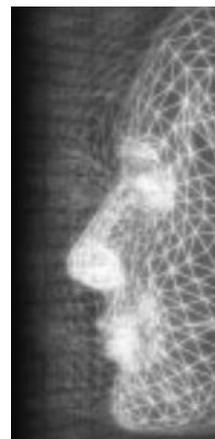# Myriad: Scalable VR via peer-to-peer connectivity, PC clustering, and transient inconsistency

By Benjamin Schaeffer*, Peter Brinkmann, George Francis, Camille Goudeseune, Jim Crowell and Hank Kaczmarski

Distributed scene graphs are important in virtual reality, both in collaborative virtual environments and in cluster rendering. Modern scalable visualization systems have high local throughput, but collaborative virtual environments (VEs) over a wide-area network (WAN) share data at much lower rates. This complicates the use of one scene graph across the whole application. Myriad is an extension of the Syzygy VR toolkit in which individual scene graphs form a peer-to-peer network. Myriad connections filter scene graph updates and create flexible relationships between nodes of the scene graph. Myriad's sharing is fine-grained: the properties of individual scene graph nodes to share are dynamically specified (in C++ or Python). Myriad permits transient inconsistency, relaxing resource requirements in collaborative VEs. A test application, WorldWideCrowd, demonstrates collaborative prototyping of a 300-avatar crowd animation viewed on two PC-cluster displays and edited on low-powered laptops, desktops, and over a WAN. We have further used our framework to facilitate collaborative educational experiences and as a vehicle for undergraduates to experiment with shared virtual worlds. Copyright © 2006 John Wiley & Sons, Ltd.

Introduction

How should we handle sharing of data across collaborative virtual environments when sharing complete data would lead to unacceptable performance degradation? We describe an approach to solving this problem; our implementation of this solution is called *Myriad*.

Myriad achieves scalability for collaborative virtual worlds by supporting: (1) peer-to-peer connectivity using point-to-point communications; (2) fine-grained sharing, controllable at the level of individual scene graph nodes; (3) transient inconsistency; (4) self-regulating feedback between peers; and (5) PC cluster visualization. This paper uses 'world' and 'scene graph' interchangeably.

*Correspondence to: B. Schaeffer, Beckman Institute, University of Illinois at Urbana-Champaign, 405 N. Mathews St, Urbana IL 61801, USA. E-mail: ben.schaeffer@gmail.com

We call each scene graph, together 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* or *worlds*.[1,2] A *reality map* filters messages at each end of the connection between reality peers (see the section on fine-grained sharing). The reality peers in a given network need not all have the same content. Each might hold only part of a larger world, different versions of the same world, or partially shared versions of a single world. All these conditions create inconsistency. However, Myriad lets its users introduce, manage, and remove inconsistency. Because this inconsistency is tolerable, correctable, and even sometimes desirable, we say that Myriad has *transient inconsistency*.

Virtual worlds are updated by various information sources with different temporal properties. These include the transform updates that move an avatar's limbs, the slow changes when a scene is edited, or the

fast changing of a mesh in 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 builds on the last decade of work in distributed and collaborative virtual environments, particularly distributed scene graphs and update message filtering.[3] It supports asymmetrical networking, computation, and visualization within a single collaborative session. The filtering and transient inconsistency that make this possible enable another important tool for collaborative prototyping: namely, local modification of a reality peer without propagation of the change to other peers.

To create Myriad's reality peers, we extended the open-source VR toolkit Syzygy.[4] This library contains a distributed scene graph intended for tightly synchronized display within a PC cluster. 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. A Myriad reality peer is a generalization of the 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 interface lets users manipulate individual scene graph nodes within a peer, even remotely.

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

In addition to stress testing, our framework with WorldWideCrowd, we have tried testing its usability. To that end, Myriad has formed part of the curriculum of two semester long classes.

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 and containing objects such as human avatars, vehicles, buildings, or more abstract visualizations of data. Additionally, some CVEs allow variation between instances of a shared world, giving rise to *subjective views* or *local variations*.[5,6] CVEs differ in the kinds of objects that are shared, how sharing occurs, how objects are updated, and how local variations are supported.

Some CVEs share domain-specific objects and information: early versions of NPSNET,[7] for example, were tuned for vehicles or avatars. In contrast, Continuum[8] and CAVERNsoft[9,10] use general shared objects. A middle ground is found in CVEs that share scene graphs in a way suitable for generic tasks: DIVE,[2,11] Repo-3D,[12] Avango[13] (formerly Avocado), and Distributed Open Inventor.[6] Myriad also shares data through a general scene graph, so it is most closely related to these latter systems.

Other differences involve how changes propagate through a CVE. Updates might propagate from point to point (reliably or unreliably), or via multicast for scalability (DIVE,[2] MASSIVE-2,[14] NPSNET[7,15]). 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. NPSNET's world is regularly tiled by subworlds, each with its own multicast group. In contrast, CAVERNsoft's user applications manually specify how object updates travel, for example, over TCP, UDP, or multicast. Myriad also lets updates use any transport mechanism, but it 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 require updates at a lower rate. A peer might need reduced sharing—particularly in the context of a very large, active CVE such as WorldWideCrowd—because it renders slowly, has less bandwidth, or has limited update-processing power. Myriad handles such situations.

MASSIVE-3 also addresses these concerns to some extent.[1] Its large virtual worlds are composed of *locales*. Each local has multiple *aspects*. When a user connects to a locale, bandwidth determines which aspect (how much detail) the user sees. This subworld-level sharing contrasts with Myriad's finer-grained object-level sharing and connection feedback, described below.

CVEs also vary in how they filter updates. MASSIVE-3 has been augmented with extensible message processing that uses *deep behaviors*, properties of an object that control how it processes scene graph updates.[3] For example, changing how an object handles updates can reduce the disk access cost of the object's persistence. Another change could let the object support reliable transactions. Myriad's message-filtering methods are described in the section on fine-grained sharing.

Myriad's reality peers and their connections form a general network propagating scene graph updates. Each peer may alter or even discard these updates in the process of relaying them to other peers. The MASSIVE systems connect virtual world databases more simply, with paths in the connection graph of at most one hop (from client database to server database).[1,3] Its architects may have 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.[1] This second idea is similar to the DIVEBONE, a network of special applications that DIVE uses as an application-level tunnel for multicast traffic through the internet.[16] DIVEBONE inspired Myriad's network of reality peers, although Myriad 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 (see the section on peer-to-peer connectivity), this broker plays no part in the peers' subsequent interaction.

Message processing determines the path updates take through a CVE. For example, an update message might have to travel to a server and back before effecting a change in the database of the originating program. Even multicast-based systems have this property, because this property guarantees total ordering of world events and thus world consistency. Repo-3D and Distributed Open Inventor both have it to some degree. Unfortunately it adds latency, creates a bottleneck at the sequencer, and doubles bandwidth usage.

To avoid latency under these circumstances, programs like Repo-3D allow *local variations* in the shared world; changes immediately affect the local copy before taking effect elsewhere.[12] This reduces interaction latency among geographically dispersed participants. Local variations can be more elaborate. DIVE can embed completely different subjective views in a single world: a viewing program displays one of several views.[5] Distributed Open Inventor also has highly developed support for local variations.[6] In this case, unshared subgraphs can be grafted onto a shared scene graph, or several peers' originally unconnected scene graphs can be composed into a single scene graph by yet another peer. Myriad's transient inconsistency accomplishes these things and generalizes previous work (see the section on transient inconsistency).

Worldwidecrowd

Reality peer networks let us construct CVEs over low-bandwidth networks; these can be experienced and modified using a wide variety of devices. In the WorldWideCrowd application (Figure 1), we can collaboratively edit the crowd in real time over a WAN by sharing avatars' geometry and positions but not their animation clips, even though the aggregate scene graph updates exceed the WAN's capacity. Each site can separately generate crowd motions, or motions of small sets of avatars can be shared, with the degree of sharing under user control. This methodology works equally well in the presence of other bottlenecks, for example, the underpowered (CPU-bound) displays on laptops and PDAs. Figure 1, 3 and 5–8

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 3×2 video wall and a six-sided CAVE (Figure 2).[17] The video wall was driven by an 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.

3

Comp. Anim. Virtual Worlds 2007; **18**: 1–17

Figure 1. A VR view of stylized segmented avatars in the WorldWideCrowd prototype.
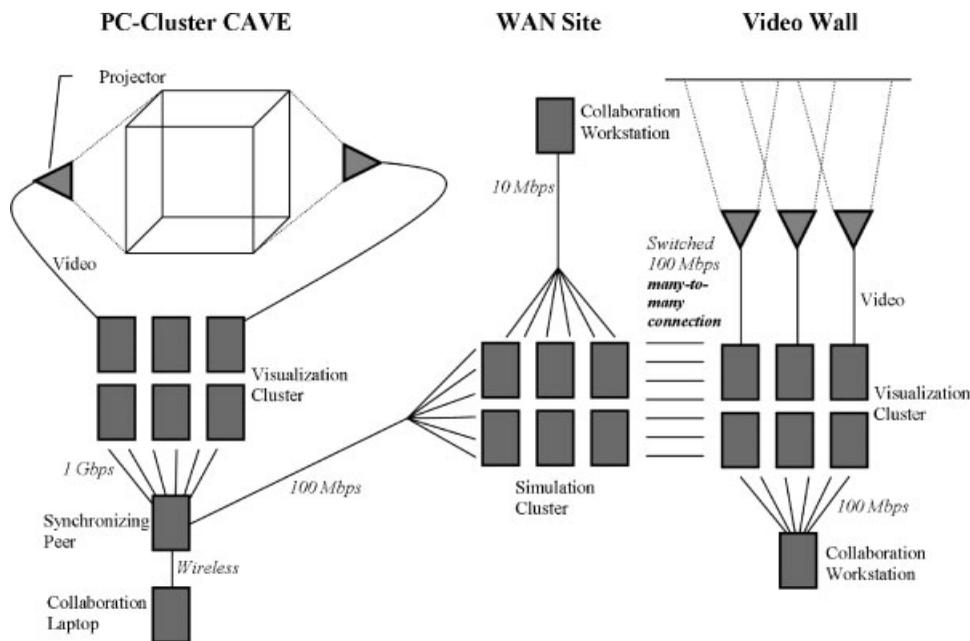

Figure 2. WorldWideCrowd: the validation configuration.

The video wall provided an overhead crowd view. Due to the variation in speed of the PCs powering its displays, OpenGL buffer swaps were not synchronized. When the crowd uniformly covered all six displays, frame rates were 10–25 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. 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.

A cluster of six PCs, with genlocked video cards for active stereo, drove the six-sided CAVE (Figure 3). This let users navigate through the crowd at ground level. The buffer swaps of the wall displays were synchronized and a special reality peer (a synchronizing peer) mirrored its scene graph state in each display. The CAVE graphics cards were more powerful than those in the video wall, but they still constituted a bottleneck at ~24 stereo fps. The synchronizing peer consumed ~24 000 updates/second; due to the animation culling mentioned above, this is much less than the quantity

4

Figure 3. WorldWideCrowd: two walls of the six-sided PC cluster CAVE.

Figure 4. Avatar manipulation.

produced by the whole crowd. As a whole, the cluster processed about 165 000 scene graph updates/second.

We also displayed the ground-level VR view on a 3×2 video wall, using a synchronizing peer. View-frustum culling let the video wall display about 40 monoscopic fps (depending on the precise point of view), even with 2002-era 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 animation information into the system. For instance, users gave a particular avatar (or even a whole set of avatars) different animation clips 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. Thus, the local animations had no net effect on overall network usage. If the users liked the new animation, they could easily push it into the core simulation for others to see.

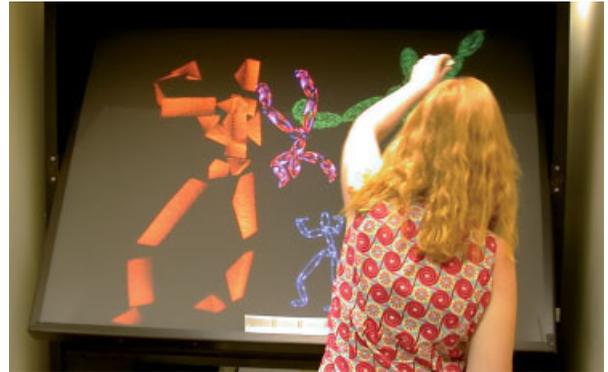This construction also benefits CVEs that include a slow network link. In WorldWideCrowd, the simulation cluster's avatar-animation programs choose animation clips based on the contents of special *info nodes* in the scene graph. Thus, simulation clusters on each side of a slow link can both drive their respective crowds even if this link does not stream animations (each cluster stores a local copy of the animation clips). A collaborator over a WAN who cannot stream the full crowd animation can still share the animation clip names embedded in info nodes and thus influence both simulation clusters.

In our test of WorldWideCrowd, users also changed avatar geometry 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 (see the section on peer-to-peer connectivity), or by 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 and update the other sites. If not, they could restore their peer's original appearance from the shared world.

Users moved avatars by manipulating local copies of transform nodes inside their peers from the Python interpreter, with scripts and 6DOF tracking devices (Figure 4). In the latter case, a manipulator object used the tracker input to modify the transform node's contents, changing the avatar's position, orientation, and scale. As with other operations, changes in avatar position could be performed locally and then either shared or discarded.

Classroom Collaboration

Following the success of the WorldWideCrowd application, which tests Myriad's scalability, we adapted our framework to an educational setting. The Python tools mentioned in the previous section formed the base upon
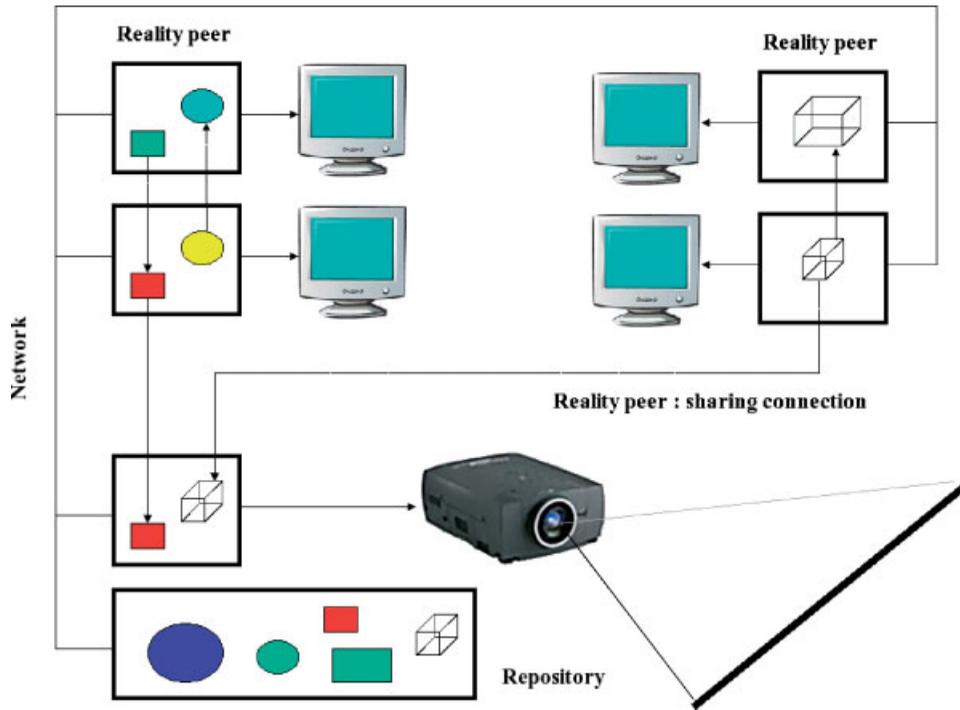
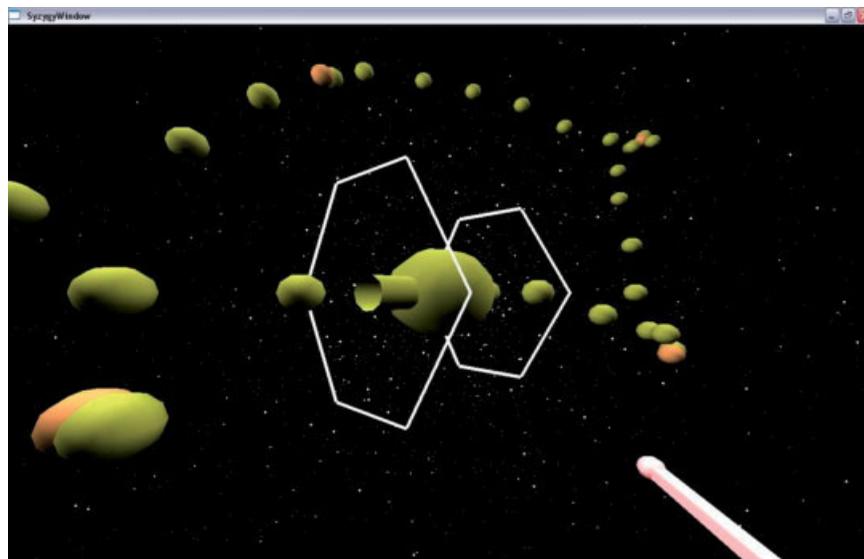*Figure 5. Classroom collaboration: systems diagram.*

Figure 6. Student application: roller coaster.

which we built specialized environments for teaching concepts in mathematics and computer graphics. For this use of Myriad, we had a room full of networked workstations, which additionally, contained a compu- ter/projector combination that displayed a shared reality peer (Figures 5 and 6). For each educational exercise, the pattern was the same. Students would wrestle with a problem individually at their own
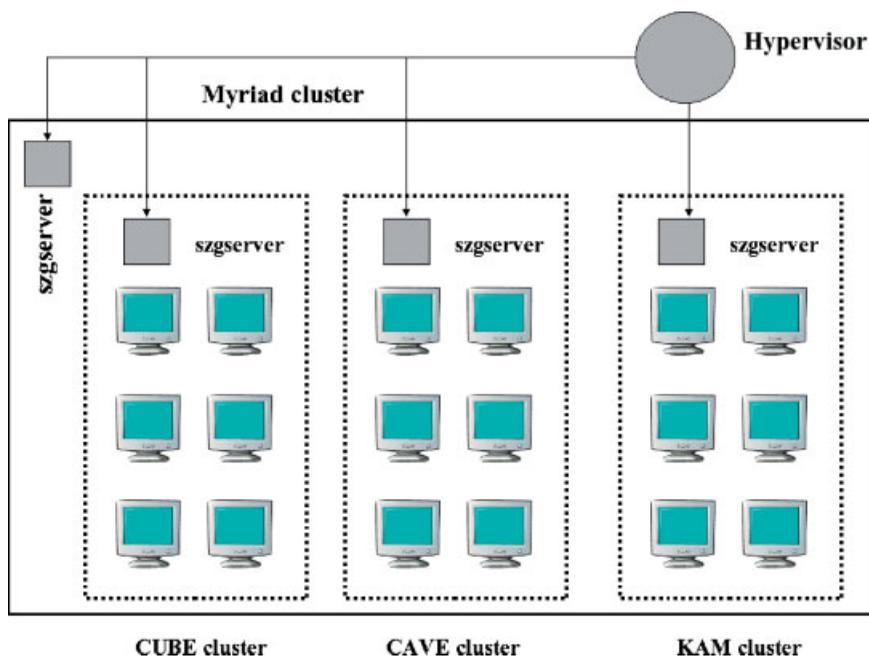
6

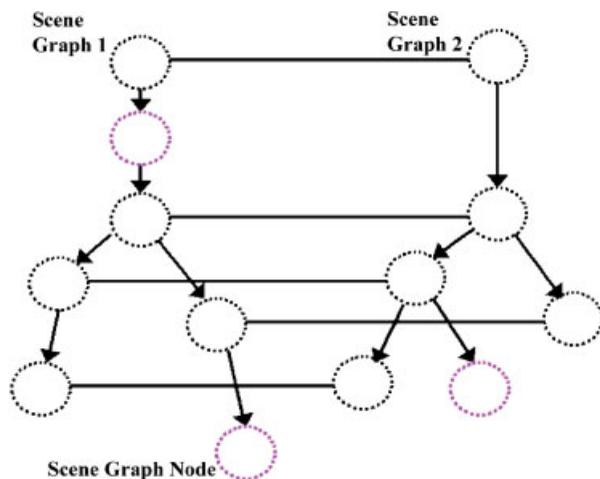*Figure 7. Hypervisor: managing the Syzygy clusters.*

Figure 8. Reality mapping: the relationship between nodes of connected scene graphs.

computers but have the ability to push their solution out to the projector's shared view or pull in information from one of their colleagues.

Mesh Modeling

In this exercise, the students must create a mesh illustrating some mathematical concept. At an elementary level, they might create a conic section, while, at a more advanced level, they might make an isosurface for a given scalar data field. The Syzygy graphics database provides a standard format for the mesh information, the coordinates of its vertices, the arrangement of those vertices into triangles or triangle strips, and so on. Myriad's reality peers allow the students to share their work in progress with the group. Each student adds to her mesh interactively using Python.

Once a student has built an initial mesh in her own reality peer, she can transfer a copy into the peer displayed through the projector. There, anyone in the class can manipulate it, via rotating it, zooming it, removing geometry, or changing the existing geometry's coloration. As more students publish their attempts, the instructor can lead the class through the different solutions, commenting on the relative success or failure of each. Those who were unable to generate the mesh correctly can pull a successful example back from the shared space into their own peer and use it as a template for further investigation.

Myriad makes more complex collaborations possible. An ad hoc subgroup of students might decide to work together, sharing their meshes only between one another's reality peers and waiting until all agree that they have a solution before pushing it out to the peer shared by the entire class. Since an individual student can have multiple versions of the mesh in her peer, each student can participate in several subgroups

simultaneously, limited only by her ability to keep track of who is doing what and by the power of her workstation to render geometry. Within her reality peer, she can, for instance, arrange her mesh versions independently of her classmates, allowing her to control her overall perspective on the problem.

Finally, in addition to publishing static copies of their meshes, the students can choose to share their work as it progresses in real-time. As a student works on getting the mesh just right, she might iteratively make small modifications here and there. Another student can copy that portion of her colleague's peer containing the mesh and elect thereafter to receive those modifications as they are made, thus being able to (metaphorically) look over her classmate's shoulder from the comfort of her own reality peer on her own workstation.

The scripting environment supports these activities by automating several interactions with the underlying Myriad system. Students can query the reality peer network to find out what meshes exist to be shared and the peers in which they are located. Such searches also reveal the owner of each mesh, comments about it, and its descriptive name. Scripts support taking a mesh snapshot and transferring it into another reality peer, either ensuring future changes are transmitted or immediately breaking the connection between the copy and its original. Since this exercise requires students to examine and modify meshes they have not created, there are scripts for browsing the scene graph structure of a mesh and extracting data, such as vertex lists or coordinates, from the individual scene graph nodes. Finally, in order to allow the lesson to extend across multiple days, there are tools for moving mesh snapshots to and from disk.

A World with Interlocking Parts

For this exercise, the collaborative task is more complex. The students must create a world with multiple independent parts, each of which is animated by a separate program, but which work together as a whole. A simple example is a physics-based simulation of N balls traveling through a maze, bouncing off the walls. A more complex example is a simulation of a sequence of gears, where turning the first causes the others to turn as well.

Each student is responsible for a subset of the world's geometry along with its physics. In this exercise, the shared reality peer displayed by the projector shows all of the independent parts working together. At checkpoints of the world's collaborative development, the

students transfer geometry from their local reality peers, possibly overwriting objects they have previously transfered. Furthermore, from time to time, a student will attach her animation scripts to her piece of the world inside the shared peer. These scripts cannot only animate her geometry but also that of the other students, as is necessary to handle, for instance, collisions between geometry in different peers.

Each student can work in isolation, within her local reality peer, on the physics expressed by her animation scripts, but she will invariably confront obstacles when trying to compose her work with that of the others. For instance, one of the objects animated by student A might penetrate one of the objects created by student B. Perhaps A was working with an old copy of B's models, having pulled them from the shared reality peer before B published changes. While the students could avoid this situation by exclusively working with the shared peer or by having their local peers continuously receive updates from those of their classmates, this methodology has drawbacks as well. Physics routines are difficult to program because of numerical instability, and, as a consequence, it is advantageous to keep their effects from the shared environment until they have been at least partially debugged.

Since the pieces of the shared world are, in fact, dependent on one another, Myriad must support a protocol through which different scripts, when operating on the same scene graph nodes within a peer, can cooperate with one another. An elementary example occurs during object collisions. When a script detects a collision involving an object it controls, the script should temporarily take control of the second object as well, altering its position and velocity according to the collision algorithm. A more sophisticated example occurs if one object can grab another, as with a simulated manipulation tool. Myriad implements these behaviors via scene graph node locking, unlocking and notifications. Script A can lock the scene graph node controlling the position of an object, ensuring that only A will alter the object's position while it holds the lock. Script B can register to receive a notification when A unlocks the object, at which point it can get the lock itself. In this way, scripts can cooperatively hand control of a shared object from one to another.

Student Applications

Building on the tools created for WorldWideCrowd and our collaborative education environments, students

participating in a University of Illinois REU (Research Experiences for Undergraduates) wrote several new-shared worlds using the Myriad framework. Their work further validates our framework as a practical tool for shared world construction. There are trade-offs, after all, between providing more power/options to the user and making the resulting system too unwieldy for all but experts. We feel that our Python tools, as a layer on top of raw Myriad functionality, implement the simplifications that make Myriad accessible.

The first application is a roller coaster constructor. Here, multiple users collaborate in laying down knot sequences that define roller coaster tracks via splines. The world can contain multiple coaster tracks, different users can modify each of the tracks by moving the knots, and the individual coaster trains start and stop at the user's command. Also, as the coasters run, users can choose to ride any of them, switching freely between the train's local viewpoint or a global viewpoint floating in space outside all tracks. The roller coaster constructor demonstrates free-form creation and collaboration.

The second student application is an animation application through which users pose a series of humanoid and equine figures and construct the simple virtual world in which they exist. The animations are created frame by frame via attaching a manipulator to a particular bone in a skeleton hierarchy, moving the bone, going to the next one, moving it, and so forth, until the new pose is complete. Sequences of these poses can be saved, spliced together, diced apart, and applied to different models inhabiting the shared world. This contrasts with WorldWideCrowd in that, there, the bulk of the application concerned arranging canned animations in an aesthetic fashion, and the tools we constructed for it were mainly for managing scalability. Here, the collaborative process of creating an individual animation is emphasized.

Cluster Hypervisor

Before moving on to a detailed description of Myriad's internals, some practical considerations should be addressed. There are several issues with deploying Myriad in a production environment. Our organization, the University of Illinois' Integrated Systems Lab, has three immersive virtual environments powered by PC clusters, called CUBE, CAVE (following the term coined by EVL), and CANVAS. The CUBE is used for psychology experiments, the CAVE for tours and demonstrations, and the CANVAS is a permanent VR art exhibit in the campus art museum. Each visualization cluster is managed by a separate instance of the Phleet distributed OS (with its kernel an szgserver), as described in the initial Syzygy paper.[4]

While these environments are mostly used for their respective functions, we sometimes want to deploy Myriad-based applications across all of them simultaneously. This must occur without disrupting their normal operations. Since different staff are responsible for each cluster, they have different maintenance schedules, and one of them (CANVAS) has intermittent network connectivity (due to its location at the art museum), joining all three visualization clusters into a single 'metacluster' running one instance of the Phleet OS is inconvenient. However, in order to run a Myriad application across all of them, this is exactly what needs to happen (Figure 7).

Consequently, we developed a simple approximation to hypervisor technology for Syzygy. Normally, a *hypervisor* is a software component that allows multiple operating system to run simultaneously on a given computer. In our case, the hypervisor allows multiple distributed operating systems to share a cluster. Each cluster participates in two instances of the Phleet OS. One instance works locally for that cluster only (there are three Phleet instances like this) and a fourth Phleet runs across all the clusters at once, making Myriad applications that utilize all three clusters possible.

The Phleet hypervisor prevents these OS instances from interfering with one another by hooking into the application-launching interface. Normally, the Phleet running a particular cluster knows whether an application is running and whether needed resources like tracking or sound are available. When a Myriad-based application is launched, the launcher first contacts the hypervisor, which in turn, contacts the Phleets on the individual clusters and has them shut down any running applications. After this has occurred, the launcher is informed and begins using the 'global' Phleet to run the Myriad application. The reverse happens when going from metacluster Myriad to single cluster application.

Myriad Scene Graph

Now that we have described some of Myriad's applications and the practical challenges to deploying them, we shift to an in-depth examination of the technical features that make the applications themselves possible.

Each reality peer contains a Syzygy scene graph whose node types correspond to OpenGL commands. Nodes can store geometry (vertex arrays), 3D transformations, texture maps, OpenGL materials, etc.[4] Myriad extends Syzygy by adding an *info node* which stores a string and can add semantics to a scene graph, similar to MASSIVE-3 and DIVE.[1,11] Each node has an ID, unique within its owning scene graph; the scene graph uses this ID to route messages generated when node methods execute. Each node also has a name that is not necessarily unique. As shown below, node names help construct the reality maps between nodes in different peers (Figure 8).

Myriad adds a new property to all Syzygy scene graph nodes: *sharing level*. Its value can be *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 the section on fine-grained sharing) and how its updates are dynamically filtered before propagating to connected peers (see the section on peer connection feedback). Nodes with transient sharing level are called *transient nodes*.

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 peers, choose a peer by name, get its IP address and port, and then connect directly to it. Each reality peer listens for update messages on its various connections, then filters them and passes them on. 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.

A *pull peer* connects to a remote peer and synchronizes its scene graph with a subgraph of 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 let us build a new crowd that tracked the evolution of an existing one but within which we could make local changes (e.g., adding avatars) without modifying the original. A pull peer

might even elect to receive updates from particular remote scene graph nodes: For example, ignoring updates from transient nodes greatly decreases 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. The push peer affects the remote peers without itself being affected. In WorldWideCrowd, the crowd animation and user navigation programs embedded push peers.

A *feedback peer* synchronizes its scene graph with a subgraph in a single remote peer upon connection and subsequently exchanges updates with it. In WorldWideCrowd, feedback peers allowed collaborative editing of a shared scene graph. A *shadow peer* is a special kind of feedback peer that shares scene graph structure (a relatively small amount of data) but not the contents of the nodes. A shadow peer quickly synchronizes its scene graph structure with a remote peer, even over a low-bandwidth link. In WorldWideCrowd, shadow peers were useful for editing scene graphs over the WAN.

Scriptability is useful in creating large-scale distributed applications such as WorldWideCrowd. 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. The simplest way a peer joins a collection of peers is by being part of an executable program. The program containing the peer can alter it directly. But more powerfully, a peer can be part of a *workspace*, a generic container for peers. A workspace can contain many peers, render their scene graphs, and be controlled remotely through RPC. Workspaces avoid the proliferation of ad hoc executables, one per peer.

Fine-Grained Sharing

Fine-grained sharing lets peers control information sharing at the level of individual scene graph nodes. 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 is inspired by recent incarnations of MASSIVE, but unlike MASSIVE's deep behaviors Myriad's filters are properties of connected pairs of nodes and peers, not of the nodes themselves.[3]

Also, Myriad's fine-grained sharing contrasts with MASSIVE-3's coarse-grained locales and aspects, which are essentially entire scene graphs.[1]

A reality map relates two peers' 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 below. Distributed Open Inventor has a similar construct, except that its shared subgraphs have identical structure (only the location of the root node changes).[6]

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 variation. For example, a peer could locally insert a transform node and add unshared rotational motion to an object's shared translational motion, or it could locally recolor a uniformly-colored group of shared objects by inserting local material nodes.

Another important element of fine-grained sharing is node locking, which maintains 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 ignore updates to these nodes that do not come from A, allowing A to make changes deterministically. Locking is implemented cooperatively: Any peer connected to B that holds a mapped image of the locked node can take the lock from A and unlock it.

The following sequence of events occurs each time an update message reaches a reality peer: First, each message checks its history of peers it has updated; a message revisiting a peer that has been already updated is discarded in order to prevent 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, or discards the message if no such node exists. The message is also discarded if the destination node is locked by a peer other than its originator. The user can define a set of message filters for each node and peer-connection pair; these are applied before the update message is sent to its destination node in the local scene graph.

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

The reality map between two connected peers can be built in two different ways. One peer can create a new copy of its scene graph in the other (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 based on each node's sharing level. The sending peer traverses the scene graph below the root node it had specified. It sends a node creation message for each node traversed; if the node's sharing level is no less than the map's sharing level, the sending peer also sends a node state serialization message. The node creation messages extend the reality mapping in the receiving peer, 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 receiver.

Consider a map trying to associate existing nodes on the receiving peer with nodes on the sending peer. The node creation message contains the name and type of the 'created' node and the ID of its parent node. The receiving peer discards the message if its reality map for the connection cannot translate the parent ID to that of one of its own nodes. Otherwise, the receiving peer searches depth-first below the mapped parent for an existing node with the same name and type as specified by the creation message. If it finds such a node, it extends the reality map to include this node and the creation message is discarded; if not, a new node with the specified name and type is created as a child of the mapped parent and the receiving peer's reality map is updated. In either case, the receiving peer returns a message to the sender describing the newly created extension for the sending peer's reality map. A reality map automatically extends itself over time, with new children of mapped nodes in one peer making new mapped nodes in the other peer.

The reality-mapping algorithm only depends on nodes having unique IDs, not unique names. Thus, the names can encode useful structural information. For example, the avatar skeletons in WorldWideCrowd all contained a standard set of node names, making it easy to associate nodes of an avatar in one peer with those in another. This standardization allowed

WorldWideCrowd users to easily replace any avatar body with any other embedded in another peer.

Reality maps have several useful properties besides supporting local variations. They support creation of larger virtual worlds by combining the scene graphs from a group of peers inside a single peer. They enable sharing of a small part of a peer's scene graph with connected peers; combined with message filtering, this facilitates low-bandwidth collaboration. In particular, reality maps support dynamically modifiable sharing, because a user or application need not decide in advance how to divide a peer's scene graph into sharing units; the user can limit shared data to precisely the area of interest at any given moment.

Peer Connection Feedback

If several reality peers participate in a multicast group, all peers see the same updates. This is good for similar peers; however, peers' resources and capabilities might vary substantially. For example, peers on a LAN or a high-speed WAN might have good network connectivity among themselves but much less to a peer outside their subgroup. Furthermore, peers may display their virtual worlds at different frame rates. If each peer receives the same scene graph updates, slower peers waste time processing updates only needed by 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 (recall the section on scene graphs). 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, foe example, requesting only one update per second. Each reality peer stores the update rate requested by each connected peer and updates transient nodes no faster than the requested rate.

If a reality peer's primary function is to render its scene graph, transient node updates matter only for nodes likely to be viewed. Consequently, the user can configure a peer to automatically test whether particular subgraphs of its scene graph are within a certain distance of the viewing frustum. Connected peers alter the relevant reality maps so as not to send transient node updates to subgraphs that fail this test. The reality maps are restored when the subgraph passes the test. Under this scheme, all viewed transient nodes contain timely values, if user travel speed is bounded. Specifically, when a subgraph approaches the viewing frustum, one message round trip to a connected peer must occur before the subgraph will again get updates from that peer. Thus, the speed limit is equal to the user-defined proximity threshold divided by the maximum ping time to a connected peer. Myriad does not enforce such speed limits; it accepts that inconsistencies can arise. Developers may of course enforce speed limits in particular applications.

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

Transient Inconsistency

We treat consistency as a local property of connections between peers, not as a global property of the whole peer network. Consider subgraphs in each of two connected peers that are reality-mapped onto each other. We call the subgraphs *consistent* if they have identical structure and all pairs of mapped non-transient nodes have the same values. If the full scene graphs in both peers are consistent, we say that the connection itself is 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 the section on peer-to-peer connectivity). Myriad reduces its use of networking resources and facilitates virtual world prototyping by allowing linked but inconsistent versions of a world.

However, 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 by the application or by user input (although they may be by Myriad itself). Myriad provides an API for launching *consistency processes*, background tasks that modify the subgraphs in the direction of consistency. We call this effect *transient inconsistency.*

Transient inconsistency imposes fewer resource requirements on a CVE than does strict consistency. For example, a substantial but intermittent resource strain occurs when new users join the virtual world; this resource strain is dubbed the *late-joiner problem.* In many CVEs, all updates pause while the world state transfers

atomically to the newcomer. This freezes all viewers for a substantial fraction of a second, even if the world state is only a few megabytes and the LAN has 100 Mbps speed. Usability significantly deteriorates on a slower WAN with frequent newcomers. 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 different peers.

Myriad solves the late-joiner problem with a consistency process that uses 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 the section on peer-to-peer connectivity); we call the consistency process *strict* if every node's state is transferred. Non-strict consistency processes need less bandwidth. For example, if a user needs to manipulate only the structure of a scene graph, node creation messages suffice to map it to a remote peer: 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. New node updates are interleaved with state-transfer updates, so the transfer may be greatly prolonged without impacting usability. Once a node has been mapped, its future updates are immediately sent to the connected peer; the reality-map filtering discards updates to nodes that haven't been mapped yet.

Can we prove 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 bounds the duration of inconsistencies; in other words, the inconsistencies are transient.

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 to. (The order of messages from one peer to another is preserved. Thus, if a node is created and soon deleted, its creation and deletion messages cannot be reversed in transit and thereby confuse the receiving peer.)

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, running in the background, the consistency process will map the parent and then the child. Otherwise, the child is created after the parent has been mapped and is immediately added to the connected peer.

Myriad's solution to the late-joiner problem is similar to MASSIVE-3's, where state transfer to the late joiner also occurs in the background without locking the scene graph and freezing other users.[1] 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's consistency processes also reconcile existing inconsistent subgraphs in connected reality peers. Once users have stopped altering the subgraphs, a background consistency process will eventually make them consistent. At its most stringent, the consistency process's goals are: (1) mapped nodes on both sides of the connection should have the same values; (2) every node should map to some remote node; and (3) every remote node should map to some local node. Under these circumstances and when not competing with user changes, an inconsistency's lifetime is bounded by how quickly the consistency process traverses the scene graph, which depends on 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, for example, compare different versions of a world. These features are supported in CVEs like Distributed Open Inventor[6] and DIVE.[5] 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 if the world is static. 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 common local variation was to animate a collection of avatars with new animation clips not available on the simulation cluster. It would have been expensive to transmit all of these variations to every peer, since each variation used substantial bandwidth. Myriad can partition scene graph updates so that they go only where needed, reducing required bandwidth. Its transient inconsistency let users locally evaluate animation clips, and then either change the global shared state on acceptance of a clip, or revert to the global state if they rejected the clip.

PC Cluster Visualization

Many researchers have created cluster graphics solutions in the last few years, such as Chromium[18] (formerly WireGL), Cluster Juggler,[19] and Syzygy.[4] 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.

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 in each frame, achieving this by synchronizing their buffer swaps and their consumption of update messages.[4] Unfortunately, processing this many updates can create a CPU bottleneck at the synchronizing peer. A 3 GHz 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 WorldWide-Crowd, which required maximum scalability, was not synchronized frame by frame. Instead, the video wall's reality peers displayed scene graph updates as they were received. The overhead view helped to partition the drawing and message processing among the peers, increasing each one's potential frame rate. This loose synchronization among screens let each render PC connect separately to the crowd simulation peers, and to a navigation controller for panning across and zooming around the scene. This worked well despite the
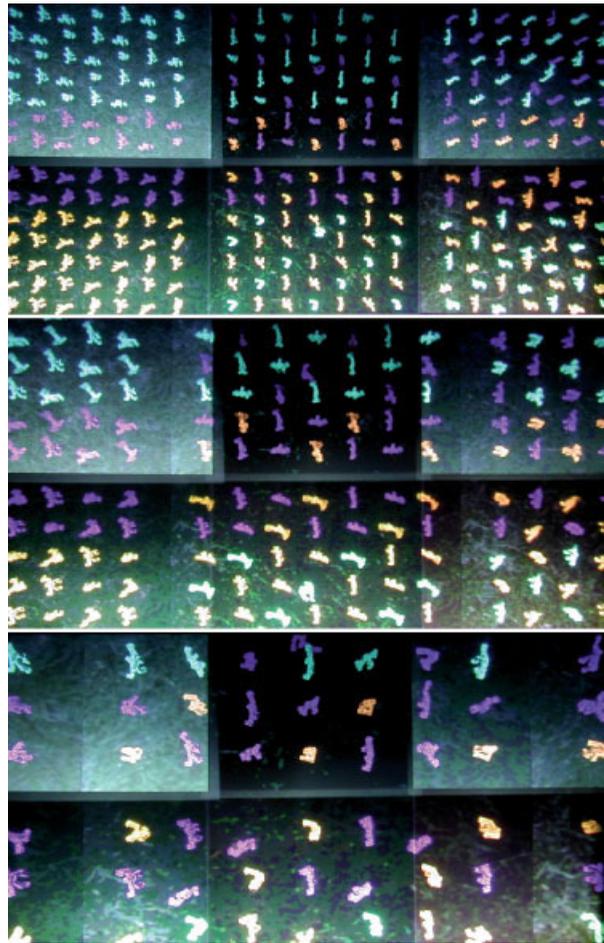

Figure 9. WorldWideCrowd: a smooth zoom into the crowd. Three steps in a time series.

performance differences between the video wall PCs, as shown in a slow zoom into the crowd (Figure 9).

However, frame synchronization was needed for the CAVE PC cluster's active stereo projectors (alternating left- and right-eye frames). The peers defining the crowd of avatars changed as users joined the world and edited it. It would be tedious, whenever this changed, to update every CAVE display PC's list of peers. The insertion of a synchronizing peer decoupled where the world's total scene graph originated, from how that scene graph was displayed. This was because Myriad's sharing of subgraphs is a property of individual pairs (in this case, 'editing peer'—synchronizing peer pairs and synchronizing peer—'display peer' pairs). Changes outside the CAVE cluster are hidden from the cluster itself. In contrast, sharing is a global property of subgraphs in Distributed Open Inventor. While an application's scene graph can be composed of any

number of separately shared and unshared subgraphs, the overall scene graph is not transparently shareable.

Interaction with the Python Interpreter

Users can interact with Myriad from Python at two different levels. 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 users manipulate them from anywhere in the network via an RPC interface. This makes it easy to set up, tear down, and manage a distributed application like WorldWideCrowd. The RPC interface is also useful for creating ad hoc workflows. For example, 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 different computers, each a pull peer (defined in the section on peer-to-peer connectivity) with respect to the original. These new peers can be altered to create new scene graph versions; if they run on different computers, each will run with high performance, simplifying comparisons. The user can automate killing views that are no longer needed via the same scripting interface. They can also remotely serialize any peer's scene graph and save it to a file, thus preserving intermediate prototypes.

The second level of interaction is direct manipulation of Syzygy/Myriad objects, such as the reality peers and their scene-graph nodes. We use SWIG-generated Python wrappers for these objects, letting them be created at a Python prompt or from script. For example, a user can start a reality peer from a Python prompt and connect it to another as a feedback peer (see the section on peer-to-peer connectivity). The user can then alter the remote peer's nodes by changing mapped nodes in the local peer from the Python prompt, encouraging free-form experimentation.

Each scene graph node's complete 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). Mapping the local transform node to a remote peer enables manipulation of any transform node in the peer network.

Future Work

This work can be extended in many ways. We could explore system performance on modern hardware: 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 six-node display cluster connected by a gigabit switch to a modern six-node compute cluster. On such hardware, Myriad might animate 1200 avatars at interactive frame rates. Furthermore, while the Myriad application described in this paper was 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 for virtual worlds. A practical use for such power would be 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 associated with, for example, segmented avatars. Furthermore, Myriad's scene graph update messages are unnecessarily general in some cases; for example, it sends a 4×4 matrix to control each avatar bone instead of three Euler angles. The latter approach would reduce bandwidth at a cost of increased computation to reconstitute the rotation matrix for OpenGL. Experiments would help in understanding these trade-offs.

Myriad's frame-synchronizing for rendering on a PC cluster could also be improved. Currently a single synchronizing peer guarantees consistency of each cluster-rendered frame (all CAVE walls display the same world state). But the PC running the synchronizing peer has to push the entire world's updates to the rendering PC's. This one-peer bottleneck would vanish if multiple data sources independently synchronized and desynchronized with the rendering PCs in the cluster. The difficulty in this approach comes from coordinating the synchronized video frames in the PC cluster with the intermittently synchronized data sources. Note that conventional parallel programming APIs like MPI cannot let synchronization groups change, after they are created.

Finally, Myriad's API can be refined. The underlying scene graph API comes from Syzygy and is relatively

computer animation
& virtual worlds

mature, but the Myriad-specific APIs for manipulating connections between reality peers are still evolving. Myriad's ability to finely manipulate the peer network makes it more complex than other CVE systems. It is still unclear how best to manage that complexity.

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 the experiments.

References

1. Greenhalgh C, Purbrick J, Snowdon D. Inside MASSIVE-3: flexible support for data consistency and world structuring. *Proc. CVE* 2000; 119–127.
2. Carlsson C, Hagsand O. DIVE—A platform for multi-user virtual environments. *Computers & Graphics* 1993; **17** (6): 663–669.
3. Purbrick J, Greenhalgh C. An extensible event-based infrastructure for networked virtual worlds. *Proc. IEEE Virtual Reality* 2002; 15–21.
4. Schaeffer B, Goudeseune C. Syzygy: native PC cluster VR. *Proc. IEEE Virtual Reality* 2003; 15–22.
5. Smith G. Cooperative virtual environments: lessons from 2D multi user interfaces. *Computer Supported Cooperative Work* 1996; 390–398.
6. Hesina G, Schmalstieg D, Fuhrman A, Purgathofer W. Distributed open inventor: a practical approach to distributed 3D graphics. *Proc. VRST* 1999; 74–81.
7. Macedonia M, Zyda M, Pratt D, Barham P, Zeswitz S. NPSNET: a network software architecture for large-scale virtual environments. *Presence* 1994; **3** (4): 265–287.
8. Dang Tran F, Deslaugiers M, Gerodolle A, Hazard L, Rivierre N. An open middleware system for large-scale networked virtual environments. *Proc. IEEE Virtual Reality* 2002; 22–29.
9. Leigh J, Johnson A, DeFanti T. CAVERN: a distributed architecture for supporting scalable persistence and interoperability in collaborative virtual environments. *Journal of Virtual Reality Research, Development, and Applications* 1997; **2** (2): 217–237.
10. Park K, Cho Y, Krishnaprasad N, Scharver C, Lewis M, Leigh J, Johnson A. CAVERNsoft G2: A Toolkit for High Performance Tele-Immersive Collaboration. *In Proceedings of ACM Symposium on Virtual Reality Software and Technology* 2000; 8–15.
11. Hangsand O. Interactive multiuser VEs in the DIVE system. *IEEE Multimedia* 1996; **3** (1): 30–39.
12. MacIntyre B, Feiner S. A distributed 3D graphics library. *Proc. ACM SIGGRAPH* 1998; **32**: 361–370.
13. Tramberend H. Avocado: a distributed virtual reality framework. *Proc. IEEE Virtual Reality* 1999; 14–21.
14. Greenhalgh C. Awareness management in the MASSIVE systems. *Distributed Systems Engineering* 1998; **5** (3): 129–137.
15. Capps M, McGregor D, Brutzman D, Zyda M. NPSNET-V: a new beginning for dynamically extensible virtual environments. *IEEE Computer Graphics and Applications* 2000; **20** (5): 12–15.
16. Frecon E, Greenhalgh C, Stenius M. The Divebone—an application-level network architecture for internet-based CVEs. *Proc. VRST* 1999; 58–65.
17. Cruz-Neira C, Sandin D, DeFanti T. Surround-screen projection-based virtual reality: the design and implementation of the CAVE. *Proc. ACM SIGGRAPH* 1993; **27**: 135–142.
18. Humphreys G, Eldridge M, Buck I, Stoll G, Everett M, Hanrahan P. WireGL: a scalable graphics system for clusters. *Proc. ACM SIGGRAPH* 2001; **35**: 129–140.
19. Olson E. *Cluster juggler—PC cluster virtual Reality*. M.Sc. thesis, Iowa State University, 2002.

Authors' biographies:

Benjamin Schaeffer received a PhD in mathematics from the University of Illinois at Urbana-Champaign and subsequently worked there developing virtual reality technology.

Peter Brinkmann is an Assistant Professor of mathematics at the City College of New York, where he is currently setting up a new virtual environment. Originally from Germany, he earned his PhD at the University of Utah and did post-doctoral work at the University of Illinois, the Technical University of Berlin, and the Max-Planck-Institute of Mathematics in Bonn.

George Francis is a Professor of mathematics. He is a faculty of the Campus Honors Program and the Beckman Institute. He is a senior research scientist at the NCSA. University of Illinois at Urbana-Champaign. His

interests include geometrical computer graphics and virtual environments. He engages undergraduate students in his research program. Springer Verlag published a paperback edition of his popular *Topological Picturebook* in the fall of 2006.

Camille Goudeseune keeps several large-scale virtual environments running in the Beckman Institute at the University of Illinois at Urbana-Champaign. From Waterloo (Canada) he earned a BMath.; from Illinois a MMus. (piano) and a DMA. (composition). His name is hard to spell and commensurately easy to google.

Jim Crowell is a researcher with the Beckman Institute at the University of Illinois at Urbana-Champaign.

Hank Kaczmarski is the Director of the Integrated Systems Laboratory at the Beckman Institute at the University of Illinois, Urbana-Champaign. Kaczmarski has supervised the construction of the nation's first six-rigid-walled CAVE, funded through an NSF grant, a driving simulator suite consisting of multiple vehicle and console simulators, is experimenting with ultra-high resolution display walls, integrating optical motion capture into virtual environments, building audio spatialization environments, and developing PC cluster display technology.