A Parallel Remote Visualization Server for Clusters

Matthew Everett Ian Buck Milton Chen Greg Humphreys Pat Hanrahan *

Stanford University

Abstract

Recent research has shown that scalable, high-performance graphics systems may be built from commodity graphics hardware running on clusters. The rendered images are then displayed on nodes directly connected to the local system area network. However, many application scenarios require remote display across low speed networks. In this paper we describe a remote, image-based, display system for a central, scalable visualization server. The remote display system involves two components: an image merging stage that assembles tiles from a logical framebuffer distributed across many nodes, and a compression subsystem that compresses the final image before transmitting it over a lower speed network. In our system the number of rendering and compression nodes may be independently varied to meet the demands of the application and the constraints of the compression and networking technologies. We leverage existing multimedia libraries, in particular, DirectShow, which allows 3D graphics to be integrated into a wide range of Internet streaming capabilities, such as video filters and compression algorithms. Using current technology, we demonstrate interactive, low-latency rendering and display of 1024 by 768 images across a 100Mbit network at approximately 13 Hz.

CR Categories: I.3.2 [Computer Graphics]: Graphics Systems—Distributed/network graphics; I.3.4 [Computer Graphics]: Graphics Utilities—Software support, Virtual device interfaces; C.2.2 [Computer-Communication Networks]: Network Protocols—Applications; C.2.4 [Computer-Communication Networks]: Distributed Systems—Client/Server, Distributed Applications

Keywords: Scalable Rendering, Cluster Rendering, Parallel Rendering, Tiled Displays, Remote Graphics, Virtual Graphics

1 Introduction

Graphics resources have typically been tightly coupled to a host CPU and display. This view of graphics is very restrictive; it imposes limits on resolution, distance, and usage models. Most importantly, it implies that the rendering power is *not shared*. Graphics is fairly unique in this regard; most other major components of a computer, including the CPU, disk, and memory, can be easily shared by other computers on the network, often transparently to the application.

*{meverett|ianbuck|miltchen|humper|hanrahan}@graphics.stanford.edu

Ideally, it would be convenient to fill a machine room with "visualization servers" that could be easily shared and partitioned among a group of users. The collective processing and rendering resources present in these servers would be aggregated in a way that is convenient both to the programmer and to the user. This would allow users to access applications and datasets that cannot reside on their local machines. To achieve this, rendering resources must be decoupled from the eventual display and also from the CPU running the application. In addition, any system that provides rendering resources as a remote service should be sufficiently flexible to meet the needs of a variety of applications, be they compute-bound or graphics-bound.

Resent research in scalable, cluster-based graphics systems has brought these goals closer to reality. One system developed at Stanford called "WireGL" is a system for scalable distributed rendering on clusters of workstations[5]. WireGL is capable of rendering over 100 million triangles per second on a 32-node cluster. In distributing rendering work across multiple graphics accelerators, WireGL creates a logical framebuffer that is physically fragmented across the cluster nodes. Each rendering node contains one or more disjoint tiled regions of the framebuffer. This distributed framebuffer is necessary to achieve scalable rendering rates for parallel graphics applications, but it must be reassembled to produce a final display for viewing. This reassembly can be performed in hardware (e.g. the DVI-based Lightning-2 image composition network [16]) or through high-speed cluster interconnects. However, these solutions are expensive and bandwidth intensive.

In order to display WireGL's distributed framebuffer, we have designed and implemented a remote display system. The remote display system generates a stream of compressed images over a network from the cluster's framebuffer tiles. This stream of images can be decompressed and viewed remotely while user input is relayed back to the running application. Although the streaming image approach to remote rendering has been considered before, our system has a few features that make it unique.

First, unlike previous approaches to remote visualization, the remote display system is designed to work with scalable, graphics clusters, in particular, WireGL. Besides providing scalability, WireGL also allows great flexibility in the allocation of compute and graphics resources.

Second, the remote display system parallelizes image compression across multiple nodes. This technique allows the server to achieve much higher resolutions and frame rates than what is possible with a single node. Furthermore, parallelism provides flexibility to add or remove compute power in order to match the system's resource requirements or constraints.

Finally, the remote display system is built from commodity parts, software, and protocols. We use Microsoft DirectShow filters to create standardized compressed image streams, including industry standards such as MPEG-4 or H-263, or proprietary streams such as Intel's Indeo. By converting the output of our visualization server to a standard multimedia stream, we connect two relatively independent worlds, 3D graphics and streaming multimedia; we also leverage a great deal of compression and networking infrastructure. The DirectShow infrastructure also makes it easy to modify the image stream with custom or 3rd party multimedia filters. By using rapidly evolving technology found in commodity parts, we are able

to focus our efforts only on the portions of the visualization server that are unique to our view of 3D graphics as a scalable remote resource.

2 Background and Related Work

Rendering at a distance has been an important research topic for several years. X Windows[11] and GLX[7] provide a solution for rendering 2D and 3D data over a network. In both cases, remote rendering is accomplished primarily by sending *commands* to generate an image, rather than sending the image itself. This trade-off assumes either that the commands to generate the image will occupy less space than the image or that the server rendering the final image has more powerful rendering resources than the client.

Other solutions target image-based transmission rather than commands such as Sun Ray[14] and VNC[12]. These systems require the server to perform all of the rendering while the client processes a stream of image updates from a network. This allows for large servers with relatively thin client capabilities. However these solutions are typically not designed with high-performance 3D graphics in mind.

The GLR[8] system provides image-based transmission for OpenGL applications. This system renders OpenGL commands on a high-end rendering server and transmits the resulting images to any X server. While this provides similar functionality to our visualization server, it does not support distributed framebuffers or compressed image transmission.

The work done by Samanta et al.[13] on parallel rendering system on PC clusters explores image compositing over a high-speed network. A sub-portion of the frame is rendered by each node, and the frames are then transmitted to a simple display node for compositing. The limitation is that the display node must be on the fast cluster interconnect, which requires the end user to be located near the cluster.

SGI's OpenGL VizServer[15] is a software solution for remote visualization that generates video streams from an Onyx2 server framebuffer to a desktop display. The video streams are compressed with Color Cell compression and Interpolated Color Cell compression achieving compression rates ranging from 8:1 to 4:1. This system provides seamless OpenGL remote rendering from a powerful shared resource. However, the scalability of the system is inherently limited to the rendering performance of an Onyx2 multipipe system, which has both physical and economical constraints. The compression and network protocols are also relatively nonstandard.

2.1 WireGL: Cluster Rendering System

The visualization server is a component of a larger system called WireGL. WireGL is a graphics system for clusters that provides scalable output resolution as well as scalable rendering rates. WireGL transmits an encoded representation of OpenGL API commands from a set of application nodes to a set of server nodes where those commands are executed on the client's behalf. The final image is fragmented across multiple physical framebuffers in the server nodes. WireGL has been described in detail elsewhere[2, 4, 5]; a brief overview will be presented below.

WireGL replaces the system's OpenGL library, allowing unmodified serial applications to render to a virtualized framebuffer that has been distributed across a cluster. In addition, WireGL allows multiple client nodes to issue OpenGL commands in parallel to render to the same final output image.

WireGL accomplishes this goal by analyzing each client's stream of OpenGL commands as they are issued and routing them to the appropriate "pipeservers" based on the screen-space extent of the resulting fragments. In the taxonomy proposed by Molnar et al. [10], this is a parallel implementation of a *sort-first* parallel rendering architecture.

At the end of a frame, the final image is distributed across the framebuffers of the pipeservers. To ensure good load balance, multiple tiles are assigned to each pipeserver. These distributed tiles must be recombined to form a final image. One method for performing this image reassembly is to combine the video output from the pipeserver graphics cards, as was done with the Lightning-2 [16]. This approach has the disadvantage that it requires special-purpose hardware. Another approach is to reassembly the tiles in software. This requires that the tiles be read from the graphics card, sent over the system area network, and then reassembled into the final image and drawn into another graphics card. The disadvantage of this approach is that it requires a fast pixel readback and a fast system area network. The combination of these two factors limits the image resolution and frame rates of software only solutions. Neither of these approaches work for remote display, since wide area networks have less bandwidth.

2.2 Streaming Multimedia

The remote display system is largely built using Microsoft's DirectShow infrastructure[3]. DirectShow is a streaming media software API that consists of "streams" and "filters." A flow of media data such as pixels or MPEG-4 frames is called a stream. A filter is a functional unit that operates on a multimedia stream. For example, a DirectShow display filter may take a video stream and display it in a window. Likewise, a video splitter filter may take a single video stream and produce two identical output streams. Each of these filters can be arranged in a "filter graph," which establishes a DirectShow application.

3 Design Goals

The remote display subsystem provides a software solution to remote visualization by recombining framebuffer tiles in software, performing parallel compression, and formatting a stream of images. In this section, we will present the design goals for the system.

- The system is targeted toward a typical work environment with a 100Mbit Ethernet connection between the compute cluster and user's personal computer. The bandwidth required by our system should not exceed this limitations.
- We target an output resolutions of 1024×768 at roughly 30Hz. This rate was chosen by measuring the decompression performance of a year 2000 PC, and is roughly equivalent to HDTV. We also target an overall latency from user input to final image display of 100 msec.
- The remote display system should work transparently with WireGL. Transparency means that the application does not need to be rewritten. This allows unmodified serial OpenGL applications to be used with our system. WireGL also provides a parallel OpenGL so that the rendering performance may be scaled. The system should provide transparent display of images produced by any combination or number of compute and rendering nodes.
- The remote display subsystem itself should be scalable. As image size increases and cluster rendering rates grow, it is important that we are able to parallelize the encoding of the compressed image streams. Modern processors can compress streams only at relatively low image resolutions (320×240) in real time. With the ability to encode streams in parallel on a cluster, we can provide scalable image sizes.

- The remote image stream should use standard streaming multimedia codecs and protocols.

- Although the remote imagery may be compressed, the compression should be of very high quality. No objectionable artifacts should be visible when displaying scientific datasets.

These specific goals were set for the proof of concept system described in this paper. An additional goal was to build a prototype and gain experience with visualization servers. Finally, we wanted to identify the ultimate constraints and limitations of different methods for building visualization servers.

4 System Description

A diagram of the remote visualization system is shown in figure 1. The system is divided into four components: WireGL clients, WireGL pipeservers, compression nodes, and the display node. We describe how the remote rendering back-end expands WireGL's capabilities to enable remote interaction.

4.1 Composition Network

The first stage of the system transfers the distributed frame buffer to the compression nodes. When a pipeserver decodes a `SwapBuffers` message marking the end of a frame, it transmits the contents of its framebuffer to the compression node responsible for compressing that portion of the framebuffer. Each compression node is responsible for a rectangular portion of the output. These pixels are sent uncompressed over the high-speed cluster interconnect and placed into the compressor's OpenGL framebuffer.

The number of compression nodes is configurable. If the system only contains a single compression node, it would be responsible for gathering the entire framebuffer and compressing it. Adding additional compression nodes distributes the framebuffer and the compression workload. While it would be possible to integrate the WireGL pipeserver and compression node onto the same machine, separating the two allows for pipelining of the rendering and the compression in addition to separate load balancing.

These nodes are designed as extended WireGL pipeservers. This closed system allows WireGL to communicate to the compression nodes without the need to a custom protocol. The disadvantage to this approach is that the pixels to be compressed reside in a local OpenGL framebuffer on the compression node. Those pixels must be read out of the framebuffer before compressing, a significant operation.

4.2 Streaming Multimedia: DirectShow

In order to leverage standard streaming multimedia codecs and protocols, the system uses Microsoft's DirectShow framework as illustrated in figure 1. We have implemented variety of filters as well leveraged existing filters available with DirectShow. The first filter is the source filter, which generates a DirectShow stream from the frame data collected on the compositing network. This stream is read by the DirectShow compressor filter, which outputs compressed video data. The compressed data is then read by the network send filter, which transmits the frame data via UDP messaging over a 100Mbit Ethernet network to the display node.

By encapsulating the compression algorithm in a DirectShow filter, we allow the user to replace the compression algorithm to match the needs of the application. Since DirectShow is an established standard, there are dozens of compressors available for use with the system to match user needs.

4.3 Display Node

The display node receives the UDP packets via a DirectShow network receive filter. The messages are streamed to the decompressor, which is able to restore the compressor's frame. While there exists only a single display node, there is a one-to-one relationship between compressor and decompressor filters in the system. In other words, multiple instances of the decompressor filter must be running on the display node if multiple compressor nodes exist.

While this relationship is a requirement of the commodity codec, it has the added benefit on multiprocessor display machines. If multiple decompressors are needed, the frame decompression can occur in parallel. Secondly, by adding more compression streams, we increase the bitrate to the decompressor. By introducing additional compressor nodes, we are able to aggregate the bitrate of all the compressors until we are limited by the network link.

Once the pixels are decompressed, the output of the decompressors needs to be integrated into a single image. This composition is done by an image reassembly filter that draws the complete frame to the screen. This filter completes the remote visualization rendering path.

We can also add pre-existing DirectShow filters at the display or compression nodes to perform additional tasks such as edge-detection, chroma-keying, and any other multimedia filter. Since DirectShow is a commodity standard, the system can easily integrate any 3rd party filters by editing the filter graph. This modularity and flexibility are key features of the remote rendering system design.

4.4 User Input

To allow for user interaction with the graphics application, the system relies on the windowing system for input event management. Since WireGL behaves as an OpenGL driver, applications can simply use the standard X-Windows interface for events, and the rendering will occur on the graphics cluster while transmitting the frames back to the user's display.

5 Results

To demonstrate the viability of the remote rendering system, we have tested it with an application "March," a parallel implementation of the marching cubes volume rendering algorithm[9] designed for WireGL. A $64\times64\times64$ volume is divided into subvolumes which are processed in parallel by a number of isosurface extraction and rendering processes distributed across the graphics cluster. This application renders a 1024×768 or 640×480 size frame (distributed) of a view of a skull dataset in 40ms. The application accepts mouse events from the user to direct the viewing angle.

March was selected to test the limits of the system. High-quality volume visualizations, even on parallel systems, may only be able to render a single frame per second. March however uses a deliberately small dataset to stress the system by driving a fast real-time graphics application to a remote display. This allows for a better understanding of the limitations of each component for interactive remote rendering.

5.1 Testing Environment

The cluster we used for all our experiments, called "Chromium," consists of 32 Compaq SP750 workstations. Each node has two 800 MHz Pentium III Xeon processors, 256 megabytes of RDRAM, and an NVIDIA Quadro2 Pro graphics adapter. 28 SP750s are running RedHat Linux 7.0 with NVIDIA's 0.9-769 OpenGL drivers which are used as WireGL rendering clients and servers. The remaining

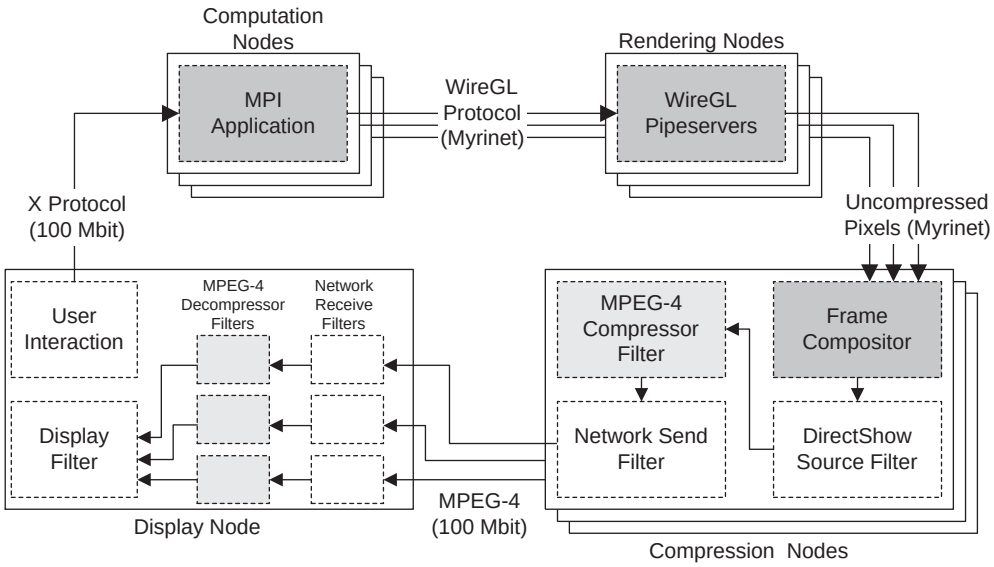

Figure 1: The remote visualization system. The system is constructed of four types of nodes: WireGL client or computation nodes, WireGL rendering nodes, Compression nodes, and a Display node. All nodes except the display node are allocated to machines in the cluster, which are connected via a Myrinet high-speed interconnect. The system begins with the display node sending X events to a parallel graphics application. Frames generated at the WireGL pipeservers are sent, via Myrinet, to the compression nodes, which composite the framebuffer and compress the pixels. The compressed bits are sent via 100Mbit Ethernet to the display nodes, which decompress and display the rendered image.

4 machines, running Windows 2000, are configured as DirectShow compression nodes.

Each node has a Myricom high-speed network adapter [1] connected to its PCI bus. Each network card has 2MB of local memory and a 66 MHz LANai 7 RISC processor. The cluster is fully connected using two cascaded 16-port Myricom switches. Using the 1.4pre37 version of the Myricom Linux drivers, we are able to achieve a bandwidth of 101 MB/sec when communicating between two different hosts.

A final external SP750, running Windows 2000, is the display node connected to the cluster via standard 100Mbit Ethernet, a typical connection for a remote user. This node performs the decompression and display of the final image.

5.2 Compositing Performance

First, we evaluate the compositing performance of the remote rendering system. This stage includes the pixel readback from the WireGL rendering servers and the transmission of the pixels to the compression nodes over the high-speed interconnect. In order for the system to be effective, it should support a variety of WireGL configurations at different resolutions.

To examine our system's flexibility in this area, we measure the time required to perform the pixel readback and send with a variable number of WireGL pipeservers each sending to a single compression node. The results are shown in figure 2. Each line represents a different output resolution with a different number of pipeservers.

The initial performance improvement from one to four pipeservers is due to the parallelization of the readback from the pipeserver framebuffers. As we increase the number of pipeservers, the portion of the logical framebuffer managed by each node decreases. For example, in the single pipeserver configuration, the complete framebuffer must be read out from the graphics card memory by a single node, an expensive operation. However, as

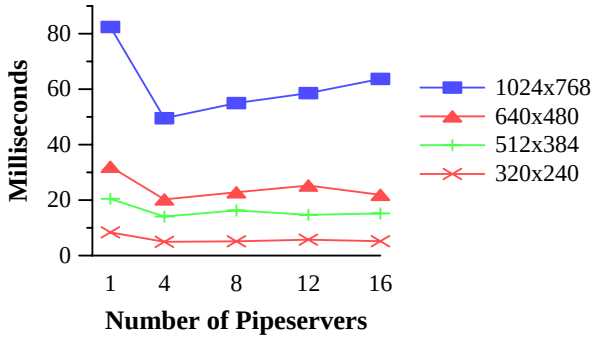

Figure 2: The compositing results. For each resolution, the number of WireGL pipeservers is varied and the time required to read out the framebuffer and send to a single compression node is recorded. The initial improvement in performance is due to parallelization of the framebuffer readback. The slight time increase as we add pipeservers is due to network fan-in congestion at the single compressor.

we add more pipeserver nodes, the number of pixels read back by each node decreases and the system becomes dominated by the network transmission of the pixels to a single compression node over the high-speed cluster interconnect.

The time slightly increases as we add more pipeservers. While we are still transmitting the same number of pixels in each case, the network fan-in can reduce the efficiency of the network bandwidth into a single compression node. We are also susceptible to load imbalances within WireGL since we must wait for the slowest pipeserver to complete before compressing a frame. However even in the worse case of 1024×768 with 16 pipeservers, we see a only 28% increase from the best case.

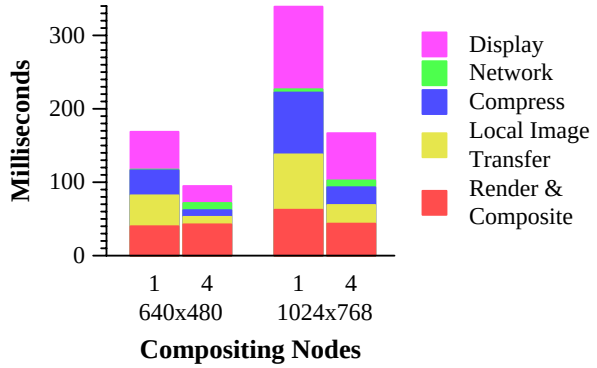

Figure 3: The end-to-end system timing results for two output resolutions with one and four compression nodes. This shows the breakdown of the end-to-end throughput of the system of the four components of the system: WireGL "Render & Composite" time, "Local Image Transfer" time spent reading pixels from compressor framebuffers to the compressor, MPEG-4 frame "Compress" time, "Network" time spent transmitting the compressed image and X events, and "Display" time spent MPEG-4 decompressing and drawing the output. As we increase the number of compression nodes, the end-to-end latency improves. The table below provides an exact breakdown of the timing information (in milliseconds).

	640x480		1024x768	
	1	4	1	4
Render & Composite	40.00	42.59	62.50	43.59
Local Image Transfer	42.09	10.20	75.38	25.34
Compress	34.05	9.14	84.30	24.21
Network	1.08	9.76	4.47	9.27
Display	51.46	23.11	112.51	64.38
Total	168.69	94.80	339.15	166.79

These results also demonstrate the need to parallelize the compositing phase of the system, especially at high resolutions. At 1024×768 with 16 pipeservers, we can only composite a frame in 64ms or at a rate of 15.6 frames per second. With a single compression node, this places a lower limit on the rendering rate of the system. If we increase the output resolution further, the rate will continue to decrease linearly with the number of pixels sent over the network[5]. Therefore, it is imperative that we parallelize this compositing step across multiple nodes to prevent the image composition from becoming the limiting operation.

5.3 System Performance

To test the throughput of our system, we ran two different configurations with the March application:

16-12-1-1: As a baseline configuration, we have 16 WireGL clients with 12 rendering nodes, 1 compression node, and 1 display node. This configuration demonstrates a serial compression pipeline.

16-12-4-1: This configuration is the same as above, except with four compression nodes. This change parallelizes the compression and compositing operations. This demonstrates the parallel pipeline.

Both configurations are run at 640×480 and 1024×768 output resolution for comparison.

For our tests, we needed to decide upon a compression codec. The ideal compression implementation should minimize the required network bandwidth and processor utilization while maximizing image quality. We evaluated five realtime DirectShow codecs with similar processor utilization: Microsoft Video1, Microsoft Motion-JPEG, Microsoft MPEG-4, Intel Indeo 5.10, and Intel H263. Many other codecs were obtainable, but some were not realtime or freely available. We used a video sequence from our test scene with large differences between successive frames to measure the worst-case behavior. Video1 and H263 generated many blocky artifacts while the other three codecs produced images visually similar to the original. Of the three high quality codecs, MPEG-4 required the least bandwidth.

The breakdown of the system is divided into five parts as shown in figure 3. "Render & Composite" consists of the time March takes to produce a frame in the pipeserver's distributed framebuffer and to perform the compositing into the compression node framebuffer. "Local Image Transfer" is the time required to read the pixels out of the compression framebuffer into DirectShow infrastructure. "Compress" is the time for the DirectShow MPEG-4 compressor to produce a compressed frame. "Network" in transmitting the compressed frame to the display node and the X events, all occurring over the 100Mbit Ethernet. "Display" is the time required for a received compressed stream to be decompressed and rendered. Each portion represents a different operation required to produce an output image on the remote display. Furthermore these were suitable locations within the system where timings could be evaluated.

The comparison between the single compression node and the four node configurations demonstrates the scalability of the compression subsystem. First, the render time decreases slightly in the high resolution case since we have divided up the compositing operation across four nodes.

The Local Image Transfer time improves four-fold in the 640×480 case and three-fold in the 1024×768 case. This is due to the parallelization of the pixel readback from the compressor framebuffers into DirectShow. The compression time also exhibits a 3.7 and 3.4 speedup since we have parallelized the compression across four nodes. Finally, the display timing exhibits a 2.2 and 1.7 speedup since the four compressed data streams can be decoded in parallel on the two processors available on the display node.

The Network time increases since we are sending 4 MPEG-4 bitstreams rather than a single stream however we are well below the Ethernet link limitations. Each MPEG-4 bitstream outputs an observed maximum bitrate of a 500Kbits per second. Since we have divided the frame between four compressors, each compressor can output the maximum MPEG-4 bitrate for its corresponding subregion of the output. The bandwidth limit for the entire output is therefore four times the limit of a single compressor. However, even with four saturated MPEG-4 streams, this results in roughly 2Mbits per second of data received by the display node. Given that our link is 100Mbit, we certainly are not limited by the network layer.

The results generated above used a lossy MPEG-4 compression algorithm which did produce minor artifacts in the final image. However by introducing multiple compression nodes, the aggregate compression bitrate was increased. This reduced the image artifacts and demonstrates an additional benefit of parallel compression nodes. The results of MPEG-4 compression are shown in figure 4.

The remote rendering components are pipelined across multiple nodes. As a result, the framerate of the system is determined largely by the slowest node in the system. We observed average framerates of 25 fps for the 4 compression node 640×480 March and 13 fps for the 1024×768 configuration. Furthermore, the end-to-end latency of the system is only 95ms for a 640×480 output and 167ms for 1024×768 case. These values clearly allow interactive remote rendering with our system.

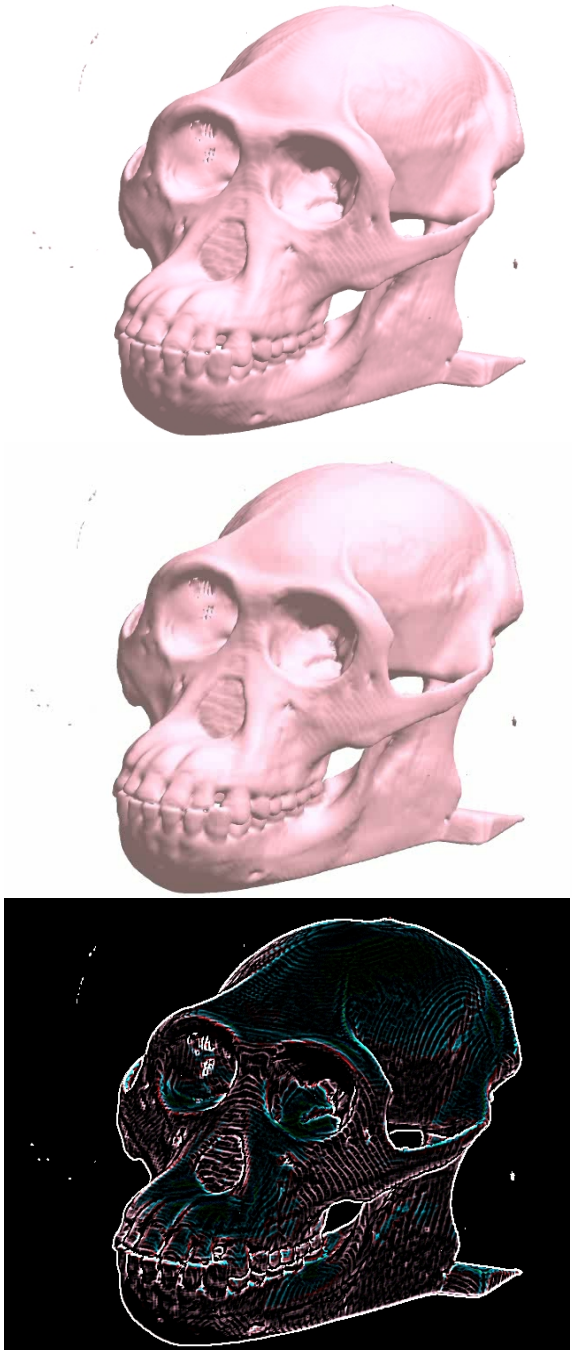

Figure 4: A DirectShow image processing tool which used in the visualization system. The top image is the original input. Next is an MPEG-4 compressed version followed by an edge detector. This last component was added by simply editing the DirectShow filter graph on the display node.

5.4 Image Streaming Flexibility

In order to demonstrate the flexibility afforded through the use of DirectShow, we created a set of filters based on the Intel Image Processing Library (IPL)[6]. IPL provides a suite of highly optimized image operators implemented as DirectShow filters. Figure 4 shows the operations of a filter performing high-pass filtering to reveal the image edges. Since IPL is designed to support DirectShow, only a trivial amount of coding is required to insert these filters. These filters operate in realtime and can be dynamically inserted into our remote visualization framework. We have also explored other filters such as file streaming filters to save and playback renderings, network broadcast filters for multiple viewing, and image analysis tools such as histogram filters.

6 Discussion and Future Work

Our results demonstrate that by combining cluster-based rendering with remote display services, we can provide high performance visualization services to the desktop. This type of system is ideal for visualizing large datasets. The system is very flexible, allowing for different numbers of compute, rendering and compression nodes. This flexibility allows the system to support different applications and situations with different resource requirements or constraints. It also allows the system to easily track new technologies.

Currently the system has been optimized around the capabilities of the most common client display server, a modern PC. This obviously has practical advantages, but some disadvantages. Since commodity PCs are oriented around applications with large markets, such as digital television, they are optimized for displaying HDTV-1 resolutions and frame rates – roughly 1024×768 at 30Hz. These image display rates are determined by the speed of a software decompressor running on the main processor. However, graphics accelerators are now beginning to implement MPEG decode in hardware. This should allow significantly higher performance and hence higher image resolutions. This assumes the system is designed to support variable resolution imagery, which surprisingly is not part of many video standards. However, if higher resolutions or frame rates becomes possible, our system could easily be reconfigured to support this by adding more compressors.

Somewhat surprisingly, by choosing to use commodity technology, we are not bandwidth limited. The widely available codecs are optimized for video applications. In particular, MPEG-4 is very good at providing relatively high quality at low bitrates. On the negative side, the MPEG-4 codec is very compute intensive. Thus, we need to devote significant resources to compression. On the positive side, because the compression rate of commercial codecs is so good, our system is able to run very well on low bandwidth networks. For example, we could use the system described here with 10 Mb/s networks and a television channel (which delivers 19.2 Mb/s). On the other hand, it is difficult right now to adapt our system to higher bandwidth networks. Ideally the compression codec should be able to trade-off quality and bitrate to adapt to the computing resources and the network. We are currently investigating codecs with these properties.

Finally, we think it is very important to couple 3D graphics with streaming multimedia. This brings together in a single system two very powerful capabilities. DirectShow has been an excellent system for rapid prototyping. The implementation was very simple because we could utilize preexisting components and did not have to build them ourselves. More importantly, combining systems adds a great deal of additional functionality. For example, we demonstrated how local image enhancement is possible by inserting a DirectShow edge detection filter into the display pipeline. Other possibilities include combining images generated from multiple sites, combining data display with videoconferencing, and so on. These are all interesting directions for future research.

Acknowledgments

The authors would like to thank Matthew Eldridge for his contributions to the WireGL system. This work is funded by DOE VIEWS

Program.

References

[1] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, pages 29–36, February 1995.

[2] I. Buck, G. Humphreys, and P. Hanrahan. Tracking graphics state for networked rendering. *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, August 2000.

[3] Microsoft DirectShow Application Programming Interface. http://msdn.microsoft.com/library/psdk/directx/dx8_c/ds/default.htm.

[4] G. Humphreys, I. Buck, M. Eldridge, and P. Hanrahan. Distributed rendering for scalable displays. *IEEE Supercomputing 2000*, October 2000.

[5] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan. WireGL: A scalable graphics system for clusters. *Proceedings of SIGGRAPH 01*, August 2001.

[6] Intel image processing library features/benefits. http://www.intel.com/software/products/perflib/ipl/iplfeatures.htm.

[7] M. Kilgard. *OpenGL Programming for the X Window System*. Addison-Wesley, 1996.

[8] M. Kilgard. GLR, an OpenGL render server facility. *Proceedings of X Technical Conference, The X Resource, issue 17*, Winter 1996.

[9] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *Proceedings of SIGGRAPH 87*, pages 163–169, July 1987.

[10] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Algorithms*, pages 23–32, July 1994.

[11] Adrian Nye, editor. *X Protocol Reference Manual*. O'Reilly & Associates, 1995.

[12] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and Andy Hopper. Virtual network computing. *IEEE Internet Computing Vol.2 No.1*, pages 33–38, Jan/Feb 1998.

[13] R. Samanta, T. Funkhouser, K. Li, and J.P. Singh. Sort-first parallel rendering with a cluster of pcs. *Sketch at SIGGRAPH 2000*, 2000.

[14] B. Schmidt, M. S. Lam, and J. D. Northcutt. The interactive performance of SLIM: a stateless, thin-client architecture. *17th ACM Symposium on Operating Systems Principles*, December 1999.

[15] SGI Vizserver. http://www.sgi.com/software/vizserver/.

[16] G. Stoll, M. Eldridge, D. Patterson, A. Webb, S. Berman, R. Levy, C. Caywood, M. Taveira, S. Hunt, and P. Hanrahan. Lightning-2: A high-performance display subsystem for PC clusters. In *Proceedings of SIGGRAPH 01*, 2001.