Distributed Rendering for Scalable Displays

Greg Humphreys Ian Buck Matthew Eldridge Pat Hanrahan

Computer Science Department
Stanford University

Abstract

We describe a novel distributed graphics system that allows an application to render to a large tiled display. Our system, called WireGL, uses a cluster of off-the-shelf PCs connected with a high-speed network. WireGL allows an unmodified existing application to achieve scalable output resolution on such a display. This paper presents an efficient sorting algorithm which minimizes the network traffic for a scalable display. We will demonstrate that for most applications, our system provides scalable output resolution with minimal performance impact.

Keywords: Remote Graphics, Cluster Rendering, Tiled Displays, Distributed Rendering

1 Introduction

Modern supercomputers have allowed the scientific community to generate simulation datasets at sizes which were not feasible in previous years. The ability to efficiently visualize such large, dynamic datasets on a high-resolution display is a desirable capability for researchers. However, the display resolution offered by today's graphics hardware is typically not sufficient to visualize these large datasets. Large format displays such as the PowerWall [12] and CAVE [4] support resolutions well beyond the common desktop but are limited in their scalability. The work presented in this paper provides a scalable display technology which decouples output resolution from rendering performance. This offers a solution for effectively visualizing large and dynamic datasets by constructing a system that complements the large compute power from which the data was generated.

Previous attempts to solve the resolution challenge have faced a number of limitations. One method is to provide a parallel computer (e.g, SGI Origin2000) with extremely high-end graphics capabilities (e.g, SGI InfiniteReality). This approach is limited by the number of graphics accelerators that can fit in one computer and is often prohibitively expensive, potentially costing many millions of dollars. At the other end of the spectrum are clusters of workstations, each with a fast graphics accelerator. Most previous efforts to allow fast rendering on clusters have dealt with static data and could not handle dynamic scenes or time-varying visualizations. Other systems require the use of a custom graphics API to achieve scalable rendering, meaning that existing applications must be ported to the API in order to use the system.

The WireGL project at Stanford is researching the systems aspects of high-performance remote rendering. We hope to provide a graphics system that allows the user to render to a tiled display with none of the aforementioned limitations. Our goals are as follows:

- Provide a scalable display solution. For most graphics applications, we can scale the resolution of the output display without affecting the performance of the application.
- Each node within the system should be inexpensive. We use commodity PC graphics cards like the NVIDIA GeForce2 GTS, each of which costs less than \$300 and offers performance comparable to or faster than high-end workstation graphics.
- Support an immediate-mode API like OpenGL rather than requiring a static scene description. By not requiring the application to describe its scene *a priori*, we enable more dynamic, interactive visualizations.
- Finally, support existing, unmodified applications on a variety of host systems. This relieves the programmer from having to learn a new graphics API or system architecture to take advantage of scalable displays. This also allows non-programmers to use new display technology with their existing applications.

{humper|ianbuck|eldridge|hanrahan}@graphics.stanford.edu

0-7803-9802-5/2000/\$10.00 ©2000 IEEE.

2 Related Work

The WireGL project grew out of Stanford's Interactive Mural group, which described techniques for treating a physically disjoint set of displays as a single logical display, as well as algorithms for efficiently sharing the display between a number of applications [6]. WireGL retains many of the techniques and metaphors present in our original system but with an emphasis on performance rather than interaction techniques. The major systems advancement of WireGL over previous work at Stanford is the transition from SGI InfiniteReality based tiled displays to a PC cluster approach. This dramatically improves the scalability of our system, allowing the system to scale to hundreds of nodes without becoming prohibitively expensive.

Due to the relatively recent appearance of networks with a gigabit or more of bandwidth, most previous attempts at rendering over a network have not been focused on high-performance rendering. In fact, remote rendering has historically been an antonym of fast rendering. GLX [13] and X Windows [11] both provide remote rendering capabilities, but neither was designed to operate efficiently over very high-speed networks. WireGL introduces new techniques for data management and distribution to make effective use of today's high-speed networks.

Much research has been done on systems for large display walls, such as the PowerWall [12] and the InfinityWall [5]. However, many of these systems require that each display component of the wall be connected to a large SMP system. WireGL works well with a cluster of workstations, which significantly reduces the total cost of the display wall. Also, other solutions often require the use of a custom API, which may limit the abilities of the graphics system [15].

Rendering on clusters of workstations is not a new idea. Many of the studies done on load-balancing graphics with clusters have typically focused on scene-graph based applications [14]. Since the scene-graph data is typically static, it can be distributed to the cluster before rendering. However, one of the main goals of WireGL is to support dynamic scenes which cannot be represented with a fixed scene-graph.

Sorting algorithms for computer graphics [3] have been explored by many different graphics architectures. Mueller presents a parallel graphics architecture using a sort-first algorithm which takes advantage of frame-to-frame coherence of scenes [10]. The display wall group at Princeton explored a load-balancing algorithm which combines sorting of primitives with routing pixel data over the cluster network [14]. Our system differs from previous research in this area in that it presents a sort-first algorithm which works with immediate-mode rendering; It is designed to work with an arbitrary stream of graphics commands and no prior knowledge of the scene to be rendered.

3 System Architecture

WireGL provides the familiar OpenGL API [13] to users of our tiled graphics system. OpenGL is a graphics API for specifying polygonal scenes and imagery to be rendered on a display. WireGL is implemented as a driver that stands in for the system's OpenGL driver so applications can render to a tiled display without modification. The WireGL driver is responsible for sending commands to the graphics servers which will do the rendering on the client's behalf. A diagram of our system is shown in figure 1. The rendering servers form a cluster of PCs, each with its own graphics accelerator. The output of each server is connected to a projector which projects onto a common screen. These projectors are configured into a tiled array so that their outputs produce a single large image.

The client and servers are connected via a high-speed network. Our current installation uses Myrinet [1], but WireGL is not bound to a specific network technology. As long as the network can support basic message passing, our system can easily be modified to support it. WireGL runs on TCP/IP and shared memory in addition to Myrinet GM, and a VIA port is in progress. WireGL also makes no assumptions about the graphics hardware present in the client or the server. Since the servers simply make calls to their native OpenGL driver, the graphics card can be upgraded without any changes to WireGL. Also, the system is largely OS independent. Only a small set of functions that connect OpenGL to the window system vary on each platform.

In order to send the client's OpenGL calls to the servers most efficiently, WireGL needs a network protocol that is compact and straightforward to encode. Given such a protocol, we must also manage the total amount of data sent to each server. To avoid unnecessarily retransmitting data to multiple servers, we use a combination of geometry bucketing and state tracking. These tools allow us to send to each server a subset of the application's commands necessary for it to render its portion of the display.

3.1 Network Protocol

In an immediate-mode graphics API like OpenGL, each vertex is specified by an individual function call. In scenes with significant geometric complexity, an application can perform many millions of such calls per frame and will be limited by the available network bandwidth. Therefore, the amount of data required to represent the function calls, as well as the amount of time required to construct the network stream, will determine the overall throughput of the system. Likewise, the time required to unpack the network commands also directly affects our overall performance.

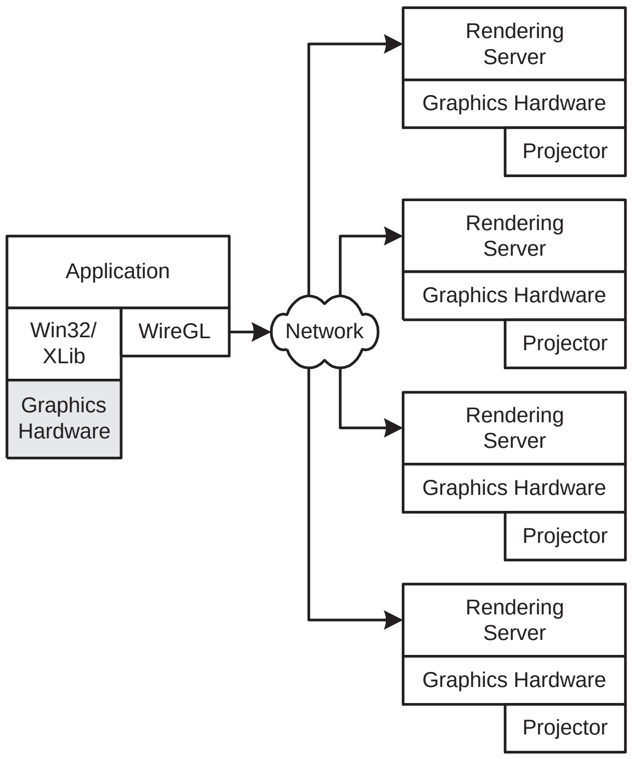


Figure 1: Diagram of the system architecture. WireGL is implemented as a graphics driver which intercepts the application's calls to the graphics hardware. WireGL then distributes the rendering to multiple rendering servers. These servers are connected to projectors which display the final output.

Our network stream representation keeps function arguments naturally aligned. For example, float arguments are aligned on 4-byte boundaries and short arguments are aligned on 2-byte boundaries. This is important on the Intel architecture because misaligned reads cost 4 to 12 times as much as aligned reads [8]. On other architectures, reading unaligned data requires manual shift and mask operations.

By eliminating unnecessary OpenGL commands (e.g. by collapsing `glVertex3f` and `glVertex3fv`), the remaining 224 commands can be encoded using a single byte opcode. In most cases, the type and number of arguments is implicit. For example, a `glVertex3f` call will generate a 1 byte opcode and 12 bytes of data for the three floating-point vertex coordinates. For functions which require a variable length data field such as `glTexImage2D`, the representation explicitly encodes the size of the arguments.

Because an opcode is a single byte, interleaving opcodes and data would introduce up to 3 wasted padding bytes per function call. By packing opcodes and data separately, we can avoid these wasted bytes. For example, a `glVertex3f` call can be encoded in 13 bytes instead of 16 bytes when opcodes and data are packed separately. We avoid the overhead of sending the opcodes and data as two separate messages on the network by packing the opcodes backwards in the same network buffer as the data, as shown in figure 2.

This simple network representation allows for fast packing and unpacking of the graphics commands. In most cases, packing simply requires copying the function arguments to the data buffer and writing the opcode. To unpack the commands on the server, the opcode is used as an offset into a jump table. Each unpacking function reads its arguments from the data buffer and calls its corre-

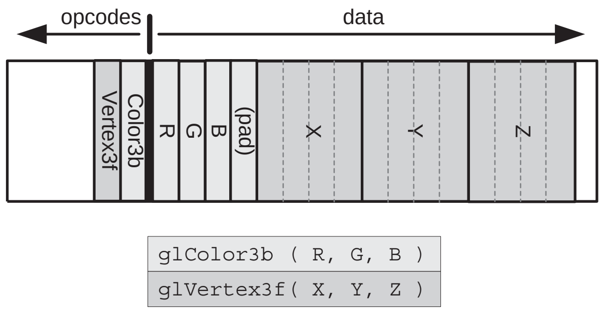


Figure 2: A packed network buffer. Each thin rectangle is one byte. The data are packed in ascending order and the opcodes in reverse order. A header applied to the network buffer before transmission encodes the location of the split between opcodes and data. Only the shaded area is actually transmitted.

sponding OpenGL function. In many cases, the explicit reading of arguments can be avoided by using the function's vector equivalent. For example, a `glVertex3f` call on the client can be decoded as a `glVertex3fv` call on the server.

3.2 Bucketing

In order to render a single graphics stream on multiple servers, the OpenGL commands need to be transmitted to each server. One simple solution would be to broadcast the commands to each server. However, high-speed switched networks like Myrinet generally do not support efficient broadcast communication. To build a system which scales effectively, we must limit the data that is transmitted to each server.

Because the WireGL driver manages the stream of OpenGL calls as they are made, it can exploit properties of that stream to more efficiently use the network. Ideally, we would like to send the minimal number of commands to each server for it to properly render its portion of the output. WireGL achieves this by sorting the geometry specified by the application and sending each server only the geometry which overlaps its portion of the output display space. Unlike solutions that require a retained-mode scene-graph, the WireGL driver has no advance knowledge of the scene being rendered.

As the application makes OpenGL calls, WireGL packs the commands into a geometry buffer and tracks their 3D bounding box. The bounding box represents the extent of the primitives in their local Cartesian coordinate system. The cost of maintaining the object space bounding box is low; only 6 conditional assignments are needed for each vertex. When the network buffer is flushed, WireGL applies a 3D transformation to that bounding box to compute the screen extent of the buffer's geometry. For each server whose output area is overlapped by the transformed bounding box, WireGL copies the packed geometry opcodes and data into that server's outgoing network buffer. These outgoing buffers are also populated by the state tracking system, described in section 3.3. Since geometry commands typically make up most of the OpenGL commands made per frame, this method provides a significant improvement in total network traffic over simply broadcasting.

Ideally, WireGL should transmit each primitive to only those servers whose managed area it overlaps, and only those primitives that span multiple managed areas would be transmitted more than once. However, performing bucketing on a per-primitive granularity would be very costly, since bucketing requires multiple matrix multiplications. Instead, we transform the bounding box only when the packed geometry buffer is flushed, which amortizes the transformation cost over many primitives.

The success of this algorithm relies on the spatial locality of successive primitives. Many large scale visualization applications (e.g. volume rendering) exhibit this behavior. These applications issue millions of small, spatially coherent primitives. Given the small primitive size, the resulting transformed bounding box is much smaller than the screen extent of a projector to which it will be sent. As a result, most bucketed geometry is only sent to a single projector, as demonstrated in the results shown later.

Although bucketing is critical to achieving output scalability, it is important to note that it does not allow us to scale the size of the display forever. As the display gets bigger (that is, as we add more projectors), the screen extent of a particular bounding box increases relative to the area managed by each rendering server. As a result, more primitives will be multiply transmitted, limiting the overall scalability of the system.

3.3 State Tracking

OpenGL is a large state machine. State parameters (lighting parameters, fog settings, etc.) can be set at almost any time and will persist until they are reset in the future. For example, instead of specifying the color of each primitive as the primitive is issued, the user sets the "current" color and then draws many primitives to appear in that color. This design has some advantages over stateless graphics systems since the state does not need to be continually retransmitted to the graphics hardware.

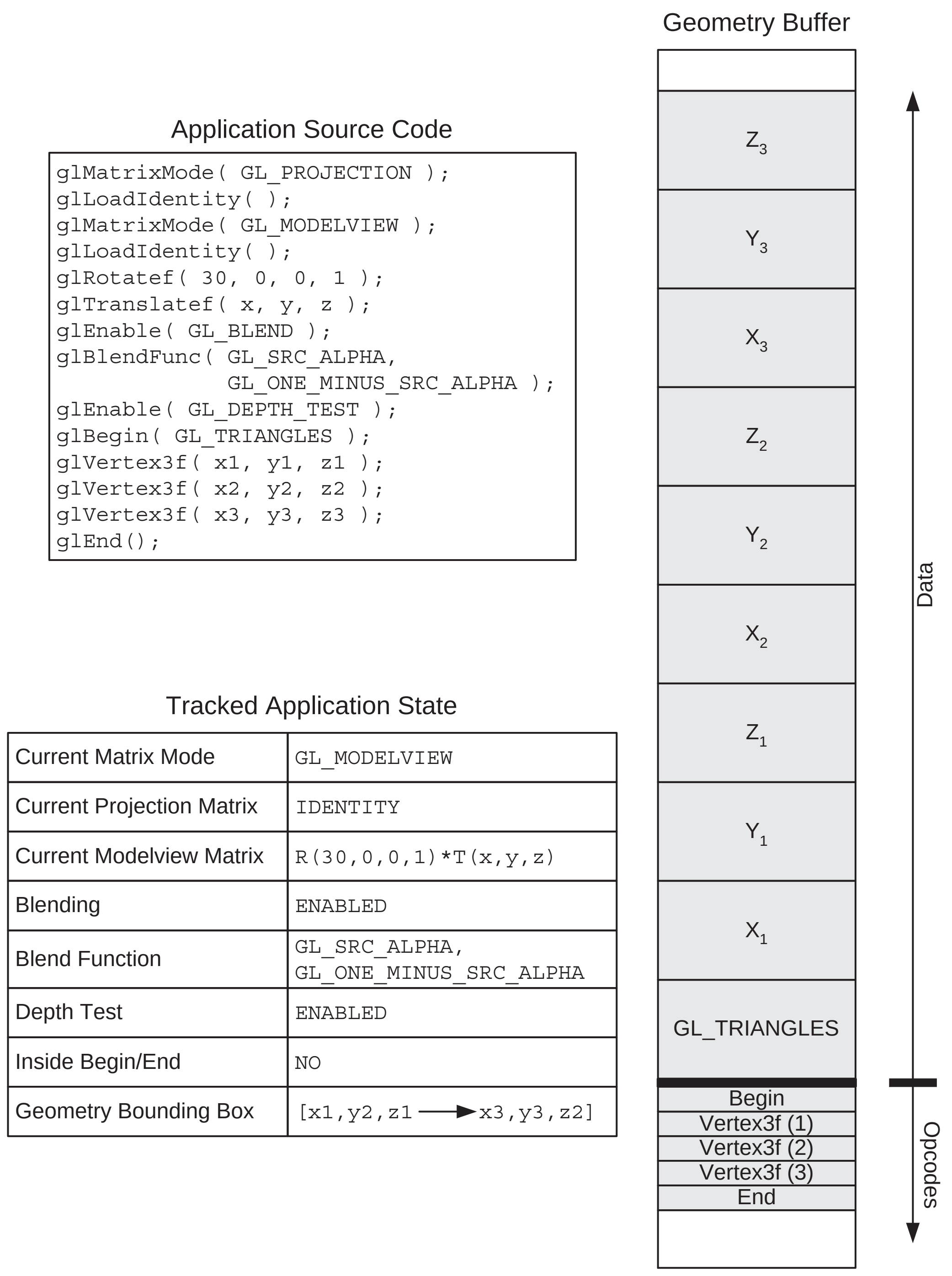
State commands cannot be packed and bucketed immediately, because they do not have any geometric extent and they may affect future geometry which is destined for a different server. One solution would be to simply broadcast state commands to all rendering servers, as proposed by Torborg [16]. Broadcasting state would be less costly than broadcasting geometry since the state data usually comprise a much smaller portion of the total frame data. For example, the OpenGL `Atlantis` demo issues 3,020 bytes of state commands and 375,223 bytes of geometry data per frame. Despite this difference of two orders of magnitude, broadcasted state traffic could quickly overtake bucketed geometry traffic as the display gets larger. Ideally, we would like a technique similar to bucketing for state commands.

WireGL solves this problem by tracking the entire OpenGL state of the application as well as the current state of each of the servers. When a state call is made by the application, instead of packing the call into its network representation, the WireGL driver merely updates a data structure containing the application's graphics state, as shown in figure 3. When WireGL transmits a network buffer of geometry commands, it first determines the overlapped servers as described in section 3.2. For each server that the buffer overlaps, WireGL then computes a difference between the application's state and the server's state and transmits the minimal set of updates to the server to synchronize it with the application. Once these differences have been sent, the packed geometry can follow. The buffer management scheme used to achieve this behavior is shown in figure 4.

We can compute the difference between the application's state and a server's state very quickly using a hierarchical scheme; in practice the entire state tracking system occupies less than 10 percent of the total frame time for any application. For a complete description of the state tracking subsystem of WireGL, see Buck, Humphreys and Hanrahan [2].

4 Results

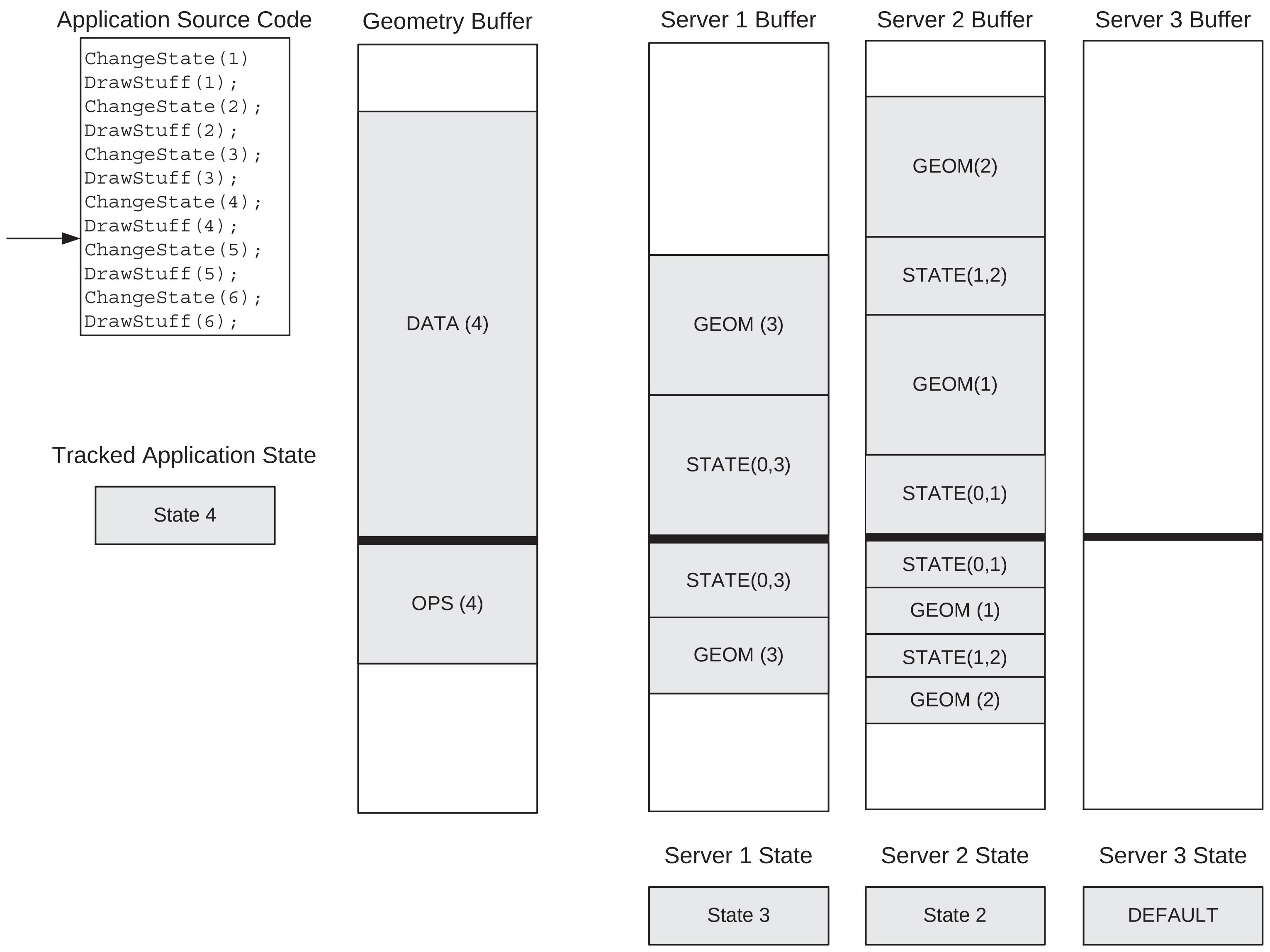
We have built a 36-node cluster named "Chromium." Chromium consists of 32 rendering workstations and 4 control servers. Each workstation contains dual Pentium III 800MHz processors and an



```
glMatrixMode( GL_PROJECTION );
glLoadIdentity( );
glMatrixMode( GL_MODELVIEW );
glLoadIdentity( );
glRotatef( 30, 0, 0, 1 );
glTranslatef( x, y, z );
glEnable( GL_BLEND );
glBlendFunc( GL_SRC_ALPHA,
             GL_ONE_MINUS_SRC_ALPHA );
glEnable( GL_DEPTH_TEST );
glBegin( GL_TRIANGLES );
glVertex3f( x1, y1, z1 );
glVertex3f( x2, y2, z2 );
glVertex3f( x3, y3, z3 );
glEnd();
```

Current Matrix Mode	GL_MODELVIEW
Current Projection Matrix	IDENTITY
Current Modelview Matrix	R(30,0,0,1)*T(x,y,z)
Blending	ENABLED
Blend Function	GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA
Depth Test	ENABLED
Inside Begin/End	NO
Geometry Bounding Box	[x1,y2,z1 $\longrightarrow$ x3,y3,z2]

Figure 3: Packing and state tracking. The "tracked application state" boxes contain the complete OpenGL state of the running application. Note that the OpenGL state is actually quite large; state elements not shown in this figure are assumed to be the OpenGL default. Once the source code shown in the upper left has executed, the geometry buffer contains just the opcodes and data that appeared between the `glBegin`/`glEnd` pair, while all other calls have recorded their effects into the state structure. When the geometry buffer is transmitted, it will be preceded by the necessary commands to bring the rendering server's state up to date with the tracked application state.



```
ChangeState(1)
DrawStuff(1);
ChangeState(2);
DrawStuff(2);
ChangeState(3);
DrawStuff(3);
ChangeState(4);
DrawStuff(4);
ChangeState(5);
DrawStuff(5);
ChangeState(6);
DrawStuff(6);
```

Figure 4: A snapshot of WireGL's buffer management scheme. Geometry commands are packed immediately into the geometry buffer. State commands record their effect into the client's tracked state structure. When the geometry buffer is flushed, the bounding box of the geometry buffer is used to determine which servers will need to receive the geometry data. Note that when the application changes state, the geometry buffer must be flushed because that state change will only apply to subsequent geometry, not to already packed geometry. The ordering semantics of OpenGL dictate that each such server must first have its state brought up to date before the geometry commands can legally be executed. In the figure, geometry blocks 1 and 2 have fallen completely on server 2, so those data and the associated state changes only appear in its associated buffer. Geometry block 3 falls completely on server 1. Note that server 3 has not had any geometry fall on its managed area, so no data have been sent to it, and its state is falling further and further behind the application. If a geometry block were to fall on more than one server, the geometry would be copied to multiple outgoing buffers. The STATE(A,B) blocks represent the opcodes and data necessary to transition from state A to state B.

Network	No Bounding Box	Bounding Box
Ideal	22.6 MVerts/sec 293.5 MB/sec	12.0 MVerts/sec 156.0 MB/sec
Myrinet Synchronous	4.73 MVerts/sec 61.5 MB/sec	4.10 MVerts/sec 53.3 MB/sec
Myrinet Asynchronous	7.70 MVerts/sec 100.1 MB/sec	7.68 MVerts/sec 99.8 MB/sec

Figure 5: Packing rates for WireGL. `glVertex3f` packing was tested with an ideal network (infinite bandwidth) and Myrinet with synchronous and asynchronous overlapped sends. The rates were calculated with and without bounding box calculation.

NVIDIA GeForce2 GTS graphics accelerator. The cluster is connected with a Myrinet network which provides a point-to-point observed bandwidth of 100MB/sec using our software. Each workstation outputs a 1024x768 resolution video signal, which can be connected to a large tiled display.

In this section, we analyze both the base protocol performance and the overall scalability of WireGL.

4.1 Protocol Performance

The overall performance of the WireGL software is directly related to the speed at which OpenGL commands can be processed by the system. To evaluate the speed of our implementation, we tested a simple application which draws a finely tessellated cube. This application was tested against three different network models: “Ideal” assumes an infinite bandwidth network, “Myrinet Synchronous” which performs a synchronous send using the Myrinet GM library, and “Myrinet Asynchronous” which performs asynchronous overlapped sends. In order to evaluate the overhead of computing the object-space bounding box, vertex rates were measured with and without bounding box calculation. All experiments were performed with a 1x1 tiled display configuration.

Figure 5 shows the results of these tests. On the ideal network, WireGL is capable of packing 22.6 million vertices, or 293.5MB per second (recall that each vertex occupies 13 bytes). This rate is halved when bounding boxes are computed, due to the extra computation which is performed at each `glVertex3f` call. These packing rates are at or above the observed peak bandwidths of high-speed networks available today.

Using Myrinet, it is clear that we are limited by the bandwidth of the network. The synchronous send model uses blocking sends which wait until the packet is placed on the physical network before returning. As a result, the packing and bounding box calculation costs are not overlapped with the network send time. This can be seen in the drop in packing rate when we maintain a bounding box. With asynchronous sends, the packing of `glVertex3f` calls, including the bounding box calculation, is overlapped with the network transmission. The results show that the bounding box calculation does not impact performance since we are limited by the bandwidth of the network, not by the WireGL implementation.

4.2 Scalability

To demonstrate scalability, we tested our system with three different applications:

- `March` extracts and renders an isosurface from a volumetric data set, a typical scientific visualization task. Our dataset is a 400^3 volume, and the corresponding isosurface consists of 1,525,008 triangles.

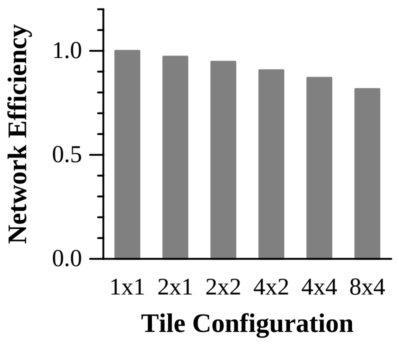


Figure 6: The network efficiency of `March`. 1x1 rendering demonstrates ideal efficiency since no geometry is transmitted more than once. At 32 rendering nodes, the efficiency has dropped to 81% due to overlapping geometry with more than one rendering server.

- `Nurbs` tessellates a set of patches into triangle strips. This application is typical of higher level surface description applications used in animation. Our dataset consists of 102 patches, with 1,800 triangles per patch, for a total of a 183,600 triangles.

- `Quake` is the first person action game *Quake III: Arena* by Id Software. `Quake` is essentially an architectural walk-through with visibility culling, and its extensive use of OpenGL’s feature set stresses our implementation of WireGL.

Each application was run in six different tiled display configurations: 1x1, 2x1, 2x2, 4x2, 4x4, and 8x4, shown in figure 7. To quantify the cost of state tracking and bucketing, we repeated our experiments in “broadcast mode”, which disables state tracking and bucketing and broadcasts the application’s command stream to all of the rendering servers.

`March` and `Nurbs`, shown in figure 7(a,b), clearly demonstrate WireGL’s scalability. As we increase the output size of the display, WireGL achieves a nearly constant frame rate regardless of the number of rendering servers. In comparison, the frame rate of the broadcast method drops with every server added to the configuration, as expected. All of the geometry rendered by both `March` and `Nurbs` is visible, so there is no benefit to bucketing when only rendering to a single display. In fact, broadcast mode has slightly better performance than WireGL when rendering to a 1x1 tiled display because it does not incur the overhead of state tracking and maintaining the bounding box. However, as soon as we add a second display WireGL quickly overtakes the broadcast method due to its more efficient network usage.

Broadcasting commands to twice the number of servers halves the rendering speed, as expected. For `March`, WireGL’s rate for 32 rendering servers only decreases by 13% from the single server configuration, compared to broadcast, which runs 25 times slower. WireGL’s performance decrease is due to a small number of geometry primitives crossing multiple server outputs, resulting in additional transmissions of the geometry buffer. As with any scene that covers the entire output area, a certain number of primitives will need to be sent to multiple servers. In general, the slowdown for broadcast will be proportional to the number of rendering servers, while the slowdown for WireGL is proportional to the multiply transmitted geometry for each application.

In order to understand the cost of geometry that overlaps multiple projectors, we measured the network efficiency of WireGL on various tiled display configurations. Network efficiency is defined to be the ratio of data sent to a tiled display versus data sent to a 1x1 configuration. As we add more tiles, primitives tend to overlap more

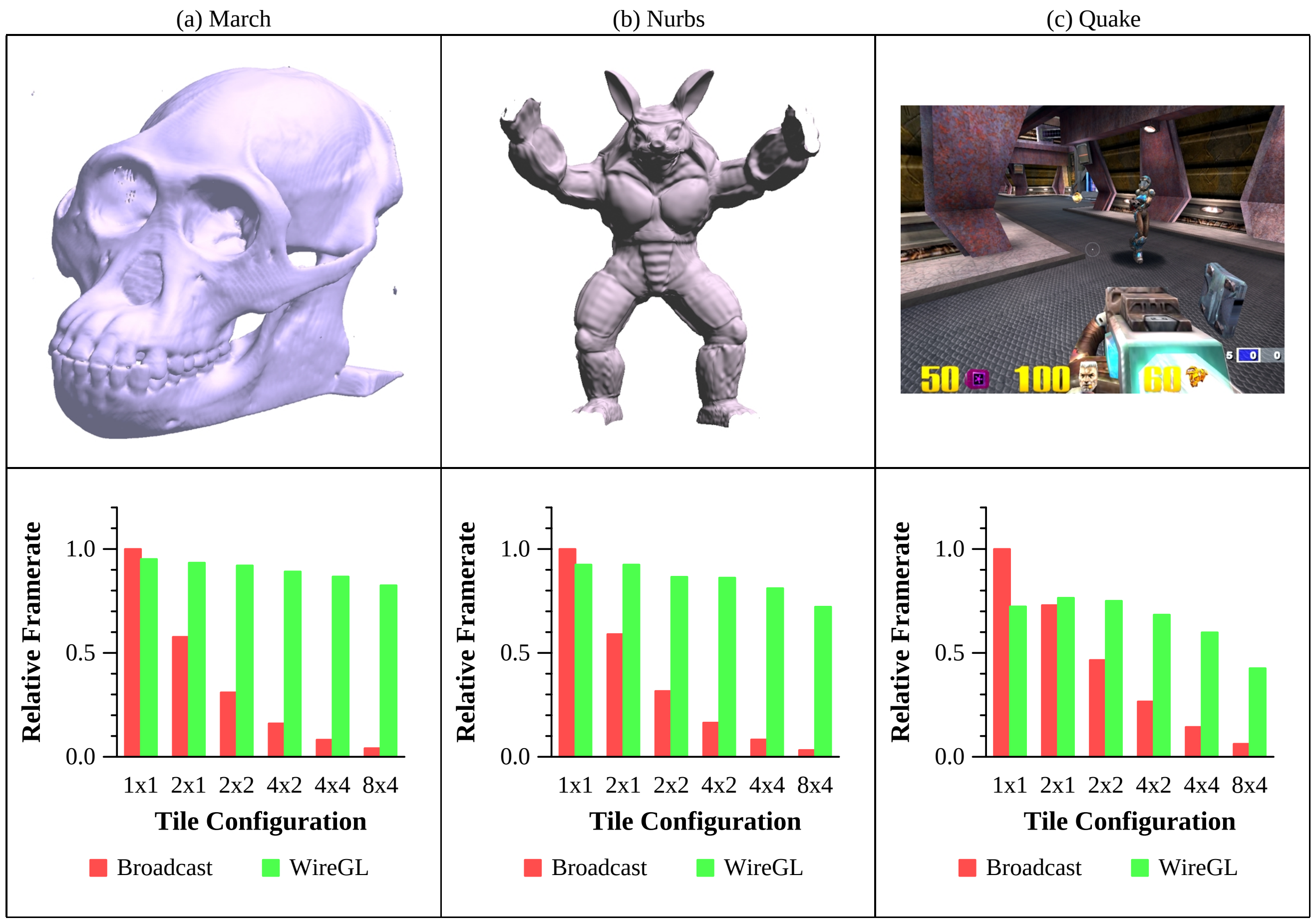


Figure 7: Frame rate comparisons between WireGL and broadcast. The broadcast frame rate falls off quickly as expected, while WireGL's frame rate falls much more slowly. The more rapid falloff of `Quake` is due to larger primitive sizes, which need to be transmitted to many servers. The actual frame rates for each application in the 1x1 broadcast configuration are 0.25, 4.93, and 48.1 frames per second, respectively. The 8x4 configuration forms a 25 megapixel display.

projectors, and we expect the network efficiency to decrease because those primitives must be sent multiple times. Figure 6 shows the network efficency for `March`. Note that this corresponds almost exactly to the relative framerates shown in figure 7(a), as expected.

`Quake`, shown in figure 7(c) does not scale as well as the other two examples, but WireGL still has a large advantage over broadcast. `Quake` makes use of complex textures rather than incorporating additional geometry to represent detail, and as a result, issues many large polygons. These large polygons cause the network efficiency of `Quake` to suffer because they must be sent to more than one server. Evidence of this can be seen in the larger configurations.

Furthermore, `Quake` has been optimized to minimize the number of OpenGL state changes by rendering all of the elements in the scene with similar state configuration at the same time. While this can significantly improve performance on conventional graphics hardware, sorting by state can be detrimental to WireGL's bucketing scheme, since geometry which has similar state does not necessarily exhibit good spatial locality. We address this problem by bucketing more frequently. Despite these limitations, figure 7(c) clearly demonstrates that our system can still efficiently manage `Quake` rendering and is superior to basic broadcasting.

5 Conclusions and Future Work

We have described WireGL, a system for providing output resolution scalability to unmodified OpenGL applications. We have been using WireGL since 1999 to research novel interaction metaphors for large format displays. We have demonstrated that output scalability can be achieved by tracking graphics state and sorting geometry into buckets. The key to performance when rendering remotely is managing the data that is sent between machines, since the network is (today) the slowest link in the entire system.

As the display is scaled larger, each individual graphics accelerator in the cluster is receiving less of the scene, and each incoming network link is getting less traffic. So while the overall performance of the client application remains nearly constant, the efficiency of each element in the cluster decreases. The natural next step in this research is to address "input scalability." If we could keep each graphics and network card in the cluster busy, the total rendering rate of an application would increase with the size of the cluster, which is the ultimate goal of our research.

To achieve this, WireGL will incorporate the OpenGL parallel API extensions proposed by Igehy, Stoll and Hanrahan [7], which allow a node in a parallel application to express the ordering requirements of its outgoing stream with respect to its peers' streams. We can then run parallel applications on the cluster, and each node can submit its own portion of the total scene to be rendered. By changing the communication from a one-to-many to a many-to-many model, we hope to realize a fully scalable rendering system. We have already incorporated a preliminary implementation of the parallel API with promising results.

Once input scalability is in place, we intend to add dynamic configurability to our cluster rendering solution. For example, depending on available network bandwidth and the nature of the data being rendered, it may be more efficient to render the stream locally and transmit an image-based representation of the commands rather than the commands themselves. We intend to add a layer of configurability so that connections between graphics streams and their various transformation engines (render, sort, extract image, composite, etc.) can be easily expressed. An ideal system would be able to choose these configurations automatically while the application is running. For example, we had to instruct WireGL to sort geometry more frequently to improve the performance of `Quake` in figure 7(c); it would be desirable for WireGL to make such decisions automatically based on the observed behavior of a running application.

Acknowledgments

The authors would like to thank Chris Niederauer for his help in setting up the Chromium cluster. This work was supported by US Department of Energy contract B504665.

References

[1] Nanette Boden, Danny Cohen, Robert Felderman, Alan Kulawik, Charles Seitz, Jakov Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, pages 29–36, February 1995.

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

[3] Michael Cox, Steven Molnar, David Ellsworth, and Henry Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Algorithms*, pages 23–32, July 1994.

[4] Caroline Cruz-Neira, Daniel J. Sandin, and Tom DeFanti. Surround-screen projection-based virtual reality: The design and implementation of the CAVE. *Proceedings of SIGGRAPH 93*, pages 135–142, August 1993.

[5] M. Czernuszenko, D. Pape, D. Sandin, T. DeFanti, G. Dawe, and M. Brown. The ImmersaDesk and InfinityWall projection-based virtual reality displays. In *Computer Graphics*, May 1997.

[6] Greg Humphreys and Pat Hanrahan. A distributed graphics system for large tiled displays. *IEEE Visualization '99*, pages 215–224, October 1999.

[7] Homan Igehy, Gordon Stoll, and Pat Hanrahan. The design of a parallel graphics interface. *Proceedings of SIGGRAPH 98*, pages 141–150, July 1998.

[8] Intel Corporation. *Intel Architecture Software Developer's Manual*, chapter 14, pages 9–10. 1999.

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

[10] Carl Mueller. The sort-first rendering architecture for high-performance graphics. *1995 Symposium on Interactive 3D Graphics*, pages 75–84, 1995.

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

[12] University of Minnesota. PowerWall. http://www.lcse.umn.edu/research/powerwall/powerwall.html.

[13] OpenGL Architecture Review Board. *OpenGL Reference Manual: the Official Reference Document for OpenGL, Release 1*. Addison–Wesley, 1993.

[14] Rudrajit Samanta, Jiannan Zheng, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh. Load balancing for multi-projector rendering systems. *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 107–116, August 1999.

[15] Daniel R. Schikore, Richard A. Fischer, Randall Frank, Ross Gaunt, John Hobson, and Brad Whitlock. High-resolution multi-projector display walls and applications. *IEEE Computer Graphics Applications*, July 2000.

[16] John G. Torborg. A parallel processor architecture for graphics arithmetric operations. *Proceedings of SIGGRAPH 87*, pages 197–204, 1987.