aiff), volume(sphere -61.113
                  -389 0 15), falloffDistance(30))
      }

userTrigger (volume(sphere 279 50 0 40), when(enter,cellarsound.play))
sound cellarsound (file(edmunds_cellar.aiff), volume(sphere 279 50 0 25),
                  falloffDistance(50))

userTrigger (volume(sphere 148.161 47.5 0 50), when(enter,JellyRollsound.play))
sound JellyRollsound (file(pod_jerrys.aiff), volume(sphere 148.161 47.5 0 30),
                  falloffDistance(70))

userTrigger (volume(sphere 78.464 -37.6 0 40),
            when(enter,tapdancingsound.play))
sound tapdancingsound (file(connies.aiff), volume(sphere 78.464 -33.515 0 20),
            falloffDistance(70))

userTrigger (volume(sphere -78.5 178.933 0 45),
when(enter,lafayettesound.play))
sound lafayettesound (file(lafayette.aiff), volume(sphere -78.5 178.933 0 30),
            falloffDistance(70))

object (file(Vehicles21.pfb))
```

APPENDIX B (continued)

Buildings.yg

```
space space-12 (volume(box -10000 385 -1000 -90 10000 1000))
space space02 (volume(box -90 385 -1000 90 10000 1000))
space space12 (volume(box 90 385 -1000 10000 10000 1000))
space space-11 (volume(box -10000 130 -1000 -90 385 1000))
space space01 (volume(box -90 130 -1000 90 385 1000))
space space11 (volume(box 90 130 -1000 10000 385 1000))
space space-10 (volume(box -10000 -130 -1000 -90 130 1000))
space space00 (volume(box -90 -130 -1000 90 130 1000))
space space10 (volume(box 90 -130 -1000 10000 130 1000))
space space-1-1 (volume(box -10000 -390 -1000 -90 -130 1000))
space space0-1 (volume(box -90 -390 -1000 90 -130 1000))
space space1-1 (volume(box 90 -390 -1000 10000 -130 1000))
space space-1-2 (volume(box -10000 -10000 -1000 -90 -390 1000))
space space0-2 (volume(box -90 -10000 -1000 90 -390 1000))
space space1-2 (volume(box 90 -10000 -1000 10000 -390 1000))

Visibility block0 (inside,space(space-12),space(space02),space(space12),
                   space(space01),space(space00))
      {
       /* 1 */ object (file(PitchedRoofApartments1.pfb), floor(1), wall(1))
       /* 2 */ object (file(EndoftheWorld1.pfb), floor(1), wall(1))
       /* 3 */ object (file(BoringApartmnts1.pfb), floor(1), wall(1))
       /* 4 */ object (file(BottomShitbuilding2.pfb), floor(1), wall(1))
      }
Visibility block1 (inside,space(space-12),space(space02),space(space12),
                   space(space-11),space(space01),space(space11),space(space00))
      {
       /* 5 */ object (file(g19721.pfb), floor(1), wall(1))
       /* 6 */ object (file(g19722.pfb), floor(1), wall(1))
       /* 7 */ object (file(g19501.pfb), floor(1), wall(1))
       /* 8 */ object (file(g19502.pfb), floor(1), wall(1))
      }
Visibility block2 (inside,space(space-12),space(space02),space(space12),
                   space(space01),space(space00))
      {
       /* 9 */ object (file(g11061.pfb), floor(1), wall(1))
       /* 10 */ object (file(JacquiesApartmentsAgain1.pfb), floor(1), wall(1))
      }
Visibility block3 (inside,space(space-12),space(space02),space(space12),
                   space(space-11),space(space01),space(space11))
      {
       /* 11 */ object (file(GreyApartmentsA-21.pfb), floor(1), wall(1))
       /* 14 */ object (file(SmallsParadise1.pfb), floor(1), wall(1))
      }
```

APPENDIX B (continued)

```
Visibility block4 ()
      {
       /* 12 */ object (file(GiantApartmentComplex1.pfb), floor(1), wall(1))
       /* 13 */ object (file(FatherDivines1.pfb), floor(1), wall(1))
       /* 15 */ object (file(ApolloTheatre1.pfb), floor(1), wall(1))
      }
Visibility block5 ()
      {
       /* 16 */ object (file(Savoys1.pfb), floor(1), wall(1))
      }
Visibility block6 ()
      {
       /* 17 */ object (file(PhatApartments1.pfb), floor(1), wall(1))
      }
Visibility block7 ()
      {
       /* 18 */ object (file(GoldDust1.pfb), floor(1), wall(1))
       /* 19 */ object (file(ConstructionFence1.pfb), floor(1), wall(1))
      }
Visibility block8 ()
      {
       /* 20 */ object (file(g21661.pfb), floor(1), wall(1))
       /* 22 */ object (file(Building103Complex1.pfb), floor(1), wall(1))
       /* 24 */ object (file(g23231.pfb), floor(1), wall(1))
      }
Visibility block9 ()
      {
       /* 21 */ //object (file(Buildings1011.pfb), floor(1), wall(1))
       /* 23 */ object (file(AbyssianChurch1.pfb), floor(1), wall(1))
       /* 25 */ object (file(Rennaissance1.pfb), floor(1), wall(1))
      }
Visibility block10 ()
      {
       /* 26 */ object (file(Bamboo1.pfb), floor(1), wall(1))
       /* 28 */ object (file(Apartments121.pfb), floor(1), wall(1))
      }
Visibility block11 ()
      {
       /* 27 */ object (file(CornerBuilding1.pfb), floor(1), wall(1))
       /* 29 */ object (file(CottonClub1.pfb), floor(1), wall(1))
       /* 30 */ object (file(AfterHoursApartments1.pfb), floor(1), wall(1))
       /* 31 */ object (file(Apartments1.pfb), floor(1), wall(1))
       /* 32 */ object (file(School1.pfb), floor(1), wall(1))
      }
Visibility block12 ()
      {
       /* 33 */ object (file(Apartments11.pfb), floor(1), wall(1))
      }
```

APPENDIX B (continued)

```
Visibility block13 ()
      {
       /* 34 */ object (file(BunchoPhats1.pfb), floor(1), wall(1))
       /* 35 */ object (file(g10341.pfb), floor(1), wall(1))
      }
Visibility block14 ()
      {
       /* 36 */ object (file(g15311.pfb), floor(1), wall(1))
      }
Visibility block15 ()
      {
       /* 37 */ object (file(JacquiesApartments1.pfb), floor(1), wall(1))
       /* 38 */ object (file(Billiards1.pfb), floor(1), wall(1))
       /* 39 */ object (file(g23261.pfb), floor(1), wall(1))
      }
Visibility block16 ()
      {
       /* 40 */ object (file(RedUglyBuilding1.pfb), floor(1), wall(1))
       /* 43 */ object (file(AnotherBuilding_391.pfb), floor(1), wall(1))
       /* 46 */ object (file(KaisersnApartments1.pfb), floor(1), wall(1))
       /* 47 */ object (file(Building301.pfb), floor(1), wall(1))
      }
Visibility block17 ()
      {
       /* 41 */ object (file(NegroWorld1.pfb), floor(1), wall(1))
       /* 42 */ object (file(Look-a-like1.pfb), floor(1), wall(1))
       /* 44 */ object (file(g7751.pfb), floor(1), wall(1))
       /* 45 */ object (file(g7771.pfb), floor(1), wall(1))
      }
Visibility block18 ()
      {
       /* 48 */ object (file(g17281.pfb), floor(1), wall(1))
       /* 49 */ object (file(g20161.pfb), floor(1), wall(1))
       /* 50 */ object (file(End_of_the_LineApartments1.pfb), floor(1),
                        wall(1))
      }
Visibility block19 ()
      {
       /* 51 */ object (file(Buildings1021.pfb), floor(1), wall(1))
       /* 52 */ object (file(TheresaHotel1.pfb), floor(1), wall(1))
      }
Visibility block20 ()
      {
       /* 53 */ object (file(building_391.pfb), floor(1), wall(1))
       /* 54 */ object (file(building_3921.pfb), floor(1), wall(1))
       /* 57 */ object (file(g15791.pfb), floor(1), wall(1))
       /* 58 */ object (file(g15792.pfb), floor(1), wall(1))
       /* 59 */ object (file(FlamingBuilding1.pfb), floor(1), wall(1))
      }
```

APPENDIX B (continued)

```
Visibility block21 ()
      {
       /* 55 */ object (file(g16082.pfb), floor(1), wall(1))
       /* 56 */ object (file(g16081.pfb), floor(1), wall(1))
      }
Visibility block22 ()
      {
       /* 60 */ object (file(LongFlats1.pfb), floor(1), wall(1))
      }
Visibility block23 ()
      {
       /* 61 */ object (file(A-61.pfb), floor(1), wall(1))
      }
Visibility block24 ()
      {
       /* 62 */ object (file(BrownCornerBuilding1.pfb), floor(1), wall(1))
       /* 63 */ object (file(AnotherBrownCornerBuilding1.pfb), floor(1),
                        wall(1))
      }
Visibility block25 ()
      {
       /* 64 */ //object (file(LongApartmentsOne1.pfb), floor(1), wall(1))
       /* 65 */ //object (file(LongApartmentsOne1_1.pfb), floor(1), wall(1))
       /* 66 */ //object (file(LongerApartmentsTwo1.pfb), floor(1), wall(1))
      }
 /* 67 */ //object (file(EndCaps1.pfb))
 /* 68 */ object (file(Streets1.pfb), floor(1))
 /* 69 */ object (file(Medians1.pfb), floor(1))
 object (file(BlockBottoms1.pfb), floor(1))
```

APPENDIX B (continued)

CottonClub.yg

```
space CottonClubViewSpace (volume(box 165 -35 -20  240 120 48))
object (file(Cotton_LobbyOnly1.pfb), floor(1), wall(1))
Visibility clubSwitch (space(CottonClubViewSpace))
      {
      selector movieSelect
            {
            wandTrigger movie0 (volume(box 165 60 -10  240 120 48),
                        when(button1, movie.show(CottonClubMovie.mov)
                                    movieSound.play(CottonClubMovie.aiff)
                                    cottonclubsound.stop
                                    cottonclubsoundSwitch.off
                                    movieSelect.select(movie1)))
            wandTrigger movie1 (volume(box 165 60 -10  240 120 48),
                        when(button1, movie.show(calloway.mov)
                                    movieSound.play(calloway.aiff)
                                    cottonclubsound.stop
                                    cottonclubsoundSwitch.off
                                    movieSelect.select(movie2)))
            wandTrigger movie2 (volume(box 165 60 -10  240 120 48),
                        when(button1, movie.show(billy_eckstein.mov)
                                    movieSound.play(billy_eckstein.aiff)
                                    cottonclubsound.stop
                                    cottonclubsoundSwitch.off
                                    movieSelect.select(movie3)))
            wandTrigger movie3 (volume(box 165 60 -10  240 120 48),
                        when(button1, movie.show(Lindy_hoppers.mov)
                                    movieSound.play(Lindy_hoppers.aiff)
                                    cottonclubsound.stop
                                    cottonclubsoundSwitch.off
                                    movieSelect.select(movie4)))
            wandTrigger movie4 (volume(box 165 60 -10  240 120 48),
                        when(button1, movie.show(nicholas_brothers.mov)
                              movieSound.play(nicholas_brothers.aiff)
                              cottonclubsound.stop
                              cottonclubsoundSwitch.off
                              movieSelect.select(noMovie)))
            wandTrigger noMovie (volume(box 165 60 -10  240 120 48),
                        when(button1, movie.hide
                                    movieSound.stop
                                    cottonclubsoundSwitch.on
                                    cottonclubsound.play+2
                                    movieSelect.select(movie0)))
            }
      object (file(ClubSplit.0.pfb))
      object (file(ClubSplit.1.pfb))
      object (file(ClubSplit.2.pfb), floor(1), wall(1))
      object (file(ClubSplit.3.pfb), floor(1), wall(1))
```

APPENDIX B (continued)

```
      transform (position(195 111 -2))
          {
            transform (size(20 1 15))
                  {
                  movieScreen movie ()
                  }
            sound movieSound (volume(sphere 0 0 0 40))
          }
      }

switch cottonclubsoundSwitch
      {
      userTrigger (volume(sphere 212 61.5 0 60),
                  when(enter,cottonclubsound.play))
                        sound cottonclubsound (file(cotton_club.aiff),
                        volume(sphere 212 61.5 0 33))
      }
```

APPENDIX B (continued)

Trolly.yg

```
simpleMover trolleyRide (start(-20 0 0), end(212 0 0), time(9))
      {
      userTrigger TrolleyAttachTrigger (volume(box -13 -4 0  13 4 10),
                  when(enter,$user.attach(trolleyRide)))
      userTrigger TrolleyAttachTrigger (volume(box -20 -8 0  20 8 10),
                  when(exit,$user.release))
      switch TrolleyMoveTriggerSwitch (on)
            {
            userTrigger TrolleyMoveTrigger (volume(box -13 -4 0  13 4 10),
                        when(enter,trolleyRide.go+.25),
                        when(enter,TrolleyMoveTriggerSwitch.off),
                        when(enter,TrolleyMoveTriggerSwitch.on+9.5)
                        )
            }
      object (file(trolly-green.pfb))
      object (file(trolleyFloor1.pfb), floor(1), draw(0))
      }
```

APPENDIX B (continued)

User.yg

```
User User1 (showlocal(no))
      {
      MiletusNavigator (fly(off), speed(20), teleport(0 -10 0),
                        collideRadius(0.5))
      caveHead ()
            {
            object  (file(ThaddeusHead1.pfb),size(2))
            }
      caveTracker (sensor(1))
            {
            caveWand  ()
            object  (file(ThaddeusArm1.pfb),size(2))
            }
      body  ()
            {
            object  (file(ThaddeusAv.pfb),size(2),position(0 0 3))
            }
      keyboard (when(nkey, User1.printnav),
                  when(fkey, User1.toggleFly),
                  when(ckey, User1.toggleCollide),
                  when(rkey, User1.release))
      }
```

VITA

NAME:	David Eric Pape
EDUCATION:	B.S., Computer Science, Rensselaer Polytechnic Institute, Troy, New York, 1988 (summa cum laude)
	M.S., Computer Science, Rensselaer Polytechnic Institute, Troy, New York, 1990
	Ph.D., Electrical Engineering and Computer Science, University of Illinois at Chicago, Chicago, Illinois, 2001
TEACHING EXPERIENCE:	Department of Electrical Engineering and Computer Science, University of Illinois at Chicago, Chicago, Illinois: Introduction to Programming Languages, August 1993 – May 1994
	Department of Computer Science, Rensselaer Polytechnic Institute, Troy, New York: Compiler Design, and Fundamental Structures of Computer Science, September 1988 – May 1990
EMPLOYMENT:	Electronic Visualization Laboratory, University of Illinois at Chicago, Chicago, Illinois: Research Assistant, August 1994 – May 2001
	Ars Electronica Center, Linz, Austria: Artist in Residence, August – September 1997
	NASA/Goddard Space Flight Center, Greenbelt, Maryland: Computer Engineer, September 1986 – August 1993
	General Electric, Syracuse, New York: Software Engineering Summer Intern, June – August 1986
PROFESSIONAL MEMBERSHIP:	Association for Computing Machinery
	SIGGRAPH
	IEEE Computer Society
HONORS:	Foreign Title Award in Theater and Exhibition, Multimedia Grand Prix '97, Tokyo, Japan.

University Fellowship, University of Illinois at Chicago, Chicago, Illinois. 1993,1994,1995.

NASA/GSFC Exceptional Achievement Award. 1993.

NASA/GSFC Space Data and Computing Division Peer Award. 1993.

Pi Mu Epsilon mathematics honorary society

PUBLICATIONS:

Pape, D., Anstey, J., Carter, B., Leigh, J., Roussou, M., and Portlock, T.: Virtual Heritage at iGrid 2000, *Proceedings of INET 2001*, Stockholm, Sweden, 5-8 June 2001.

Pape, D., D'Souza, S., Anstey, J., DeFanti, T., Roussou, M., and Gaitatzes, A.: Shared Miletus: Towards a Networked Virtual History Museum, *Proceedings of the International Conference on Augmented, Virtual Environments and Three-Dimensional Imaging*, Mykonos, Greece, 30 May – 1 June 2001.

Anstey, J., and Pape, D.: Being There: Interactive Fiction in Virtual Reality, *Consciousness Reframed 3*, Newport, Wales, UK, August 2000.

Pape, D., and Sandin, D.: Quality Evaluation of Projection-Based VR Displays, *4th International Immersive Projection Technology Workshop*, Ames, Iowa, 19-20 June 2000.

He, D., Liu, F., Pape, D., Dawe, G., and Sandin, D.: Video-Based Measurement of Tracker Latency, *4th International Immersive Projection Technology Workshop*, Ames, Iowa, 19-20 June 2000.

Johnson, A., Sandin, D., Dawe, G., Pape, D., Qiu, Z., Thongrong, S., and Plepys, D.: Developing the PARIS: Using the CAVE to Prototype a New VR Display, *4th International Immersive Projection Technology Workshop*, Ames, Iowa, 19-20 June 2000.

Anstey, J., Pape, D., and Sandin, D.: The Thing Growing: Autonomous Characters in Virtual Reality Interactive Fiction, *Proceedings of IEEE Virtual Reality 2000*, New Brunswick, NJ, 18-22 March 2000.

Anstey, J., Pape, D., and Sandin, D.: Building a VR Narrative, *Proceedings of SPIE Vol. 3957 Stereoscopic Displays and Virtual Reality Systems VII (The Engineering Reality of Virtual Reality 2000)*, San Jose, CA, Jan 2000.

Imai, T., Johnson, A., Leigh, J., Pape, D., and DeFanti, T.: Supporting

Transoceanic Collaborations in Virtual Environment, *Asia-Pacific Conference on Communications / OptoElectronics and Communications Conference (APCC/OECC) '99*, Beijing, China, 18-22 October 1999.

Imai, T., Johnson, A., Leigh, J., Pape, D., and DeFanti, T.: VR Mail System, *Correspondences on Human Interface*, Vol. 1 No. 4, September 1999.

DeFanti, T., Sandin, D., Brown, M., Pape, D., Anstey, J., Bogucki, M., Dawe, G., Johnson, A., and Huang, T.: Technologies for Virtual Reality/Tele-Immersion Applications: Issues of Research in Image Display and Global Networking, *EC/NSF Workshop on Research Frontiers in Virtual Environments and Human-Centered Computing*, Chateau de Bonas, France, 1 - 4 June 1999.

Pape, D., Anstey, J., Bogucki, M., Dawe, G., DeFanti, T., Johnson, A., and Sandin, D.: The ImmersaDesk3 - Experiences With A Flat Panel Display for Virtual Reality, *3rd International Immersive Projection Technology Workshop*. Stuttgart, Germany, 10 - 11 May 1999.

Imai, T., Johnson, A., Leigh, J., Pape, D., and DeFanti, T.: The Virtual Mail System, *IEEE Virtual Reality '99*, Houston, TX, March 13 - 17, 1999.

Pape, D., Sandin, D., and DeFanti, T.: Transparently Supporting a Wide Range of VR and Stereoscopic Display Devices, *Proceedings of SPIE Vol. 3639 Stereoscopic Displays and Virtual Reality Systems VI (The Engineering Reality of Virtual Reality 1999),* San Jose, CA, Jan 28, 1999.

Pape, D., Imai, T., Anstey, J., Roussou, M., and DeFanti, T.: XP: An Authoring System for Immersive Art Exhibitions, *Proceedings Fourth International Conference on Virtual Systems and Multimedia*, Gifu, Japan, Nov 18-20, 1998.

Czernuszenko, M., Pape, D. Sandin, D., DeFanti, T., Dawe, G., Brown, M.: The ImmersaDesk and Infinity Wall Projection-Based Virtual Reality Displays, *Computer Graphics*, Vol. 31 No. 2, May 1997.

Pape, D.: A Hardware-Independent Virtual Reality Development System, *IEEE Computer Graphics and Applications*, Vol. 16.4, July 1996.

Krishnamoorthy, M. S., Oxaal, F., Dogrusoz, U., Pape, D., Robayo, A., Koyanagi, R., Hsu, Y., Hollinger, D., and Hashimi, A.: GraphPack: Design and features, *Software Visualization: Series on Software Engineering and Knowledge Engineering, Vol. 7,* World Scientific, 1996.

Krishnamoorthy, M. S., Oxaal, F., Dogrusoz, U., Pape, D., Robayo, A., Koyanagi, R., Hsu, Y., Hollinger, D., and Hashimi, A.: Further improvements to GraphPack, *Graph Drawing `94 Poster Gallery, DIMACS Workshop on Graph Drawing*, Princeton, New Jersey, October 1994.