Myriad: Scalable VR via Peer-to-Peer Connectivity, PC Clustering, and Transient Inconsistency

Benjamin Schaeffer[†] schaeffr@uiuc.edu
Peter Brinkmann[††] brinkman@math.tu-berlin.de
George Francis[†] gfrancis@uiuc.edu
Camille Goudeseune[†] cog@uiuc.edu
Jim Crowell[†] jimc@uiuc.edu
Hank Kaczmarski[†] hank@isl.uiuc.edu

ABSTRACT
Distributed scene graphs are important in virtual reality, both in collaborative virtual environments and in cluster rendering. In Myriad, individual scene graphs form a peer-to-peer network whose connections filter scene graph updates and create flexible relationships between scene graph nodes in the various peers. Modern scalable visualization systems often feature high intracluster throughput, but collaborative virtual environments (VEs) over a WAN share data at much lower rates, complicating the use of one scene graph system across the whole application. To avoid these difficulties, Myriad uses fine-grained sharing, whereby sharing properties of individual scene graph nodes can be dynamically changed from C++ and Python, and transient inconsistency, which relaxes resource requirements in collaborative VEs. A test application, WorldWideCrowd, implements these methods to demonstrate collaborative prototyping of a 300-avatar crowd animation viewed on two PC-cluster displays and edited on low-powered laptops, desktops, and even over a WAN.

Categories and Subject Descriptors
I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism – *virtual reality*

General Terms
Performance, design.

Keywords
Virtual environments, PC cluster, peer-to-peer.

1. INTRODUCTION
Myriad achieves scalability for collaborative virtual worlds by (1) peer-to-peer connectivity using point-to-point communications, (2) fine-grained sharing controllable down to the level of an individual scene graph node, (3) transient inconsistency, (4) self-regulating feedback along connections between peers, and (5) PC cluster visualization.

We call each scene graph, along with its network connections and machinery for filtering update messages, a *reality peer*. It is implemented as a C++ object. Reality peers are analogous to constructions in other distributed VR systems such as *locales* [7] or *worlds* [2]. A *reality map* filters messages on each end of the connection between reality peers, enabling Myriad's fine-grained sharing (see Section 6). The reality peers in a given network need not all have the same contents. Each might hold only part of a larger environment, different versions of the same environment, or partially shared versions of a single environment. All of these conditions create inconsistency. However, Myriad provides an API by which the system and its users can manage inconsistency, both creating and destroying it. Since this inconsistency is tolerable, correctable, and even sometimes desirable, we say that Myriad has *transient inconsistency*.

These concepts are explored in WorldWideCrowd, a collaborative prototyping application for assembling avatar mobs. Our test case uses 300 avatars, each with 20 bones animated by 60 frame per second (fps) motion-capture clips. Within the reality peers hosting the mob's pieces, the avatar motions generate 360,000 scene graph updates per second. In the current unoptimized system, each bone update creates a separate message containing a 4x4 matrix, about 100 bytes including the message header. To view, navigate, and collaboratively edit this scene over low bandwidth WAN connections requires all of Myriad's features.

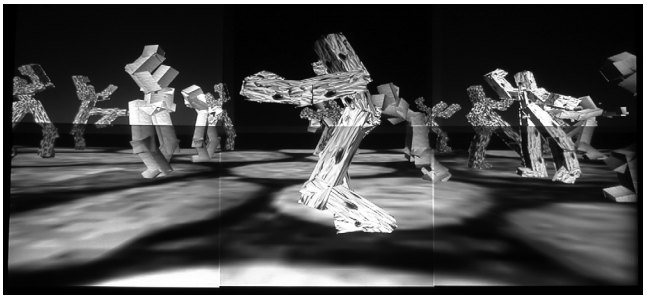

Figure 1. A VR view of Cubist-like segmented avatars in the WorldWideCrowd prototyper.

In general, a virtual world is not only its current graphical state but also the update sources that modify it. These include the transform updates that move an avatar's limbs, the slow changes when a user edits the scene, or the fast changing of a mesh in

VRST'05, November 7-9, 2005, Monterey, CA, USA.

[†] University of Illinois at Urbana-Champaign, 405 N. Mathews St, Urbana IL 61801, USA; 217-333-1000.

[††] TU Berlin, Inst. für Mathematik, Sekretariat MA 3-2, Straβe des 17. Juni 136, D-10623 Berlin, Germany; +49-30-314-29281.

response to a cloth simulation. In Myriad, scene graph information flows through a decentralized network of reality peers, which can be configured to meet the demands of particular applications. Each peer in the network can filter or modify an update to the scene graph before passing it on to other peers.

Myriad focuses on the mechanics of managing this peer-to-peer network of virtual worlds. It builds on the work done over the last decade in distributed and collaborative virtual environments, especially in terms of distributed scene graphs and update message filtering [16]. It supports asymmetrical networking, computation, and visualization capabilities within a single collaborative session. The filtering and transient inconsistency that make this possible also support another important tool for collaborative prototyping as in WorldWideCrowd, namely locally modifying a reality peer without propagating the change to other peers.

To create Myriad's reality peers, we extended the open source VR toolkit Syzygy [17]. This library contains a distributed scene graph intended for tightly synchronized display across PC clusters. In Syzygy, a scene graph application alters a local (server) scene graph copy, producing a sequence of update messages. These are sent to the cluster's render computers, where client scene graphs synchronize themselves with the server copy. Under our extensions, a reality peer simply generalizes a Syzygy scene graph.

Any computer running Python can script or interactively manipulate the network of reality peers. The peers are thus building blocks for constructing ever more elaborate virtual environments. Users can create new peers, make and break connections between them, and alter how those connections filter update messages. Additionally, Myriad's Python bindings can manipulate individual scene graph nodes within a peer, even remotely. This lets users prototype virtual worlds over a full spectrum of scales, all from within a Python interpreter.

2. PREVIOUS WORK

VR researchers have long studied Collaborative Virtual Environments (CVEs). In general, CVEs conceive of a virtual world or collection of worlds connected by portals or other mechanisms, containing defined objects. Additionally, some let instances of a shared world have variations not present in other copies, giving rise to *subjective views* [18] or *local variations* [9]. Differences between systems arise in what kinds of objects are shared, how that sharing occurs, how objects are updated, and how local variations are supported.

Some systems are domain-specific in terms of the objects and information they share: older incarnations of NPSNET [13] are tuned for vehicles or avatars. In contrast, projects like Continuum [4] or CAVERNsoft [11, 15] use a general shared object system. A middle ground is found in scene-graph-based CVEs that share graphics information, but in a way suitable for generic tasks: DIVE [2, 8], Repo 3D [12], Avango (formerly Avocado) [20], and Distributed Open Inventor [9]. Since Myriad also shares data through a general scene graph, it is most closely related to these systems.

Other differences involve how changes propagate through the CVE. Updates might propagate by a point-to-point protocol (reliable or unreliable) or via multicast for scalability (DIVE [2], MASSIVE-2 [6], NPSNET [1, 13]). In each of these cases, multicast groups are closely tied to shared world structure. In DIVE and MASSIVE-2, specific scene graph nodes correspond to multicast groups, with nodes below them shared in that group. In NPSNET, the world is regularly tiled by subworlds, each with its own multicast group. In contrast, user applications manually specify how CAVERNsoft's object updates travel, e.g. over TCP, unicast UDP, or multicast. Myriad also lets updates use any transport mechanism, but pays special attention to point-to-point connections and filtering thereon.

Multicast communications unfortunately present the same packet flow to all peers, even if some want updates at a lower rate. A peer might need reduced sharing because it renders slowly, has less bandwidth, or has limited update-processing power. Myriad handles such situations because they arise with very large or very active worlds like in our WorldWideCrowd application.

MASSIVE-3 [7] also addresses these concerns somewhat. Its large virtual worlds are composed of *locales* which themselves can have multiple *aspects*. To improve accessibility, when a user connects to a locale, bandwidth considerations can affect which aspect the user sees. This coarse-grained sharing contrasts with Myriad's fine-grained sharing and use of connection feedback, as described below.

CVEs also differ in how they filter updates. MASSIVE-3 has been augmented by an extensible message processing framework that uses *deep behaviors* [16], properties of an object that control how it processes scene graph updates. In this way the programmer can, for instance, change the logging behavior of particular objects, possibly enabling persistence at lesser disk access cost or creating objects that support reliable transactions. Myriad's message filtering methods are described in Section 6.

Myriad's reality peers and their connections form a general network propagating scene graph updates. Each peer may alter or even discard these updates. The MASSIVE systems [7, 16] connect virtual world databases more simply, essentially with paths in the connection graph of at most one hop (from client database to server database). However, Myriad supports arbitrarily long paths.

Still, remarks in [7] suggest that the architects considered erasing the distinction between database clients and servers (making every node a potential server), or using a tree of servers to efficiently propagate changes via TCP. This second idea is similar to the DIVEBONE [5], a network of special applications that DIVE uses as an application-level tunnel for multicast traffic through the internet. This construction in particular inspired Myriad's network of reality peers, though our implementation differs in its uniformity. Each object relaying or processing scene graph updates is a full reality peer. While Myriad does connect peers using Syzygy's connection broker (Section 5), this service plays no part in their subsequent interaction.

Message processing determines the path updates take through the CVE. For instance, a change might have to round-trip to a server before heading back to its originating program and finally registering its update. Even multicast-based systems have this property, since it guarantees total ordering of world events and thus world consistency, and Repo-3D and Distributed Open Inventor both have it to some degree. Unfortunately this adds

latency, creates a bottleneck at the sequencer, and doubles the bandwidth usage.

To avoid latency under these circumstances, programs like Repo-3D [12] can create *local variations* in the shared world whereby changes immediately affect the local copy before taking effect elsewhere. This reduces interaction latency for geographically dispersed participants.

Local variations can be more elaborate. DIVE can embed completely different subjective views in a single world [18]: a viewing program uses an identification string to choose which view to display. Distributed Open Inventor also has highly developed support for local variations [9]. In this case, unshared subgraphs can be grafted onto a shared scene graph or different, separately shared, scene graphs can be composed into a single entity. Myriad's transient inconsistency accomplishes these things and generalizes previous work (see Section 8).

3. WORLDWIDECROWD

Reality peer networks let us construct CVEs over low-bandwidth networks which can be experienced (or more importantly, modified) on a wide variety of devices. In the WorldWideCrowd application, even though the aggregate scene graph updates exceed a WAN's capacity, we can collaboratively edit the crowd in real time over the WAN by sharing avatar geometry and positions but not their motions. Instead each site can separately generate crowd motions, or motions of small sets of avatars can be shared, with the sharing dynamically controllable. This methodology also applies to other system bottlenecks, like those confronted when using underpowered (CPU-bound) displays like laptops and PDAs.

In WorldWideCrowd, we edited a shared world containing 300 segmented avatars. The entire crowd could not run on a single computer because of bottlenecks in processing scene graph updates and sending them over the network. Consequently, each of six crowd pieces ran within a reality peer on a different computer (a simulation cluster). For high-end visualization, there were two PC cluster displays, a 3x2 video wall and a six-sided CAVE [3] (Figure 2). The video wall was driven by a motley assortment of PCs, mostly 1 GHz Pentium-IIIs with GeForce 2 graphics cards.

Switched 100 Mbps Ethernet connected these resources, with 1 Gbps within each visualization cluster. In addition, a workstation near the video wall and a wireless laptop near the CAVE let users interactively change the CVE from the Python interpreter. Inside the CAVE, a participant navigated the world in standard VR fashion; another participant panned and zoomed the video wall from a desktop interface. Finally, a remote user manipulated the CVE over an 800-mile 10 Mbps link.

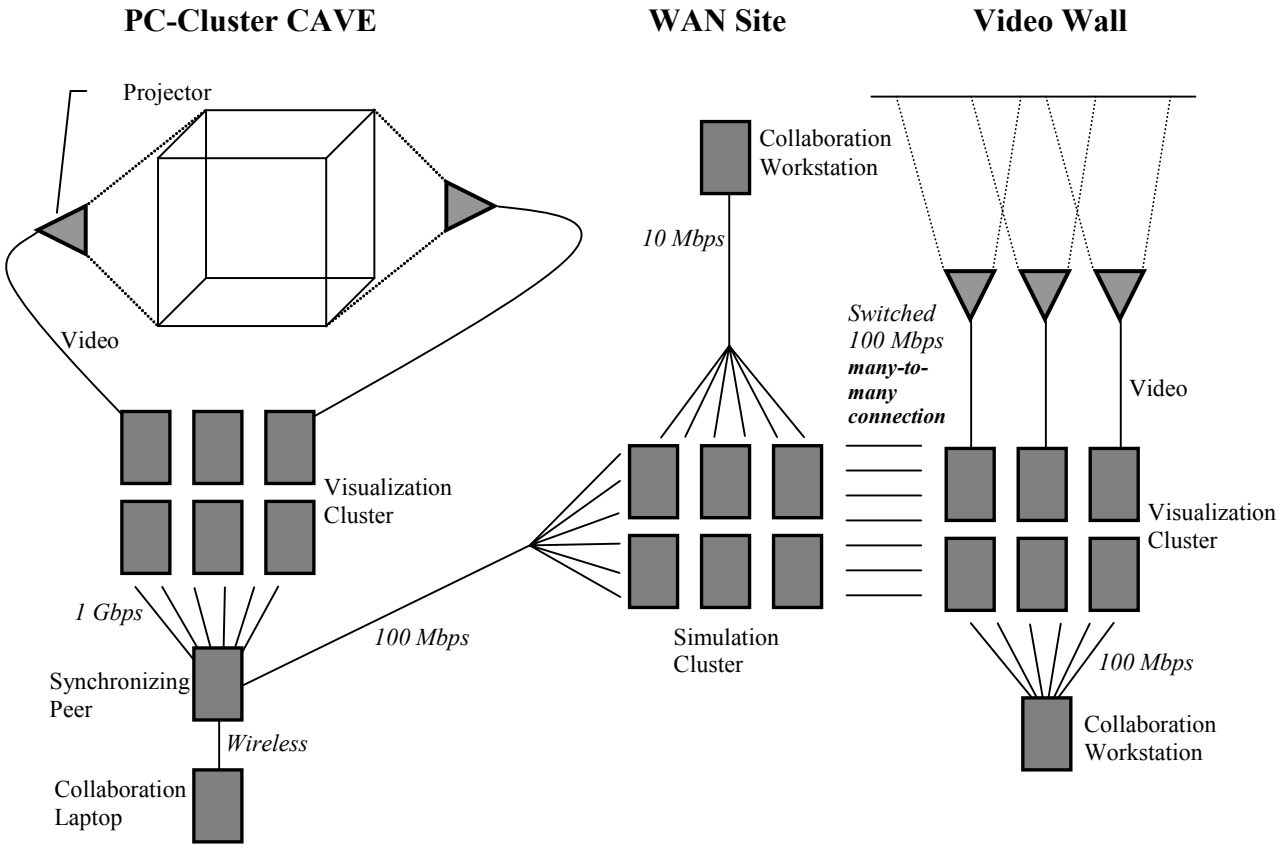

Figure 2. WorldWideCrowd: The validation configuration.

The video wall provided an overhead crowd view. Due to the variety of PCs powering its displays, OpenGL buffer swaps were not synchronized. When the crowd uniformly covered all six displays, frame rates were 10 to 25 fps, and the visualization cluster processed about 80,000 scene graph updates per second. This was sustained while panning across the scene and zooming the camera in and out, as demonstrated in the accompanying video. The overall update rate was limited chiefly by the video cards and secondarily by the 100 Mbps link between the video wall and the simulation cluster.

Figure 3. WorldWideCrowd: Two walls of the 6-sided PC cluster CAVE.

A cluster of 6 PCs, with genlocked video cards for active stereo, drove the six-sided CAVE. This let users navigate through the crowd at ground level. Here, for best visual quality, the buffer swaps on the wall displays were synchronized, and a special reality peer (a synchronizing peer) mirrored its scene graph state in each display. While the CAVE graphics cards were more powerful than those in the video wall, they were still a bottleneck, running at about 24 stereo frames per second. The synchronizing peer consumed about 24,000 updates per second, much less than that produced by the whole crowd due to motion culling. If we count updates to the 6 walls and the synchronizing peer separately, the cluster processed about 165,000 scene graph updates per second.

For the accompanying video, we also displayed the ground-level VR view on a 3x2 video wall, using a synchronizing peer. The video wall's increased view frustum culling resulted in about 40 monoscopic fps (depending on the precise point of view), even with three-year-old video cards. Again, the graphics cards and not the networking or update processing limited the frame rate.

The CAVE, the video wall, and the simulation cluster produced and exchanged the vast majority of scene graph updates. The total traffic for all reality peers within the core system was about 600,000 updates per second, with the CAVE cluster accounting for 165,000, the simulation computers 360,000, and the video wall 80,000.

Under some circumstances, reality peers outside the core described above would send avatar motion information into the system. For instance, users gave a particular avatar (or even a whole set of avatars) different motions than those shared globally. This local variation was accomplished by running a reality peer that streamed the desired clips, connecting it to the desired display peers, and turning off the transmission of avatar limb updates from the core simulators. Since this new motion was not transmitted globally, it did not increase overall network usage. If the user liked the new motion, they easily pushed it into the core simulation for others to see.

This same construction benefits CVEs which include a slow network link. In WorldWideCrowd, the simulation cluster's avatar animation programs choose motion clips based on the contents of special scene graph *info nodes*. Thus, if all data except the motion streaming is shared across the slow link, then two simulation clusters, one on each side of the slow link, can drive their respective crowds. Even if a collaborator over a WAN cannot stream the full crowd animation, he can certainly share each avatar's motion clip name, as embedded in an info node, and thus influence both simulation clusters.

In our test of WorldWideCrowd, in addition to trying different motion clips, users changed avatar geometry, even over running motions, either in the globally shared world or in their own local variations. Avatar geometry in a given reality peer was changed either by merging geometry from a file through that peer's RPC interface, or by directly manipulating its scene graph from the Python prompt. If users liked the avatar's new appearance, they could push it into the globally shared world, updating the other sites. If not, they could restore the old appearance from the shared world.

Figure 4. Avatar Manipulation.

The users moved the crowd's avatars by manipulating transform nodes inside the peers. As with other operations, changes in avatar position could be either local or shared, with new local changes able to become shared or to revert to the current shared state. In order to move avatars, users worked from the Python interpreter, getting local copies of the relevant transform nodes. From this point, users arranged the avatars with scripts as well as 6DOF input devices (Figure 4). In the second case, a manipulator object connected a transform node to the input device input and

subsequently coordinated changing the avatar's position, orientation, and scale.

4. MYRIAD SCENE GRAPH

Each reality peer contains a Syzygy scene graph, whose node types correspond to OpenGL commands. Nodes store geometry in a vertex array style, and can specify 3D transformations, texture maps, OpenGL materials, etc. [17]. Myriad extends Syzygy by adding an *info node* which stores a string and can add semantics to a scene graph, much like constructions in [7] and [8]. Each node has an ID, unique in the context of its owning scene graph. When an application executes a node method, it generates update messages, which the scene graph routes using this ID. Each node also has a name, possibly not unique. As shown below, node names help construct the reality maps between nodes in different peers.

Myriad adds a new property to all Syzygy scene graph nodes: *sharing level*. Its value is one of *transient*, *stable-optional*, or *stable-required*. The application marks a node as transient if its value changes rapidly over time; stable-optional if its value remains more or less fixed but is not critical to the proper functioning of the application; and stable-required if its value is critical. A node's sharing level affects how reality maps treat it (see Section 6) and how its updates are dynamically filtered before propagating to connected peers (see Section 7). Nodes with transient sharing level are called *transient nodes*.

5. PEER-TO-PEER CONNECTIVITY

When a reality peer starts, it registers a service with a connection broker provided by Syzygy. This broker gives the peer a unique ID for subsequent processing of the update stream passing through it. Other peers can query the broker's service registry, retrieve a list of all reality peers in the peer network, choose a remote peer by name, get its IP address and port, and then connect directly to it.

Each reality peer, then, connects to other peers. It listens for update messages on its various connections, filters them, and then propagates them. While arbitrarily complex networks are possible, several simple constructions illustrate important Myriad features: push peers, pull peers, feedback peers, and shadow peers. WorldWideCrowd uses all of these types.

A *pull peer* connects to a single remote peer and synchronizes its scene graph with a subgraph contained in the remote peer. Thereafter, it receives updates from the nodes in the remote subgraph but does not send any of its own. In WorldWideCrowd, this lets us build a new crowd that tracks the evolution of an existing one but within which we can make local changes, like adding new avatars, without modifying the original. The pull peer might even elect to only receive updates from certain remote scene graph nodes. For instance, ignoring all updates from transient nodes would drastically decrease its bandwidth requirements.

A *push peer* connects to remote peer(s) and synchronizes its own scene graph with a subgraph in each remote peer, afterwards sending updates but not receiving them. So the push peer affects the remote peers without being affected itself. In WorldWideCrowd, the crowd animation programs embed push peers, as do navigation utilities for moving through the virtual worlds.

A *feedback peer* connects to a single remote peer; synchronizing its scene graph with a subgraph in the remote peer upon connection, and subsequently both sends its own updates to the remote peer and receives updates from it. In WorldWideCrowd, feedback peers allow collaborative editing of a shared scene graph. A *shadow peer* is a special kind of feedback peer that shares only scene graph structure but not the contents of the nodes themselves. Since it does not download geometry, a shadow peer can quickly synchronize its scene graph structure with a remote peer, even over a low bandwidth link. In WorldWideCrowd, shadow peers are useful for editing scene graphs over the WAN.

To create large-scale distributed applications like WorldWideCrowd, the reality peer network must be scriptable. To this end, reality peers implement an RPC interface for loading and saving scene graph information, managing connections, and adjusting the sharing along each connection. Peers can enter the system in different ways. A program can embed a reality peer and, as part of its operation, alter that peer. On the other hand, *workspaces* act as generic containers for reality peers, allowing them to exist independently of other controlling programs. A workspace can create and delete peers residing within it, and, as with the reality peers themselves, it has an RPC interface to control these methods remotely.

6. FINE-GRAINED SHARING

Fine-grained sharing allows control over how connected reality peers share information, down to the level of an individual scene graph node. This section describes its three functional components: *message filtering*, *reality mapping*, and *node locking*. These features let peers support local variations, adapt to low-bandwidth network links, and combine with other peers to form larger virtual worlds.

Myriad's message filtering was inspired by recent incarnations of MASSIVE [16], but Myriad's filters are not properties of the scene graph nodes themselves (like deep behaviors). Instead, they are properties of node and peer connection pairs. Also, Myriad's fine-grained sharing contrasts with MASSIVE-3's coarse-grained locales and aspects [7], which are essentially entire scene graphs.

A reality map relates two peer scene graphs. It is stored on each side of the peer connection, associates remote node IDs with local ones, and determines if a given local node is *mapped* remotely. Updates to a scene graph node are propagated to mapped nodes in connected peers via a message filtering system, as outlined later in this section. Distributed Open Inventor [9] has a similar construct, except that its shared subgraphs have identical node structure (only the location of the root node changes).

Consistency requirements are relaxed in Myriad, stipulating only that if node B is a descendant of node A and both map into a connected peer, then B's image is also a descendant of A's image. This allows unmapped node insertions within either of the corresponding (mapped) subgraphs, facilitating fine-grained local variations. For example, a peer could locally insert a transform node and add unshared rotational motion to an object's shared translational motion, or, by locally inserting new material nodes, a peer could locally recolor a (shared) uniformly colored group of objects.

Another important element of fine-grained sharing is node locking, which provides an API to maintain consistency across peers. Suppose A and B are connected peers. Peer A can take one of its scene graph nodes and lock changes on mapped nodes in connected peer B. Subsequently, B will only let message updates from A change, delete, or add children to this node; this lets A make changes deterministically. Locking is implemented cooperatively. Any peer connected to B that holds a mapped image of the locked node can grab the lock from A, or even unlock it.

Given this introduction to fine-grained sharing, we describe what happens when an update message reaches a reality peer. First, each message stores a history of the peers it has updated; a message revisiting an already updated peer is discarded, preventing infinite loops. Next, the reality map associated with the message's incoming connection changes the update message's embedded node ID to the ID of the mapped local node, if any, and discards the message if none such exists. The message is also discarded if the newly determined destination node is locked by a peer other than its originator. Next, a sequence of user-defined message filters, which is set per node and peer connection pair, is applied before the update message is finally sent to its destination node in the local scene graph.

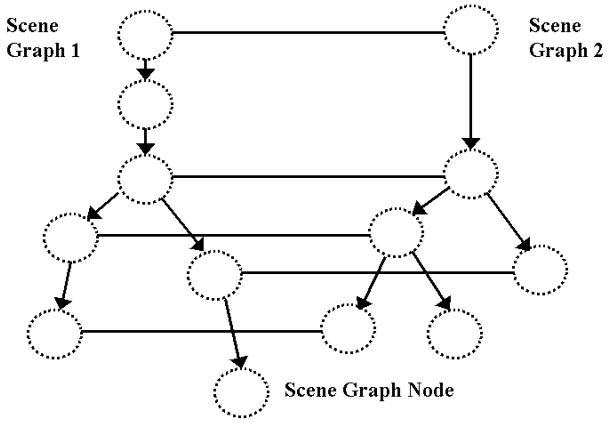

Figure 5. Reality mapping: The relationship between nodes of connected scene graphs.

After updating the local node, the peer executes the following for each outgoing connection. If the local node maps to a node in the connected peer, and the node is transient, and it is too soon to send a fresh update (based on the connected peer's update rate, see Section 7), then the message is discarded. Then another sequence of user-defined message filters is applied, and the message is sent if no filter discards it.

Once two peers connect, the reality map between them can be built in two different ways. One peer can create a new copy of its scene graph in a remote peer (but rooted at an arbitrary remote node), or it can attempt to associate local nodes with remote ones. In either case, the sending peer tells the receiving peer that it is constructing a reality map and specifies a remote node to be its root. This map has an associated sharing level that affects its construction, node by node, based on each node's sharing level.

The sending peer starts at a specified local node, traversing the scene graph below it. At each node, it sends a node creation message and, if the node's sharing level is greater than or equal to the sharing level specified for the map, a node state serialization message as well. Inside the receiving peer, the node creation messages extend the reality mapping, either by creating new nodes (as when building a copy) or by trying to associate the sending peer's node with an existing node in the receiving peer.

Consider the case where existing remote nodes (on the receiving peer) are associated with local ones (on the sending peer) if possible. The node creation message contains the ID of the parent node, the name of the new node, and the new node's type. Upon receipt of such a message, if the receiving peer's current reality map (for the given connection) cannot translate the parent ID to that of one of its own nodes, the node creation message is discarded. Otherwise, the receiving peer searches depth-first below the mapped parent for an existing node with the same name and type as the creation message. If such is found, the reality map is extended to include this node and the creation message is discarded; otherwise, a new node is created as a child of the mapped parent and the reality map (on the receiving peer) is updated. Either way, a message returns to the sending peer with the newly created extension for the sending peer's reality map. As constructed, reality maps automatically extend themselves over time, with new children of mapped nodes in one peer making new (mapped) nodes in connected peers.

The reality mapping algorithm depends on nodes having only unique IDs, not unique names, so the names can encode useful structural information. For instance, the avatar skeleton used in WorldWideCrowd uses a standard set of node names, making it easy to associate an avatar in one peer to an avatar in another via this mapping process. Together with reality mapping, this standardization lets WorldWideCrowd users replace any avatar body by any other embedded in another peer.

As described here, reality maps are a general concept with several useful properties in addition to supporting local variations. They let the scene graphs within a group of peers be combined inside a single peer, thus creating larger worlds. They can also share only a small part of a peer's scene graph with connected peers. When combined with message filtering, this greatly assists low-bandwidth collaboration. In particular, since a user or application need not decide in advance how to divide a peer's scene graph into sharing units (like locales), reality maps support dynamically modifiable sharing, which can limit shared data to precisely the area of interest at any given moment.

7. PEER CONNECTION FEEDBACK

If a group of reality peers all participate in a multicast group, all peers see the same updates, which makes the most sense when peers are similar. However, peer resources and capabilities might differ substantially. For instance, peers on a LAN or a high-speed WAN tested might have good network connectivity among themselves but have much less bandwidth to a peer outside their subgroup. Furthermore, peers may display their virtual worlds at different frame rates. Under this scenario, if each peer receives the same scene graph updates, the ones with slower update rates waste time processing updates only needed by the faster ones.

Myriad uses feedback to adjust the message flow along peer connections, dynamically adapting to changing peer networking and graphics performance. Feedback messages regulate the data

transfer between Myriad's reality peers by controlling the filtering of scene graph updates to transient nodes (see Section 4). A reality peer can send its preferred update rate to connected peers. This preferred update rate might match its graphics frame rate (which would be sent automatically), or it might explicitly throttle incoming bandwidth, e.g. requesting only one update per second. Each reality peer stores, per connection, the update rate requested by the remote peer. On that connection, it sends transient node updates only at the given rate.

If a reality peer's primary function is rendering its scene graph, transient node updates matter only for nodes likely to be viewed. Consequently, the user can configure a peer to automatically test if particular subgraphs of its scene graph are "near" the viewing frustum, where the proximity threshold is user-adjustable. If a subgraph is not near enough, connected peers alter the relevant reality maps so that they stop sending it transient node updates. When a subgraph is again in proximity to the viewing frustum, the reality maps are restored and updates from transient nodes in connected peers will again be received.

Under this simple scheme, every transient node a user views will have timely values, given appropriate bounds on user travel speed. Specifically, when a subgraph enters proximity with the viewing frustum, one message round trip to a connected peer must occur before the subgraph will again receive updates from that peer. Consequently, the user-defined proximity distance divided by the maximum ping time to a connected peer gives the speed bound. Note that Myriad itself does not enforce user speed limits (though a particular application might). Instead, in keeping with its overall philosophy, Myriad accepts that inconsistencies might arise.

Connection feedback is critical for WorldWideCrowd. Without it, our video wall could not smoothly pan the overhead crowd view, nor could a CAVE user navigate seamlessly through the whole crowd.

8. TRANSIENT INCONSISTENCY

We treat consistency as a local property of connections between peers, rather than a global property of the whole peer network. Consider two subgraphs (one in each of two connected reality peers) mapped onto each other by the connection's reality map. We call them *consistent* if they have identical structure and all pairs of mapped non-transient nodes have the same values. We call a connection consistent if the full scene graphs in connected peers are consistent.

Connected reality peers need not have consistent scene graphs, though configurations of reality peers may have varying degrees of guaranteed consistency (see the discussion of locking in Section 5). Myriad's controlled breakdown of consistency reduces its use of networking resources and facilitates virtual world prototyping by allowing linked (though inconsistent) versions of a world. On the other hand, some degree of consistency is obviously needed for collaboration and communication.

Consider two inconsistent subgraphs in connected reality peers, neither of which is currently being altered (except by the Myriad system itself). A *consistency process* is a Myriad background task that eventually, assuming it is sufficiently *strict*, makes these subgraphs consistent. We call this effect *transient inconsistency*. Myriad provides an API for launching consistency processes.

Transient inconsistency relaxes the resource requirements imposed on a CVE by strict consistency. For instance, a substantial though intermittent resource strain occurs when new users join the virtual world, dubbed the *late-joiner problem*. In many CVEs, all updates pause while the world state transfers atomically to the newcomer. Even if the world state is only a few megabytes and the LAN has 100 Mbps speed, this freezes all viewers for a substantial fraction of a second. On a slower WAN with frequent newcomers, usability significantly deteriorates. While atomic state transfer ensures that all users always see the same world, it limits scalability. By allowing gradual, non-blocking background transfer of world state, Myriad eliminates these delays at the cost of temporarily presenting different pictures of the world to peers.

Myriad's solution to the late-joiner problem uses a consistency process and builds on its concept of reality maps. When a peer maps part of its scene graph into another peer (transferring world state), it does so in the background and without locking the whole scene graph. The originating peer traverses its scene graph depth-first. It maps nodes, transferring node state according to the map's sharing level (see Section 5); we call the process *strict* if every node's state is transferred. Non-strict consistency processes need less bandwidth. For instance, if a user only needs to manipulate a scene graph's structure (as with shadow peers), node creation messages suffice to map it to a remote peer and no internal node state needs to be sent.

The application controls the mapping's traversal rate so that state transfer does not overload network or CPU resources. The transfer may even be greatly prolonged without impacting usability, since new node updates interleave with the transfer-specific updates ("streaming VR"). During the transfer, once a node has been mapped, its future updates are immediately sent to the connected peer, and filtering discards updates to as yet unmapped nodes.

We now argue that inconsistencies during a strict scene graph transfer are, in fact, transient. Suppose that the new nodes in the target scene graph remain unaltered during the mapping (if they change, a subsequent consistency process can *reconcile* them). After the mapping finishes, unless nodes were created or deleted in the original scene graph during the mapping, the new scene graph must be consistent with the original. This gives a bound on the duration of inconsistencies; i.e., the inconsistencies are transient.

We now show node creation and deletion in the originating scene graph are also acceptable. Myriad handles node deletion like any other update. Suppose a node in the original peer is deleted during the mapping. If it has already been mapped, the update message deleting it is passed on as well, deleting the node's image. If it has not been mapped, the delete message is not sent to the connected peer, which will never have a corresponding node because no local copy now exists to map.

The scene graph transfer similarly allows interleaved node creation. If a new child node is created before its parent has been mapped, the message adding the child is discarded by the sending peer's outbound filter. Later, the consistency process will map the parent and then the child. Otherwise, the child is created after the parent has been mapped, and so it is immediately added in both the local and connected peers.

Myriad solves late-joiner problem similarly to MASSIVE-3 [7], where state transfer to the late joiner also occurs in the background without locking the scene graph and freezing other users. However, MASSIVE-3 sends a full scene graph to the new user before sending new updates, whereas Myriad interleaves new updates with the initial state transfer.

In addition to supporting incremental background transfer of scene graph information to late-joiners, Myriad also lets consistency processes reconcile existing inconsistent subgraphs in connected reality peers. Once users have stopped altering the subgraphs, the background consistency process will eventually make them consistent. It traverses a reality peer's scene graph and compares it to a connected scene graph, bringing (part of) the remote scene graph into a preset level of agreement with the first.

At its most stringent, the consistency process's goals are: mapped nodes on both sides of the connection have the same values; every node has a mapped remote node; and every remote node maps to some local node. Under these circumstances and when not competing with user changes, an inconsistency's lifetime is bounded by how long the consistency process takes to traverse the scene graph, which depends on its assigned CPU and networking resources.

An important type of inconsistency is a local variation or subjective view. This may be long lasting, since users specifically create it to e.g. compare different versions of a world. These features are supported in CVEs like Distributed Open Inventor [9] and DIVE [18]. Myriad supports analogous constructions, tolerating inconsistencies and providing a means to reconcile them.

DIVE's subjective views assume that each shared view is simultaneously available to all participants. This single shared world can become a bottleneck when many subjective views exist, even for static content. Moreover, if each shared subjective view generates many updates, these can strain the network because they are forced into a single multicast group and sent to every peer.

This case occurs in practice. In WorldWideCrowd, a commonly tried local variation is animating a collection of avatars with new motion clips not available on the simulation cluster. Since each variation uses substantial bandwidth, it would be expensive to transmit them all to every peer, although some sharing may be desired between certain peers. Myriad can partition scene graph updates so that they go only where they are needed, reducing required bandwidth.

Also, a user may want to try a local variation, change the globally shared state if it succeeds, and otherwise return the world to the globally shared state. Here the world's division into shared and unshared pieces changes dynamically. Myriad's transient inconsistency explicitly lets worlds lose and regain synchronization like this.

9. PC CLUSTER VISUALIZATION

Many researchers have created cluster graphics solutions in the last few years, like Chromium (formerly WireGL) [10], Cluster Juggler [14], and Syzygy [17]. There are various cluster visualization situations. Visual quality might be the most important consideration, requiring all screens to display the virtual world synchronized frame by frame. On the other hand, synchronization might be sacrificed in order to display as much information as possible at one time.

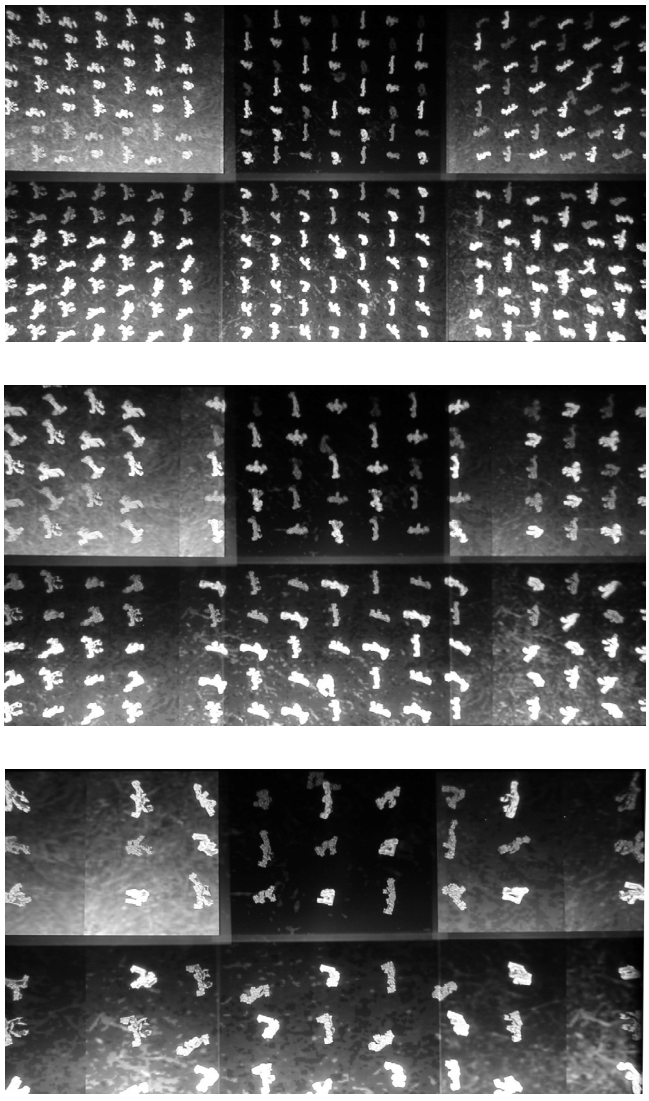

Figure 6. WorldWideCrowd: A smooth zoom into the crowd.

When displays must be synchronized, the slowest graphics cards limit the overall frame rate, though view frustum culling can ameliorate this. Myriad's cluster synchronization scheme routes all update messages through a central synchronizing peer. This peer ensures that each render PC's drawn scene graph is identical to the others at each frame, achieving this by synchronizing their buffer swaps and their consumption of update messages, as in [17]. Unfortunately, processing this many updates can create a CPU bottleneck at the synchronizing peer. A 3 MHz Pentium-IV processes about 400,000 scene graph updates per second, which is similar to the update rate of the 300-avatar crowd. Network utilization at the synchronizer is also a problem.

Consequently, the overhead view in WorldWideCrowd, which requires maximum scalability, does not use frame by frame synchronization. Instead, the video wall's reality peers display

scene graph updates as they receive them. The overhead view helps partition the drawing and message processing among the peers, increasing each one's potential frame rate. By reducing inter-screen synchronization, each render PC can connect separately to the crowd simulation peers (many-to-many) and to a navigation controller that allows, for instance, panning the scene back and forth. Even though the computers driving the video wall have significantly different performance, this works well in practice, as shown in the accompanying video's slow zoom into the whole 300-avatar crowd (Figure 6).

The PC cluster CAVE, however, requires a frame-synchronized display. This critically depends on the fact that, in Myriad, sharing is a local property of connections between peers instead of a global property of the peers themselves. Consequently, the synchronizing peer can just push the updates it receives to the display peers, without regard to their origin, even while dynamically changing its sharing with peers outside the cluster. In contrast, sharing is a global property (of subgraphs) in Distributed Open Inventor. There, while an application's scene graph can be composed using any number of separately shared and unshared subgraphs, the overall scene graph is not transparently shareable.

10. INTERACTION VIA PYTHON INTERPRETER

Two levels of interaction are supported in Python, both using scripts and in the interpreter. The first involves manipulating Myriad's distributed system at a high level, managing workspaces, reality peers, and their connections. A collection of Python proxy objects for these entities lets the actual running objects, located anywhere in the network, be manipulated using an RPC interface. This makes it possible to set up, tear down, and manage a distributed application like WorldWideCrowd.

The RPC interface is also useful for creating ad hoc workflows. For instance, a user might experiment with local variations of the scene graph contained in a particular reality peer. A Python script can start several new peers on a collection of specified computers, each a pull peer with respect to the original (defined in Section 5). These new peers can then be altered to create new scene graph versions, and, if they run on different computers, each will run with high performance, aiding their comparison. Using the same scripting infrastructure, the user can automate killing no longer needed views and can also, remotely, serialize any peer's scene graph and save it to a file, preserving successful intermediate prototypes.

The second level of interaction is manipulating the Syzygy/Myriad objects themselves. We use SWIG-generated Python wrappers for the relevant objects, such as the reality peers and the nodes their scene graphs contain. With the resulting Python modules, these objects can be created at a Python prompt or called from a Python script. In this way, a user can, for instance, start a reality peer from a Python prompt and connect it as a feedback peer (see Section 5) to another peer. The user can then alter the remote peer's nodes by changing mapped nodes in the local one, all from the Python prompt, encouraging free-form experimentation.

Each scene graph node's full C++ interface is available from Python. The user can set material properties, alter lighting, change transform matrices, and even move individual points within triangle meshes. The Python bindings let users connect 6DOF input devices to transform nodes in local peers, with special manipulator objects forming the bridge and providing a rudimentary interface for the interaction (Figure 4). Since the local peers can affect remote ones, any transform node in the peer network can be manipulated in this way.

11. FUTURE WORK

This work can be extended in many ways. First and foremost, we could explore system performance on modern hardware. For instance, experiments indicate that two 3 GHz Pentium computers with recent GeForce FX graphics cards can transfer and display a 50-avatar crowd over a 100 Mbps link at 40 fps, processing 40,000 scene graph updates per second on the display end. We could measure performance on a modern 6-node display cluster connected by a gigabit switch to a modern 6-node compute cluster. On such hardware, Myriad might animate 1200 avatars at interactive frame rates.

Furthermore, while the Myriad applications described in this paper are fairly large, comprising 22 computers for the full WorldWideCrowd demo, we have yet to see how Myriad scales to hundreds or thousands of computers. Such a system would harness supercomputer-level power when rendering and driving virtual worlds. This leads to a potential practical use of Myriad: real-time visualization and collaborative prototyping of large, data-intensive worlds for the movie industry.

Optimizing the underlying scene graph could significantly increase rendering speed. Vertex programs running on programmable GPUs might accelerate deep scene graph traversals like those associated with segmented avatars. Furthermore, Myriad uses overly general scene graph updates in some cases, sending a 4x4 matrix to control each avatar bone instead of only three Euler angles. The latter approach would reduce bandwidth, but it might increase computation when reconstituting the rotation matrix for OpenGL. Experiments would help in understanding these trade-offs.

Myriad's high-quality cluster rendering, which features frame-by-frame synchronized scene graphs, could also be improved. Currently a single synchronizing peer guarantees consistency of each set of cluster-rendered frames. This potential network and computational bottleneck is removed if we let multiple data sources independently (and even dynamically) synchronize and desynchronize with the cluster rendering PCs. The difficulty comes from coordinating the already synchronized video frames from the PC cluster with the data sources as they come and go. Note that conventional parallel programming APIs like MPI fix synchronization groups at their creation.

Finally, the API seen by developers can always be refined. The underlying scene graph API comes from Syzygy and is relatively mature, but the Myriad-specific APIs for manipulating connections between reality peers are still evolving. Relative to other CVE systems, Myriad exposes additional complexity in its ability to make fine manipulations of the peer network, and the best methods for managing that complexity in a wide range of situations are still unknown.

The software described in this paper is open source and is available at http://www.isl.uiuc.edu, along with all of the data files necessary to reproduce our experiments.

12. REFERENCES

[1] Capps, M., McGregor, D., Brutzman, D., and Zyda, M. 2000. NPSNET-V: A New Beginning for Dynamically Extensible Virtual Environments, *IEEE Computer Graphics and Applications*, *20*, 5, 12-15.

[2] Carlsson, C. and Hagsand, O. 1993. DIVE- A Platform for Multi-User Virtual Environments, *Computers & Graphics, 17, 6*, 663-669.

[3] Cruz-Neira, C., Sandin, D., and DeFanti, T. 1993. Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE. *Computer Graphics*, ACM SIGGRAPH 1993, 135-142

[4] Dang Tran, F., Deslaugiers, M., Gerodolle, A., Hazard, L., and Rivierre, N. 2002. An Open Middleware System for Large-scale Networked Virtual Environments, In *Proc. IEEE Virtual Reality 2002*, 22-29.

[5] Frecon, E., Greenhalgh, C., and Stenius, M. 1999. The DIVEBONE- An Application-Level Network Architecture for Internet-Based CVEs. In *Proc. VRST 1999*, 58-65.

[6] Greenhalgh, C.M. 1998. Awareness Management in the MASSIVE Systems, Distributed Systems Engineering, 5, 3, 129-137.

[7] Greenhalgh, C., Purbrick, J., and Snowdon, D. 2000. Inside MASSIVE-3: Flexible Support for Data Consistency and World Structuring, In *Proc. CVE 2000*, 119-127.

[8] Hangsand, O. 1996. Interactive Multiuser VEs in the DIVE System, *IEEE Multimedia, 3, 1*, 30-39.

[9] Hesina, G., Schmalstieg, D., Fuhrman, A., and Purgathofer, W. 1999. Distributed Open Inventor: A Practical Approach to Distributed 3D Graphics. In *Proc. VRST 1999*, 74-81.

[10] Humphreys, G., Eldridge, M., Buck, I., Stoll, G., Everett, M., and Hanrahan, P. WireGL: A Scalable Graphics System for Clusters. *Computer Graphics*, ACM SIGGRAPH 2001:129-140.

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

[12] MacIntyre, B. and Feiner, S. 1998. A distributed 3D graphics library. In *Proc. ACM SIGGRAPH 1998*, ACM Press / ACM SIGGRAPH, 361-370.

[13] Macedonia, M., Zyda, M., Pratt, D., Barham, P., and Zeswitz, S. 1994. NPSNET : A Network Software Acrhitecture for Large-Scale Virtual Environments, *Presence*, *3*, 4, 265-287.

[14] Olson, E. 2002. *Cluster Juggler – PC Cluster Virtual Reality*. M.Sc. thesis, Iowa State University.

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

[16] Purbrick, J. and Greenhalgh, C. 2002. An Extensible Event-Based Infrastruture for Networked Virtual Worlds, In *Proc. IEEE Virtual Reality 2002*, 15-21.

[17] Schaeffer, B. and Goudeseune, C. 2003. Syzygy: Native PC Cluster VR, In *Proc. IEEE Virtual Reality 2003*, 15-22.

[18] Smith, G. 1996. Cooperative Virtual Environments: Lessons From 2D Multi User Interfaces, in *Computer Supported Cooperative Work '96*, 390-398.

[19] Snowdon, D., Greenhalgh, C., and Benford, S. 1995. What You See is Not What I See: Subjectivity in Virtual Environments, In *Proc. Framework for Immersive Virtual Environmments – FIVE '95*, London.

[20] Tramberend, H. 1999. Avocado: a distributed virtual reality framework. In *Proc. IEEE Virtual Reality 1999*, 14-21.