RING: A Client-Server System for Multi-User Virtual Environments

Thomas A. Funkhouser
AT&T Bell Laboratories [‡]

Abstract

This paper describes the client-server design, implementation and experimental results for a system that supports real-time visual interaction between a large number of users in a shared 3D virtual environment. The key feature of the system is that server-based visibility algorithms compute potential visual interactions between entities representing users in order to reduce the number of messages required to maintain consistent state among many workstations distributed across a wide-area network. When an entity changes state, update messages are sent only to workstations with entities that can potentially perceive the change – i.e., ones to which the update is visible. Initial experiments show a 40x decrease in the number of messages processed by client workstations during tests with 1024 entities interacting in a large densely occluded virtual environment.

CR Categories and Subject Descriptors: **[Computer Graphics]:** I.3.7 Three-Dimensional Graphics and Realism – *Virtual Reality.*

Additional Key Words and Phrases: Visual simulation, multi-user systems, virtual reality, 3D virtual environments, real-time graphics, client-server design, distributed systems.

1 Introduction

In a multi-user visual simulation system, users run an interactive interface program on (usually distinct) workstations connected to each other via a network. The interface program simulates the experience of immersion in a virtual environment by rendering images of the environment as perceived from the user's simulated viewpoint. Each user is represented in the shared virtual environment by an entity rendered on every other user's workstation, and multi-user interaction is supported by matching user actions to entity updates in the shared virtual environment. Applications for these systems include distributed training simulations, collaborative design, virtual meetings, and multiplayer games.

A difficult challenge in multi-user visual simulation is maintaining consistent state among a large number of workstations distributed over a wide-area network. Since three dimensional rendering at interactive rates requires fast access to the geometric database, shared portions of the virtual environment (including dynamic entity states) are replicated on every participating workstation. As a result, whenever any entity changes state (e.g., moves) or modifies the shared environment, an appropriate update must be applied to every copy of the database in order to maintain consistent state (see Figure 1).

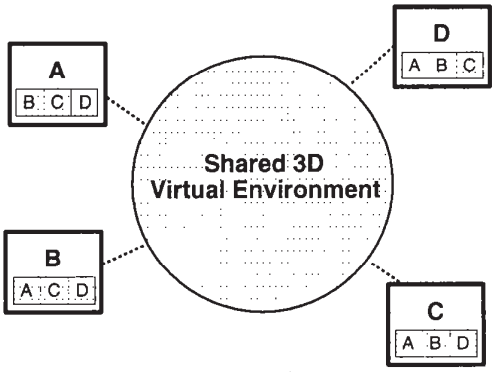

Figure 1: Multi-user systems must maintain consistency between entities (A, B, C, and D) replicated on multiple workstations.

Implementing visual simulation systems for large numbers of users is especially challenging because updates can occur at extremely high rates. If N entities move through a shared virtual environment simultaneously, each modifying its position and/or orientation M times per second, then $M * N$ updates are generated to a shared database per second. Moreover, updates must be propagated to participating workstations in near real-time since large variances or delays in updates can result in visually perceptible jerky or latent motion, and thus may be disturbing to users. As a result, general-purpose distributed database systems are not adequate for use in multi-user visual simulation applications, and special-purpose messaging protocols are typically used to maintain consistent state in multi-user visual simulation systems [9, 13].

[‡]600 Mountain Avenue, 2A-202, Murray Hill, NJ 07974, funk@research.att.com

1995 Symposium on Interactive 3D Graphics, Monterey CA USA
© 1995 ACM 0-89791-736-7/95/0004...$3.50

2 Previous work

Numerous experimental virtual reality systems and multi-player games have been developed for real time interaction in shared virtual environments. Unfortunately, most existing systems do not scale well to large numbers of simultaneous users.

Reality Built For Two [2], VEOS [4], and MR Toolkit [14] are multi-user virtual reality systems that maintain consistent state among N workstations by sending a point-to-point message to each of N-1 workstations whenever any entity in the distributed simulation changes state. This approach yields $O(N^2)$ update messages during every simulation step (see Figure 2), and thus does not scale to many simultaneous users before the network gets saturated.

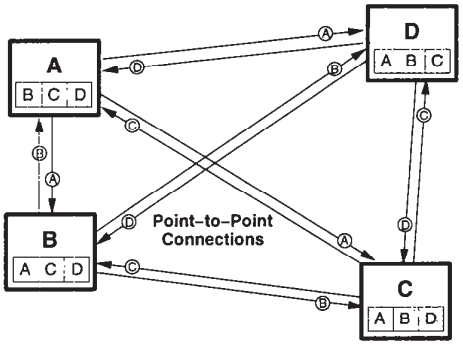

Figure 2: Systems using point-to-point connections pass $O(N^2)$ update messages (labeled arrows) during each simulation step.

SIMNET [5], NPSNET [17], and VERN [3] use broadcast messages to send updates to all other workstations participating in a virtual environment at once. Although, this approach cuts down on the total number of messages transmitted to $O(N)$, every workstation still must process a message whenever any entity in the distributed simulation changes state (see Figure 3). Since every workstation must store data and process update messages and/or simulate behavior for all N entities during every simulation step, these systems do not scale beyond the capabilities of the least powerful participating workstation. Experiences with SIMNET and NPSNET show that a significant percentage of every workstation's processing capability is used just to read update messages from other workstations during large simulations; and, therefore, broadcast protocols are not practical for more than a few hundred users on inexpensive workstations [17].

In order to support very large numbers of users (> 1000) interacting simultaneously in a distributed virtual environment it is necessary to develop a system design and communication protocol that does not require sending update messages to all participating hosts for every entity state change. Kazman has proposed a system design, called WAVES, in which message managers mediate communication between hosts, possibly culling irrelevant messages [10, 11]. His approach is very similar to the one presented in this paper. One difference is that this paper presents algorithms and experimental results for visibility-based message culling during large simulations.

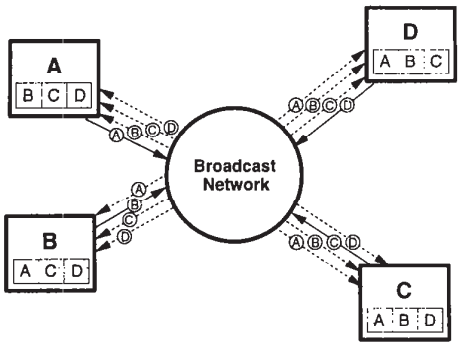

Figure 3: Systems using broadcast messages pass only $O(N)$ updates each simulation step. But, every workstation still must process every update message.

3 Overview of Approach

This paper describes a system (called RING) that supports interaction between large numbers of users in virtual environments with dense occlusion (e.g., buildings, cities, etc.). RING takes advantage of the fact that state changes must be propagated only to hosts containing entities that can possibly perceive the change – i.e., the ones that can see it. Object-space visibility algorithms are used to compute the region of influence for each state change, and then update messages are sent only to the small subset of workstations to which the update is relevant.

The key idea is illustrated in Figure 4. Although entities A, B, C, and D (filled circles) all inhabit the same virtual environment, very little visual interaction (hatched polygons) is possible due to the occlusion of walls (solid lines). In fact, in this example, only one visual interaction is possible - entity A can see entity B. Therefore, only one update message must be sent for each update to entity B's position in real-time (to the workstation with entity A). All other entities need not distribute any update messages in real-time since they are not visible to any other entity. From this example, we see that it is possible to greatly reduce the number of messages passed in real-time to maintain consistent state among multiple entities in a densely occluded environment using line-of-sight visibility to determine the region of influence for each update.

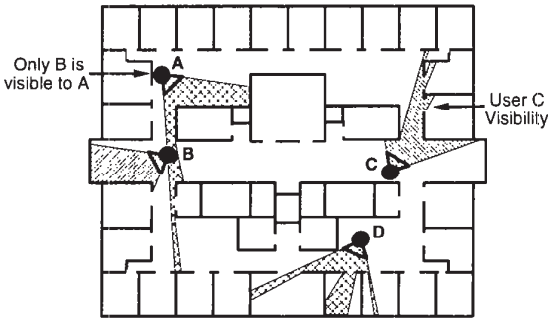

Figure 4: A system that culls messages based on entity-entity visibility may be able to reduce the number of messages processed by each workstation in densely occluded environments.

The following section describes the RING system design. Results of experiments with the system are presented in Section 5, while a discussion of alternate approaches and possible future work appears in Section 6. Finally, Section 7 contains a brief summary and conclusion.

4 RING System Design

RING represents a virtual environment as a set of independent *entities* each of which has a geometric description and a behavior. Some entities are static (e.g., terrain, buildings, etc.), whereas others have dynamic behavior that can be either autonomous (e.g., robots) or controlled by a user via input devices (e.g., vehicles). Distributed simulation occurs when multiple entities interact in a shared virtual environment by sending messages to one another to announce updates to their own geometry or behavior, modifications to the shared environment, or impact on other entities.

Every RING entity is managed by exactly one *client* workstation. Clients execute the programs necessary to generate behavior for their entities. They may map user input to control of particular entities and may include viewing capabilities in which the virtual environment is displayed on the client workstation screen from the point of view of one or more of its entities. In addition to managing their own entities (local entities), clients maintain surrogates for some entities managed by other clients (remote entities). Surrogates contain (often simplified) representations for the entity's geometry and behavior. When a client receives an update message for an entity managed by another client, it updates the geometric and behavioral models for the entity's local surrogate. Between updates, surrogate behavior is simulated by every client.

Communication between clients is managed by *servers*. Clients do not send messages directly to other clients, but instead send them to servers which forward them to other client and server workstations participating in the same distributed simulation (see Figure 5). A key feature of this client-server design is that servers can process messages before propagating them to other workstations, culling, augmenting, or altering them. For instance, a server may determine that a particular update message is relevant only to a small subset of clients and then propagate the message only to those clients or their servers. In addition, a server may send clients auxiliary messages that contain status information helpful for future client processing. Finally, a server may replace some set of messages intended for a client with another (possibly simpler) set of messages better suited to the client's performance capabilities. The aim of this client-server design is to shift some of the processing burden away from the client workstations and into servers so that larger, more affordable, multi-user visual simulation systems can be built using primarily low-cost client workstations.

In the current implementation, RING servers forward update messages in real-time only to other servers and clients managing entities that can possibly "see" the effects of the update. Server-based message culling is implemented using precomputed line-of-sight visibility information. Prior to the multi-user simulation, the shared virtual environment is partitioned into a spatial subdivision of *cells* whose boundaries are comprised of the static, axis-aligned polygons of the virtual environment [1, 15]. A visibility precomputation is performed in which the set of cells potentially visible to each cell is determined by tracing beams of possible sight-lines through transparent cell boundaries [15, 16] (see Figure 6). During the multi-user simulation, servers keep track of which cells contain which entities by exchanging "periodic" update messages when entities cross cell boundaries. Real-time update messages are propagated only to servers and clients containing entities inside some cell visible to the one containing the updated entity. Since an entity's visibility is conservatively over-estimated by the precomputed visibility of its containing cell, this algorithm allows servers to process update messages quickly using cell visibility "look-ups" rather than more exact real-time entity visibility computations which would be too expensive on currently available workstations.

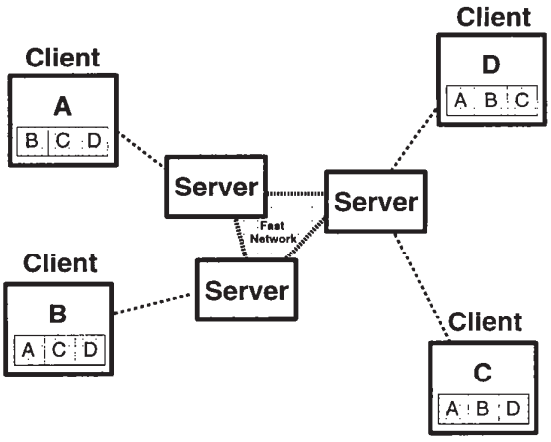

Figure 5: RING servers manage communication between clients, possibly culling, augmenting, or altering messages.

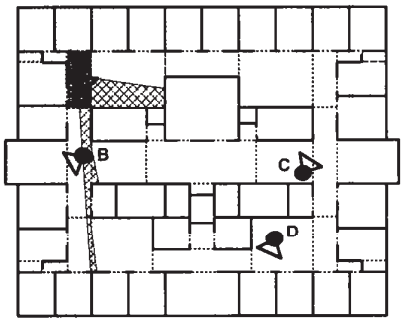

Figure 6: Cell-to-cell visibility (stipple) is the set of cells reached by some sight-line from anywhere in the source cell (dark box) passing only through transparent portals (dash lines) and no opaque walls (black lines). It is a useful, precomputed overestimate of the visibility of any entity resident in the source cell.

As an example of RING server operation, consider the flow of messages between clients A, B, C, and D for the entities shown in Figure 4 connected to servers in the topology shown in Figure 5. Figure 7 shows the surrogates (small squares labeled by entity) and flow of update messages (arrows labeled by entity) for each of the four entities in this example.

- *If entity A is modified:* client A sends an update message to server X. Server X propagates that message to server Y, but not to server Z because entities C and D are not inside cells in the cell-to-cell visibility of the cell containing entity A. Server Y forwards the message to Client B which updates its local surrogate for entity A.
- *If entity B is modified:* client B sends an update message to server Y. Server Y then propagates that message to servers X and Z, which forward it to clients A and C. Server Z does not send the update message to client D because the cell containing entity D is not in the cell-to-cell visibility of the cell containing entity B.
- *If entity C is modified:* client C sends an update message to server Z. Server Z propagates that message to server Y, which then forwards the message to Client B. Server Z does not send the message to either server X or client D because neither is managing entities in the visibility set for entity C.
- *If entity D is modified:* client D sends an update message to server Z. Server Z does not forward the message to any other server or client because no other entity can potentially see entity D.

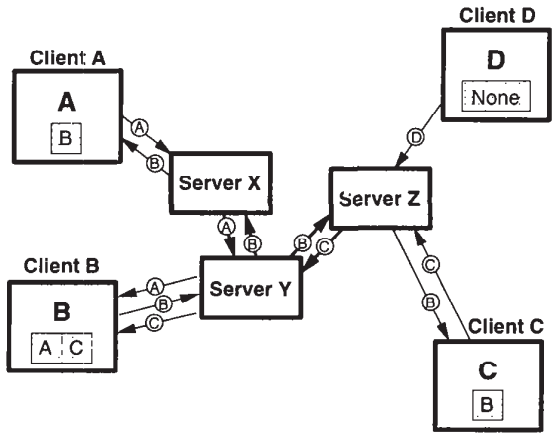

Figure 7: Flow of update messages (labeled arrows) for updates to entities A, B, C, and D arranged in a virtual environment as shown in Figure 4.

RING servers allow each client workstation to maintain surrogates for only the subset of remote entities visible to at least one entity local to the client. All other remote entities are irrelevant to the client so there is no need to waste storage space or behavioral simulation processing for them. To support this feature, servers send their clients an "Add" message each time a remote entity enters a cell potentially visible to one of the client's local entities for the first time. A "Remove" message is sent when the server determines that the entity has left the client's visible region. As entities move through the environment, servers augment update messages with "Add" and "Remove" messages notifying clients that remote entities have become relevant or irrelevant to the client's local entities. Since the system uses an unreliable network protocol, the "Add" and "Remove" messages are considered hints and need not necessarily be processed by clients. However, they allow a client to store and simulate a small subset of the entities with little additional processing or message traffic.

The primary advantage of the RING system design is that the storage, processing, and network bandwidth requirements of the client workstations are not dependent on the number of entities in the entire distributed simulation. Client workstations must store, simulate, and process update messages only for the subset of entities visible to one of the client's local entities. In densely occluded virtual environments, visible sets tend to be constant size (e.g., how many rooms you can see looking into the hallway from your office usually does not depend on the size of your building or whether your building is surrounded by a large city), so the burden on individual client workstations does not grow as the entire system does.

Another advantage is that high-level management of the virtual environment may be performed by servers without the involvement of every client. For instance, adding or removing an entity to or from the virtual environment requires notification of only one server. That server handles notification of other servers and clients. Also, the client-server design allows use of efficient networks and protocols available between server workstations, but not universally available to all client workstations. For instance, clients may connect to servers via low-bandwidth networks, while servers communicate with each other via high-bandwidth networks.

The storage and processing requirements of RING servers are within practical limits. Unlike clients, servers do not have to store display data (e.g., polygons, textures, etc.). But, they must maintain spatial subdivision and visibility information for the virtual environment (typically < 20MB for large environments) and a surrogate representation for every entity in the environment (currently 48 bytes per entity). As server storage requirements grow linearly with the total number of entities, the size of server workstation memory may theoretically limit the number of entities that are able to share a virtual environment simultaneously. However, this is not likely to be a problem in practice since a workstation with 64MB of memory can accommodate nearly one million entities.

Server workstation processing is also within reasonable bounds. Servers must process messages in real-time only for entities visible to some entity managed by one of their clients; they are not required to simulate entity behavior between updates; and, they do not render images of the virtual environment. As a result, the memory capacity and processing power of standard UNIX workstations are adequate for RING servers in densely occluded virtual environments with very large numbers of simultaneous users.

The disadvantage of the RING system design is that extra latency is introduced when messages are routed through servers. Rather than sending messages directly between clients, RING routes each one through at least one server, and possibly two. Computations are performed in the servers before messages are propagated further adding to latency. So far, the extra latency due to server processing has not been noticeable during experiments. Additional work will have to be done to quantify the latency costs and to determine which types of entity interactions are sensitive to latency issues.

5 Experimental Results

A prototype multi-user simulation system has been implemented with the client-server design described in the previous section. The system runs on Silicon Graphics workstations and uses UDP/IP datagrams for message passing. This section presents results of experiments with this system managing many entities interacting in large densely occluded virtual environments. The virtual environments used in these experiments were mazes of "rooms" connected by "hallways." They were constructed by instancing a simple floor-plan 1, 2, 4, 8, 16, and 32 times in a square tiling pattern. Each tile contained 25 rooms (counting hallways) and had 724 polygons (see Figure 8). The largest environment used in these tests had 23,168 polygons which formed 2,219 cells. The spatial subdivision and visibility information for this environment took 99 seconds to compute and required 11.2MB of storage.

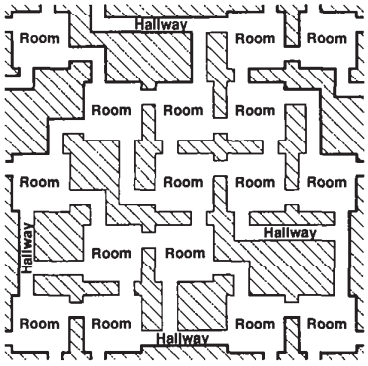

Figure 8: One tile of virtual environment used in tests.

Experiments were run with several environment sizes and various numbers of entities, clients, and servers to characterize the scalability of the system design. During these experiments, entities navigated through the virtual environment "randomly" following piecewise linear paths in randomized directions for randomized distances. Clients sent update messages only for changes in derivatives of entity position and/or orientation (i.e., dead-reckoning) while other clients simulated intermediate positions with linear "smooth-back." Update messages containing 40 bytes (message-type[4], entity-ID[4], target-position[12], target-orientation[12], positional-velocity[4], and rotational-velocity[4]) were generated for each entity once every 2.3 seconds on average with this "random" navigational behavior.

To investigate the message processing requirements of a single client in RING, we performed tests measuring the rates of messages received by clients managing one entity navigating through virtual environments containing 64, 128, 256, 512, and 1024 entities managed by other clients. Each test was repeated in virtual environments containing 25, 50, 100, 200, 400, and 800 rooms. Plates I and II contain images captured during tests with 512 entities in a 400 room environment. Table 1 and Figure 9 show average rates of messages received by individual clients in each test. In Figure 9, points representing the same number of total entities are connected by lines, while points representing the same density of entities are at the same horizontal position in the plot.

Entities Per Room	# Entities	# Rooms	Client↔Server Output	Client↔Server Input
10.24	1024	100	0.44	61.37
10.24	512	50	0.43	70.43
10.24	256	25	0.47	53.68
5.12	1024	200	0.55	55.93
5.12	512	100	0.45	37.37
5.12	256	50	0.44	33.20
5.12	128	25	0.46	27.26
2.56	1024	400	0.50	24.56
2.56	512	200	0.47	19.88
2.56	256	100	0.46	23.19
2.56	128	50	0.41	17.42
2.56	64	25	0.46	13.65
1.28	1024	800	0.50	11.35
1.28	512	400	0.46	14.18
1.28	256	200	0.43	13.28
1.28	128	100	0.45	12.08
1.28	64	50	0.43	8.39
0.64	512	800	0.40	4.62
0.64	256	400	0.45	6.57
0.64	128	200	0.50	6.41
0.64	64	100	0.46	5.37
0.32	256	800	0.35	3.18
0.32	128	400	0.38	3.20
0.32	64	200	0.33	3.35
0.16	128	800	0.38	1.91
0.16	64	400	0.40	1.68
0.08	64	800	0.32	0.52

Table 1: Average message processing rates (messages per second) measured in a single client (managing one entity) during experiments with 64, 128, 256, 512, and 1024 entities in virtual environments with 25, 50, 100, 200, 400, and 800 "rooms."

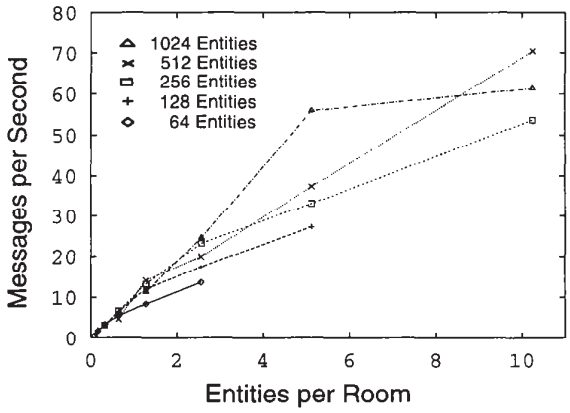

Figure 9: Average rate of messages sent to a single client (managing one entity) during tests with 64, 128, 256, 512, and 1024 entities interacting in virtual environments with 25, 50, 100, 200, 400, and 800 "rooms." Horizontal axis represents the density of entities in the environment.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.