A Scalable, Multi-user VRML Server

Thomas Rischbeck and Paul Watson
Department of Computing Science, University of Newcastle, Newcastle-upon-Tyne, UK
{Thomas.Rischbeck|Paul.Watson}@ncl.ac.uk

Abstract

VRML97 allows the description of dynamic worlds that can change with both the passage of time, and user interaction. Unfortunately, the current VRML usage model prevents its full potential from being realized. Initially, the whole world must be loaded into the user's desktop browser, and so large worlds can take a very long time to download and render, while a world cannot be shared among multiple users

This paper describes the design and implementation of a client-server architecture that was built to overcome these problems. The major novelty is the decoupling of VRML world execution from world rendering. Parallelism and information filtering are exploited to produce a highly scalable system that can support huge, highly active worlds, accessed simultaneously by large numbers of users. A cluster-based parallel server is responsible for maintaining the dynamic world state, and most of the world dynamics are evaluated on the server side. The server streams VRML to the client, using view frustum culling and dynamic LOD selection to reduce clients' network bandwidth, storage and rendering requirements. Clients with limited resources (e.g. wireless-connected PDAs) can therefore participate in highly complex virtual worlds. While the implementation of the design focuses on VRML worlds, the design ideas could be exploited in other types of VR system, e.g. X3D.

1 Introduction

The Virtual Reality Modeling Language (VRML) [2,3] allows three-dimensional worlds to be described in a platform-independent notation. A world description can be downloaded over the Internet into a VRML browser, which performs the audio-visual presentation and provides a set of navigation paradigms to the user. The first version of VRML provided only static and non-interactive worlds, but the later VRML97 standard supports "moving worlds" that are responsive to both user interaction and the passage of time. It is therefore possible to envisage the creation of huge, complex worlds with thousands of interacting users. For example, models of cities could be built to include moving vehicles as well as static buildings. In the future, with advances in traffic sensing technology, it may even be possible to build models of real cities that show accurate traffic flows in real-time.

Despite the obvious potential, the VRML worlds available on the Internet have so far been relatively limited in terms of scale, movement and interaction. The reason can be found in the nature of the VRML usage model, in which the description of the VRML world is downloaded into the user's browser and evaluated locally. This introduces a set of problems:

(1) Download and rendering time. Downloading the world description into the VRML browser takes a long time for large worlds. Once downloaded, the VRML browser is responsible for rendering the world. This can be prohibitively expensive for large scenes, despite the culling optimizations performed by most VRML browsers.

(2) Behavior evaluation. The browser must continually update the state of the world–as objects move and the user interacts–and also render a view of the world in the browser window. If a client's processing power is insufficient for complex behavior evaluation, the frame rate will drop, resulting in a low-fidelity presentation.

(3) Multi-user interaction and persistence of changes. Because the world runs locally–in the user's browser–there is no possibility of interaction between different users in the same world. This precludes both, direct interaction between users who have visible contact, but also indirect interaction. For instance, one user could introduce a new object in the world to be seen by a later passerby. The VRML usage model does not allow persistence of state: a world reverts to its initial state every time it is downloaded.

(4) External updates. It is difficult to arrange for the world to change in response to external events. For example, if a virtual world models the current state of part of the real world (e.g., traffic flow in a city, footballers on a pitch) then we would wish to move objects in the virtual world to reflect real-time changes in the real world.

In this paper we describe the design of a scalable, multi-user system that was built to overcome these problems. By decoupling world execution from world render-

ing, it allows multiple users (possibly using low-powered PDAs) to interact in huge, persistent, dynamic worlds.

The rest of the paper is structured as follows. After a comparison with related work, Section 2 examines the opportunities for exploiting parallelism within the VRML execution model, and goes on to describe the design of the scalable server. Section 3 introduces the client-server interface that allows low-powered clients to participate in large, active worlds. Performance results from an implementation of the design are given in Section 4, and then conclusions are drawn from the work (Section 5).

1.1 Related work

Support for scalability is one of the main research interests in the area of concurrent virtual environments [1]. However, while other work deals mainly with the dimensions of graphical complexity or simultaneous user support [9], we are also interested in scalability to complex behavior and dynamic evaluation. Such complexity might arise from arbitrary update code as it might be contained in VRML Scripts. A number of VRML-based networked virtual environments exist. Typically these are implemented as client-server systems. VNET [13] for example, makes use of the EAI interface on the client-side to collect information about a user's movements and update the avatar state of other simultaneous participants. Unlike the system described in this paper, the VNET server acts merely as a communications relay for such updates to avatars. Commercially available implementations of VRML multi-user systems extend this scheme with authentication and simple, server-controlled artifacts, so called robots (e.g. blaxxun, ParallelGraphics). These systems are neither scalable to large-scale worlds, nor to complex behaviors as both scene graph and behavior evaluation are actively replicated on all clients. I.e., the complete scene graph is downloaded to all clients and the server is responsible for the replication of avatar movements across all clients.

ActiveWorlds [10] is also based on the idea of a central server to broadcast avatar movements. In addition, the ActiveWorlds server performs distance based culling: clients are only informed about artifacts and other client avatars that are in their respective proximity. Therefore, ActiveWorlds is scalable in terms of world size and number of participants. Participants can upload VRML files as new artifacts to the server and thereby actively "build" the world. Besides artifact addition and avatar movement, dynamics support is limited to simple animations. Unlike the servers described in this paper, the ActiveWorlds server does not perform any behavior evaluation.

Chenney et al. [4] examine dynamics culling for invisible parts of a scene. While this approach is useful for physics-based simulations, which can easily be extrapolated over time, this method cannot be used for general, unpredictable behavior evaluation.

In the domain of computer games, BSP traversal with pre-computed visibility sets and portal rendering are frequently used to greatly reduce rendering time for large worlds. These techniques work well for mainly static worlds with movement restricted to simple animations and human or computer controlled actors. In multi-player network games, such as Quake or Doom, the dataset describing the world and avatars is available to the client a-priori; no dynamic download takes place. The server (usually one of the player's machines) performs replication of client state (e.g., position, weapons usage) across all participants. This scheme restricts maximum world size to the clients' storage capacity. Levels are therefore rather small and uniform.

Peer-to-peer VR systems in generally use IP multicast to propagate user information to other simultaneous participants. In NPSNet, the world is divided into hexagonal cells with one multicast address associated to each cell. MASSIVE [5] uses multicast between participants according to an aura/nimbus model of awareness. Static world content in this system is known in advance, whereas client state is shared in a scalable manner. NPSNet and Simnet [11] evaluate behaviors (e.g., weapons ballistics) on the participant's machine. This is a scalable approach in terms of world size and numbers of users, but suffers from the fact that a malicious participant can easily break the rules.

2 Potential parallelism in the VRML execution model

This section examines how parallelism can be exploited within VRML in order to achieve scalable behavior evaluation. We begin with a very brief overview of the key attributes of VRML.

2.1 VRML attributes

Nodes are the fundamental building block of VRML models. Various node types exist, with semantics that abstract from various real-world objects and concepts. A scene is represented as an ordered collection of nodes, known as a *scene graph.* Nesting within the scene graph

is based on special *grouping nodes*, which may contain other nodes as children. Each node contains a set of typed fields that hold its state.

A field marked with the *eventIn* or *eventOut* tag can participate in the event-driven VRML execution model. An *event* may be sent from one node's eventOut to another node's eventIn field if a *route* exists. Reception of an event may trigger processing at the destination node and possibly the creation of further events. Routes are described in the *routing graph*, separate to the scene graph. In response to an incoming event, a node can update its state, and also generate a set of outgoing events. A *fan-out* of events takes place if a node generates more than one outgoing event.

2.2 Event cascades

An *event cascade* is the set of events generated as the result of a single initial event propagating over the routing graph. Sensor nodes generate initial events in response to user interaction (e.g., a mouse click), the passage of time, or external inputs. Event cascades allow an initial event to trigger a set of changes in the world, for example when the user presses a button object.

Events consist of a typed value (matching the type of the connected eventOut and eventIn fields) and a timestamp. All events in an event cascade are considered to occur simultaneously with the initial Sensor event [3], and therefore carry the same timestamp. A loop-breaking rule prevents livelock in cyclic routing graphs.

Whereas most nodes perform only fine-grained processing when an event arrives (e.g. updating the co-ordinates of an object in the world), Script nodes may execute arbitrarily complex Java and JavaScript code (e.g., to provide real world sensors or database queries, knowledge discovery visualization, complex physics-based simulation or AI control). These operations may induce significant load on the VRML browser. In conventional VRML browsers, the complexity and number of Script nodes is consequently limited.

2.3 Discrete and continuous events

Most events produced during world execution are *discrete*: they happen at well-defined world times, e.g. as determined by the time of user interaction. However, TimeSensor nodes also have the capability to model continuous changes over time: A browser generates sampling events on the *fraction_changed* and *time* eventOut fields of TimeSensors. The sampling frequency is implementation dependent, but typically samples would be produced once per frame, e.g., once for every rendering of the user's view on the world.

As event cascades become computationally more demanding, a conventional VRML browser will compensate through a reduced sampling frequency for continuous events, and therefore jerky scene updates. Further, the system may become over-saturated with discrete events if they are generated at a higher rate than the system can evaluate their event cascades.

2.4 Fan-out parallelism and parallel sensor evaluation

Evaluation of event cascades must follow the causal ordering of events. This is a partial order, as the VRML97 specification does not define a processing order for fan-out events. Further, there is no interference between two fan-out branches, except for fan-in event routing to a shared node. The processing order for fan-ins is equally undefined. Parallelism within a single event cascade can therefore be exploited in conformance with the VRML97 specification. It is also possible to compute independent event cascades in parallel [8].

Our server architecture has been designed to exploit this potential parallelism (see Figure 1). The set of nodes in the world is partitioned over the local memories of a set of processors in a parallel system. In our experiments we use a cluster of PCs interconnected by Fast Ethernet, but the precise architecture is not central to the design—any scalable distributed memory parallel machine is suitable. One processor maintains the global clock for the system, generating regular time beats that are sent out to each of the other processors in the parallel machine. These beats are used to trigger the time sensors that initiate event cascades. Parallelism is exploited because, at any one time, all the processors can be sending and receiving multiple events, and executing the scripts that they trigger. As the source and destination of an event may be held in the memories of different processors, some event communications will result in network messages between nodes in the parallel machine. A further degree of scalability is introduced because the memories of all the processors are available for holding the world's nodes.

We have investigated an alternative to having one global clock. Instead, each processor could be driven by its own local clock. However, this raises a problem: if the clocks are out of synchronization then events generated, and timestamped on one processor, and then sent to a another processor, may arrive in "the past".

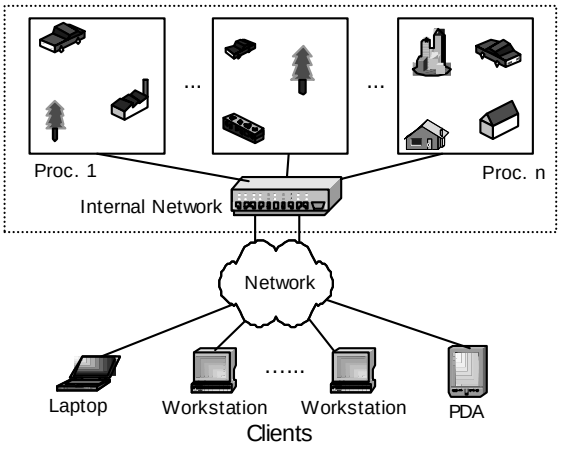

Figure 1 The client-server architecture

3 The client-server interface

The parallel server, as described above, maintains evolving world state and performs event cascades in a distributed manner. This removes from the clients the need to compute world changes, so reducing their required processing power. In this section we describe the client-server interface. The design aim was to produce a system that allows large numbers of (possibly low-powered) clients to interact in huge, dynamic worlds.

Information must be sent from the server to the client so that the user can view the world. At one extreme, the server could render each client's view of the world and so only send clients a stream of frames, represented as bitmaps, to be displayed. This limits the work of the client to reading frames from the network and displaying them. However, it would place a large load on the server and network. Further, many clients have specialized graphics hardware for rendering 3-D scenes efficiently. It was therefore decided that each client should render its own view of the world.

This requires the world's objects to be sent from the server to each client for rendering. At one extreme, all the world's objects could be sent to a client when it connects to the server, and then any subsequent updates could be forwarded (for example when objects move). This would create three problems for very large worlds: firstly they would take a very long time to download, secondly, the client would need sufficient memory capacity to store them, and thirdly rendering could take a very long time. In order to avoid this, it was decided to design a client server interface that limited the amount of world information sent to the client. This was achieved by exploiting two techniques: frustum culling, and dynamic level of detail selection, both performed on the server side.

3.1 Server-side culling

When a client initially connects to the server, one processor is chosen to act as a proxy for it, handling all subsequent client-server communications. A load-balancing scheme is used to spread the set of client proxies evenly across the parallel server's processors, to promote scalability. Once connected, the client reports to its proxy any changes to its position, orientation and horizontal and vertical field of view. Based on this information, the server updates the position and orientation of the avatar that represents the user, so that other online users can see those movements. The client proxy also computes the four planes describing the client's view frustum. This is then used to determine which objects in the world are visible to the user, as only these objects are sent to the client.

Two options were considered for selecting objects. In the first, the world is represented as an unstructured set of objects. Therefore, every time a client connects, or moves in the world, all the worlds' objects must be compared with the client's view frustum to determine which need to be sent to the client for rendering. While this can be done in parallel, with each processor in the parallel machine comparing the frustum with the objects held in its own memory, for a large world, this requires a very large amount of processing, and this would limit the scalability of the system. Therefore, an alternative was designed.

In this scheme, the VRML world is structured as a set of complex objects. It makes use of VRML's "PROTO" statement that enables the encapsulation of a partial scene graph, with a well-defined interface of fields and events [2,3]. In addition, this allows a world programmer to move responsibility for repetitive and low-granularity behaviors to the client side.

3.2 PROTO encapsulation

In our system, PROTOs are used to contain a set of mandatory fields that are accessed both by the client and server, and used to implement an efficient interface between them (see Figure 2 for an example PROTO node definition). Whereas the client deals with the contents of the PROTO, the server sees it as a "black box". The man-

```
PROTO House002 [
 # mandatory fields for shared PROTOS ...
 field SFVec3f   bboxCenter        2.5 2.5 2.5
 field SFVec3f   bboxSize          5   5   5
 field MFFloat   levels            [ 100, 200, 300 ]
# default shared fields
 field MFString  urls [
     "http:////www.cs//vrml//House002_LOD0.wrl",
     "http:////www.cs//vrml//House002_LOD1.wrl",
     "http:////www.cs//vrml//House002_LOD2.wrl"
 ]
 exposedField  SFVec3f    position         0 0 0
 exposedField  SFRotation orientation      0 1 0 .5
 exposedField  SFVec3f    scale            0 0 0

 # optional shared eventIn/eventOut fields, specific
 # to the described object.
 exposedField SFBool      doorOpen         FALSE

] {…}
```

Figure 2 An example PROTO node definition

datory fields are: the node's position, orientation and scale; the size and position of its bounding box; and, references (in the form of URLs) to files that contain the encapsulated scene graphs at different Levels of Detail (LOD). As is described in detail below, the server determines the appropriate level of detail for every PROTO and only the corresponding LOD file is downloaded to the client.

It is important that a client's copy of a PROTO is synchronized with the server's version of the same PROTO. Therefore, if the server-side state of the PROTO is updated, the client is notified of the change by the server. Vice versa, a notification message is sent to the server if the user interacts with the PROTO, leading to an eventOut being generated on one of the exposed fields. All behavior encapsulated in the PROTO is evaluated on the client side, which is more efficient in terms of network utilization for e.g., simple animations. Complex and non-repetitive behavior however should be defined outside of PROTOs, so that evaluation takes place on the server. This gives a world designer control over the location of behavior evaluation.

Structuring the world in terms of complex nodes defined by PROTO descriptions is the key to an efficient client-server interface. Performing culling is reduced to checking if the bounding box of each PROTO is visible to the client. For a typical world, this reduces the number of comparisons by a factor of 10 to 1000 when compared to culling every individual node that makes up a scene.

When a client connects to the server, the set of PROTOs that are visible to a client are determined, and the server sends the contents of all default-shared and optional fields. The server knows the distance of each object from the client's viewpoint, and so can determine the appropriate LOD level for a PROTO, which it communicates to the client. LOD levels for a PROTO can change dynamically as a result of the client navigating in the world. On the basis of URLs the client can perform efficient and easy to implement caching of LOD files, as is now explained.

The client sends requests to the server for the files whose URLs are contained in the PROTO it has been sent to render. The requests go through the standard Web browser cache running on the client and so when the server (which runs a Web server for this purpose) returns the files, they are stored in the cache. This has the advantage that all instances of the same complex object at a particular LOD level required by a client can share the same cached file, and so a server request is only required for the first access to a file. Of course, if the cache fills and the file is rejected then it will have to be re-fetched on its next access. A further advantage is that if a client disconnects from the server and then connects again later then many of the files it requires may still be cached, so reducing load time, and network bandwidth. By using the client's Web cache to store the LOD files, the benefits of cache management were obtained without any extra development being required. It is important to note that the client handles all PROTO downloads and additions to the client scene asynchronously and without blocking the user's navigation. PROTOs are streamed on demand into the client scene.

The client proxy on the server keeps a record of all the PROTOs that it has sent to the client. If the state of the complex object is changed on the server, as the result of an event, then the changes are propagated to the client. Similarly, if a user interacts with an object on the client, then all resulting events are routed to the server so that they can cause the necessary change in the world. Consequently, all other users will observe a change to the world made by one user.

After the initial loading of objects into the client, further objects are transmitted if the client moves, so that new objects are visible, or if moving objects come into the client's field of view. Similarly, if objects move out of

Figure 3 Screenshot of the browser interface—other online users are represented through their avatars

visibility then the server informs the client so that the client can remove PROTOs from the local scene graph.

One disadvantage with this approach is that the world has to be represented in a particular, structured format in the VRML file. This means that existing large-scale VRML scenes need to be manually reworked to represent the scene as a collection of PROTO nodes at the server. For each of these components, bounding boxes must be created and, for optimum performance, a set of LODs should be provided. However, it is possible to envisage an automatic tool carrying out these tasks, though none currently exists.

3.3 Client-side implementation

The client-side implementation of this system is based around a VRML97 browser component controlled and monitored by a lightweight application layer. Three client versions exist: a Java applet, using the VRML browser as plug-in on a web page, a Java stand-alone application and a Visual Basic implementation. Figure 3 shows a screenshot of the browser interface.

The EAI [6] protocol is used in the Java versions. EAI defines a protocol for external applications to manipulate a scene graph loaded in a VRML browser. In particular, this mechanism allows the dynamic addition and removal of nodes from a scene and manipulation of a node's field values. In addition, callback methods can be set up to listen on eventOut fields, so that the server can be informed of client interaction.

The Visual Basic client uses a COM component produced by Parallel Graphics. The server was implemented using novel, parallel object-oriented middleware—SODA [8]. This is written in Java, so that the server is very portable. Most of the prototyping has been carried out on a cluster of Linux PCs, interconnected by a standard (Ethernet) network. Results show that this type of low-cost parallel system can operate as a very efficient, scalable VR server.

4 Results

In this chapter we present performance results for the system. They were obtained with the server running on a cluster of 16 PCs, each with dual-Pentium III processor, running at 866 MHz and with 512 MB main memory. The client was running on a Pentium II, 300 MHz machine with 192 MB main memory. The client-server interconnection network is a shared 10 Mbit LAN, with a ping-latency of about 1 ms between client and server.

4.1 World loading time/streaming benefits

Initial world loading time for a scene on the client-side is largely determined by the download time of the files

describing the complex objects from the web server. As described in Section 3, a major benefit of our scheme is that only those object that are visible to the user need be downloaded. To demonstrate this Figure 4 shows a comparison of loading time for varying world sizes. The example world used for the graph contains a variable number of PROTOs (each about 23 KB, with 87 faces). The advantages of culling are clear for worlds of moderate to large complexity. Once the initial view is loaded, further PROTOs are streamed to the client without interrupting navigation.

4.2 Frame rates for a static world

If a world is static, then the frame rate depends on the time taken to render the objects in the world. The server-side culling of the parallel server reduces the time required by the client to render the world, and so increases the frame rate. Figure 5 shows that the result is much better than that of a conventional VRML browser for worlds that are of moderate to large size. This problem had however, already been solved very efficiently by techniques such as BSP pre-computed visibility sets (see related work), but the parallel server is also able to provide scalability in the level of world activity, as the following section shows.

4.3 Dynamic world update rates

Figure 5 and Figure 6 measure the performance of the system for dynamic worlds. The world size was kept constant while the level of activity was varied. This was achieved by changing the number of Script nodes in the world. Each script node number executed code with fixed granularity of roughly 500 µs. Event cascades were initiated by a TimeSensor and their outputs controlled various PROTOs in the scene.

The results show the excellent levels of scalability that can be achieved with the parallel server. The speedup increases as the activity of the world increases, suggesting that the system should be highly scalable when managing huge, active worlds. Of course, if disjoint event cascades [8] are executing in separate parts of the system without any communication between processors in the parallel system, speedup is even higher since communication and synchronization overheads are removed.

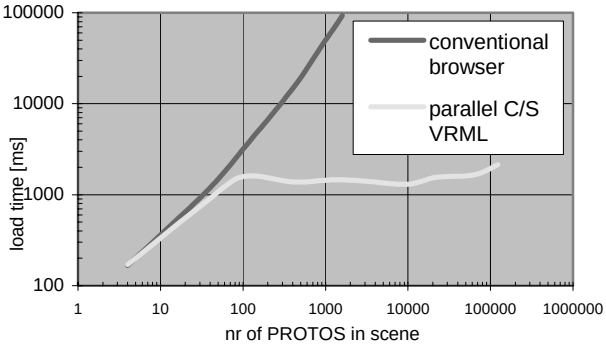

Figure 4 Load time vs. world size

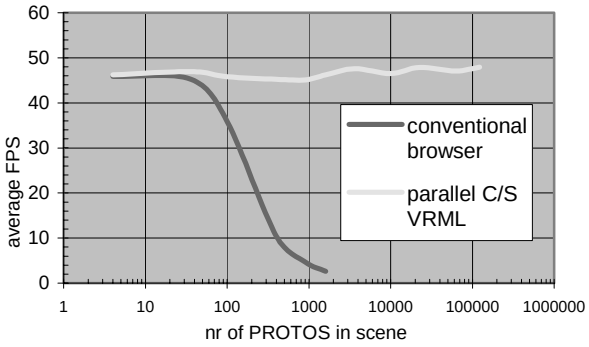

Figure 5 Average FPS vs. world size

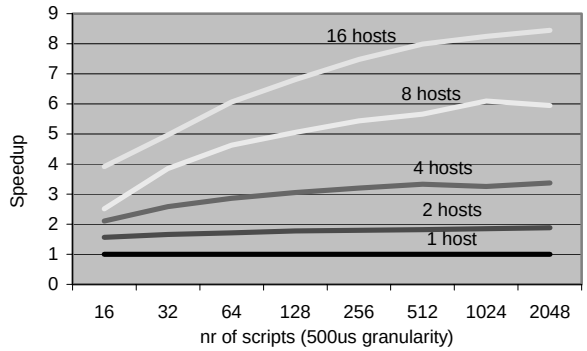

Figure 6 Speed-up vs. world activity

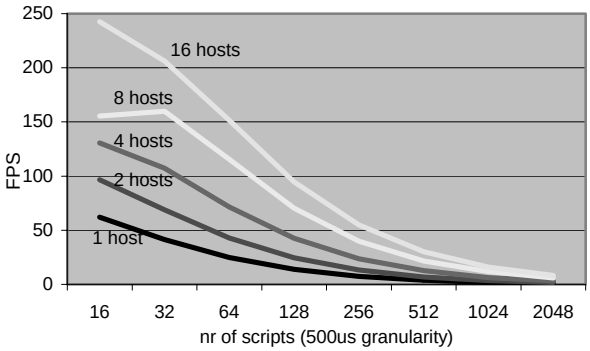

Figure 7 FPS vs. world activity

5 Conclusions and future work

This paper explores the design of a client server system that provides scalability in various dimensions: number of users, world size, and world dynamics. The performance results show that the approach is successful, and that the parallelism of the server can be efficiently exploited. This opens up the possibility of deploying systems that allow many users, connected through a variety of clients, to interact in a huge virtual world. While the concepts shown have been implemented and tested with VRML, the solutions are general, and so the same approach could be applied to other VR systems, e.g. X3D.

We are currently further evaluating the system, and exploring several enhancements to it. Recent technological advances are leading to systems that combine handheld computing devices with global positioning systems and mobile data transmission. The parallel VR server could serve as an excellent basis for augmented reality systems, delivering information to mobile users. There are many potential applications for such a system including visualizing hidden features during construction (e.g. underground pipes, and electricity cables); and, providing personal guides to travelers (including both pedestrians and drivers). Efficient support for mobile clients connected through low-bandwidth networks requires the ability to adapt the amount of information sent, to the available bandwidth, as described in Section 3.1.

While the server maintains world state between consecutive user sessions, it does so in volatile memory. This means that any changes to the world are lost if the system fails, or has to be taken down for maintenance. We are therefore investigating how to best maintain an up-to-date copy of the world in non-volatile storage. This will have the option of maintaining a log of changes, so that the evolution of the world could be replayed.

We have also been investigating support for highly geographically dispersed users. This could be done via “local servers”, which are located in the client’s network proximity. Based on our system, two replication approaches can be envisaged to synchronize such a server-federation: replication of the complete scene graph with active replication of events; or, partially replicated servers (both are described in [7]).

References

[1] Benford, S., Greenhalgh, C., Rodden, T., and Pycock, J., Collaborative Virtual Environments *CACM*, vol. 44, no. 7, pp. 79-86, Jul, 2001.

[2] Carey, R. and Bell, G. *The Annotated VRML 2.0 Reference Manual*, Addison-Wesley, 1997.

[3] Carey, R., Bell, G., and Marrin, C., ISO/IEC 14772-1: 1997 Virtual Reality Modelling Language (VRML97) 1997.

[4] Chenney, S., Ichnowski, J., and Forsyth, D., Dynamics Modelling and Culling *IEEE Computer Graphics and Applications*, vol. 19, pp. 79-87, 1999.

[5] Greenhalgh, C., Large Scale Collaborative Virtual Environments 1997. University of Nottingham.

[6] Marrin, C., Proposal for a VRML 2.0 Informative Annex, External Authoring Interface Reference 1997.

[7] Rischbeck, T. and Graham, M., "Implementing Scalable Networked Virtual Environments Using Replicated VRML Servers," *SRDS 2000 Workshop on Object-Oriented Reliable Distributed Systems (WOODS 2000)*, Nuremberg, Germany.

[8] Rischbeck, T. and Watson, P. A Parallel VRML97 Server Based on Active Objects. In: *VECPAR'2000 Selected Papers and Invited Talks from the 4th International Conference on Vector and Parallel Processing*, eds. Palma, J. M. L. M., Dongarra, J., and Hernandez, V. FEUP, Porto, Portugal: Springer Verlag, 2001. pp. 47-60.

[9] Singhal, S. and Zyda, M. *Networked Virtual Enviroments*, ACM Press, 1999.

[10] Tatum Mandee, Active Worlds *Computer Graphics, ACM SIGGRAPH*, vol. 34, pp. 56-57, May, 2000.

[11] Miller, D. and Thorpe, J. A., SIMNET: The advent of simulator networking. *Proceedings of the IEEE*, vol. 83, pp. 1114-1123, Aug, 1995.

[12] Macedonia, M. R., Brutzman, D. P., Zyda, M. J., Pratt, D. R., Barham, P. T., Falby, J., and Locke, J., "NPSNET: A multi-player 3D virtual environment over the Internet," *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, Monterey, CA.

[13] White, Stephen. VNET website: http://www.csclub.uwaterloo.ca/u/sfwhite/vnet/.