Semotus Visum: A Flexible Remote Visualization Framework

Eric J. Luke* Charles D. Hansen†

Scientific Computing and Imaging Institute
School of Computing, University of Utah

ABSTRACT

By offering more detail and precision, large data sets can provide greater insights to researchers than small data sets. However, these data sets require greater computing resources to view and manage. Remote visualization techniques allow the use of computers that cannot be operated locally. The Semotus Visum framework applies a high-performance client-server paradigm to the problem. The framework utilizes both client and server resources via multiple rendering methods. Experimental results show the framework delivers high framerates and low latency across a wide range of data sets.

CR Categories: C.2.0 [Computer-Communication Networks]: General—Data communications; C.2.4 [Computer-Communication Networks]: Distributed Systems—Client/server;

Keywords: remote visualization, client/server

1 INTRODUCTION

Remote visualization is an increasingly important aspect of scientific computing. Advances in computing power enable researchers to utilize progressively larger and more refined data sets. Obtaining the insights offered by these data sets requires full use of modern supercomputers. These computers typically have many CPUs, a large amount of memory, increased I/O capability, and may have special graphics hardware. We assume that such hardware is typically available on a high-end server. Difficulty arises when the scientist or engineer who wants to visualize simulation data must use a computer that they cannot operate locally. Transferring the full data set to the researcher's desktop for visualization is prohibitively slow; furthermore, many desktop machines lack the memory and disk space to hold a multi-gigabyte data set.

Recent visualization research applies a client-server paradigm to the problem. In figure 1, we see that there are many different strategies for remote visualization. In the first scenario, the server renders images and streams them to the client. The second scenario has the server performing some rendering calculations, such as geometry transformation or visibility determination; the client finishes the rendering locally. Another possibility is scenario 3, where the server performs only the large-scale computations, and leaves the client to handle all the rendering computations. Finally, scenario 4 outlines the situation where the server only provides raw data to the client; not only does the client handle the visualization, but also performs the scientific computation.

Each of the above-mentioned scenarios has tradeoffs. For example, image streaming works well for thin clients, but can require

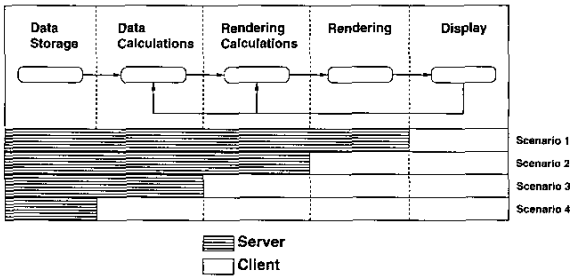

Figure 1: Dataflow in scientific visualization applications. In scenario 1, images are streamed from server to client. In scenario 2, part of the rendering calculations are done on the server. Scenario 3 allows the client to do all rendering calculations. Scenario 4 uses the server for data storage only; all calculations are done on the client.

significant network bandwidth. Performing some rendering calculations on the server can greatly speed up isosurface visualization [11]; however, low-end clients may not have the resources to finish the rendering in a timely manner. With these issues in mind, the value of a multifaceted approach becomes clearer.

We present a remote visualization framework that implements a number of rendering methods. The framework, dubbed "Semotus Visum" from the Latin for "remote visualization", encompasses a server middleware package, a set of communications protocols, and a Java client. By utilizing both client and server resources, the Semotus Visum framework can produce high framerates across a range of data sets.

In the following sections, we first discuss related work, then present an architectural overview of our framework. We then present experimental results demonstrating the framework's performance on three scientific data sets. Finally, we discuss our conclusions and give suggestions for future work.

2 RELATED WORK

A common approach to remote visualization is to use an X server to forward the local display. Such an approach requires no effort on the part of the programmer; a simple environment variable provides the interface. However, the protocol used by the server is not optimized for high-performance visualization; rendering N polygons remotely requires O(N) internal X protocol messages. This high overhead can severely curtail the performance of X-based applications.

A simple way to perform remote visualization is by image streaming (scenario 1, above). Engel et al. [5] used a high-end graphics server to stream images to a simple Java or C++-based client. In [6], Engel et al. created a Common Object Request Broker Architecture (CORBA)-based framework that allows a server to transmit images to a client. The server, running an Open Inven-

*luke@sci.utah.edu
†hansen@cs.utah.edu

IEEE Visualization 2002 Oct. 27 - Nov. 1, 2002, Boston, MA, USA
0-7803-7498-3/02/$17.00 © 2002 IEEE

tor or Cosmo3D application, sends a series of images to the Java client. These images can be transmitted raw, or compressed using Zlib [19], Lempel-Ziv-Oberhumer (LZO) [17], or run-length encoding (RLE).

Ma and Camp [13] also stream compressed images in their parallel volume rendering scheme. Unlike other implementations of this strategy, image compression is parallelized. Each processor begins compressing its subimage when it finishes rendering its volume partition. These subimages are also decompressed in parallel on the viewer. The speedup from parallel compression is nullified to a great extent by the poorer compression rates of the smaller images, as well as the overhead of compositing the subimages on the viewer. To improve performance, they suggest a hybrid approach of combining subimages before compression.

Another visualization option is an all-Java implementation (encompassing scenario 4). Hibbard et al. [10] converted their VisAD visualization system to Java. They utilize Java's Remote Method Invocation (RMI) system to provide access to remote data. Another pure Java approach was taken by Michaels et al. [16] in Vizwiz. Vizwiz was designed to be run as an applet, using a CGI script to upload the user's data. The lack of a usable 3D API caused significant rendering problems; further, Java's interpreted nature greatly limited its performance in calculations, especially those related to isosurface construction.

In [3], Engel et al. used VRML and Java to create a volume visualization system viewable by any web browser. This technique involves creating three stacks of texture planes in orthogonal orientations (ie, the XY, XZ, and YZ planes). When a user wishes to view the model, only the plane most perpendicular to the line of sight is rendered. In addition to the 3D view, the user can also view the orthogonal 2D slices independently. By selecting an isovalue on any of the 2D slices, a client-side module can compute and display a full isosurface, eliminating the need for further data transfers. This technique roughly corresponds to scenario 3, shown in figure 1.

An interesting approach to isosurface visualization was presented by Engel et al. [4]. By allowing the user to interactively control the level of detail, rough approximations to isosurfaces could be quickly viewed, with finer detail available upon command. Engel et al. [7] make efficient use of both local and remote resources in their isosurface extraction package. By dividing the computation load between server and client, they are able to transmit and reconstruct isosurfaces very quickly. In addition, they incorporate a focus point, allowing the user to increase the level of detail in a limited area for more precise viewing.

Multi-modal systems such as OpenDX [20] allow the user to choose between X-based rendering and local OpenGL to maximize rendering performance. More intelligent packages such as ARTE [15] analyze bandwidth and client capabilities to automatically select the most appropriate mode of transmission.

Not all remote visualization packages focus solely on software. In their Visapult [1] visualization framework, Bethel and Tierney et al. combine software with massively parallel hardware. The hardware component, the Distributed Parallel Storage System (DPSS) [2], is built of commodity components linked with custom software.

Friesen and Tarman [8] describe a hardware-assisted system in use at Sandia National Laboratories. This system uses arrays of hardware scan converters to convert computer-generated RGB images to NTSC format. The NTSC video is then fed into an array of hardware image compressors; the compressed output is transmitted along a shared ATM network to the viewer, where similar hardware decompresses the stream. By pushing the remote visualization components from the server into hardware, the researchers were able to sustain good quality 1280 x 1024 frames at 30 Hz.

3 The Semotus Visum Framework

Effective remote visualization frameworks must address a number of system issues. Due to its distributed nature, a framework must be portable; the choice of development language, graphics API, network access, and thread implementation can all affect portability. As will be shown shortly, a monolithic rendering scheme cannot guarantee good performance in all conditions. Thus, a sensible rendering framework should allow renderers to be placed at different points in the rendering pipeline given in figure 1. More explicit details for the algorithms described in this section can be found in [12].

Collaboration is an increasingly important part of visualization. Collaborative tools are useful in a remote visualization system because data features can be subtle and difficult to verbalize. Integrated tools can aid in communicating these features to peers. Persistent annotations can also serve as a reminder for researchers when they revisit older data sets.

Remote user interfaces, though not needed for batch systems, are vital to interactive systems. The difficulty lies in providing an interface that is not specific to a single application, but is still useful across multiple applications. Remote users need to access the server's user interface, preferably in its original form. Furthermore, a remote user interface needs to show interconnection data (for dataflow systems), and provide some method for choosing a subinterface from multiple options.

3.1 Client Architecture

Based on the issues discussed above, the Semotus Visum client was implemented in Java. Java offers the platform independence of interpreted languages, along with performance approaching that of compiled languages. Virtual machines can be obtained for nearly every modern computing platform. Development kits also come with a large set of support and development libraries, speeding development time. Java applications can also utilize native code for speed-critical routines using the Java Native Interface (JNI).

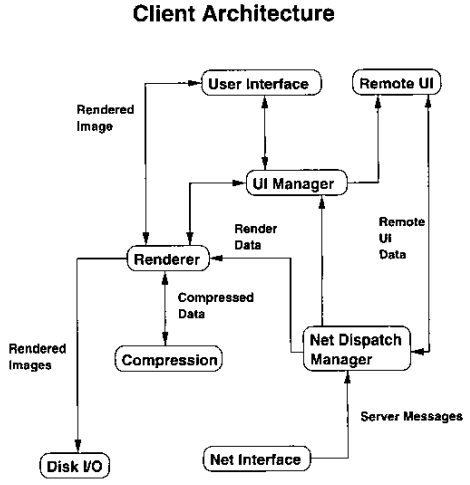

Figure 2: Detailed client architecture.

The client architecture, shown in figure 2, is highly modular. This modularity allows clean communication between unrelated sections of the client, quick additions or modifications to modules, and independent development and testing. The client abstracts the network into a Network Interface layer. This layer uses standard Java sockets to transmit and receive data from the network. There

are also utilities to handle byte order conversion. In addition to a generalized socket interface, Java also provides threading and synchronization primitives. Unrelated modules are decoupled from each other by running in separate threads. Within a given module, high- and low-performance sections are also decoupled into separate threads. This helps to ensure maximum throughput and performance.

To facilitate communication with the server, the client uses a Message abstraction for received and transmitted data. There are several message classes, each of which implements a particular communication protocol. Incoming message requests are registered with a callback dispatch manager; the manager calls a callback function when the message arrives. Outbound messages are sent directly to the network interface.

Renderers share a common interface; this common interface facilitates the addition of new renderers. Network data is sent to the renderer from the callback manager. If the data is compressed, the renderer sends it to a decompression module which decompresses the data, and returns it to the renderer. After performing any needed operations on the data, the renderer sends a rendered image to the user interface. Optionally, the rendered image can also be written to disk for permanent storage.

To protect the client's internals from user interface changes, the client includes a user interface manager. Aside from the manager, only renderers, in the interest of efficiency, have access to the user interface. The manager also interacts with the remote user interface, though this interaction is largely independent from the rest of the client.

3.2 Server Architecture

The Semotus Visum server is implemented in C++ as a middleware software package. In this case, "server" refers to part of the application process. Multiple servers may run on the same machine, and multiple clients may interact with a single server. The design, shown in figure 3, is focused as a general-purpose remote visualization solution rather than restricted to a single scientific computing package. Because the server interfaces between the application and multiple clients, it should not impose significant processing overhead. Thus, the package gives high priority to performance and scalability.

Like the client, the server abstracts network access into a Network Interface layer. This layer utilizes a wrapper for standard Berkeley sockets, both connection-oriented and connectionless. The same wrapper also offers host and network byte-order conversion. In addition to the socket interface, the server also uses threading and synchronization wrappers; these wrappers abstract away the underlying thread and synchronization primitives. Unrelated modules are decoupled from each other by running in separate threads. Within a given module, high- and low-performance sections are also decoupled into separate threads. This helps to ensure maximum throughput and performance.

Messages are also used on the server for transmitted and received data. These abstractions maintain a one-to-one mapping between message types and communication protocols. Incoming client messages are registered with a callback dispatch manager; the callback manager can either call a callback function or send a synchronization message when data arrives. Outbound messages are sent directly to the network interface.

Rendering support is provided through a consistent interface across renderers. The application provides the renderer with the raw visualization data, and the renderer does any specialized processing. After processing, the data is sent to a compression module; the compressed data is forwarded to the network interface for transmission to clients.

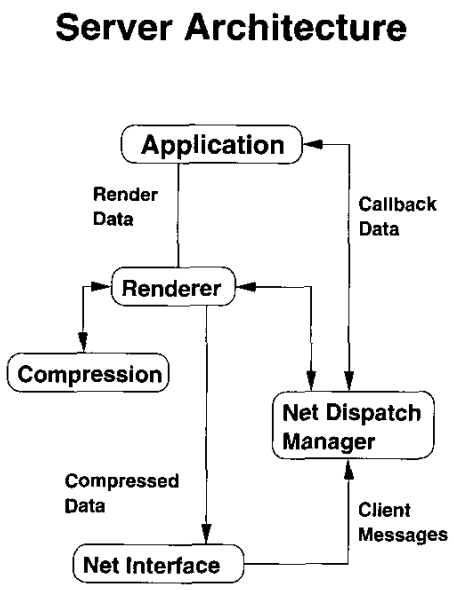

Figure 3: Detailed server architecture.

3.3 Communication

Communication between client and server utilizes the eXtensible Markup Language (XML). XML was chosen because it is a well-documented protocol for information exchange. Well-supported, XML can be created and parsed on virtually any platform in many programming languages. Its human-readable form is useful in both debugging and offline analysis of communication information.

Client-server messages fall into a number of categories: rendering, collaboration, and miscellaneous. The client uses rendering messages to request rendering parameters from the server; these messages also supply the server with any needed local information. The server also makes use of rendering messages by including an XML header for binary viewing data. These headers typically comprise a very small percentage of the total data transmitted, on the order of less than 0.1%. Collaboration messages allow the client and server to exchange collaboration data such as text, pointers, and drawings. Other messages provide a framework for an initial client-server handshake, multicast and compression parameters, and remote user interface data.

3.4 Rendering

Image streaming, shown in figure 4, uses the server to provide most of the rendering power. This visualization method is simple to implement; the server need only access a rendered frame, and the client need only draw these pixels to the screen. Such simplicity allows a client to support this type of rendering with minimal local resources. Furthermore, the server's internal data representation is hidden from the client. The server is thus free to use any available methods (such as ray tracing, volume rendering, or polygon rasterization) to render the data. The Semotus Visum client uses Java's *BufferedImage* class to hold streamed images prior to rasterization.

Geometry rendering, shown in figure 5, uses the client to provide most of the rendering power. Unlike image streaming, this scheme takes advantage of client rendering resources. With the growing power of 3D graphics accelerators, desktop machines can render large numbers of polygons at interactive rates. Network resources are involved only in the initial setup; after the geometry is trans-

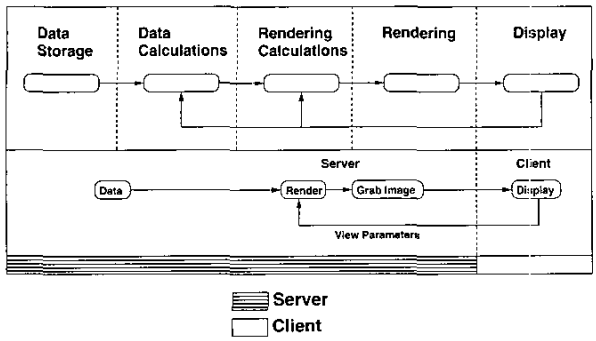

Figure 4: Image streaming. The server does all the calculations and rendering; the client simply displays the rendered image. Viewing parameters are relayed back to the server.

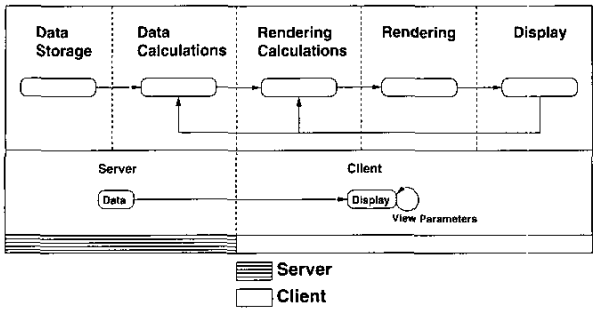

Figure 5: Geometry rendering. Geometric data is transmitted to the client. Any changes in viewing parameters are implemented locally on the client.

mitted, the client need not access the network again. The Semotus Visum client uses the Java3D API to render geometry.

ZTex rendering, shown in figure 6, divides the rendering load between client and server. This method of rendering builds on image-based rendering techniques and the relief texture mapping ideas discussed in [18, 14]. We first observe that a scene rendered from a given viewpoint can be thought of as a dense mesh (represented by the depth buffer) and as associated texture (represented by the color buffer). If the dense mesh is simplified and the color buffer appropriately applied as a texture map, from the identical viewpoint the framebuffer results would be unchanged. Having such a representation allows the client a small amount of 3D interaction, albeit with significant error from oblique views.

The ZTex renderer is an implementation of these ideas. After rendering the data from a particular view, the server reads the color and depth buffers. The color buffer is sent to the client in the same manner as in image streaming. The server creates a triangle mesh from the depth buffer height field using a modified Garland-Heckbert algorithm [9]. This mesh encompasses only the visible rendered data; depth values of infinity, or polygons removed by the Z buffer are not present in the mesh. The server then sends this mesh to the client as it would in the geometry rendering scheme. After receiving the viewing data, the client calculates texture coordinates for the mesh vertices using local viewing parameters. These texture coordinates correspond to the appropriate locations in the image texture. The client subsequently renders the mesh using the image as a texture. Like the Geometry renderer, the client ZTex renderer uses the Java3D API to render the images.

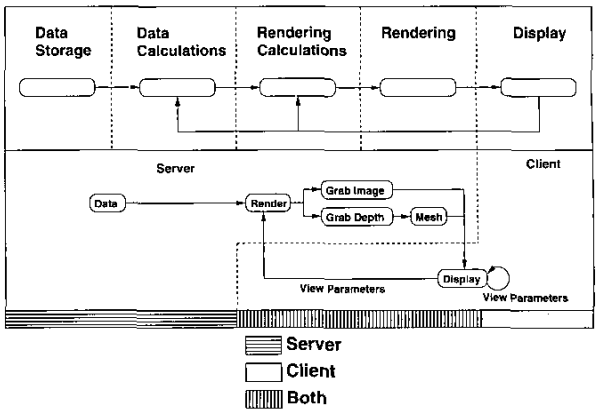

Figure 6: ZTex rendering. Geometric data is rendered on the server. The server generates a mesh from a simplified height field read from the depth buffer; the resulting mesh and rendered image are transmitted to the client. The client can alter the viewing parameters locally, or request another ZTex object with the current local viewing info.

4 Results

The Semotus Visum framework was tested using a number of scientific data sets. The Aneurysm data set is a 250x250x125 magnetic resonance angiogram of the vasculature of a patient's head. The Jet Shockwave data set contains 256^3 cells. This data set simulates the Kelvin-Helmholz instability in a supersonic jet. The final data set is derived from the Visible Male frozen CT. The raw Visible Male data were scaled and padded to a uniform 369x475x254. From this data, we derived a bone isosurface using a standard Marching Cubes isosurface generator. This dataset is labeled 'VisBone'. All the data sets use rectilinear grids to implicitly denote positions of the data points in space, and all isosurfaces were generated prior to performance testing. Table 1 gives the size of each dataset in memory footprint and polygons. Representative images from each data set can be found in figures 7-9.

Data Set	Size (MB)	Polygons
Jet	67.1	148802
Aneurysm	31.3	502936
VisBone	178.1	2458966

Table 1: Data sets range in size from a thirty to several hundred megabytes. The polygon count ranges from one hundred fifty thousand to two and a half million. Note that even though the Jet data set has a greater memory footprint than the Aneurysm, the specific isovalues chosen for each dataset lead to inversely proportioned polygon counts.

All the experiments used an SGI Origin 2000 (32 250MHz R10000 CPUs, 12 GB of memory) as the server machine. The primary client in many of the tests is a dual 550MHz Pentium III with 512 MB of memory running a 2.2.17 Linux kernel. This client also has an NVIDIA Quadro graphics card. Because it consistently outperformed other similar virtual machines in industry benchmarks and preliminary testing, the Linux client utilized the IBM 1.3.9 JVM. The Linux client was tested using 10, 100, and 1000 Mbit ethernet connections. These connections will be labeled 'Linux10', 'Linux100' and 'Linux1000', respectively. All tests were run on a local network, with no more than one hop between server and client.

Results in this chapter represent three types of experiments. The first tests measure the global throughput of the system; this represents the maximum frame rates that the system can produce. As much of the framework is multithreaded, we measure throughput as a function of the slowest stage in the rendering pipeline. However, frame rate alone does not give the complete picture of a rendering system. The latency between client request and display of the corresponding data also affects the perceived interactivity of a system. Thus, we also conducted tests to measure the latency of different rendering requests. Though not directly affecting the user's perception of interactivity, the bandwidth requirements for each data set and viewing method were also measured.

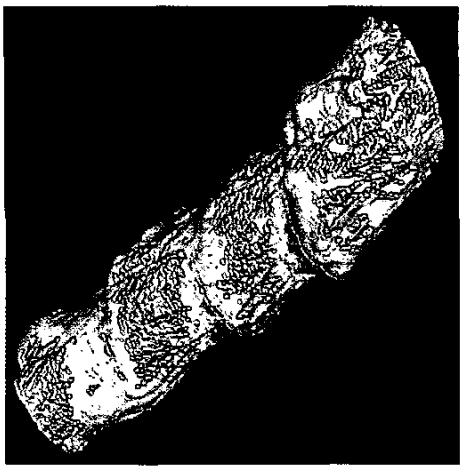

Figure 7: Jet data set.

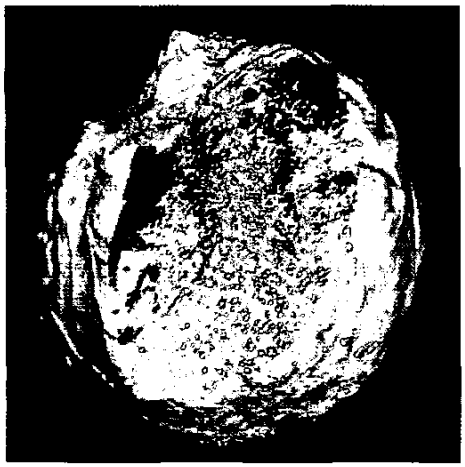

Figure 8: Aneurysm data set.

4.1 Frame Rates

The basis for all framerate comparisons is the server's local rendering rate. Framerates were measured on the server with no remote services running; this will be referred to as the 'Local' framerate. As the simplest way to do remote visualization is to use the X Windows protocol [21], framerates achieved while using this protocol

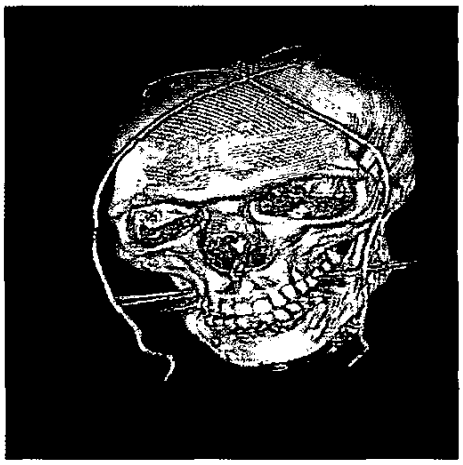

Figure 9: Visible Human Bone data set.

were also measured. To perform this measurement, we set the display of the server to the local machine, and rendered images as normal. Frame rates measured using this technique will be referred to as the 'X' frame rates.

Frame rates for image streaming are shown for the case when the server compresses the images using LZO. Further, to reduce decompression and byte-order conversion time on the client, the client utilizes the Java Native Interface (JNI). The proportion of native code used in the client is very small; it represents less than 2% of the entire code base. Geometry and ZTex renderers convert the geometry to triangle strips before rendering. This can improve performance by as much as 50%.

Tables 2-4 give the frame rates for the Jet, Aneurysm, and VisBone data sets. Severely limited by network bandwidth, the 10 Mbit X frame rates are consistently lower than those of the 100 and 1000 Mbit clients. Image streaming matches the performance of the server; the local rendering rate is the bottleneck in this case. Interestingly, compression reduces the bandwidth needs of the image streaming clients enough that the 10 Mbit client can offer the same performance as the other clients. Geometry rendering consistently offers twice the local rendering rate. However, the VisBone data set cannot be rendered due to size restrictions.

ZTex rendering provides interactive frame rates for each of the data sets. On the Jet and Aneurysm data sets, ZTex renders at 10-20 times the local rate; on the VisBone data set, this speedup improves to 50-120. These frame rates correspond to speedups of 50-700 over X-based visualization.

4.2 Latency

Though frame rate is an important measure of renderer performance, latency cannot be ignored. In many cases, interactivity requires the user to see the results of viewing changes without a sizeable delay. We define the latency of an X request as the inverse of the framerate - ie, we ignore the time required for the client to request an image from the server. This is a reasonable assumption, as such requests require negligible processing and network time. To keep comparisons equitable, we also ignore the small Semotus Visum request time.

Though each stage in the framework's rendering process runs in parallel, our latency measurements must include the full time a request takes from start to finish. Thus, image streaming latency is the sum of server rendering and processing time, network transmission

Jet Dataset			
Local Frame Rate	4.3 fps		
10 Mbit X Frame Rate	0.08 fps		
100 Mbit X Frame Rate	0.6 fps		
1000 Mbit X Frame Rate	0.8 fps		
Renderer	FPS	Local Speedup	X Speedup
Linux10 Image	4.3	1.0	53.75
Linux100 Image	4.3	1.0	7.16
Linux1000 Image	4.3	1.0	5.38
Linux Geom	9.5	2.2	15.7
Linux ZTex	78 / 45	18.1 / 11.3	130 / 80.8

Table 2: Basis results for the Jet dataset. ZTex results are for 3/6 concurrently rendered ZTex objects.

Aneurysm Dataset			
Local Frame Rate	1.6 fps		
10 Mbit X Frame Rate	0.02 fps		
100 Mbit X Frame Rate	0.3 fps		
1000 Mbit X Frame Rate	0.3 fps		
Rendering Method	FPS	Local Speedup	X Speedup
Linux10 Image	1.6	1.0	80
Linux100 Image	1.6	1.0	5.3
Linux1000 Image	1.6	1.0	5.3
Linux Geom	2.8	1.75	9.4
Linux ZTex	27.4/15.5	17.1 / 9.7	91.3 / 51.7

Table 3: Basis results for the Aneurysm dataset. ZTex results are for 3/6 concurrently rendered ZTex objects.

VisHuman Bone Dataset			
Local Frame Rate	0.3 fps		
10 Mbit X Frame Rate	0.002 fps		
100 Mbit X Frame Rate	0.05 fps		
1000 Mbit X Frame Rate	0.06 fps		
Rendering Method	FPS	Local Speedup	X Speedup
Linux10 Image	0.3	1.0	150
Linux100 Image	0.3	1.0	6.0
Linux1000 Image	0.3	1.0	5.0
Linux ZTex	35.4/15.5	.118 / 51.7	708 / 310

Table 4: Basis results for the Visible Human Bone dataset. Client could not render geometry due to size constraints. ZTex results are for 3/6 concurrently rendered ZTex objects.

time, client processing time, and client rendering time. Geometry latency is the sum of the server data marshalling and processing time, network transmission time, and client processing time. ZTex request latency is the sum of server rendering and processing time, network transmission time, and client processing time.

Table 5 shows the latency of the X protocol and each Semotus Visum rendering method. The X protocol's latency increases linearly as data set size increases. Though their latency also increases with data set size, the image renderers are dominated by the server's rendering rate. As it is image-based, ZTex latency is also directly affected by the server's rendering time. In addition, the overhead of transmitting ZTex objects increases the overall latency. The renderer with the worst latency performance is geometry rendering. As data sets increase in size, the latency of the geometry renderer increases exponentially. This is due to nonlinear effects of handling large data sets: networks approach saturation, caches are overwhelmed, and the client must start swapping the data to disk.

Latency Per Dataset (seconds)			
Rendering Method	Jet	Aneurysm	VisBone
Linux X	1.6	3.3	20
Linux10 Image	1.45	1.9	4.6
Linux100 Image	0.42	0.83	3.5
Linux1000 Image	0.44	0.84	3.5
Linux10 Geom	3.3	12.7	131
Linux100 Geom	2.7	9.9	122
Linux1000 Geom	2.6	9.4	120
Linux10 ZTex	3.5	6.6	9.1
Linux100 ZTex	2.2	4.8	7.8
Linux1000 ZTex	2.2	4.7	7.7

Table 5: Latency for all datasets.

4.3 Bandwidth

Measuring the bandwidth requirements for the Semotus Visum rendering methods is mostly straightforward. The required bandwidth for geometry and ZTex is simply the data transferred per request divided by the network time. Image streaming is more complex; this rendering method is usually constrained by the client or server. So, though the peak bandwidth requirements will be similar to other rendering methods, the sustained bandwidth requirements are calculated using this formula:

$$Sustained = (ActualFPS/MaxNetworkFPS) * Peak.$$

In other words, the sustained rate is the fraction of time spent transmitting data times the peak rate.

The bandwidth used by the X server is not easily measured directly. Thus, results for X-based visualization are derived from inspection of the source code and X protocol definitions. The X protocol transfers roughly 28 bytes of information per double-precision vertex. The data required for each frame is therefore approximately the number of vertices times the 28-byte packet size. The results of these bandwidth measurements and estimations are given in table 6.

X requires 10.1 MB/sec on average, or roughly 81% of the available bandwidth on a 100 Mbit network. It is this high bandwidth requirement that causes X to perform so poorly on a 10 Mbit/s network. Though LZO compression can reduce the size of an image stream by up to 90%, the high server frame rates for the Jet and Aneurysm data sets cause image renderers to consume bandwidth on the order of X. As the server rendering rate becomes the bottleneck, the peak bandwidth demand subsides. The sustained bandwidth requirements of these image streams are a few hundred kilobytes per second, or only 1.5% of the bandwidth of the X-based

Bandwidth Per Dataset (MB/second)			
Rendering Method	Jet	Aneurysm	VisBone
X	7.5	12.5	10.3
Image (Peak)	9.5	10	3.9
Image (Sustained)	0.2	0.2	0.03
Geometry	10.3	10	10.7
ZTex	3.7	4.2	7.3

Table 6: Bandwidth requirements for all datasets. Tests were run on a 100 Mbit/s network.

approach. Geometry rendering transmits the entire data set in a single burst; therefore, even compressed geometry streams still require roughly 10 MB/second of bandwidth. As ZTex objects are partially composed of compressible images, they require less bandwidth than geometry. However, as data set complexity increases, the ZTex objects will be contain a greater proportion of geometry. Thus, the total compressibility of the object decreases, and bandwidth needs increase.

5 Conclusions and Future Work

We have presented a remote visualization framework that offers high frame rates and low latency on a variety of data sets. The Semotus Visum framework supports multiple rendering methods, each of which exploits resources found in a stage of the rendering pipeline. This approach delivers significant speedups over previous methods. By trading latency for throughput, the framework can outperform the server's local rendering rate even for large data sets. Though image streaming makes heavy use of network resources, compression algorithms like LZO can liberate this method from bandwidth considerations.

Future work on the Semotus Visum framework encompasses several directions. First, the framework would benefit from extensive wide-area network testing, as well as multi-client tests conducted on a larger scale. This testing could identify scalability issues in the server and communication scheme. The current shared-memory server design could be extended to include distributed and cluster machines. Further, the framework could be extended to utilize multiple independent servers. Peer-to-peer client interaction, such as sharing rendering or collaboration information, could reduce the resource demand on the server.

The three rendering methods included in the framework only encompass a small fraction of the imaginable rendering schemes. For example, variants of the geometry renderer could implement the isosurface extraction schemes discussed by Engel et al. [7]; the geometry renderer could also be extended to support geometry compression. New rendering methods could accommodate those supercomputers that do not possess specialized graphics hardware. The framework could also be augmented with an automated heuristic which recommends a rendering method based on server resources, network bandwidth, client power, and user needs.

6 Acknowledgments

The authors would like to acknowledge Peter-Pike Sloan, David Hart, and David McAllister for their work on the ZTex rendering scheme. This work was supported by the DOE Corridor One project and the DOE Advanced Visualization Technology Center (AVTC). The Jet data set was obtained from the Advanced Visualization Technology Center data repository at Argonne National Labs. The Visible Human data set was obtained from the National Library of Medicine's Visible Human project. The Aneurysm data set was provided by the University of Utah's Medical Imaging and Research Laboratory (MIRL), and Department of Radiology.

References

[1] "W. Bethel, B. Tierney, J. Lee, D. Gunter, and S. Lau". Using High-Speed WANs and Network Data Caches to Enable Remote and Distributed Visualization. In *"Supercomputing"*. IEEE, November 2000.

[2] Distributed Parallel Storage System. "http://www-didc.lbl.gov/DPSS".

[3] K. Engel and T. Ertl. Texture-based Volume Visualization for Multiple Users on the World Wide Web. In Gervautz, M. and Hildebrand, A. and Schmalstieg, D., editor, *Virtual Environments '99*, pages 115–124. Eurographics, Springer, 1999.

[4] K. Engel, R. Grosso, and T. Ertl. Progressive Iso-Surfaces on the Web. In *Late Breaking Hot Topics*. IEEE Visualization, 1998.

[5] K. Engel, O. Sommer, C. Ernst, and T. Ertl. Remote 3D Visualization using Image-Streaming Techniques. In *Advances in Intelligent Computing and Multimedia Systems (ISIMADE '99)*, pages 91–96, 1999.

[6] K. Engel, O. Sommer, and T. Ertl. An Interactive Hardware Accelerated Remote 3D-Visualization Framework. In *Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym '00*, May 2000.

[7] K. Engel, R. Westermann, and T. Ertl. Isosurface Extraction Techniques for Web-based Volume Visualization. In *Proc. Visualization '99*, pages 139–146. IEEE, 1999.

[8] J. Friesen and T. Tarman. Remote High-Performance Visualization and Collaboration, 2000.

[9] M. Garland and P. Heckbert. Fast Triangular Approximation of Terrains and Height Fields.

[10] W. Hibbard, J. Anderson, and B. Paul. A Java and World Wide Web Implementation of VisAD. In *Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology*, pages 174–177, 1997.

[11] Y. Livnat and C. Hansen. View Dependent Isosurface Extraction. In *IEEE Visualization '98*, pages 175--181. IEEE, 1998.

[12] Eric J. Luke. Semotus Visum: A Flexible Remote Visualization Framework. Master's thesis, School of Computing, University of Utah, January 2002.

[13] Kwan-Liu Ma and David M. Camp. High Performance Visualization of Time-Varying Volume Data over a Wide-Area Network Status. In *Supercomputing*, 2000.

[14] William R. Mark, Leonard McMillan, and Gary Bis hop. Post-rendering 3d warping. In *1997 Symposium on Interactive 3D Graphics*, pages 7–16. ACM SIGGRAPH, April 1997. ISBN 0-89791-884-3.

[15] I.M. Martin. ARTE - An Adaptive Rendering and Transmission for 3D Graphics. In *ACM Multimedia 2000*, November 2000.

[16] C. Michaels and M. Bailey. Vizwiz: A Java Applet for Interactive 3D Scientific Visualization on the Web. In *Proc. Visualization '97*, pages 261–267. IEEE Computer Society Press, 1997.

[17] M.F.X.J Oberhumer. Lempel-Ziv-Oberhumer Data Compression Library. http://wildsau.idv.uni-linz.ac.at/mfx/lzo.html.

[18] M. Oliveira, G. Bishop, and D. McAllister. Relief Texture Mapping. In *Computer Graphics Proceedings*. ACM SIGGRAPH, 2000.

[19] G. Roelofs. Zlib. http://www.cdrom.com/pub/infozip/zlib/.

[20] Visualization and Imagery Solutions. OpenDX. http://www.opendx.org.

[21] The X Windows System. http://www.x.org.