Ygdrasil - a framework for composing shared virtual worlds

Dave Pape [a] Josephine Anstey [a] Margaret Dolinsky [b]
Edward J. Dambik [c]

[a] *Department of Media Study, University at Buffalo, Buffalo, NY*
[b] *Henry Radford Hope School of Fine Arts, Indiana University, Bloomington, IN*
[c] *Knowledge Acquisition and Projection Lab, Indiana University, Bloomington, IN*

Abstract

Ygdrasil is a programming framework for creating networked, multi-user virtual worlds, especially interactive artistic worlds. It provides a shared scene graph, a plug-in system for adding new behaviors, and a high-level script interface for composing these worlds. We describe the architecture of Ygdrasil, and its use in creating two applications that were demonstrated at the iGrid 2002 workshop.

Key words: Virtual reality, Shared virtual environments, Computer art

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 [6]. 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. 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.

Email address: dave.pape@acm.org (Dave Pape).
URL: http://resumbrae.com (Dave Pape).

This paper describes Ygdrasil, a framework for building networked virtual environments. The framework takes elements of existing systems for virtual reality programming, but focuses on enabling rapid and easy development of environments 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. A specific focus in the development of Ygdrasil has been for creating artistic environments for the CAVE VR system [1]. These applications are often built by groups of developers with a wide range of programming skills; hence, we want a system that is easy for non-expert programmers to work with, but that still provides the sort of power expected by more advanced programmers.

1.1 Previous Work

Alice is a programming environment for interactive virtual environments, from the Stage 3 Research Group at Carnegie Mellon University [8]. 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. Alice's use in undergraduate courses has demonstrated that it is very possible for beginning programmers to create interesting VR worlds when given the right tool.

Avango, formerly known as Avocado, is an object-oriented, shared scene-graph framework [7]. It was created to provide a transparent method for building networked VR applications. It is based on OpenGL 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. 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. 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 neither part of the scene graph, nor 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.

Bamboo is a portable system that uses a plug-in architecture for building networked virtual environments [9]. 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.

2 Architecture

The primary features of Ygdrasil are: a distributed scene graph, dynamically loaded extensions, and scripting. 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 the networking can be automated. Dynamic loading of new code, and the script interface for defining a virtual world, make it possible for components (node classes) to be easily shared and re-used by world authors.

Ygdrasil is built around SGI's OpenGL Performer visual simulation toolkit [4] and the CAVERNsoft G2 networking library [3]. Performer provides the basis for a hierarchical scene graph representation of the virtual world database. Necessary data, such as lists of nodes' children, transformation matrices, and model information, are automatically distributed among participants in the application via CAVERNsoft. CAVERNsoft is a networking toolkit for VR that emphasizes integrating VR with high-performance and data-intensive computing over high-speed networks.

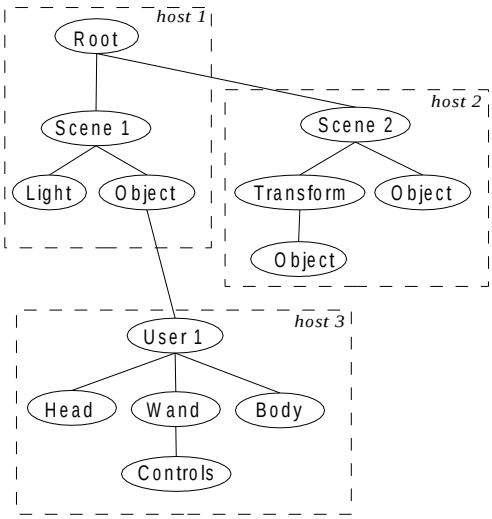


Fig. 1. A distributed scene graph

2.1 Scene Graph

We use a scene graph structure for the Ygdrasil world database. However, it is a distributed scene graph, which does not require a central server for storage. 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 1 shows an example scene graph for a world that contains two distinct scenes and a user, and shows how it 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. Each host will have its own version of the overall scene graph, consisting of those parts that it owns, and proxies of the other parts.

In Ygdrasil, in addition to the basic graphical data used in Performer, any scene graph node can have behaviors added to it. Each particular node is considered to be owned by the host that creates it. This host executes any behavior associated with the node. All other hosts will create proxy versions of the node, and only receive data for it; they do not directly modify the node, except by sending messages to the master copy to request changes. Because the basic scene graph data – that which is sufficient to render the scene – is shared automatically, new behavioral components generally do not have to include any networking themselves.

```
light sun (position(-1 0 1), color(1 .5 .2))
spinner (period(5))
      {
      object (file(top.iv))
      }
UserTrigger trig1 (volume(sphere 0 0 0 100),
                   when(enter, $user.teleport(1000 0 0)))
```

Fig. 2. Example scene file

2.2 Node Programming

Most behaviors in Ygdrasil applications are built as simple components in C++. They are new node classes that extend other, existing classes. The individual node classes are compiled into dynamically loaded objects (DSOs) (sometimes referred to as "plug-ins"). Included with each DSO is a list of other classes that it is derived from or depends on. DSOs for these dependencies are automatically, and recursively, loaded prior to loading a requested DSO.

Because they are dynamically loaded, classes can be rapidly added to a world or modified. New DSOs can also be shared among developers and re-used in different applications. The system includes a number of pre-made classes (also DSOs) that implement common virtual world interactions; these include such things as users' avatars, navigation controls, and triggers that detect when a user enters an area. We have further begun maintaining a web-accessible archive of new classes created by various people, containing about 70 different classes so far. These standard tools simplify the quick construction of many applications.

2.3 Scripting

The actual composition of a virtual world in Ygdrasil is done using a higher level, scripting-like layer. Other toolkits have used traditional procedural or object-oriented scripting languages, such as Scheme in Avango, or VisualBasic in WorldUp[5]. The scripting layer in Ygdrasil is a simple textual representation of the scene graph layout (or a fragment of a scene graph), similar to an OpenInventor object file; figure 2 is an example of a partial scene. It tells the system what kinds of nodes to create, and includes commands with each node to control its behavior.

Besides laying out the scene graph, programming 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; for example, a UserTrigger node will generate an event any time that a user enters its area. In the

scene graph script, the "when" command is used to send messages in response to these events. The messages can be sent to any node to change a value or to start some action. When creating a new C++ class, all that an application programmer has to do is to process the new, class-specific messages in a message() function, and to signal events using the eventOccurred() function; the connection between the events and messages is done separately, by the scene author. This has made it possible for experienced programmers and non-programming designers to work together in creating a world — experienced programmers create new behavior components when necessary, while others can create a world by simply plugging together the components.

Events are represented by an Event class; each occurrence generates a separate Event object. At its most basic, an Event is simply an arbitrary string. For example, in the UserTrigger node, the string "enter" is an event that indicates that a user just entered the trigger region. However, some events 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. Scene file messages can then make use of these arguments through their labels. For example, the UserTrigger command "when(enter, $user.teleport(1000 0 0))" will cause any user entering the trigger space to be teleported to the location (1000 0 0).

2.4 Networking

For any given node there is a host that owns that node; this is the host that performs calculations to update it (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 only contains the data that are 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 are used internally by the code that controls a node are not shared, and only exist on the owning host. If another node wishes to change the state of a node, it does not change the data directly, but sends a message 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

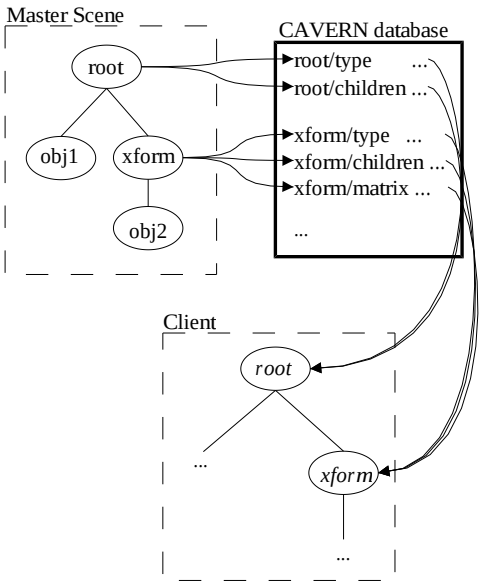


Fig. 3. Ygdrasil scene graph database

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 3 shows an example of the attributes that are shared for a group and a transformation node. The host running the master scene owns the nodes, so it initially adds keys to its CAVERNsoft database for their attributes; any time that the data changes, it will be written to the database. The client hosts registers their interest in this data, and create a local copy; whenever new data is received for a key, it will be stored in the local scene graph. Every node attribute has a separate key; different keys can be shared in different manners — an array of children pointers could use a reliable (TCP) connection, while frequently updated matrix data could use an unreliable (UDP) 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 network address for the database, and the name of the root node. In earlier versions of Ygdrasil, the client would create a local copy of the root node and begin receiving the root's data keys; this would give it a list of the root's children nodes; following these node references recursively eventually produced a copy of the entire scene. However, in large environments (with hundreds or thousands of nodes), if network latency was large this process was unacceptably slow, since as the client learned of each new node, a separate round-trip communication with the database server was necessary to register interest and receive that new node's data. Hence, we modified the design so that when a client starts up,

the server will send it a complete copy of the current state of the database in one initial transfer. In typical applications, this has improved the startup time from several minutes to a few seconds.

3 iGrid 2002 Applications

At the iGrid 2002 event, we demonstrated two applications built on Ygdrasil - "Beat Box" and "PAAPAB". These two environments were originally created as part of "EVL: Alive on the Grid," a networked virtual reality art show that premiered at the Ars Electronica Festival in September 2001 [2].

For iGrid 2002 we networked the applications between CAVEs and other VR systems at SARA in Amsterdam, NYSCEDII at the University at Buffalo, Indiana University Bloomington, the Electronic Visualization Laboratory (EVL) at the University of Illinois at Chicago, and NCSA at the University of Illinois at Champaign Urbana. The network connections from Amsterdam to the four US sites all went through iGrid's SurfNet connection between Amsterdam and Chicago, and then either through the STAR TAP connection point or the Abilene (Internet 2) network. Prior to the conference, we measured raw bandwidths between the computers at the various sites of roughly 20 to 35 megabits/second, with round-trip latencies of 120 milliseconds between Amsterdam and the US and 10 to 15 milliseconds between US sites.

3.1 Beat Box

"Beat Box" presents collaborative CAVE participants with a playful interactive audio environment. The entry scene is a grassy jungle area with colored spotlights over three virtual machines which control percussion sounds, ambient loops and bass sounds (figures 4 and 5). A fourth area provides a variety of drums. The machines act as sequencers, each with a unique periodic duration. Visually the machines are constructed of rows of thoroughly odd indigenous heads, some with long necks, some with musical instruments. Each head represents a "beat" on the sequencer. Users can generate a sound-scape by a graphically scheduling selected sound events on particular heads.

For example, at the drumming machine, using the wand interface, participants cycle through audio samples which are represented graphically by brightly colored rings. The participant can place the rings around the long necks of the heads on the machine. The sound ring takes the appearance of a necklace. The participant chooses at what interval to put the rings and can put multiple rings on one neck. Then the samples are played in sequence while the heads

Fig. 4. Beat Box drumming machine

react graphically. The bass and ambient sound machines play sounds of longer duration, which can be "mixed and matched" in order to create interesting sonic compositions that may or may not be traditionally musical. In order to keep the instructions simple, all sound tasks at the machines are performed with one button on the wand. This offers a consistency in action and keeps the movement dynamic. The drums are played by beating on them with the wand.

At iGrid, the resulting compositions varied greatly depending on the individuals involved at any specific time and appeared to be affected by their initial knowledge of music and rhythm; the time they spent learning how to manipulate the sound machines and understanding how this changed the sonic composition; and how closely users cooperated with their remote counterparts. The environment contained a reset switch in front of each machine that cleared all the sound selections and reset the machine to silence. If new arrivals did not reset the scene upon entry, they were able to explore the rhythm patterns persisting from other participants. This afforded an opportunity to understand the environment and how the machines work.

In this exhibitions users joined from multiples sites in Europe and the US, but had no real sense of the other locations, how many people were in the audience, what the mood of the audience was. However, throughout the collaborative event particular avatars consistently represented each location. For example, each day we knew that Indiana was the woman in the green dress. The avatars began to take on a character of their own, as voices became identities linked to their visual, complete with head and hand gestures and individual quirky comments and movements. Typically one person, usually the most gregarious, led the pack in adventure and conversation. "Beat Box's" graphics are appealing and having others to converse with and manipulate the machines made the exploration fun and exciting.

"Beat Box" contains custom Ygdrasil code for the control and precise timing of sound events. "Beat Box's" custom code creates a matrix of sound

Fig. 5. Beat Box bass machine

events which are triggered in realtime in the server and client applications. Each sound event causes a specific sound file to be played by the server and clients' sound server. The drumming machine uses percussive sounds of much shorter duration and actually breaks up each beat into four subdivisions for the creation of more interesting timing rhythms. Due to this, the drumming machine requires updates much quicker than the other sound machines at 100 beats per minute, it generates a timing event every 150 milliseconds, which is transmitted to all active clients. Graphical updates indicating beats as well as active drum machine sound events are transmitted at this rate.

Musically, randomly missed or delayed timing events can be greatly detrimental to a sonic composition's rhythm and can even destroy it completely. Also, latency can make collaborating on a musical piece across a network in realtime quite challenging (although the significance of latency was minimized to a great extent by using sound machines which use discrete timing events). The overall results at iGrid proved quite excellent and satisfying. The network appeared to handle "Beat Box" sound events and graphics updates quite smoothly and without noticeable missed or delayed sounds or updates. The drums and drumming machine, the most sensitive components due to their timing requirements, kept steady beats when required and behaved almost as well as in a non-networked environment.

In virtual environments, the scene and its three-dimensional elements act as navigational icons to lead participants in a direction or signal an upcoming event. Audio has a significant influence on the level of immersion. The audio can influence the participants' direction and elevate the significance of the imagery. Visuals, such as an indigenous head, act as markers or signs pointing a way to explore the world. The audio envelops the environment and carries the participants along towards successive events. This intimate coupling of visuals to audio becomes a powerful concept for exploration in virtual reality.

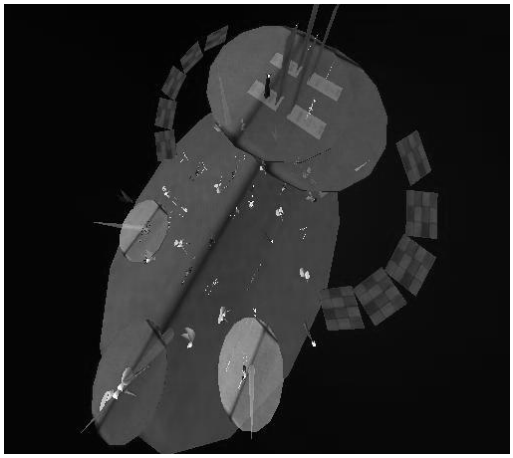


Fig. 6. The PAAPAB dance floor environment

3.2 PAAPAB

PAAPAB (Pick An Avatar, Pick A Beat) is a shared virtual reality dance club inhabited by life-size puppets that are animated by the users. The environment comprises a dance floor with raised semi-transparent platforms where the animated puppets and avatars of remotely located users can dance (figure 6). At iGrid 2002 the music was a combination of original compositions by Dan Neveu and compilations of house, techno and trance tracks from his vinyl collection.

On one platform there are are four booths where users can record their own motions and see them imitated by one of the life-size puppets. The VR display's tracking system is used for this process. The position and orientation information from the tracking sensors on the user's head and hand(s), is mapped onto the puppet's body parts. The recording lasts for 15 seconds. After the motion is recorded the puppet leaves the recording area and heads downstairs to the dance floor. There it continues dancing, continuously looping the motion information that the user has recorded. The user can go down and discover the puppets s/he has recorded.

Each booth has a different type of puppet. At iGrid the four types were a skeleton, a sex kitten, a dragon-like being and a winged puppet (figure 7). The puppets' body parts are not attached to one another, which avoids the problem of body parts joining up badly when the puppet moves. But the life-like movement that results from the motion tracking creates a strong illusion of a concrete being formed from a collection of shapes. If the puppet has more body parts than the user has tracking sensors, a system of offsets and time-lags is used to animate the extra parts. For example the sex kitten puppet has a head, two breasts, a skirt and two legs. If the user has a sensor on each hand, his hands will animate the legs, while his head movements will animate

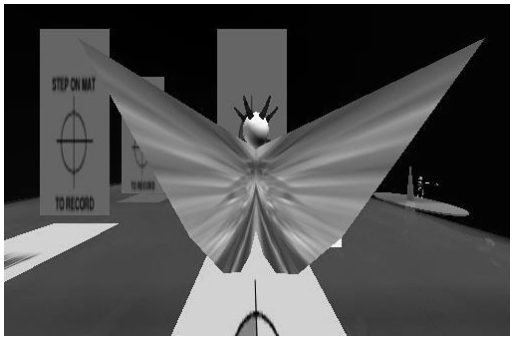


Fig. 7. A winged puppet

the puppet's head and, with offset and timelag, her breasts and skirt.

Much of our research focuses on creating interactive drama in virtual reality — immersive stories. Because access to the VR hardware needed for these projects is limited, we generally show them in a standalone form, with a single participant. However, the growth of networks and systems to support teleimmersion could provide the possibility for large-scale VR dramas featuring numerous live participants at widely distributed locations. PAAPAB is a simple demonstration/testbed of the software technology needed to achieve this. We also use intelligent agents as actors within our dramas. The design and implementation of the puppets in PAAPAB are experiments for the embodied aspect of these agents. We are interested to discover how many agents and avatars we can have running around in the networked environment. In PAAPAB users were not immediately distinguishable from the agents as their avatars were very similar to the life-size puppets - some had wings, some were skeleton-like. In the environment, users danced with both puppets and avatars, they tried to figure out which was which, they watched each other record motions.

3.2.1 Networking PAAPAB

For both the Ars Electronica show and the iGrid demonstrations all remote sites stored data such as models and textures locally. But the much larger bandwidth available at iGrid 2002, and the fact that all participating sites had Internet2 connectivity, meant that we could make some significant alterations to the application.

Typically transformation data from the tracking system, which is used to position and orient remote avatars, is sent over the network over UDP. At first we imagined that we would similarly stream the transformation data for our puppets. However at Ars Electronica we did not have the bandwidth to send all the transformation information for all the puppets over the network. Since we have 40 puppets each with at least four body parts and some with additional

spring calculations to make, the transformation data can grow quite large. Therefore for the Ars Electronica show, we used a file server to supplement the networking built into Ygdrasil. When a user recorded a dance, we packed the data in files. The file server sent the file out to all connected clients via TCP. Each client had to unpack the data, use code modules on the local machine to interpret it, and further local modules to make the spring calculations locally. This conflicted with the original design objective to have the more complex behavior modules only at the master site, while the clients would only need transformation nodes to alter the position and orientation of objects in their local graphics. At iGrid, in contrast, the master machine saved the recorded motion data of the forty puppets, and streamed it continuously (via UDP) to the client machines.

For the Ars Electronica show, each remote site also has all the sound files and played them locally — the only information that was networked was messages about what should be played when. For iGrid we were able to stream the music from a computer in Chicago at CD-quality (44.1 kHz, 16 bits). For both shows we used aconf, an audio conferencing tool written by the CAVERN group at EVL, to do audio-conferencing. We experienced some initial problems with aconf because it did not have a large enough buffer to accommodate the 44.1 kHz stream, and had to rewrite a small part of the code.

During the iGrid demonstrations, we observed the network traffic on the different computers using the "netstat" utility. The computer at SARA, which controlled the master scene and broadcast all puppet data, generated continuous traffic of roughly 3.5 megabits/second. The computer in Chicago, which streamed the music as well as avatar data for the local user, generated between 1.5 and 3 megabits/second of traffic. This amount of network use was well within the capability of the network set up for iGrid, but was in fact near to the limits of our software. We intentionally restricted the speed of updates of the puppets to 5 frames per second (even though the graphics rendering runs at about 24 frames per second), because when we initially tried to send more data to multiple clients, the database server was not able to keep up, and most of the UDP data packets were lost.

4 Conclusion

In Ygdrasil we have 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.

With Beat Box and PAAPAB, we demonstrated some of the possibilities for the use of high-speed networks and immersive displays in interactive art. The bandwidth requirements currently limit them to high-end computers and research networks, such as those that were available at the iGrid conference. Although the capabilities of mainstream computers and networks will continue to increase, further work will be needed to make the software useable in such an environment while still providing the flexibility and power that has made it possible to create these applications.

5 Acknowledgments

CAVE is a registered trademark, and STAR TAP is a service mark, of the Board of Trustees of the University of Illinois.

References

[1] C. Cruz-Neira, D. Sandin, T. DeFanti, Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE, in: Proceedings of SIGGRAPH 93 Computer Graphics Conference, 1993, pp. 135-142.

[2] D. Pape, D. Sandin, Alive on the Grid, in: Proceedings of 6th World Multiconference on Systemics, Cybernetics and Informatics 12, 2002, pp. 485-490.

[3] K. Park, Y. Cho, N. Krishnaprasad, C. Scharver, M. Lewis, J. Leigh, A. Johnson, CAVERNsoft G2: A Toolkit for High Performance Tele-Immersive Collaboration, in: Proceedings of the ACM Symposium on Virtual Reality Software and Technology 2000, pp. 8-15, 2000.

[4] J. Rohlf, J. Helman, IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics, in: Proceedings of SIGGRAPH 94 Computer Graphics Conference, pp. 381-395, 1994.

[5] Sense8 Corporation, WorldUp Users Guide, Release 4, 1998.

[6] S. Singhal, M. Zyda, Networked Virtual Environments: Design and Implementation, New York: ACM Press, 1999.

[7] H. Tramberend, Avocado: A Distributed Virtual Reality Framework, in: Proceedings of IEEE Virtual Reality '99, Houston, TX, March 1999.

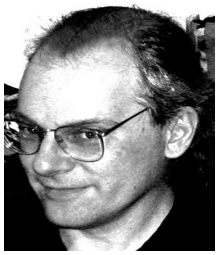
[8] UVa User Interface Group, Alice: Rapid Prototyping for Virtual Reality, IEEE Computer Graphics and Applications, 15 (3) (1995) 8-11.

[9] K. Watsen, M. Zyda, Bamboo - A Portable System for Dynamically Extensible, Real-time, Networked Virtual Environments, in: Proceedings IEEE Virtual Reality Annual International Symposium, Atlanta, GA, 1998, pp. 252-259.

Dave Pape is a Research Professor in the Department of Media Study of the University at Buffalo. He received a BS and MS in Computer Science from Rensselaer Polytechnic Institute, and a PhD in Computer Science from the University of Illinois at Chicago. His research interests include tools for and applications of virtual reality and computer graphics in the arts and sciences.

Josephine Anstey is an Assistant Professor in the Department of Media Study of the University at Buffalo. She received an MA in American Studies from the University at Buffalo, and an MFA in Electronic Visualization from the University of Illinois at Chicago. Her research interests are interactive VR drama, networked VR and low-cost VR solutions.

Margaret Dolinsky is an Assistant Professor and Research Scientist at Indiana University, Bloomington's HR Hope School of Fine Arts. She received her MFA from the University of Illinois at Chicago and is currently a postgraduate researcher at the University College of Wales. Her work concentrates on collaborative CAVE art environments, visual metaphors for navigation, and the participants' role in the experience.



Edward J. Dambik is a Research Associate/Software Developer with the Pervasive Technology Knowledge Acquisition and Projection Laboratory at Indiana University, Bloomington. Prior to Indiana University, Dambik was at Fermi National Accelerator Laboratory where he developed networked data acquisition servers and clients. His other interests include collaborative CAVE environments, human video tracking, and musical applications.