Distributed Scene Graph to Enable Thousands of Interacting Users in a Virtual Environment

Dan Lake, Mic Bowman, Huaiyu Liu
dan.lake@intel.com, mic.bowman@intel.com, huaiyu.liu@intel.com
Intel Labs, Intel Corporation
2111 NE 25th Ave
Hillsboro, OR, U.S.A.

***Abstract*—Virtual environments are currently limited to no more than a hundred interacting users by the simulator-centric server architectures used for many of these applications. There are some potential new usages such as virtual concerts and sporting events involving hundreds or thousands of users and we seek to enable these exciting new applications. We propose a distributed scene graph (DSG) architecture which enables massive scaling of scene complexity and participants with the addition of hardware. A prototype implementation of the DSG components to manage client communications demonstrates an order of magnitude increase in the number of concurrent users. We present the design of this component within the DSG architecture, prototype implementation based on an open source virtual environment server and our experimental setup, workloads and results.**

virtual environment, virtual world, multiplayer, scaling, client, server, game, distributed, scene, architecture, opensimulator

I. INTRODUCTION

General purpose virtual worlds and multiplayer online games, collectively referred to as virtual environments (VEs), have gained tremendous popularity. Some of the largest systems claim millions of registrations and tens of thousands of simultaneous online users. They allow people to represent themselves in the virtual environment as an avatar and through the avatar they may interact with other people and the environment. VEs are ever more able to immerse users using rich 3D graphics, detailed models of objects and realistic interactions between users and between users and their environment.

As the richness and complexity of a VE increases, the computation required to support the experience also increases [1]. Online games such as World of Warcraft™ feature large virtual spaces but the environment remains largely static and interactions between users and objects are pre-defined by the game designers. Many optimizations can be made in such a case and the computational load per user is relatively low. As many as a hundred users can participate and interact simultaneously. General purpose virtual worlds such as Second Life® allow users to create their own environments, objects and behaviors in real time with fewer constraints on what can be built or how avatars can interact with their environment. The computational load is higher as models, graphical textures, and object behaviors cannot generally be pre-computed by clients or the server prior to run-time. This class of general purpose virtual world is limited to well below 100 interacting users.

Interactions between the users and their environment are of critical importance and the essence of any fun game or interesting virtual world. As the complexity of the scene or the number of participating users increases, the communications and computation load on a per user basis grows with it. Additionally, interactions between users and the environment cause an exponential increase in interdependencies between inputs from network clients and server-side computation engines. Figure 1 shows the growth of outgoing bandwidth measured from an OpenSimulator [12] server with an increasing number of moving avatars.

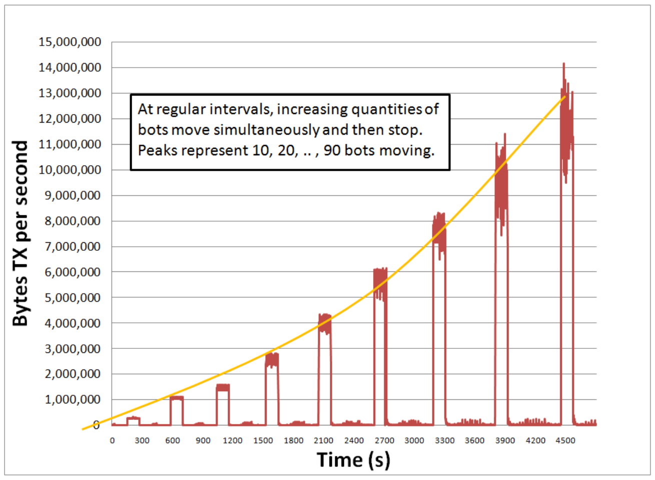

Figure 1.

The bulk of the computation in most online games and virtual worlds is done on centralized servers owned by the company providing the service. The servers receive user input, schedule and execute scripted behaviors of the environment, drive the simulation of physics or other engines and generate updates to clients. In many implementations, most of the simulation and communication processes are executing on a single piece of hardware. We refer to this as a simulator-centric architecture [2].

At the heart of the VE server is the software which drives the simulation. We refer to the components that apply operations to objects and the scene as the actors on the scene. These functions include:

- storing the objects in a portion of the scene
- handling incoming communications from users
- simulate physics of motion and gravity on objects
- run scripted behaviors of active objects
- save scene state to provide persistence
- generate update outputs to all connected clients

We found that when any one of these functions becomes overloaded on the scene server, the user experience was diminished regardless of remaining capacity for other types of work. We therefore theorized that it is the actors on the scene and not the scene storage and management itself which are the primary barriers to scalability. Regardless of the capacity of the central server, there is some quantity of networking load, scripts, or simulation elements which will fully load any system. At that point, the quality of the user experience will be diminished. The simulator-centric architecture does not scale with the addition of hardware.

There are two common approaches to dividing a VE workload across multiple servers. Both involve dividing users into different virtual spaces with each space executing on different pieces of hardware. They are known as sharding and spatial partitioning.

Sharding is when many copies of the same virtual space are created on different servers and users are segregated into different copies with very limited interactions. MMOGs such as World of Warcraft have millions of users, but load is managed by creating hundreds of copies (called shards) of the entire game world. In a VE system implementing shards, load balancing and quality of experience is managed by limiting the number of characters which can be created in any single shard and restricting the number of players that may connect to a shard at one time through login wait queues.

Spatial partitioning is when the virtual space is divided across multiple servers with each server handling only the load of the smaller space. The Second Life virtual world uses spatial partitioning. Each server is assigned a square of land 256 meters on a side and the server, known as a simulator or just sim, is responsible for everything that takes place in that area. There is little communication between servers handling different areas other than the handoff of users and objects as they move around in virtual space.

We have proposed a new architecture which we call the Distributed Scene Graph (DSG) [2]. The DSG is a spatially partitioned scene graph which eliminates the simulator-centric execution loop. The simulation work is moved from a central server to actors around the scene graph, allowing a general purpose virtual environment to scale with the addition of hardware. In this paper, we present our work of decoupling client management from the central simulator. This is a first step and proof point in implementing the DSG. With the introduction of the client manager, we are able to support 1000 concurrent users, a 10X increase over state-of-the-art.

The primary goal of the distributed scene graph architecture is to allow the VE server workload to scale up with additional hardware while maintaining a high quality user experience. A scene that once had integrated networking, physics and script engines may now have multiple client managers or script engines as needed, running on separate hardware.

The remainder of the paper is organized as follows. In Section II, we present an overview of the distributed scene graph architecture. We describe the role and operation of the client manager component as it works with the DSG in Section III and implementation details of our prototype in Section IV. Our experimental results based on the prototype are reported in Section V and we conclude with our future work in Section VI.

II. Distributed Scene Graph

In this section, we provide a brief introduction to the distributed scene graph (DSG) as a background for the client manager component presented in the following sections. A full discussion of its capabilities and operation is presented in a separate paper [2].

In the DSG architecture, the scene is no longer a single centralized and monolithic process which drives the simulation of the space and manages data. Instead, the scene is an information hub which connects various simulation components (called actors) that interact with and through the scene. It exposes the available operations on objects through the scene interface, acts in response to actor requests (add, remove, update) and distributes object updates and world events (such as collision events, sensor events) to other interested actors. By separating the scene from the actors, the scene is free to focus on data management, state synchronization, event distribution, and provide persistence of state.

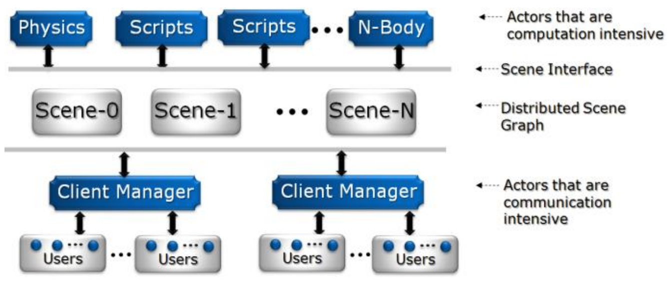

Figure 2.

As depicted in figure 2, the scene can be hosted on multiple servers, each managing a portion of the data for load balancing purposes. We call each portion of the scene a region. Regions can be of varying sizes. The regions in figure 2 are shown as Scene-0 through Scene-N, collectively called the distributed scene graph. Each scene region supports the actors by providing an interface (shown as a horizontal grey line) through which they can subscribe to interesting objects or events and publish changes to the different regions of the scene. A key challenge of our research is to develop a robust interface to the scene with efficient and scalable message exchange protocols to deliver updates between the scene and the actors with minimal communication cost. Research into pub/sub models for message distribution [3][10] and mixed synchronization for heterogeneous actors [4] have provided a basis for addressing this. The DSG aims to build on these technologies to scale virtual worlds orders of magnitude beyond their current capacity.

As an example of actors communicating with the scene and with each other, consider the case of a voice-activated door. The door has a script controlling its behavior which listens for a correct passphrase within the vicinity around the door. To enable this, the script engine subscribes to chat events within the region where the door is located. When a nearby avatar speaks, the chat activity is received by the client manager and posted into the scene as a chat event. The scene services the subscriptions for chat and forwards the message on to the script engine which then provides the message to the appropriate script for verification. If the script determines that the door should open, then the script engine posts an update of the door to the scene. That update is then broadcast to the actors which have subscribed to the state of the door, including the client manager. The updated state of the door is finally communicated back to the client who spoke the passphrase along with other nearby users who are within sight of the door.

Unlike traditional spatial partitioning, the scene regions in the DSG architecture would be split according to the number of messages that the region must process and pass in support of the actors communicating with and through each section. The size and shape of each region of the scene is selected so that the number of messages is below the processing threshold and networking capacity of the server hosting the region. The scene regions do not directly participate in any of the simulation work. There are actors computing physics, running scripted actions, processing user inputs over the network, and any other process that may have an interest in participating in the scene. These actors may be executing on the same hardware as one of the scene regions or on separate hardware. The distribution of actors would depend on available hardware and the load balancing policy.

DSG allows for multiple actors of the same type to work on a single region of the scene. The opposite is also supported where one script engine, for example, can operate across multiple regions supporting the scripted behaviors of a much larger space. Actors can now be partitioned independently of the scene space and of other actors, allowing each simulation component to run on hardware best configured for the type of simulation being performed and in a real-world proximity that provides the best balance between cost and user experience.

III. Proof Point – Client Manager

The client manager is an actor on the scene graph which allows users to interact with the scene. It manages the connection to clients, forwards client inputs to the central scene through the scene interface and broadcasts scene updates and events to connected clients within their area of interest. In addition, it could potentially perform update filtering, detail reduction or prioritization based on distance or other factors. We are experimenting with solutions such as surface elements [6] and image based rendering [7] to apply highly compressed scene representations for distant regions that may only occupy a small number of pixels on a client's screen. The client manager can also use interest management [5][8] and visibility calculation [9] algorithms when subscribing to scene objects and events to reduce the message load it must process and communicate to clients. This offloads a considerable amount of computation from the centralized scene server to the client manager.

A. Design

As the first step in implementing the DSG, we have specified the sections of the scene interface which are needed to support client managers and script engines to act on the scene. These are the scene interface calls that are used by a client manager. Subscription calls will cause a message or callback from the scene to the client manager when one of the subscribed events occurs.

- Subscribe to new or removed avatar events
- Subscribe to updated avatar (position, properties)
- Subscribe to new, updated, or removed object events
- Subscribe to spatial events (chat, sensors)
- Subscribe to directed chat events
- Add/Remove avatar
- Update avatar (change position, velocity)
- Add/Remove objects
- Update object (size, shape, properties)
- Get region info (terrain, environment, properties)

When a client manager first comes online and connects to one or more scene regions, it will subscribe to the events which are interesting to clients. These are shown in the list above. From that point, the scene region will send the subscribed updates to the client manager so that both will have a synchronized view of the scene from a client's perspective.

When a new user wants to view or participate in the scene, they will use an appropriately configured viewer or browser application. The viewer software contacts a centralized login and authentication service and specifies the location in virtual space where the avatar should be positioned. The login service then provides connection information for an appropriate client manager server and the viewer then connects to the client manager. This is similar to how a DNS request returns server address information to a web browser.

The client manager listens for connection requests from new clients wanting to participate in a scene and authenticates those clients by communicating with central account services. It then creates a representation for new users in the scene by publishing a request to add an avatar at the appropriate location in the distributed scene graph. This is done through the scene interface. The client manager receives network inputs from connected clients including movement commands, social and chat activity, economy transactions, maps, models, textures, and other operations associated with the VE, its objects and other participants. Some of these user actions modify the scene. Examples are avatar movement or building and editing objects and the environment. In these situations, the changes are published to the central scene through the scene interface. There are also many types of client input which do not change the state of the scene or its objects. Requests which only "read" data such as fetching a map, an object model, media, or textures can be handled locally by the client manager or by another service distinct from the scene interface. Other inputs such as chat messages or touching and interacting with scene objects may not alter the visible scene but can still cause events which other users or actors may be interested in.

When the client manager gets a new connection from a client, it posts a new avatar to the scene region where the avatar is positioned. An initial view of the entire area of interest for the new avatar is sent from the client manager to the client's viewer and displayed for the user. The user is then able to move about and interact with the environment, having a current view of the scene from their position.

User inputs from clients are received by the client manager and posted to the appropriate scene region where the avatar is located. Updates from the scene are broadcast to all actors which have subscribed to avatar attributes such as position. In this case, client managers each get the update and distribute to the connected clients. The client managers subscribe to avatar positions within the area of coverage, visible object updates, etc on the scene graph interface. Client managers are responsible for establishing connections to new clients as they join the scene. They work with central grid services to get avatar appearances, capabilities, maps, permissions, social data, etc.

B. Scaling Number of Concurrent Users

Figure 3 shows how multiple client managers may connect to the Scene API on a region and each manager then hosts several client connections. This additional layer can at first be thought of as a proxy for clients but the client managers are capable of offloading more than just the simple network processing from the scene server. Almost any type of request from a client that does not modify the scene can be handled directly by the client manager.

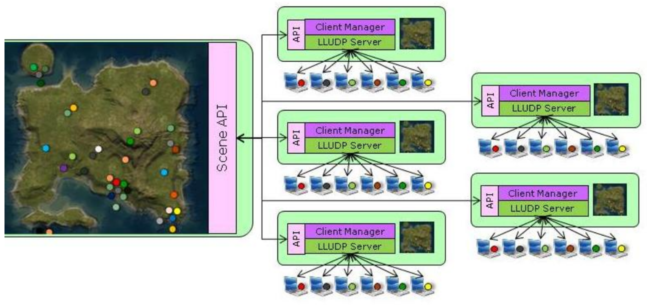

Figure 3.

Whatever quantity of messages a scene is able to process is now multiplied by the number of client managers being deployed. With five client managers and 500 clients, the central scene only needs to transmit each update or event message to five endpoints instead of 500, freeing it to continue handling transactions from other actors and manage a much larger region than current VE simulators.

When one client manager gets overloaded, it can shed load to other client managers or an additional client manager can be brought online.

IV. Client Manager Prototype

In order to measure the scalability of the distributed scene graph architecture, we built a proof of concept implementation using software from the OpenSimulator project [12]. We selected OpenSimulator because it is a mature VE server implementation with similar capabilities to the Second Life platform. It is open source so we can modify the software to meet our requirements and has a flexible framework for adding custom modules to regions of space. It is compatible with several existing viewers, eliminating the need to implement the client-side software. In its unmodified form, OpenSimulator implements spatial partitioning to distribute load. Each region of virtual space is fixed in size and is typically run on a server by itself or collocated with a few adjacent regions if server capacity allows. Different organizations or individuals can operate their own region simulators and connect them together as part of a larger virtual world known as a *grid*.

For our proof of concept, we designed two region modules for OpenSimulator that allow it to be run either as a region of the scene or as an actor providing simulation functionality to the scene. Figure 4 shows the connection between the scene server and a client manager actor through the Scene API connector. This allows identical software to run as different components in the DSG by changing configuration options. Instead of the scene and all simulation functionality running in a single executable, we can specify that an instance of OpenSimulator should only act as a script engine or client manager actor and which scene regions it should provide the functionality to.

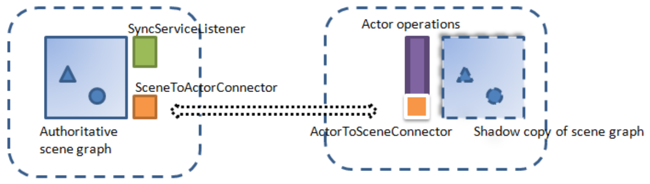

Figure 4.

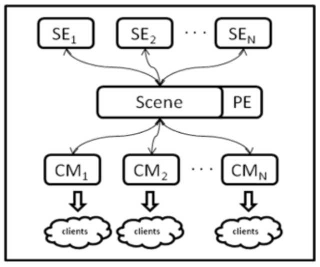

Figure 5.

We have not yet implemented the interface to support a decoupled physics engine. Figure 5 shows the prototype architecture with the physics engine (PE) still integrated with the scene service and support for multiple distributed script engines ($SE_1...SE_N$) and client managers ($CM_1...CM_N$).

A. *Scene module*

We wrapped an interface around each region through which actors can publish and subscribe to region data, objects, and avatars in the region. We did this by creating a region module on the server which accepts connections from actors (client manager, etc) and implements the scene interface. The module receives requests over the network and makes the calls into the local scene. Changes in the scene or other event notifications are pushed by the region module to the interested actors as needed. A simulator running this module thus becomes one of the regions in the distributed scene graph. It is entirely responsible for storing the authoritative copy of all objects and avatar presences in that area of virtual space, receiving updates from actors and publishing updates to subscribed actors. Other functionality other than storing and communicating object data has been disabled in this configuration.

B. *Client Manager module*

Another region module allows OpenSimulator to operate as an actor on the scene such as a client manager. In this case, all authority over the objects and avatars has been relinquished to the central scene. The client manager receives data as usual from connected clients but instead of processing that data locally, it pushes the inputs to the scene module on the central region through the Scene API. Any updates which come back from the central scene are broadcast to interested users.

C. *Scene and Client Manager Placement*

The DSG architecture is designed to run regions of the scene across multiple servers and different geographies to provide the best user experience and to scale with the addition of hardware. Different client managers can be placed in different geographic locations. Clients can connect to the closest client manger server in terms of latency and bandwidth. Communication between client managers and the central scene server will typically be over a much longer distance and more network hops. These connections can be within a high speed low latency backbone, allowing high speed, reliable communication between the scene and client managers.

V. Experimental Results

To test our implementation of the DSG with client managers, we wanted to scale up the number of connected clients and also the number of interacting clients. For our interacting client experiments, we generate a realistic client load using TestClient application from the LibOpenMetaverse project [11]. We implemented a new client action which we call *swarm bot*. Each connected bot, when given the command to swarm, picks a random waypoint within a specified area of the scene. The bot then turns and walks toward that waypoint. When it reaches its destination or has determined that it is not making progress (the path may be blocked), it picks another waypoint and repeats the process until told to stop at the end of the experiment.

Our test configuration consists of a server running the central scene region (OpenSimulator configured in scene mode), five servers running client managers (OpenSimulator configured in client manager mode), and five servers running TestClient to generate the load. Each load server can host up to 200 TestClient bots, each with a network connection to a client manager.

To evaluate improvements, we use the simulation frame rate as reported by OpenSimulator running the central scene with integrated physics. When running with no load, the baseline frame rate is above 50 fps, indicating that the inputs from all actors has been processed, the physics simulation step has been completed, and updates have been sent out to connected clients or client managers.

We ran two sets of experiments using the TestClient with increasing number of swarm bots. In the first experiment, we use an unmodified OpenSimulator server to get a baseline measurement of how the frame rate changes as the number of interacting users increases. In the second experiment we use our test bed to distribute the client management across multiple servers. We connect an increasing number of swarm clients evenly across multiple client managers.

In the case of the monolithic simulator, Figure 6 shows how the frame rate decreases as the number of connected clients increases. Once 400 clients have connected, the frame rate has begun to decrease and lag appears in the physics and network processing. With even more swarm bots, the frame rate quickly drops to where the scene is unusable. Figure 7 shows the percentage of frame time allocated to client processing continuing to increase with the number of clients. This is expected, but it also indicates that once the server becomes fully utilized at around 400 clients, both client and physics processing flatten out and the frame rate just continues to decrease.

It is this situation that the DSG is designed to eliminate by offloading and distributing the client communication processing to multiple servers, giving more processing capacity to physics and scene management. The scene server only needs to communicate with a small number of client managers rather than the hundreds of clients directly.

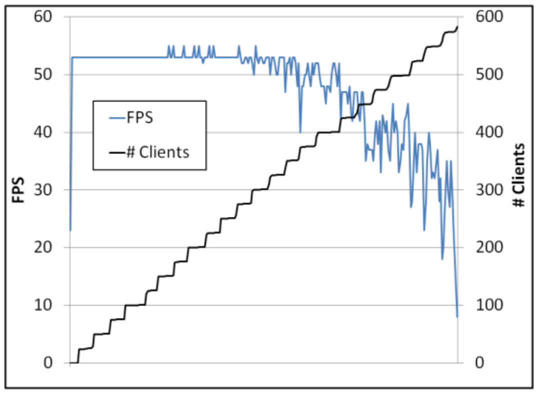

Figure 6.

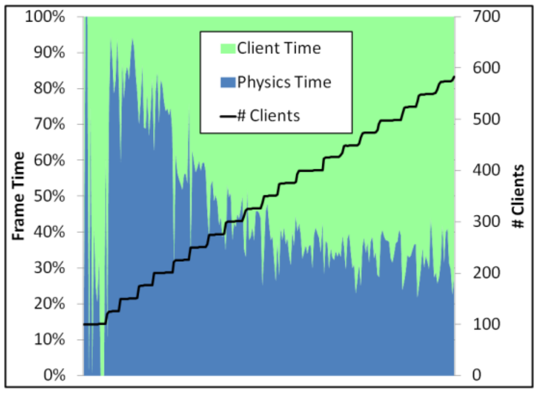

Figure 7.

In the case where we have five client managers and one central scene server, we can connect up to 750 swarm bots to the distributed client managers before the frame rate of the physics simulation drops below 40 fps. When we connected over 1000 swarming, interacting bots to the client managers, the frame rate of the physics engine dropped to below 10 fps. The OpenDynamicsEngine (ODE) physics engine used in OpenSimulator is single threaded and almost all cycles on the scene server are spent computing physics, limiting the frame rate of the scene to that of the physics engine. The DSG architecture with distributed client management has allowed us to almost entirely offload the communication processing from the central scene. These experiments confirm our ideas that offloading the simulation actors of client management, physics, script execution from the central scene will allow each actor to scale with additional hardware.

Using a configuration with client managers in multiple geographies including California, Oregon, and New York, USA, we have publicly demonstrated over 1000 swarm bots with more than a dozen human participants.

The limitations we have encountered with avatar scaling during these experiments have been in getting enough hardware to generate the load of over 1000 clients and the limited physics simulation capabilities of a single thread on the scene server.

VI. Conclusion and Future Work

We have described how the distributed scene graph with discrete actors for client management, script processing and physics simulation can convert the scene server from a monolithic VE simulator to a high speed communication bus. A publish/subscribe model from actors to the scene allow for several technologies such as spatial filtering and update detail reduction to further increase the scalability potential of the architecture.

With the client management actor in place, we have nearly doubled the number of connected clients with a "full speed" simulation frame rate by offloading the client processing from the scene. We have also demonstrated publicly over 1000 connected avatars with an interactive frame rate. These results validate our DSG proposal that offloading simulation processing from the monolithic scene server and communicating with and through the scene via a Scene API allows the entire VE scene to scale with the addition of hardware.

In other areas of our DSG work, we have demonstrated a distributed script engine which enables multiple servers to process scripts for objects within a single region of virtual space. Our research continues into creating an actor for distributed and multi-threaded physics simulations.

Future experiments will include network emulation and more testing of the distributed client managers on the open Internet. We expect to encounter additional challenges when the latency and jitter of real networks are introduced to the synchronization, publish/subscribe functions of the distributed scene graph.

References

[1] Gupta, N., Demers, A., Gehrke, J., Unterbrunner, P., and White, W. 2009. **Scalability for Virtual Worlds.** In Proceedings of the 2009 IEEE international Conference on Data Engineering (ICDE).

[2] H. Liu, M. Bowman, R. Adams, J. Hurliman, and D. Lake. **Scaling virtual worlds: simulation requirements and challenges.** To appear in Proc. of Winter Simulation Conference, Decemeber 2010.

[3] Ostrowski, K., Birman, K., and Dolev, D. 2007. **Extensible Architecture for High-Performance, Scalable,Reliable Publish-Subscribe Eventing and** Notification. International Journal of Web Services Research 4 (4), 18-58.

[4] Perumalla, K. 2006. **Parallel and Distributed Simulation: Traditional Techniques and Recent Advances.** In Proceedings of the 2006 Winter Simulation Conference.

[5] Graham Morgan , Fengyun Lu , Kier Storey, **Interest management middleware for networked games**. In Proceedings of the 2005 symposium on Interactive 3D graphics and games, April 03-06, 2005, Washington, District of Columbia

[6] Pfister, H., Zwicker, M., van Baar, J., and Gross, M. 2000. **Surfels: surface elements as rendering primitives.** In Proceedings of the 27th Annual Conference on Computer Graphics and interactive Techniques, SIGGRAPH 2000.

[7] Shum, H.-Y. and Kang, S. B. 2000. **Review of image-based rendering techniques.** In Proceedings of SPIE Int'l Conf. on Visual Communications and Image Processing, Perth, Australia, June 2000, pp. 2-13.

[8] Taylor, S. J., Saville, J., and Sudra, R. 1999. **Developing interest management techniques in distributed interactive simulation using Java.** In Proceedings of the 1999 Winter Simulation Conference.

[9] Kumar, S., Chhugani, J., Kim, C., Kim, D., Nguyen, A., Dubey, P., Bienia, C., and Kim, Y. 2008. **Second Life and the New Generation of Virtual Worlds.** Computer 41 (9), 46-53.

[10] Shun-Yun Hu, Chuan Wu, Eliya Buyukkaya, Chien-Hao Chien, Tzu-Hao Lin, Maha Abdallah, Jehn-Ruey Jiang, and Kuan-Ta Chen, **A Spatial Publish Subscribe Overlay for Massively Multiuser Virtual Environments.** In Proc. 2010 International Conference on Electronics and Information Engineering (ICEIE 2010), Aug. 2010

[11] OpenMetaverse project website: http://www.openmetaverse.org/

[12] OpenSimulator project website: http://opensimulator.org