Using RTP and RTSP for Real-Time 3D Interaction

Kristian Kuhn

A paper submitted to the
Computer Science & Electrical Engineering Department
in partial fulfillment of the requirements for the M.S. degree at
University of Maryland Baltimore County

1/7/2006

Certified by: Dr. Marc Olano
Advisor's Signature:_______________________________________
Date________________

1 Abstract

This project report presents a mechanism for allowing remote clients to navigate and interact with a 3D scene that resides entirely on a server, while having only a subset of the scene in memory at an instance. This will allow the client to traverse larger scenes than it could handle locally and does not require the client to have any persistent storage. Real-Time Transport Protocol (RTP) is used to manage data distribution and Real-Time Streaming Protocol (RTSP) for state synchronization. These protocols were originally designed for streaming audio and video feeds to a client. The described mechanism extends these protocols to allow the streaming of 3D scene data, which is rendered on a client. Unlike streaming audio and video, where the data is only valid during a given time interval, the algorithm will not dispose of the scene data once it has been received. The stream data can then be reused when necessary, reducing the transmission cost to the client. Because the overall scene may be larger than the client can maintain, the client can dispose of portions of the scene as needed. In the event that the client disposed of scene data that it now needs, the protocol will retransmit that information.

2 Introduction

The idea of remote navigation and interaction with a 3D environment has generated quite a bit of attention with the increased presence of broadband in the home and the recent surge of online gaming. In today's world, the client is required to: a) already have an existing knowledge of the scene by installing software from a provided media, or b) download the scene information prior to interacting with the environment. When interacting with a remote scene, it is likely that the scene is larger than the client could manage locally on its own.

One example of a Large Scale Virtual Environment (LSVE) is an online gaming world. An online gaming world is typically managed across a farm of servers, and it would not be possible for a client to maintain the entire world. In addition, the user must install a client application specific to that online gaming world before they can begin to interact with it. Updates to the world must be downloaded offline before the user can continue interacting with the environment.

To relieve this burden, we developed a technique to pre-fetch portions of the scene, including associated resources, and transmit them to the client on an as-needed basis. In addition to online gaming, this technique could be applied to handheld clients or clients with limited communications bandwidth.

Today many different portable devices are capable of rendering 3D graphics but lack the runtime and/or persistent storage to maintain a large scene in the memory. For example a Special Forces soldier will be in the field with a handheld computing device wirelessly linked to a ground station. Due to the nature of their work, these machines will have significant computing power, but they lack substantial persistent storage due to their size and propensity to be damaged. This technique would allow the soldier to visualize the battlefield on their computing device as if they were sitting at a terminal attached to the ground station using a high speed link. Many researchers have been working on new solutions to the problem by proposing new means for navigating an interacting with LSVE.

3 Related Work

Current large scale virtual environment systems have identified common areas of functionality^ many of which will pertain to future LSVEs. The areas of particular interest are entity management, communication model, scenegraph management and data reduction and compression. Several systems have been developed that provide significant contributions to these major functional areas.

3.1 Entity Management

In an LSVE system, entities can influence the state of other entities in the system. For large environments, managing these interactions in a real-time manner poses a significant challenge.

In 1999, Greenhalgh and Benford~~,~~ developed the MASSIVE system. The MASSIVE system defined a concept known as an "aura"~~. Every object in the MASSIVE environment has an aura.~~^ The aura defines the physical extent to which interaction with other objects in the environment is possible. When two auras collide, an interaction between those objects is possible. The aura reduces the number of possible interactions a given entity can have, which reduces the computational overhead of managing interactions within an LSVE. The effect of an interaction occurrence varies based on the application and the object's behavior. In an LSVE system, entities come in many different flavors each with unique properties and behaviors. Dynamic discovery of new entity types provides for a rich and extensible system.

The Bamboo system [Watsen and Zyda 1998] focused on dynamic configuration without explicit user interaction, allowing the system to discover new virtual environments on the Bamboo network at runtime. The Bamboo framework's network component is capable of downloading and updating the local scene with information from a remote site. This information may typically be geometry, texture and sounds, and it may also contain executable code that defines its physical behavior. Today's massively multiplayer, online games include features for developing custom modifications to the LSVE. The extensions must be acquired offline before the user can interact in an environment with them. Applying techniques from the Bamboo system would remove the offline acquisition step and improve the user experience. The Bamboo system recognizes the security problems associated with downloading binary data from un-trusted hosts, but it was left as an area of future work.

3.2 Communication Model

Large scale virtual environments require communication between a large number of client and server processes. The server entity will be composed of one or more computers designed to manage the virtual environment. As with all distributed computing applications, the network topology used can greatly affect the performance of the system.

The MASSIVE architecture utilizes a peer-to-peer (P2P) framework built upon a combination of streams and RPC for exchanging data among peers. P2P networks usually have the option of retrieving identical data from multiple locations and are designed to retrieve it from the most cost effective node. By design a P2P network is decentralized, meaning clients can talk directly to one another without relaying their messages through a common server, and because of this there is no single network bottleneck or hotspot. Due to the lack of a centralized authority, it is more difficult to maintain a consistent global state. In addition to P2P, a client/server based approach is also possible and provides some benefits over P2P.

Macedonia ~~et. al.~~ [1995] developed a system for interacting with an LSVE utilizing multicast groups. When managing an LSVE, the server must manage a large number of participants. The solution is built upon partitioning the virtual environment by associating “spatial, temporal, and functionality related entity classes with network multicast groups” [Macedonia et. al. 1995]. Each partition is associated with a multicast group. The partitioning allows them to devote an ~~entities~~ processing and networking resources for its area of interest to a specific local Area of Interest Manager (AOIM). The AOIM uses the partitioning properties to determine membership in multicast groups for entities. In their implementation, the AOIM partitions the virtual environment into hexagonal cells. They chose hexagons for two main reasons: 1) they have uniform adjacency, that is as an entity leaves one cell, it moves exactly into another, and 2) they provide better coverage of an entity’s AOI than a square would, without having to reduce the size of the cell, which would directly increase the number of multicast groups. In their implementation an entity may belong to multiple multicast groups if its AOI spans multiple cells, but it only publishes updates to the multicast group that corresponds to the cell it is in (based on location point). This model using multicast groups creates a potential network bottleneck but simplifies the state management compared to P2P since all updates come directly from the server entity. The described communication models require proprietary network configuration. A communication model that was inherently compatible with today’s web technologies would be ideal.

3.3 Scenegraph Management

Both managing interactions and communicating changes in the scene, affect the contents of the global scenegraph. Two existing systems have provided contrasting methods for managing the global scenegraph. A scenegraph is a graph structure that represents a three dimensional environment; it can be used traversed to generate a two dimensional image of the environment.

Cheng et al. [2004] proposed a real time 3D graphics streaming architecture leveraging MPEG-4. They divide their architecture into a three layered system: 1) control plane, 2) data plane and 3) transmission plane. The control plane transmits user interactivity to the server where it is interpreted by the data plane. The data plane renders the scene using the server's resources based on the user interactivity at the client. It is assumed that the client lacks the resources to render the scene (i.e. lack of necessary rendering hardware or lack of the data to be visualized). After rendering the frame, it is compressed using MPEG-4 then streamed back to the client. Cheng et al. address consistency issues with a globally replicated scenegraph by only maintaining one scenegraph on the server and performing all rendering on the server. An opposite approach would be to maintain replicas of the scenegraph on all clients and have the client's use their resources to render the scene.

Sahm and Soetebier [2004] proposed another method for managing remote scenes using a client/server scenegraph that is represented as a dynamic Space Partition Tree (SPT) on both the client and the server. Their proposed scenegraph representation allows for easier matching of objects on the client and the server, which simplifies the identification of objects being manipulated in an interactive scene. The scenegraph design also simplifies the process of managing scenes that exceed the available memory, even with regard to the server. The organization of their scenegraph allows for quicker identification of objects that need to be sent to the client, based on the client's viewing parameters. Access time to locate and update objects in the scenegraph is crucial to providing real-time performance. In addition to scenegraph design, minimizing the overall size of the scenegraph is equally important.

3.4 Data Reduction and Compression

Reducing the size of the scenegraph maintained on the client is important for both rendering speed and minimizing the memory footprint required to participate in the LSVE. Teler and Lichinski [2001] approached the problem by streaming the 3D scene using a real-time on-demand algorithm. In this approach, the server sends sections of the scene to the client, based on the client's viewing parameters, while focusing on bandwidth as the bottleneck in rendering a remote scene. By using bandwidth analysis, continuous level of detail and image imposters, the transmission cost is reduced. This approach is ideal for clients that have unlimited local resources (i.e. a state of the art home PC), because it is assumed that the server will never need to retransmit anything to the client.

In addition to trimming the client-side scenegraph, one could also compress the data stored at each node. One mechanism for doing this is geometry compression. Deering [1995] proposed a new mechanism for compressing 3D triangle data with six to ten times fewer bits than conventional techniques, with limited loss. This approach required an efficient decompression algorithm, but it did not require a real-time compression solution. In this analysis, it was assumed that all geometry was pre-compressed offline before the 3D scene was to be rendered, and that no compression would be required at runtime. In addition to discussing the software implementation of the algorithm, Deering pointed out that this design was meant for hardware decompression on the 3D accelerator, thus simulating higher bandwidth on the graphics bus.

The LSVE systems to date have presented varying solutions to these common architectural areas. Presented is a new architecture that builds on the current systems and incorporates web technology friendly transport protocols, as well as new algorithms for managing the global scenegraph.

4 Design

4.1 Overview

We present a new mechanism for navigating and interacting with a 3D environment from a remote client. This mechanism introduces two main concepts: data delivery and data synchronization. Data delivery is the ability to deliver scene information to the client in a timely manner. This information may include geometry, textures and audio. The data delivery service must be extensible to other types of binary data to ensure adequate flexibility. Unlike the approaches discussed earlier, we will utilize a standard protocol known as RTP [Schulzrinne et al. 2003] for transmitting streaming scene data to the client. Real Networks and Apple QuickTime, as well as other lesser-known streaming media applications, use this protocol to deliver streaming audio and/or video feeds. The RTP protocol is capable of streaming binary data to a client bundled with timing, sequence information and payload encoding information. In addition to delivering the scene information, the client and server must be able to synchronize their states of the scene (data synchronization). Like in the Unreal Network [Sweeney 2005] architecture, the server's state will be considered the absolute authority, and clients must correct their state as they receive updates from the server. In our approach, the client's scene state is merely an approximate replica of a subset of the scene resident on the server. The communication of state between client and server will be handled using the RTSP protocol [Schulzrinne et al. 1998]. The RTSP protocol can be thought of as the remote control for the scene playing out on the server. When using this protocol to manage streams of video or audio one can adjust the temporal properties of the stream (i.e. rewind or fast forward). In this application, instead of manipulating temporal properties, we adjust the spatial position of the client actor, which in turn will adjust the flow of data on the stream. The client will use this protocol to:

1) Set up a session with the server
2) Begin playing in and interacting with the 3D environment
3) Disconnect from the server

4.2 Communications Layer

RTP and its control counterpart, RTSP, are independent high-level transport protocols. They do not define the mechanism for delivering the packets at the network layer, and because of that, they do not depend on their capabilities in order to function properly. We transmit the RTP packets using a connectionless protocol known as UDP (User Datagram Protocol). The UDP protocol does not guarantee delivery, order or Quality of Service (QOS). Since RTP provides sequence numbers in its message headers, out of order delivery and packet loss are overcome, making UDP a viable transport protocol. The UDP protocol provides the ability to multicast data to a group of clients. Multicasting is the process of sending data to a client group address, and the underlying protocol will transmit a copy of the data to each client in the multicast group. This is ideal when managing a scene with a large number of clients.

RTSP also does not require a particular transport protocol. The RTSP protocol is a text message based protocol and was purposely designed to be similar to HTTP. Like HTTP, the RTSP implementation will run on TCP. RTSP does not have the capabilities of RTP that allowed the use of UDP, and it does not adapt well to packet loss. It does provide a means for reordering requests that was designed for pipelining purposes; that is allowing a client to send out additional requests, while it has one or more outstanding requests.

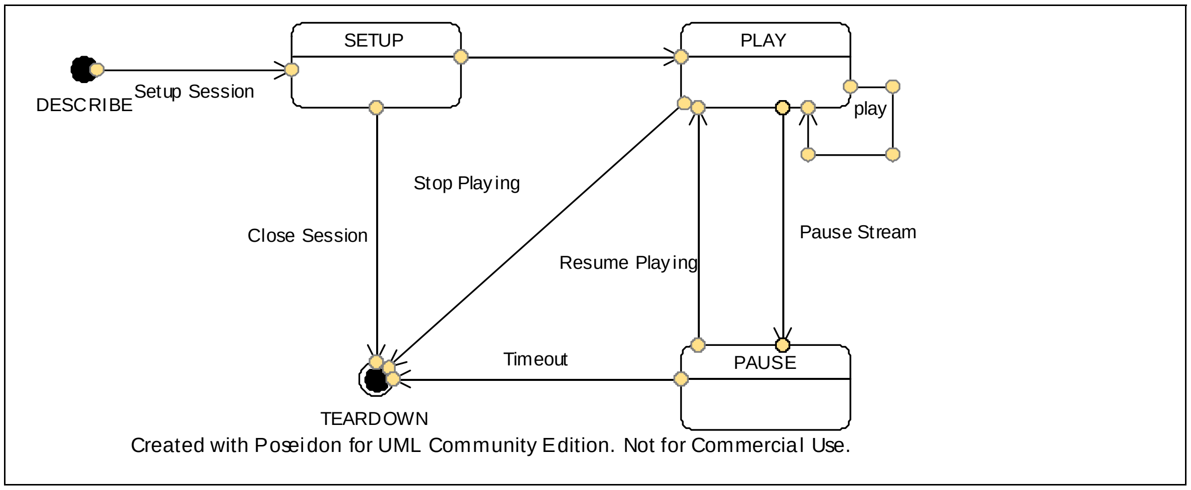

Figure 1 – RTSP Connection State Machine

4.3 Scene Management

4.3.1 Initialization

Before a client can begin rendering and interacting with a remote 3D environment, it must setup a session with the server. Setting up a session is a multiple step process using the DESCRIBE and SETUP messages described in the RTSP protocol. The DESCRIBE method is used by the server to provide the presentation details to the client. The presentation details contain information about the streams available at that server. The information includes network settings, CODECs and other implementation specific data. Typically this information is described using the SDP [Handley and Jacobson 1998] protocol, which is used in this implementation. Once the client has determined which stream it would like to listen to, it issues a SETUP message to the server. In the response message from the SETUP request, the server establishes the session and returns the session identifier to the client.

Client->Server:

```
    DESCRIBE rtsp://server.example.com/fizzle/foo RTSP/1.0
    CSeq: 312
    Accept: application/sdp, application/rtsl, application/mpeg
```

Server->Client:

```
  RTSP/1.0 200 OK
       CSeq: 312
       Date: 23 Jan 1997 15:35:06 GMT
       Content-Type: application/sdp
       Content-Length: 376

        v=0
        o=mhandley 2890844526 2890842807 IN IP4 126.16.64.4
        s=SDP Seminar
        i=A Seminar on the session description protocol
        u=http://www.cs.ucl.ac.uk/staff/M.Handley/sdp.03.ps
        e=mjh@isi.edu (Mark Handley)
        c=IN IP4 224.2.17.12/127
        t=2873397496 2873404696
        a=recvonly
        m=audio 3456 RTP/AVP 0
        m=video 2232 RTP/AVP 31
        m=whiteboard 32416 UDP WB
        a=orient:portrait
```

Figure 2 – Example DESCRIBE interaction adapted from Schulzrinne et al. [1998]

Client->Server:

```
    SETUP rtsp://example.com/foo/bar/baz.rm RTSP/1.0
    CSeq: 302
    Transport: RTP/AVP;unicast;client_port=4588-4589
```

Server->Client:

```
    RTSP/1.0 200 OK
    CSeq: 302
    Date: 23 Jan 1997 15:35:06 GMT
    Session: 47112344
    Transport: RTP/AVP;unicast; client_port=4588-
    4589;server_port=6256-6257
```

Figure 3 – Example SETUP interaction adapted from Schulzrinne et al. [1998]

4.3.2 Interaction and Scene Maintenance

Once the session has been established, the client must initiate the beginning of the data stream by issuing a PLAY message. The PLAY message contains the initial spatial properties of the client. Upon receipt of this message, the server begins transmitting the scene data to the client based on the client's initial viewing parameters. At anytime after transmitting the PLAY message, the client may issue a PAUSE message. The PAUSE message will cause the server to stop transmitting scene data to the client. During the session, the client will continuously update its spatial properties, including position and direction of movement, with the server. The server uses the update information to determine what portions of the scene the client might need next. Unlike streaming audio or video data that is only valid during a specific time period, the scene data is valid whenever it is in view of the client actor. This property can be leveraged to optimize the scene data transmitted from the server. In order to use this property, several issues must be addressed. The first is replicating scene data on the client and synchronizing that data with the server. More formally, this requires a means of correlating entities on the client and the server and exchanging updates to the scene data, the details of which are presented later. The second requires the server to have knowledge of what data is on the client so that it does not retransmit data that the client already has, unless it has been modified.

```
Client->Server:
    PLAY rtsp://audio.example.com/twister.en RTSP/1.0
    CSeq: 833
    Session: 12345678
    Range: smpte=0:10:20-;time=19970123T153600Z

Server->Client:
    RTSP/1.0 200 OK
    CSeq: 833
    Date: 23 Jan 1997 15:35:06 GMT
    Range: smpte=0:10:22-;time=19970123T153600Z
```

Figure 4 – Example PLAY interaction adapted from Schulzrinne et al. [1998]

```
Client->Server:
    PAUSE rtsp://example.com/fizzle/foo RTSP/1.0
    CSeq: 834
    Session: 12345678

Server->Client:
    RTSP/1.0 200 OK
    CSeq: 834
    Date: 23 Jan 1997 15:35:06 GMT
```

Figure 5 – Example PAUSE interaction adapted from Schulzrinne et al. [1998]

4.3.3 Uniquely Identifying Scene Entities

As presented earlier, in order to maintain consistency between the client and server, an object must be uniquely identified consistently on both the client and the server. In this solution the server will assign identifiers to all entities that exist in the scene. The server will use UUIDs (Universally Unique Identifiers) [The Open Group 1997] for all entities in the scene. The UUID specification was originally developed by the Open Software Foundation as part of its DCE (Distributing Computing Environment). It was designed to uniquely identify an entity in a distributed environment with reasonable certainty that it will not be duplicated intentionally. UUID is currently documented as part of the ISO/IEC 11578, and an effort is underway by ISO/IEC and the IETF to document it as a separate standard.

```
0679E900-A387-110F-9215-930269220000
```

Figure 6 – Example UUID

4.3.4 Sending Updates to the Server

The RTSP protocol provides a general message for setting and retrieving parameters on the server or client. The SET_PARAMETER message is used to transmit scene data updates and client spatial information to the server. In addition, the server may use the GET_PARAMETER message to request updates from the client. It may also be valuable if the client were to buffer updates to the scenegraph until another client needs them, similar to a delayed write back cache. In this implementation, updates are immediately transmitted to the server.

Server->Client:

```
GET_PARAMETER rtsp://example.com/fizzle/foo RTSP/1.0
CSeq: 431
Content-Type: text/parameters
Session: 12345678
Content-Length: 15

packets_received
jitter
```

Client->Server:

```
RTSP/1.0 200 OK
CSeq: 431
Content-Length: 46
Content-Type: text/parameters

packets_received: 10
jitter: 0.3838
```

Figure 7 – Example GET_PARAMETER interaction adapted from Schulzrinne et al. [1998]

Client->Server:

```
    SET_PARAMETER rtsp://example.com/fizzle/foo RTSP/1.0
    CSeq: 421
    Content-length: 20
    Content-type: text/parameters

    barparam: barstuff
```

Server->Client:

```
    RTSP/1.0 200 OK
    CSeq: 421
    Date: 23 Jan 1997 15:35:06 GMT
```

Figure 8 – Example SET_PARAMETER interaction adapted from Schulzrinne et al. [1998]

4.3.5 Managing Multiple Clients

So far only a methodology for controlling a 3D interactive scene using RTSP and delivering the scene data using RTP have been discussed. While our implementation functions in a single client state, its true value is in multi-client state. Managing multiple clients means handling changes to the global scenegraph from multiple sources. When managing the changes, the temporal ordering of the updates must be maintained. The architecture provides a queuing mechanism for buffering updates received from the clients. In addition, the queue is self-sorting, and it orders the messages using temporal properties maintained within the updates. The server maintains a world time for the 3D environment, and each client will approximately synchronize to that time. This allows the clients to timestamp their messages with reasonable accuracy. Applying a timestamp to a message and maintaining a session time is natively supported by RTSP due to its original design for streaming audio and video.

4.3.6 Termination of the Session

When a client disconnects from the server, it sends a TEARDOWN message to the server. When the server receives this message, it releases all resources associated with this session, and any future messages from the client with regard to this destroy session are ignored. The TEARDOWN message is only valid once the client has issued a SETUP message to server. An implicit TEARDOWN may occur if the client has paused the stream using the PAUSE message and the timeout interval has elapsed. In the event that an implicit teardown has occurred, a TEARDOWN message is sent from the server to the client.

```
Client->Server:
    TEARDOWN rtsp://example.com/fizzle/foo RTSP/1.0
    CSeq: 892
    Session: 12345678
Server->Client:
    RTSP/1.0 200 OK
    CSeq: 892
```
Figure 9 – Example TEARDOWN interaction adapted from Schulzrinne et. al. [1998]

4.4 Streaming Scene Data Using RTP

The RTP protocol is used to stream the scene data to the client. The scene data includes the scenegraph, textures, audio and any other resources that are required for the scene implementation. RTP does not concern itself with contents of the payload, making the implementation independent of the scenegraph package used on the client and the server. The payload of the RTP packets is the serialized DirectX scenegraph. The DirectX scenegraph may require other resource files. These files are also transmitted using the RTP channel.

The client does not dispose of the scene data it received from the server, unless it is forced due to a resource constraint. Using RTSP, a session between the client and the server is established. Utilizing this session, the server maintains a list of the UUID's for all entities it transmitted to the client. In the event that the client disposed of an entity, it must notify the server by sending the UUID to the server using the SET_PARAMETER message. If it does not notify the server, it may not receive that entity for the duration of the session. The list of entities transmitted to the client will allow the server to only send entities to the client that are new or have been modified since the client was last updated. When a client receives updated scene data from the server, it must determine if it received new or modified entities. This is accomplished by comparing the entity's UUID with all entities known to exist in the client's subset of the scene. If the entity already exists, it is replaced by the entity that was received. If it is not already in existence, the entity is added to the client's scene. This approach handles only new and updated entities. In the implementation a hashtable is used to quickly determine if a node exists in the scenegraph. The table provides a mapping of the UUID directly to the corresponding node in the scenegraph. It does not provide enough information to determine if an entity was removed. When an entity is removed from the scene, a remove message is sent as the payload of a SET_PARAMETER message containing the UUID of the entity to be removed.

4.4.1 Determining the RTP Payload

The previous section outlined, at a high level, the process of transmitting and incorporating scene data received from the server. This section highlights the process of determining what to send to the client and how to maintain the accuracy of that data. In order for the client to begin rendering the scene, it must have scene data based on the client's current viewing parameters. First, the client transmits its initial viewing parameters. The parameters include a spherical area of interest and a direction of movement vector. Upon receipt of the viewing parameters, the server begins transmitting the related scene data. On the client's end, the data is buffered until it receives all the necessary data to render the scene accurately based on the initial viewing parameters. Once the server finishes transmitting the initial scene data, it begins the speculative pre-fetching portion of the session lifecycle. During the speculative pre-fetching, the server makes an educated guess when sending additional data to the client. The guess is based on the client's known current position and the associated direction of movement vector. In addition to the speculative data, the server is also responsible for sending any updates to the scene that occur in the client's current area of interest. Throughout this process, the client is rendering and navigating the scene. As it navigates the scene, it continuously updates its viewing parameters with the server. The viewing parameter updates are used as the basis for the speculative pre-fetching. In order to maintain the accuracy of the client's cache, the server invalidates an entity that has been modified in the scene. For example, suppose a client had recently viewed a pencil on a table and then left the area. While the client was gone, another client removed the pencil from the table. When the first client returned to that area, it would render the pencil because the pencil is still in the cache. This problem is solved by invalidating anything in the client cache that is modified in the scene. In addition to server invalidation, an entity may be removed from the cache due to a resource constraint on the client. When this occurs, the client removes the least recently used entry in the cache. This is believed to be correct because the elements that have been used more frequently are more likely to be used again and are more likely to be in a fixed position, which eliminates the possibility that the server will invalidate them.

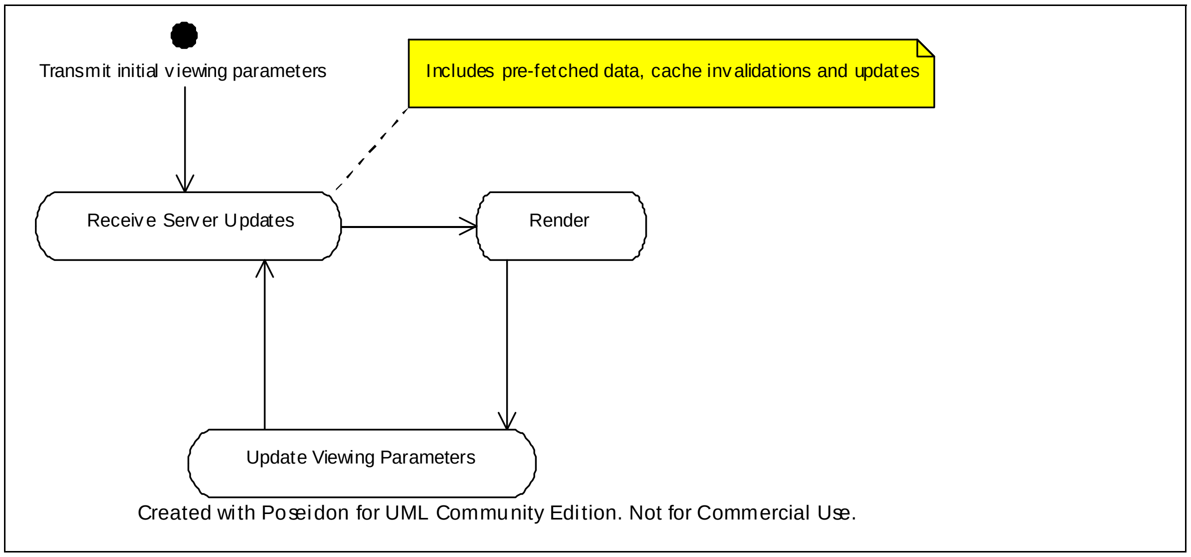
Transmit initial viewing parameters
Includes pre-fetched data, cache invalidations and updates
Receive Server Updates
Render
Update Viewing Parameters
Created with Poseidon for UML Community Edition. Not for Commercial Use.

Figure 10 – Client Management Activity Diagram (Client)

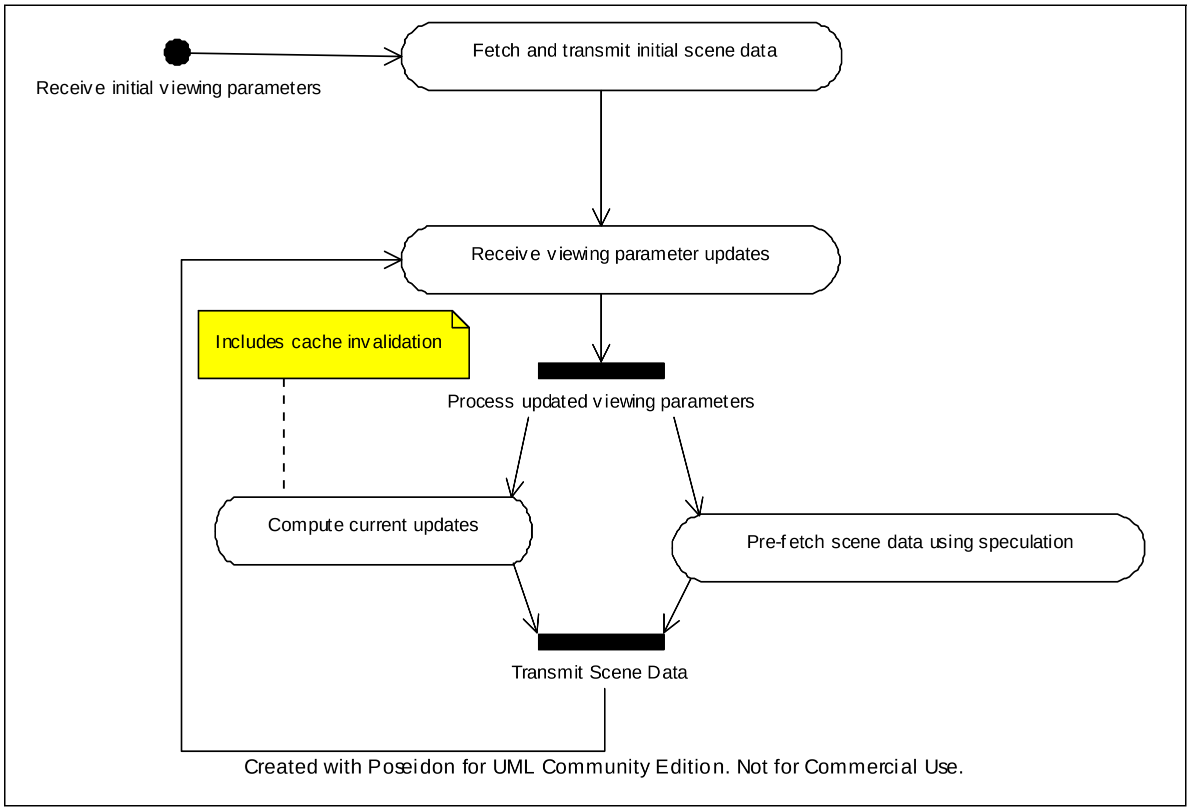
Receive initial viewing parameters
Fetch and transmit initial scene data
Receive viewing parameter updates
Includes cache invalidation
Process updated viewing parameters
Compute current updates
Pre-fetch scene data using speculation
Transmit Scene Data
Created with Poseidon for UML Community Edition. Not for Commercial Use.

Figure 11 – Client Management Activity Diagram (Server)

4.5 Rendering

Rendering the scene on the client is done in a separate thread. This keeps the rendering from being delayed by update processing. Prior to rendering, but in the context of the render thread, any entities in the scene that have a position and velocity vector will be moved, unless their positions have been updated by the server since the last render pass. This allows the client to guess as to how the entity is moving for frames that are rendered in between updates from the server. When the client receives an update for an entity, the client will correct the position and velocity vector for that entity. Rendering consists of two phases: update processing and rendering. These phases repeat through the lifecycle of the application. As updates come in, they are buffered while the current frame is being rendered. After the render completes the buffered, updates are processed and the next frame is rendered. This will continue until the application is terminated. Each update that is received from the server is an add/modify/remove a node. The node is identified using the UUID as described above. The UUID is used to locate the node being modified or deleted, or is used to add a new node to the scenegraph. As an optimization in this implementation, we added a secondary data structure that maps the UUID directly to the node in the scenegraph. An entry is made to this table each time a node is added, and the entry is removed when it's deleted. In addition to this optimization, each node maintains a reference to its parent. This back reference enables us to delete a node quickly from the scenegraph by deleting the child from its parent without traversing the tree to locate the parent of a node that is to be removed. Add and updates contain the new node as well as the UUID of the parent node. Using the "UUID to node mapping" we can find the parent and append the child in constant time.

5 Implementation

A prototype of the design was implemented, it is not as full featured as the design but it illustrates the proof of concept. The prototype implementation is capable of disseminating the initial scene state to all attached clients. The clients render the scene using a DirectX scenegraph that was developed as part of this effort. In addition, a prototype communications layer based on the RTP/RTSP protocols was developed. The communications layer is used by the clients to send updates to the server. In this prototype implementation, the communications layer is used to send the client's position to the server and it also disseminates the positional information of other objects in the scene. On the server, it receives the update messages and makes the corresponding updates to the server side scenegraph and disseminates that information to all clients.

This deviates slightly from the design, in a production implementation the updates would only be sent to clients that needed that information at that time. For example, if an object was moved by client A, client B would only receive the update from the server if the object was in client B's scenegraph. If client B receives an update message for an entity that it does not have in its local scenegraph, the update is discarded.

The full multi-user implementation as described in the design has not been fully implemented. Each message is time stamped but they are processed in the order in which they are received from the server. Due to the speed of the local network, the packets tend to match their temporal order. The following are several screenshots from the client and the server. The client images illustrate the process of moving the camera in the scene and receiving updates from the server. The server images show the server view in the scene when the camera is at the same position and direction as the client. Figures 12 and 16 show the initial scene state when the application loads on both the client and the server. In Figure 13, we have suspended updates from the server and moved the camera, when the updates are resumed the pink spaceship appears on the client as can be seen in Figure 14 which now matches the server state, Figure 17.

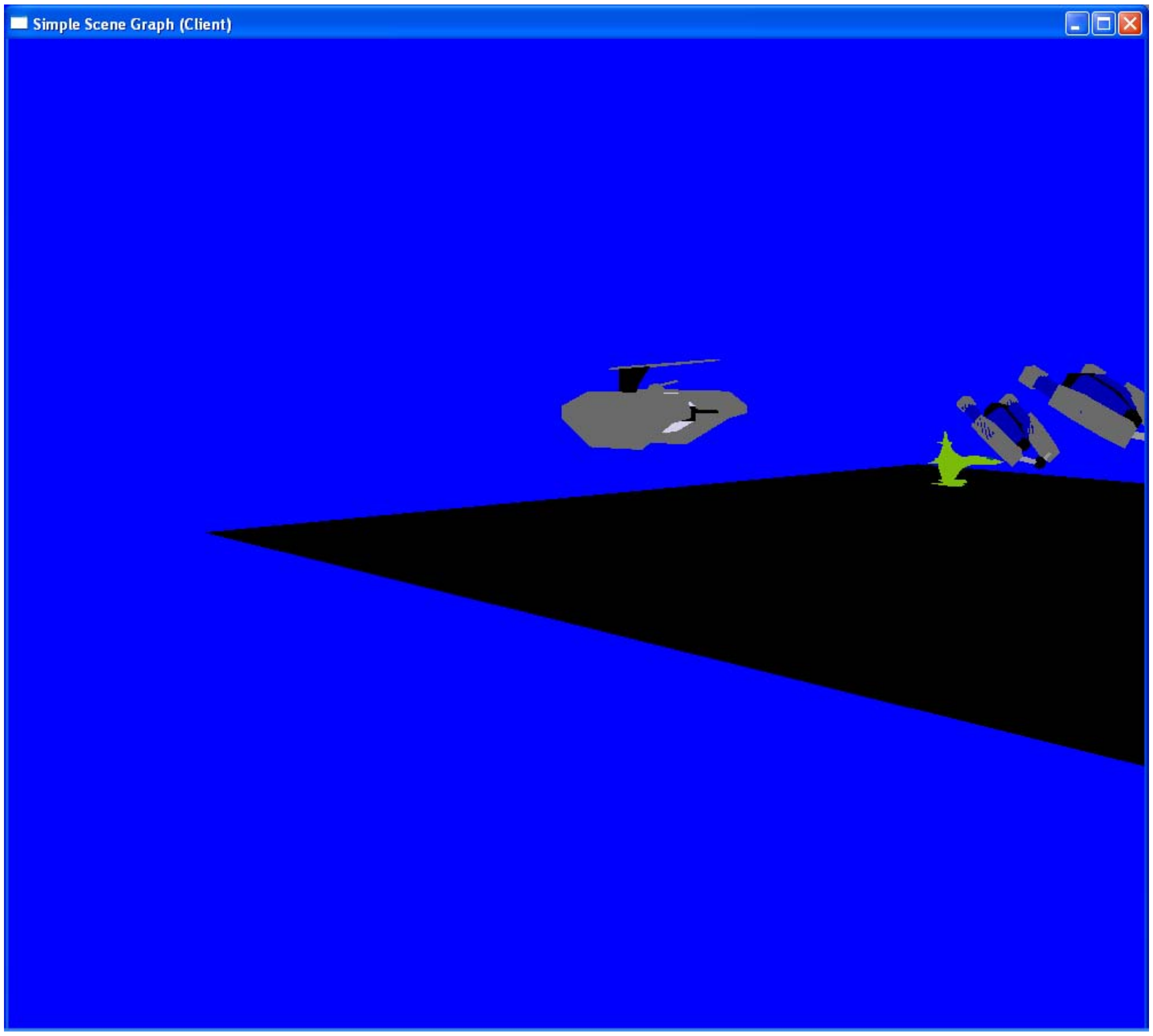

Figure 12 – Initial client scene state.

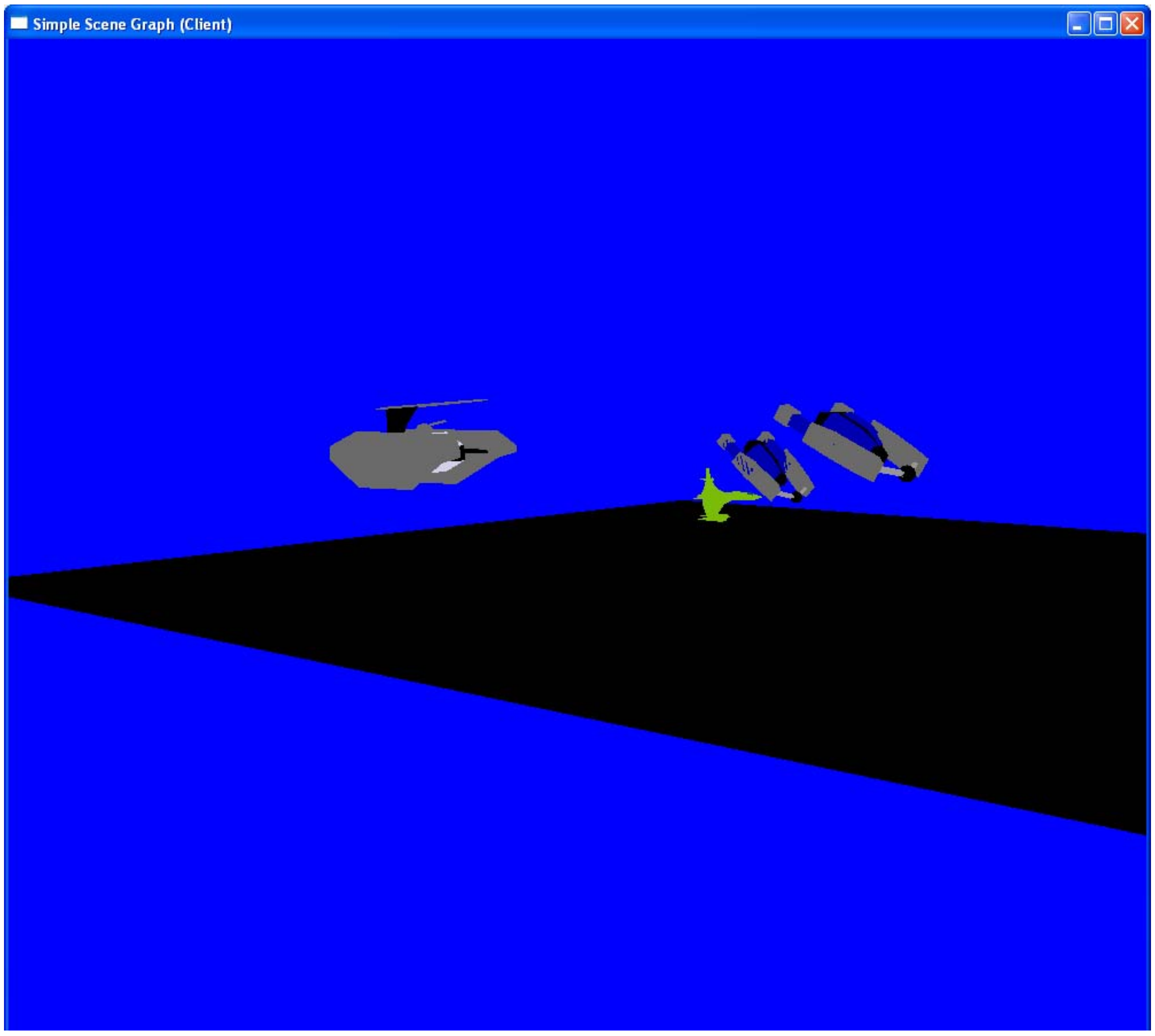

Figure 13 – Camera reposition with updates suspended.

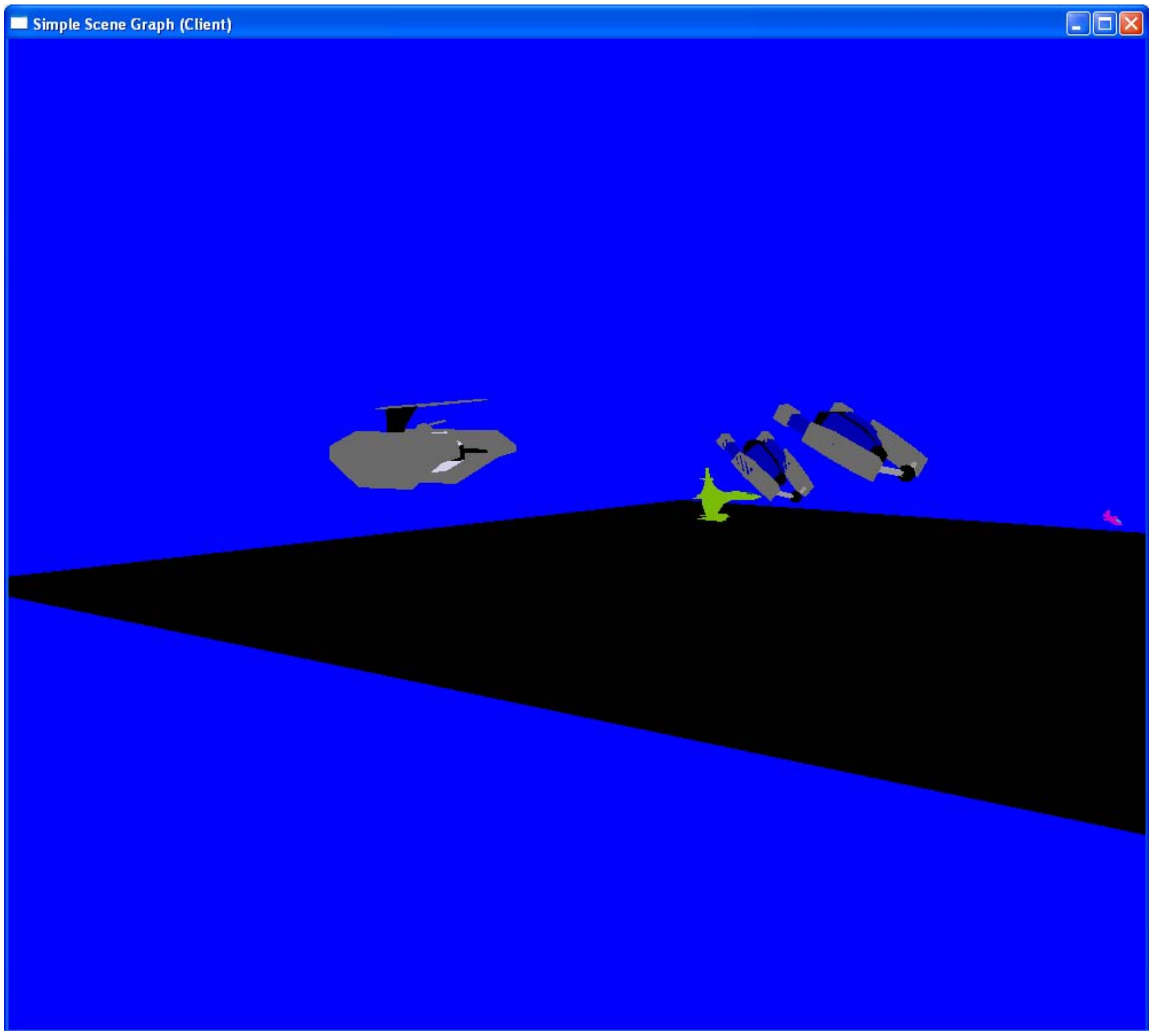

Figure 14 – Updates resumed on client, Pink Spaceship appears.

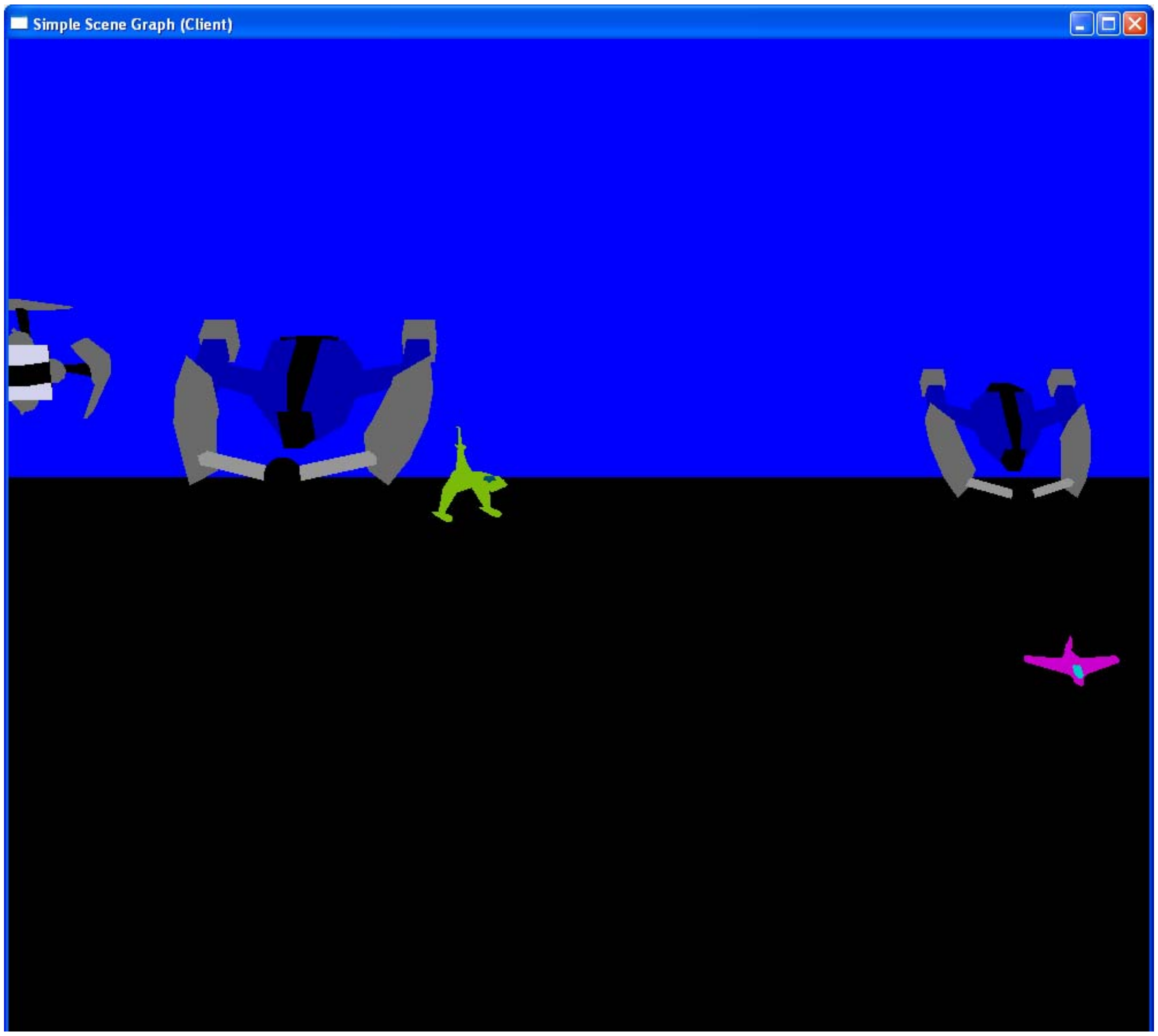

Figure 15 – Frontal view of the client scene.

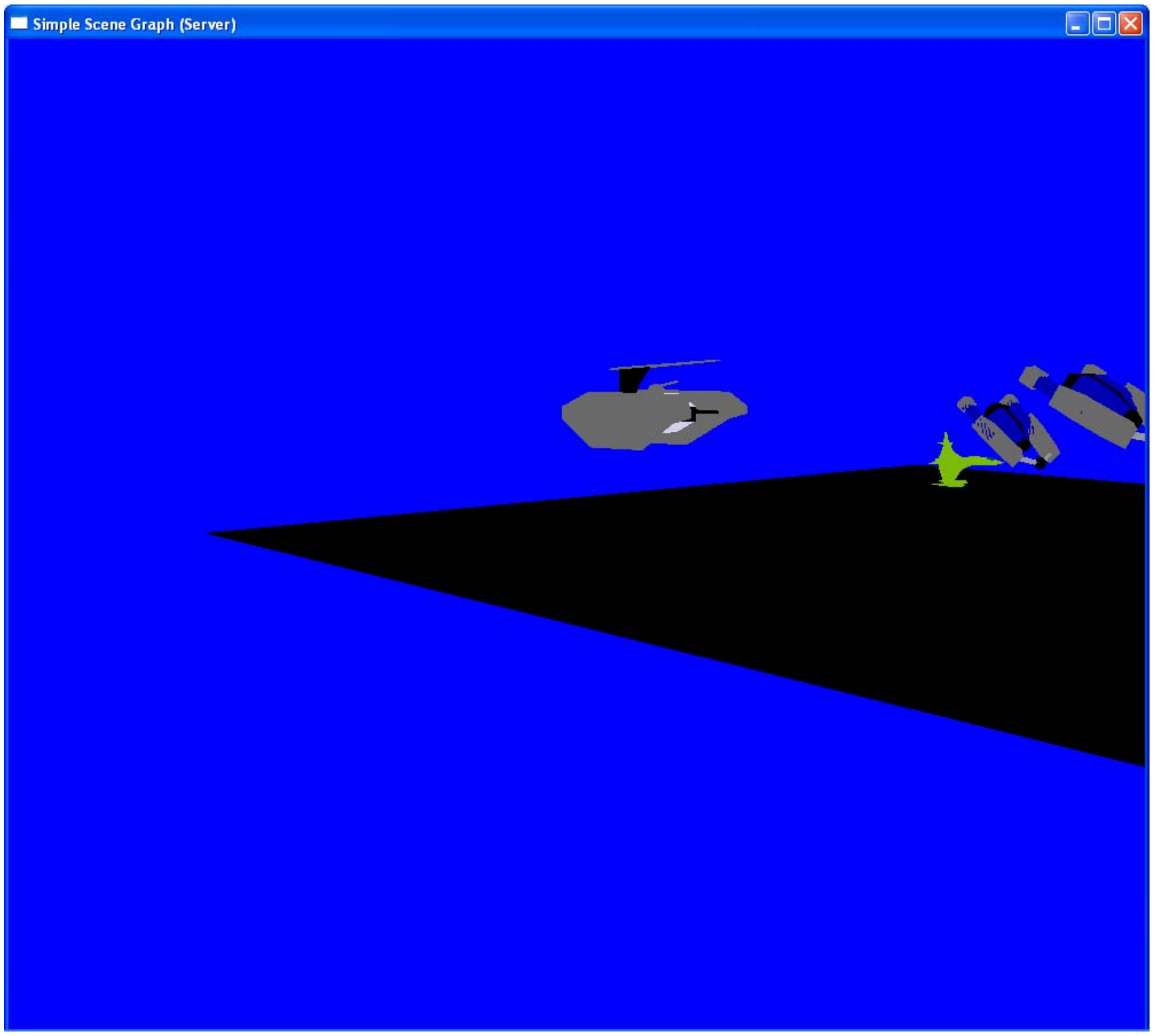

Figure 16 – Initial server scene state.

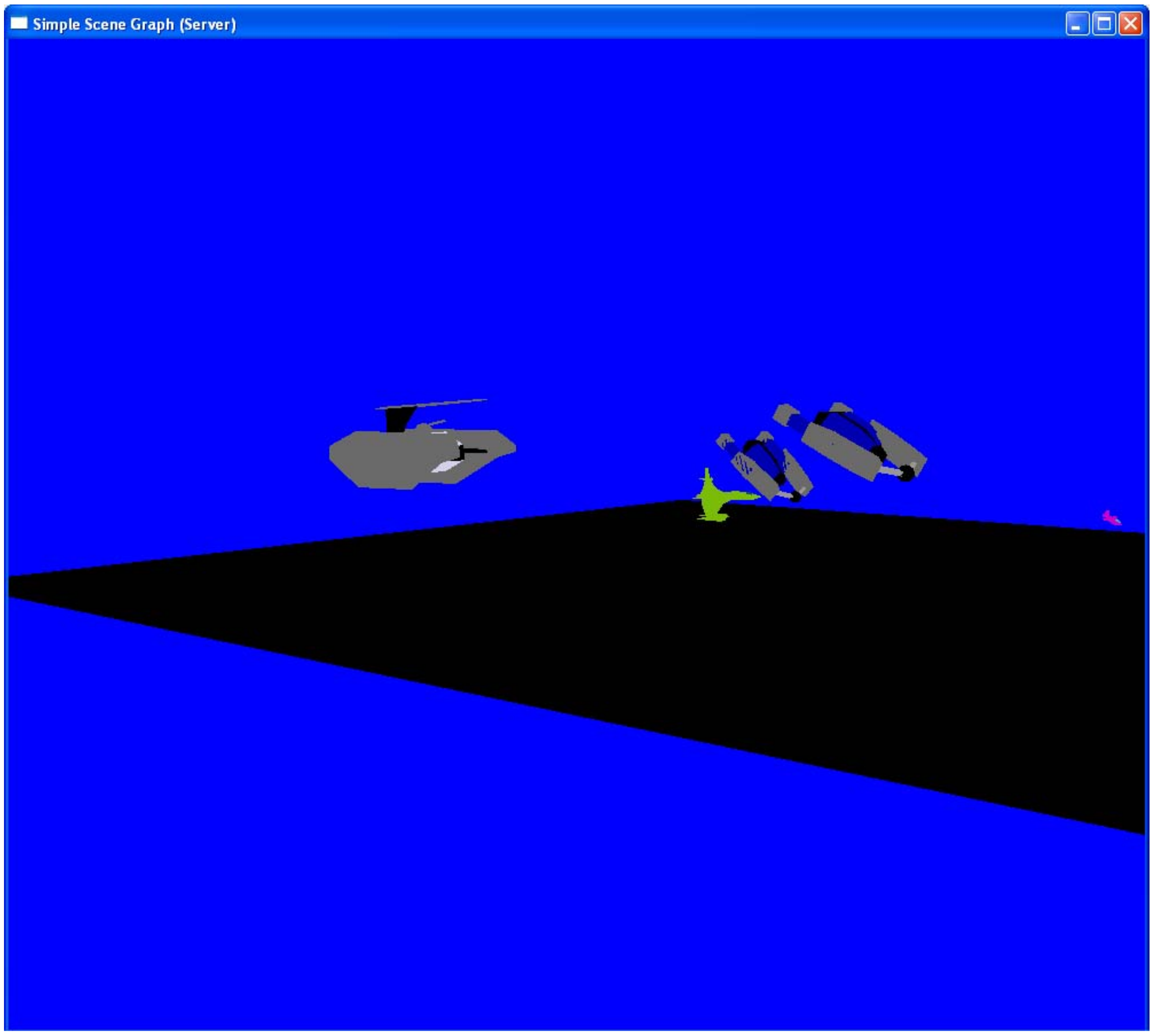

Figure 17 – Server scene state at client camera position.

6 Results

~~As a result, a~~ system design and implementation of an LSVE ~~was produced,~~ based upon the research of Teler and Lichinski [2001]. Beyond the benefits of the Teler and Lichinski design our system includes the utilization of standard multimedia communication protocols, provides a LRU cache for maintaining entities on the client and provides a scenegraph implementation that supports intermediate updates based on the clients viewing parameters.

Teler and Lichinski's implementation called for a proprietary communications protocol which was replaced with our RTP/RTSP communication framework. Utilizing RTP and RTSP, the design allowed the system to function over the internet without specific firewall and router configuration. RTP and RTSP are used today to stream audio and video over the internet, and leveraging this technology allowed us to piggyback on the already existing infrastructure, which minimizes the complexity of the system.

Teler and Lichinski's system did not address the possibility that the total scene may be to large for the client to maintain within its local resources. Developing an LSVE like other parallel programming problems requires a large number of processes to work on a common dataset (scenegraph) that is larger than single process could handle on its own. In order for a system to be a viable solution for massively multiplayer systems it must be able to handle this case. In this system the scenegraph management system was extended with a LRU cache for graphical resources. The client will ~~disposed~~ of resources (textures, sound files, models, portions of scenegraph, etc) in a least recently used manner, as it encounters its upper memory bound. This design worked well for clients with a reasonably sized cache (>2MB for our sample scene). If the cache was too small (less than 512KB for our sample scene) the client begins to thrash. It devoted most of its resources to selecting entities to remove and transmitting dispose messages to the server. This is a common limitation of caching algorithms and was expected.

The scenegraph maintenance algorithm was constructed on top of the RTP/RTSP framework and the LRU cache and minimized client/server communication, while maintaining a consistent global state. Teler and Lichinski proposed a scenegraph system that transmitted portions of the scene to the client as needed. Their design properly handled clients with slower connections. This system's design was extended to handle scenes which our larger than a client can maintain in its entirety.

This system implementation also had some limitations, some of which are present due to a simplified implementation, not of the overall design. The design required the client to process messages in temporal order from the server, which was not implemented at this time in order to reduce complexity. In this implementation, messages were processed in the order they were received, which may not have matched the actual temporal order. However, this was particularly evident when multiple clients were interacting in the scene. In the laboratory LAN the message order matched the temporal order with a single client, which allowed for this simplification. The design for maintaining global state called for transmitting updates to entities, only if a client's scenegraph had a reference to that entity. In this implementation all updates were sent to all clients, which significantly increased the message traffic and computation required at the server.

In order to evaluate the performance of this system a two pronged analysis is provided. Both the communication and computational overhead of the LSVE design ~~were~~ analyzed, and analysis to support a clustered server design is provided. The core of this algorithm involves receiving viewing updates from each client, traversing the scenegraph utilizing those parameters, and constructing an update message, which will be sent to the client at the end of the traversal. Figure 18 presents the cost (T_s) equation for the prototype implementation, which uses a single processor server. The presented equations were derived from standard formulations for analyzing sequential and parallel applications.

n = number of clients.
w = size of the viewing parameters.

$t_{latency}$ = the cost of setting of the socket connection.
t_{data} = the cost of sending one unit of data.

$t_{traversal}$ = cost of traversing the scenegraph.

$t_{comm} = t_{latency} + t_{data}$.

t_{comp} = computational cost.

$T_s = t_{comm} + t_{comp} =$
$T_s = n(t_{latency} + wt_{data}) + nt_{traversal}$

Figure 18 – Sequential cost analysis for LSVE server.

Note, that there is no cost in this equation for transmitting the scene data back to the client. Similar to a circuit switched network, a stream is dedicated to each client. ~~Increasing~~ performance could be achieved by parallelizing the server.

The equation for the single processor model shows that as n increases the communication and computational cost increase linearly. The cost can be reduced by distributing the work inside of a server cluster. If the server load is distributed over p processors, the processing of the client viewing updates can be parallelized. In addition the number of streams maintained at each server processor would be n/p instead of the original n. If the scenegraph updates to all processors in the server cluster additional costs are incurred, thus the maximum speedup of p cannot be achieved. The parallel form of the equations from Figure 18 is shown below in Figure 19. In Figure 19 a term (yt_{data}) is introduced to account for the cost of multicasting the scenegraph to the other processors in the server cluster. This architecture will provide a significant speed up over the single processor model when the number of clients and the size of the scenegraph are large. If this is not the case, the cost of the communication overhead will be greater than the savings achieved by distributing the computational component. This will occur if the ratio $t_{comp} / t_{comm} < 1$.

y = size in data units for the scenegraph.

$T = t_{comm} + t_{comp} =$
$T = n/p(t_{latency} + wt_{data}) + yt_{data} + (n/p)t_{traversal}$

Figure 19 – Parallel formulation of the equations from figure 18.

Based on this analysis, it is apparent that the server is a bottleneck in the design and it should be implemented as a cluster on a high bandwidth network in order to support the number clients required by today’s massively multiplayer online games. The future work section highlights the areas of improvement that could be researched and expanded on to make this a viable solution.

7 Contributions

This project presented a set of problems necessary to navigate and interact with LSVE. Our research contributions are in the areas related to distributed scenegraph management. Primarily we focused on researching and developing techniques for generating scenegraph updates on the server and merging those updates with the client's local scenegraph. This required algorithms and implementations for uniquely identifying all entities and resources in the scenegraph, traversing the scenegraph efficiently to determine the resources required at each client and a mechanism for constructing a subset of the scenegraph on the client. The client's local scenegraph is thought of as a cached subset of the server's scenegraph. Research was done to see how known caching algorithms for distributed systems may apply to this problem space, and an LRU-based algorithm was developed for the client side cache. Similar research has produced mechanisms for transmitting data to the client and merging it with the client's local scenegraph. Consistent with our research, we do not believe a viable solution has been found for dealing with limited resources on the client. Our algorithm addressed transmission and merging and also dealt with scene data disposal and retransmission, which we believe is necessary for interacting with LSVE. In addition, we developed prototype extensions to RTP and RTSP in order to facilitate the management and delivery of a 3D interactive world to a client. Previous research in this area produced proprietary protocols for managing the scene. By utilizing standard protocols for streaming multimedia any hardware or software optimizations developed for RTP and RTSP would also benefit this system. In addition RTP and RTSP are widely used throughout the industry for streaming audio and video. By extending this protocol we allow for bundling of video and audio streams with the 3D world content.

8 Future Work

In the future, we would expand the applications scenegraph to support more advanced features, such as additional lighting, collision detection, picking and other features necessary to facilitate additional scene interaction. In addition, the current implementation does not support interacting with objects in the scene, so future work is needed in this area to support the distributed state management required to support this feature. This implementation supported only a single user. Ideally, a multi-user implementation is needed to meet the demands of online applications.

REFERENCES

CHENG, L., BHUSHAN, A., PAJAROLA, R. and El ZARKI, M. 2004. Real-time 3D Graphics Streaming using MPEG-4. In *Proceedings of BroadWISE 2004*.

DEERING, M. 1995. Geometry Compression. In *Proceedings of ACM SIGGRAPH 1995*, ACM Press / ACM SIGGRAPH, New York, E. Fiume, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM, 13-20.

GREENHALGH, C. and BENFORD, S. 1995. MASSIVE: a Distributed Virtual Environment Reality System Incorporating Spatial Trading. In *Proceedings of DCS 1995*, 27-34.

HANDLEY, M. and JACOBSON, L. 1998. RFC 2327 - SDP: Session Description Protocol. Internet Society (IETF).

MACEDONIA, M., ZYDA, M., PRATT, D., BRUTZMAN, D. and BARHAM, P. 1995. Exploiting Reality with Multicast Groups: A Network Architecture for Large-scale Virtual Environments. In *Proceedings of VRAIS* 1995. 2-10.

MACEDONIA, M., ZYDA. 1997. A Taxonomy for Networked Virtual Environments, *IEEE MultiMedia 4*, 1, 48-56.

SAHM, J. and SOETEBIER, I. 2004. A Client-Server-Scenegraph for the Visualization of Large and Dynamic 3D Scenes, In *Journal of WSCG, 12*, 1-3.

SCHULZRINNE, H., CASNER, S., FREDERICK, R. and JACOBSON, V. 2003. RFC 3550 - RTP: A Transport Protocol for Real-Time Applications. Internet Society (IETF).

SCHULZRINNE, H., RAO, A. and LANPHIER, R.1998. RFC 2326 - Real Time Streaming Protocol (RTSP). Internet Society (IETF).

SWEENEY, T. "Unreal Networking Architecture". Retrieved March 1, 2005 from the World Wide Web: http://www.epicgames.com.

TELER, E. and LISCHINSKI, D. 2001. Streaming of Complex 3D Scenes for Remote Walkthroughs, *Computer Graphics Forum 20*, 3, 17-25.

The Open Group. 1997. Technical Standard: DCE 1.1: Remote Procedure Call.

WATSEN, K., DARKEN, R. and CAPPS, M. 1999. A Handheld Computer as an Interaction Device to a Virtual Environment. In *Proceedings of the International Projection Technologies Workshop.*

WATSEN, K. and ZYDA, M. 1998. Bamboo – A Portable System for Dynamically Extensible, Real-time Networked, Virtual Environments. In *Proceedings of VRAIS 1998*, 252-259.