A STREAM PROCESSING APPROACH TO INTERACTIVE GRAPHICS ON CLUSTERS OF WORKSTATIONS

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Greg Humphreys

July 2002

Abstract

Recent advances in VLSI technology have made it possible to implement the entire graphics pipeline on a single chip. This departure from monolithic multi-chip graphics systems has made high performance 3D graphics inexpensive and commonplace. Despite the very high performance achievable on a modern graphics accelerator, these systems have serious scalability limitations. Typically their performance or utility is limited by the serial interface between the host processor and the graphics subsystem, or the limited output resolution they provide. In this thesis, we examine the potential for constructing scalable graphics systems from multiple such graphics accelerators housed in nodes of a cluster of off-the-shelf workstations.

The system we present, called Chromium, uses the notion of *stream processing* to achieve scalability while retaining maximal flexibility. Using Chromium's building blocks, parallel graphics systems can be build that scale in both output resolution and input rate. In addition, because the graphics commands being manipulated conform to industry-standard 3D graphics API, Chromium can use unmodified programs as stream sources, allowing existing applications to be retargeted to new display environments transparently. By using a standard API, existing serial applications can often be parallelized with very little effort. Chromium provides extensions to this API to allow parallel applications to constrain the order of execution of their commands with respect to those of their peers.

Acknowledgements

I would like to thank my advisor, Pat Hanrahan, for his guidance, support, my undergraduate dorm couch, not throwing me out of the Princeton Graphics Lab for loitering, and especially for trusting me to head a risky new research project even before I had arrived at Stanford. I could not have hoped for a better mentor. The other members of my reading committee, Bill Dally and Samuel P. Uselton (SPU), gave generously of their already overcommited time to help me produce this thesis, for which I am very grateful.

Being at Stanford has been an even better experience than I had initially hoped for. The Stanford Graphics Lab is an amazing (possibly unparalleled) place, and the people I have worked with over the years have made being a graduate student truly enjoyable, especially Maneesh Agrawala, Ian Buck, Milton Chen, James Davis, Matthew Eldridge, Matt Everett, François Guimbretière, Mike Houston, Homan Igehy, Brad Johanson, Tamara Munzner, Ren Ng, Chris Niederauer, John Owens, Matt Pharr, Kekoa Proudfoot, Tim Purcell, Ravi

Rammamorthi, Jeff Solomon, Gordon Stoll, Maureen Stone, and Diane Tang.

I would also like to thank the members of the Stanford community who have helped me indulge my unhealthy obsession with duplicate bridge, especially Adam Meyerson, who taught me that "recursive" can be a dirty word. Also, without Adam, I never would have produced this textbook auction: 2♣ – 3♣ – 3♠ – 4♠ – 5♣ – 6♣. Losing 8 IMPS to Billy Miller for getting +760 in 2♢ redoubled making with an overtrick was one of the highlights of my last five years, so I guess I must also thank Ted Hwa and Qi Sun, the perpetrators at the other table. Way to go guys.

Kelly Shaw, Liadan O'Callaghan, and Aarati Parmar have been exceptional friends, always making sure I didn't take my schoolwork too seriously (which I suppose was not such a monumental task). Without Paul Martino, I wouldn't have participated in nearly as many boondoggles as I have, wouldn't know anything about game shows from the 70's, and wouldn't know nearly as much as I do about cheating the IRS. He is the perfect partner in crime, although I can't help feeling that not quitting school to create bridal registry kiosks for Sears was the big missed opportunity of my career. I suppose only time will tell.

Several people from outside Stanford have contributed to Chromium, making it ironically more robust and fragile at the same time. The main culprits were Randy Frank and Sean Ahern from Livermore, Peter Kirchner and Jim Klosowski from IBM T.J. Watson, and Brian Paul and Alan Hourihane from RedHat. I am also grateful to Nick Triantos and Gareth Hughes from NVIDIA for their support and access to inside information.

No one is more deserving of my thanks than Jessica Humphreys. Her love and support have been the most important factors in my success at Princeton and Stanford, and I suspect that will continue to be true wherever I am in the future. I'm the luckiest guy around.

For Jessica

Contents

List of Tables

List of Figures

1. Introduction

The performance of consumer graphics hardware is increasing at such a fast pace that a large class of applications can no longer utilize the full computational potential of the graphics processor. This is largely due to the serial interface between the host and the graphics subsystem. Recently, clusters of workstations have emerged as a viable option to alleviate this bottleneck. However, cluster rendering systems have been mostly focused on providing specific algorithms, rather than a general mechanism for enabling interactive graphics on clusters. The goal of our work is to allow applications to more easily utilize the aggregate rendering power of a collection of commodity graphics accelerators housed in a cluster of workstations, without imposing a specific scalability algorithm that may not meet an application's needs.

To achieve this goal, we have designed and built a system that provides a generic mechanism for manipulating streams of graphics API commands. This system, called Chromium, provides an industry-standard graphics API called OpenGL that virtualizes the disjoint rendering resources present in a cluster. Chromium can be used as the underlying mechanism for any existing cluster-graphics algorithm by having the algorithm use the OpenGL API to move geometry and imagery across a network as required. In addition, existing OpenGL-based applications can use a cluster with very few modifications; in some cases, the application does not even need to be recompiled. Compatibility with existing applications will accelerate the adoption of rendering clusters and high resolution displays, encouraging the development of new applications that exploit resolution and parallelism.

Chromium's stream processors are implemented as modules that can be interchanged and combined in an almost completely arbitrary way. By modifying the configuration of these stream processors, we have built sort-first and sort-last parallel graphics architectures that can, in many cases, support the same applications without recompilation. Unlike previous work, our approach does not necessarily require that any geometry be moved across a network (although this may be desirable for load-balancing reasons). Instead, applications can issue commands directly to locally housed graphics hardware, thereby achieving the node's full advertised rendering performance. Because our focus is on clusters of commodity components, we only consider architectures that do not require communication between stages in the pipeline that are not normally exposed to an application. For example, a sort-middle architecture, which requires communication between the geometry and rasterization stages, is not a good match for our system.

Chromium's stream processors can be extended programmatically. This added flexibility allows Chromium users to solve more general problems than just scalability, such as integration with an existing user interface, stylized drawing, or application debugging.

This extensibility is one of Chromium's key strengths. Because we simply provide a programmable filter mechanism for graphics API calls, Chromium can implement many different underlying algorithms. This model can be thought of as an extension of Voorhies's virtual graphics pipeline, which insulates applications from the details of the underlying implementations of a common API [Voorhies *et al.*, 1988].

Portions of the work presented in this thesis were described in previous publications. The design of the `tilesort` SPU (described in chapter 4) was originally published at IEEE Supercomputing [Humphreys *et al.*, 2000]. The state tracking system used in Chromium is based in large part on joint work between Ian Buck and myself, which was presented at the SIGGRAPH/Eurographics Graphics Hardware Workshop [Buck *et al.*, 2000]. The parallel interface to our sort-first architecture (described in chapter 5) was presented at SIGGRAPH [Humphreys *et al.*, 2001]. Finally, the flexible stream processing mechanism underlying all this technology, along with some of the results in chapter 6 will appear at SIGGRAPH [Humphreys *et al.*, 2002].

2. Related Work

2.1 Networked Graphics

In the area of remote rendering, GLX and X windows stand out as the two most widely used solutions. GLX is a wire protocol for OpenGL that allows an application to transmit a simple packed representation of the OpenGL command parameters to a server to be executed on behalf of the client [Segal and Akeley, 1999; Kilgard, 1996]. GLX performs the minimum amount of state tracking required for correctness; it tracks pixel formats for packing and unpacking pixel data, as well as vertex array state. All other state commands are sent directly to the server. The design of GLX was motivated by a desire to interface with the X protocol, not by a need for high-speed remote rendering.

X-Windows provides remote 2D graphics capabilities. Earlier versions of X were stateless (every drawing command carried with it all necessary state information), which was very inefficient for remote drawing. The latest version (R11) keeps all state information on the server. A single protocol request is used to change a subset of the state at once [Nye, 1995]. X-Windows has the same model of network packet generation as GLX; each state command (e.g., `XSetForeground`) causes a state changing packet to be generated. Note that if multiple state changing commands are made in a row, the X client libraries can sometimes collapse these commands into a single protocol packet.

Solutions for remote desktop access also fall into this category. The VNC system from AT&T Research provides remote access to a Microsoft Windows desktop or an X-Windows session by transmitting compressed image rectangles. Network traffic is kept to a minimum by only transmitting portions of the framebuffer that are changing, and by combining multiple such changed regions into a single packet. This design has the nice property that the client is extremely portable; it only needs to refresh a bitmap and transmit input events. Microsoft NetMeeting also provides remote access to a Windows desktop. In contrast to VNC, Netmeeting works by transmitting the Generic Device Interface (GDI) calls from the desktop to the remote client. This has the advantage of transmitting much less information than VNC, since the GDI command stream is typically much more compact than the framebuffer itself[1], but it makes the client highly non-portable.

One advantage of an image based approach to remote graphics is that the client need not have any rendering capabilities. In fact, remote framebuffer access is a convenient way to access high-performance or high-quality rendering capabilities from a lower power desktop. This approach was used by SGI's GLR and "VizServer" products [SGI Vizserver, 1999], which each allowed users to submit rendering commands to a high-performance graphics workstations and receive the resulting images as streaming video.

[1]This will depend on the total size of the image being rendered, as well as any compression strategy being employed on the image or command stream. In fact, it is an interesting problem to consider the situations when remote rendering becomes more efficient by sending the final rendered image instead of the high-level commands required to produce that image.

2.2 Parallel Graphics

There have been a number of efforts to improve the performance of a graphics system by leveraging multiple graphics engines and processors. IRIS Performer provides a scene-graph library that can parallelize scene graph traversal and also take advantage of multiple graphics accelerators in a single SMP workstation [Rohlf and Helman, 1994]. Performer requires the application to use a retained-mode interface to achieve speedups. Performer also exposes the number of graphics pipes to the application and requires the programmer to manage that parallelism explicitly.

Another approach to scalable rendering is to build standalone accelerators that have internal parallelism. The SGI RealityEngine is an example of such an accelerator [Akeley, 1993]. The RealityEngine uses a number of geometry units and rasterization units to transform and rasterize geometry in parallel. Because it uses bus-based communication to distribute work and provide strict ordering semantics, its internal scalability is limited by the difficulty of building faster and wider busses. In addition, the RealityEngine and its successor, the InfiniteReality, suffer from a problem common to modern PC graphics accelerators: the interface bottleneck. Today's accelerators are so fast that it is difficult or impossible to drive them at full rate through an immediate-mode interface.

Pomegranate is a fully scalable graphics system based on point-to-point communication [Eldridge *et al.*, 2000; Eldridge, 2001]. Pomegranate provides scalability at all points in the graphics pipeline, most notably the input interface. This is one of the first modern graphics architectures to address the problem of interface limitation by allowing multiple concurrent processes to submit graphics commands to the hardware simultaneously. Although the individual building blocks of Pomegranate are all found in current graphics accelerators, the very high bandwidths needed for communication in later stages of the pipeline make it impractical to separate multiple pipelines across nodes in a cluster.

PixelFlow is an image-composition architecture designed to support multiple inputs from a parallel host [Molnar *et al.*, 1992]. PixelFlow provided an OpenGL interface with a few mandatory extensions [Eyles *et al.*, 1997], but could not provide any guarantees about order of execution, making it difficult to write scalable applications that required ordering (e.g., back-to-front transparency).

Most parallel hardware architectures are standalone accelerators with internal parallelism. These architectures provide good scalability up to some predetermined limit and for a certain class of applications. Each architecture can be classified according to the point in the graphics pipeline at which it redistributes data [Molnar *et al.*, 1994].

2.3 Cluster Graphics

Clusters have long been used for parallelizing traditionally non-interactive graphics tasks such as ray-tracing, radiosity (e.g., [Funkhouser, 1996; Recker *et al.*, 1990]), and volume rendering (e.g., [Ma *et al.*, 1994; Giertsen and Peterson, 1993]). Other cluster-parallel rendering efforts have largely concentrated on exploiting inter-frame parallelism rather than trying to make each individual frame run faster [Pixar, 1998]. We are interested in enabling fast, interactive rendering on clusters, so these techniques tend to be at most loosely applicable to our domain.

In the last few years, there has been growing interest in using clusters for interactive rendering tasks. Initially, the goal of these systems was to drive large tiled displays. Humphreys and Hanrahan described an early system designed for 3D graphics [Humphreys and Hanrahan, 1999]. Although the system described in that paper ran on an SGI Infinite-Reality, it was later ported to a cluster of workstations. At first, their cluster-based system, called WireGL, only allowed a single serial application to drive a tiled display over a network [Humphreys *et al.*, 2000]. WireGL used traditional sort-first parallel rendering techniques to achieve scalable display size with minimal impact on the application's performance. The main drawback of this system was its poor utilization of the graphics resources available in a cluster. Because it only focused on display resolution, applications would rarely run faster on a cluster than they would locally.

Other approaches focused on scalable rendering rates. Samanta et al. described a cost-based model for load-balancing rendering tasks among nodes in a cluster, eventually redistributing the resulting non-overlapping pixel-tiles to drive a tiled display [Samanta *et al.*, 2000a; Samanta *et al.*, 1999]. They then extended this technique to allow for tile overlap, creating a hybrid sort-first and sort-last algorithm that could effectively drive a single display [Samanta *et al.*, 2000b]. All of these algorithms required the full replication of the scene database on each node in the cluster, so further work was done to only require partial replication, trading off memory usage for efficiency [Samanta *et al.*, 2001]. Although these papers present an excellent study of differing data-management strategies in a clustered environment, they all provide *algorithms* rather than *mechanisms*. Applying one of these techniques to a big-data visualization problem would require significant reworking of existing software.

A different approach to dataset scalability was taken by Humphreys et al. when they integrated a parallel interface into WireGL [Humphreys *et al.*, 2001]. By posing as the system's OpenGL driver, WireGL intercepts OpenGL commands made by an application (or multiple applications), and generates multiple new command sequences, each represented in a compact wire protocol. Each sequence is then transmitted over a network to a different server. Those servers manage image tiles, and execute the commands encoded in the streams on behalf of the client. Finally, the resulting framebuffer tiles are extracted and transmitted to a compositing server for display. Ordering between streams resulting from a parallel application is controlled using the parallel immediate mode graphics extensions proposed by Igehy et al [Igehy *et al.*, 1998]. WireGL can use either software-based image reassembly or custom hardware such as Lightning-2 [Stoll *et al.*, 2001] to reassemble the resulting image tiles and form the final output. This approach to cluster rendering allows existing applications to be parallelized easily, since it is built upon a popular, industry-standard API. However, by imposing a sort-first architecture on the resulting application, it can be difficult to load-balance the graphics work. Load-balancing is usually attempted by using smaller tiles, but this will tend to cause primitives to overlap more tiles, resulting in additional load on the network and reduced scalability. More fundamentally, WireGL requires that all geometry be moved over a network every frame, but today's networks are not fast enough to keep remote graphics cards busy.

2.4 State Management and Context Switching

State management is another critical feature of parallel rendering systems. Torborg describes a parallel graphics system in which graphics state is broadcast to each geometry processor [Torborg, 1987]. This was practical because the architecture being described used a shared bus to connect the geometry processors, so broadcasts were inexpensive. The RealityEngine system broadcasts predetermined "infrequent" state commands, but maintains a copy of frequently changed commands (e.g. color, normal, texture coordinates) near the host interface. Those commands are attached to issued geometry, preventing needless broadcast of rapidly changing state.

Michael Cox outlines a near optimal algorithm in his Ph.D. thesis for state management in a parallel implementation of Pixar's PhotoRealistic RenderMan [Cox, 1995]. His approach is conservative (i.e., it may send more data than absolutely necessary) because of the undecidability of computing state element equality in RenderMan. David Ellsworth et al. describe a system which only sends relevant state elements to rendering nodes [Ellsworth *et al.*, 1990]. Cox points out that Ellsworth's system is restricted to supporting retained mode applications. In addition, Ellsworth's algorithm still requires the broadcast of certain state elements (e.g., matrix transformations).

Finally, context switching is addressed by many papers on graphics hardware. The Apollo DN10000 supported multiple graphics contexts in hardware and could perform a context switch in 16 microseconds if the target graphics context was in its 6-context cache [Voorhies *et al.*, 1988]. Akeley and Jermoluk identified the need for fast context switching in their paper on high performance polygon rendering [Akeley and Jermoluk, 1988]. Despite apparent agreement in the hardware community on the need for fast context switching, most hardware implementations can switch contexts only a few thousand times per second. In many cases, these systems are limited by the design of the window system in which they must operate. WireGL's efficient context differencing operation provides very fast context switching performance without the need for hardware support [Buck *et al.*, 2000]. This allows multiple applications to share a remote display with very little context switching penalty, a crucial feature for supporting parallel remote rendering.

2.5 Image Composition

In the area of hardware for image composition from multiple traditional graphics accelerators, Stanford's Lightning project provided a distributed framebuffer for use in a multiprocessor workstation. Compaq Research has developed a system called Sepia for performing image composition using the Servernet-II networking technology [Heirich and Moll, 1999; Moll *et al.*, 1999]. Both of these systems use an image composition network connected to multiple rendering nodes via a PCI interface card. While these systems avoid some of the data transfers required by a pure software approach, it is still necessary for pixel data to be read from the framebuffer using the graphics I/O port and transferred to the image composition system by PCI. Bandwidth on these paths is often a critical resource for parallel visualization applications.

Blanke et al. describe the Metabuffer, a system for performing sort-last parallel rendering on a cluster using DVI to extract color and depth [Blanke *et al.*, 2000]. Unlike Sepia, the Metabuffer does not require pixel data to be transferred to the image composition network over the internal system bus, where bandwidth is often a critical resource for parallel visualization applications.

The Metabuffer is similar to Lightning-2 [Stoll *et al.*, 2001], a DVI-based pixel routing network. A Lightning-2 system is comprised of one or more pixel routing boards, which are tiled in a two-dimensional array. Each row of this tiling provides four DVI inputs and each column provides eight DVI outputs. Each input pixel is interpreted by programmable processors on the Lightning-2 boards. These processors are controlled by 48-bit wide "strip headers", which are drawn directly into the framebuffer by an Lightning-2-aware application. Typical uses of Lightning-2 are tile reassembly and depth compositing. Unlike Sepia, Lightning-2 and the Metabuffer do not require pixel data to be transferred to the image composition network over the internal system bus, where bandwidth is often a critical resource for parallel visualization applications.

The Hewlett-Packard Visualize *fx* architecture uses a custom network to composite the results of multiple graphics accelerators [Cunniff, 2000]. Sony's GSCube, demonstrated at SIGGRAPH 2000, combines the outputs of multiple Playstation2 graphics systems using a custom network, and supports both sort-first and image composition modes of operation.

2.6 Large Displays

The Interactive Mural was a graphics system for virtualizing tiled displays, with a focus on enabling research on interaction with large displays driven by large multiprocessors [Humphreys and Hanrahan, 1999]. This system provided an OpenGL interface to the display, but it required modifications to existing applications. Mural-aware applications used a custom API to create transparent "layers", which could be stacked and positioned similarly to windows. The Mural API also provided an event queue for interaction.

There are many ways to virtualize a tiled display system in software. MacOS and, more recently, Microsoft Windows have support for extending the desktop onto multiple monitors connected to a single computer system. In these cases, the device driver takes care of managing the screen real estate and the potentially distributed framebuffer memory. Similarly, the latest release of the X Window system contains the XINERAMA extension, which allows multiple displays to be combined into one large virtual desktop. The Silicon Graphics InfiniteReality system allows a single framebuffer to drive multiple displays through one large logical X server [Montrym *et al.*, 1997].

A more general approach is taken by DEXON Systems' DXVirtualWall, which provides tiled displays that support either X Windows or Microsoft Windows. Clients connect to a display proxy that broadcasts the display protocol to multiple display servers, each of which offsets the coordinates to display its own small portion of the larger desktop. Another use for protocol proxies of this nature is to duplicate a single display across several remote displays, as in Brown University's XmX project. These proxy-based systems do not currently support any high performance 3D graphics API such as OpenGL or Direct3D, and do not handle overlapping displays. Additionally, as the number of displays gets very large, the number of concurrent redraws exerts increasing pressure on the network, causing performance to suffer.

IRIS Performer provides an API for managing multiple graphics pipes within a single application [Rohlf and Helman, 1994]. However, Performer is designed for a single application driving the entire display, and most compelling Performer demos are full screen, immersive walkthrough applications. Running multiple Performer applications simultaneously incurs great context switching overhead, resulting in pronounced performance degradation. In addition, Performer is designed around hierarchically defined “scene graphs”; arbitrary OpenGL applications do not receive much of the benefit of Performer. Finally, Performer makes little attempt to virtualize the configuration of a multiple-pipe system; applications need to be aware of the number of available pipelines and use them explicitly.

The University of Minnesota’s PowerWall uses the output of multiple graphics supercomputers to drive multiple outputs, creating a large tiled display for high resolution video playback and immersive applications [PowerWall, 1994]. The University of Illinois at Chicago extended this system to support stereo display and user tracking in the InfinityWall system [Czernuszenko *et al.*, 1997]. Unlike our system, neither of these displays overlaps its projectors. These systems are designed to facilitate a single full-screen application, which is often an immersive virtual reality system. More expensive custom hardware solutions are available as well. Panoram Technologies has a suite of hardware devices designed to create large tiled displays. Their “Integrator”, a special analog feathering box, blends the outputs of individual projectors. Because of the increasing performance of graphics systems, it is now practical to perform this blending step in software.

2.7 Graphics Interfaces

To provide ordering constraints, Pomegranate used the parallel OpenGL API proposed by Igehy, Stoll, and Hanrahan [Igehy *et al.*, 1998]. This API (hereinafter referred to as the "Parallel API") extends OpenGL with traditional synchronization primitives (barriers and semaphores). These primitives allow multiple simultaneous processes to express the ordering required between their graphics streams to produce a single image. The key advantage of the Parallel API's synchronization primitives is that they do not require the issuing application to block. Instead, the synchronization primitives are encoded into the graphics stream, and their implied ordering is obeyed by the graphics system when a context switch occurs. This ability is critical for supporting parallel visualization applications that may have either partial or strict ordering requirements, particularly those that require the use of alpha-blending to render partially transparent geometry. Graphics barriers and semaphores are implicitly created in a global namespace, similarly to the way OpenGL allows texture objects and display lists to be shared between contexts. A graphics context may enter a barrier at any time by calling `glBarrierExec(name)`. Semaphores can be acquired and released with `glSemaphoreP(name)` and `glSemaphoreV(name)`, respectively. Chromium provides this same API, and is the first implementation of the Parallel API in a hardware-accelerated architecture.

Graphics APIs can provide a low-level resource abstraction such as OpenGL, or a high-level abstraction such as a scene graph library. Scene graphs and other high-level interfaces are attractive because global information can be used to automatically parallelize rendering or perform fast culling. IRIS Performer provides parallel traversal of a retained-mode scene graph, and can also take advantage of multiple graphics pipelines in a single SMP [Rohlf and Helman, 1994]. Samanta et al. describe a novel screen subdivision algorithm for load-balanced rendering of a scene graph that has been replicated across the nodes of a cluster [Samanta *et al.*, 2000a; Samanta *et al.*, 1999]. However, not all visualization tools can conveniently use a scene graph, because their data may be unstructured and time-varying. Another significant drawback of scene graphs is the lack of a standardized scene graph API.

Chromium provides the OpenGL API to each node in a cluster. The decision to use OpenGL for specifying graphics data has several advantages over using a custom API. First, we can run an unmodified application on a single node in our cluster without re-compiling it. Also, if we have access to a large display wall, we can easily interact with resolution-limited datasets that can take advantage of the larger display area. SGI also provides a library called “Multipipe” that intercepts OpenGL commands and allows unmodified applications to render across multiple graphics accelerators, providing increased output resolution [SGI Multipipe, 2000].

Many applications are must be parallelized to achieve speedup. Using Chromium, many existing serial OpenGL applications can be parallelized with minor changes to the inner drawing routines. In particular, applications that render large geometric datasets using the depth buffer to resolve visibility can simply partition their dataset across the nodes of the cluster, and have each node render its portion as before. Because such an application has almost no intra-frame ordering requirements, achieving parallelism is straightforward.

2.8 Stream Processing

Continual growth in typical dataset size and network bandwidth has made stream-based analysis a hot topic for many different disciplines, such as telephone record analysis [Cortes *et al.*, 2000], multimedia, rendering of remotely stored 3D models [Rusinkiewicz and Levoy, 2001], database queries [Babu and Widom, 2001], and theoretical computer science [O'Callaghan *et al.*, 2002]. In these domains, streams are an appropriate computational primitive because large amounts of data arrive continuously, and it is impractical or unnecessary to retain the entire data set. In the broadest sense, a stream is an ordered sequence of records. Applications designed to operate on streams only access the elements of the sequence in order, although it is possible to buffer a portion of a stream for more global analysis. Any stream processing algorithm must operate on a potentially infinite input set using only finite resources.

Many of the traditional techniques used to solve problems in computer graphics can be thought of as stream processing algorithms. Immediate-mode rendering is a classic example. In this graphics model, an unbounded sequence of primitives is sent one at a time through a narrow API. The graphics system processes each primitive in turn, using only a finite framebuffer (and possibly texture memory) to store any necessary intermediate results. Because such a graphics system does not have memory of past primitives, its computational expressiveness is limited[2]. Owens et al. implemented an OpenGL-based polygon renderer on Imagine, a programmable stream processor [Owens *et al.*, 2000]. Using Imagine, they achieved performance that is competitive with custom hardware while

[2]Because most graphics APIs have some mechanism to force data to flow back towards the host (i.e., `glReadPixels`), graphics hardware is actually not a purely feed-forward stream processor. This fact has been exploited to perform more general computation using graphics hardware [Peercy *et al.*, 2000; Proudfoot *et al.*, 2001], and extensions to the graphics pipeline have been proposed to further generalize its computational expressiveness [Mark and Proudfoot, 2001].

enabling greater programmability at each stage in the pipeline.

Mohr and Gleicher demonstrated that a variety of stylized drawing techniques could be applied to an unmodified OpenGL application by only analyzing and modifying the stream of commands [Mohr and Gleicher, 2001]. They intercept the application's API commands by posing as the system's OpenGL driver, in exactly the same way Chromium obtains its command source. Although some of their techniques require potentially unbounded memory, some similar effects can be achieved using Chromium and multiple nodes in a cluster.

3. The Chromium Architecture

Chromium is a software system for enabling scalable interactive graphics on clusters of workstations. Its design centers around the ability to efficiently manipulate streams of graphics API commands. We have chosen to make Chromium as non-invasive as possible; that is, we intercept calls made to an existing graphics API (OpenGL) rather than providing a custom API of our own. This choice has the immediate consequence that any existing application that uses OpenGL can run on top of Chromium without modification. This allows us to run an application in a new display environment such as a tiled display wall.

Chromium also provides some extensions to the OpenGL API to facilitate the development of parallel applications. In particular, Chromium supports the Parallel API proposed by Igehy, Stoll and Hanrahan. [Igehy *et al.*, 1998]. These extensions provide a parallel interface to the system, eliminating the traditional bottleneck at the command processing stage of the graphics pipeline. Because the Parallel API is a simple extension to the OpenGL specification, existing serial OpenGL applications can be parallelized with a minimum of effort. Typically applications need only partition the model to be rendered and provide a minimal amount of synchronization (if any) through the Parallel API.

In this chapter, we describe the architecture of Chromium, and how command stream transformations are expressed and implemented. In the chapters that follow, we present a few important configurations of Chromium, including one for achieving scalable display resolution, and two for scaling rendering performance.

3.1 Cluster Nodes

Chromium users begin by deciding which nodes in their cluster will be involved in a given parallel rendering run, and what communication will be necessary. This is specified to a centralized configuration system as a directed acyclic graph. Nodes in this graph represent computers in a cluster, while edges represent network traffic. Each node is conceptually divided into two parts: a *transformation* portion and a *serialization* portion.

The transformation portion of a node takes a single stream of OpenGL commands as input, and produces zero or more streams of OpenGL commands as output. The mapping from input to output is completely arbitrary. The output streams (if any) are sent over a network to another node in the cluster to be serialized and transformed again. Stream transformations are described in greater detail in section 3.2.

The serialization portion of a node consumes one or more OpenGL streams, each with its own associated graphics context, and produces a single OpenGL stream as output. This task is analogous to the scheduler in a multitasking operating system; the serializer chooses a stream to “execute”, and copies that stream to its output until the stream becomes “blocked”. It then selects another input stream, performs a context switch, and continues copying. Streams block and unblock via the Parallel API. Recall that these synchronization primitives do not block the issuing process, but rather encode ordering constraints that will be enforced by the serializer. Because the serializer may have to switch between contexts very frequently, we use a hierarchical OpenGL state tracker, described in section 3.7. In brief, our state representation permits the efficient computation of the difference between two graphics contexts, allowing for fine-grained sharing of rendering resources.

A node's serializer can be implemented in one of two ways. Graph nodes that have one or more incoming edges are realized by Chromium's network server, and are referred to as *server nodes*[1]. Servers manage multiple incoming network connections, interpreting messages on those connections as packed representations of OpenGL streams.

On the other hand, nodes that have no incoming edges must generate their (already serial) OpenGL streams programmatically. These nodes are called *client nodes*. Clients obtain their streams from standalone applications that use the OpenGL API. Chromium's application launcher causes these programs to load our OpenGL shared library on startup. Chromium's OpenGL library injects the application's commands into the node's stream transformer, so the application does not have to be modified to initialize or load Chromium. If there is only one client in the graph, it will typically be an unmodified off-the-shelf OpenGL application. For graphs with multiple clients, the applications will have to specify the ordering constraints on their respective streams.

[1]The use of "client" and "server" in networked graphics systems such as X-Windows is often a source of confusion. For our purposes, we consider a server to be an entity that provides a service, and a client to be an entity that connects to a server to take advantage of that service. In Chromium, servers provide an implementation of the OpenGL API, and clients connect to those servers to remotely make calls to that API. Note that because Chromium supports an arbitrary DAG of cluster nodes, servers can be clients of other servers.

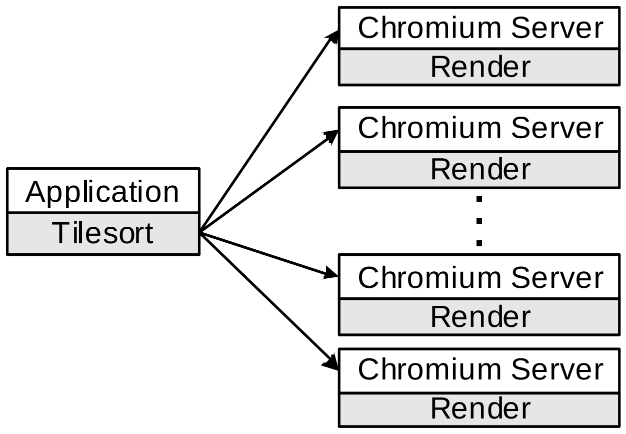


Figure 3.1: A simple Chromium configuration. In this example, a serial application is made to run on a tiled display using a sort-first stream processor called `tilesort`.

3.2 OpenGL Stream Processing

Stream transformations are performed by “Stream Processing Units”, or SPUs. SPUs are implemented as dynamically loadable libraries that provide the OpenGL interface, so each node’s serializer will load the required libraries at run time and build an OpenGL dispatch table. SPUs are normally designed as generically as possible so they can be used anywhere in a graph.

A simple example configuration is shown in figure 3.1. The client loads the `tilesort` SPU, which incorporates all of the sort-first stream processing logic from the WireGL system [Humphreys *et al.*, 2000]. The `tilesort` SPU is described in detail in chapter 4. The servers use the `render` SPU, which dispatches the incoming streams directly to their local graphics accelerators. This configuration has the effect of running the unmodified client application on a tiled display using sort-first stream processing. Notice that the graph edges originate from the `tilesort` SPU, not the application itself. This convention is used because the SPU manages its own network resources and originates server connections.

3.3 SPU Chains

A node's stream transformation need not be performed by only a single SPU; serializers can load a linear chain of SPUs at run time. During initialization, each SPU receives an OpenGL dispatch table for the next SPU in its local chain, meaning simple SPUs can be chained together to achieve more complex results. Using this feature, a SPU might intercept and modify (or discard) calls to one particular OpenGL function and pass the rest untouched to its downstream SPU. This allows a SPU, for example, to adjust the graphics state slightly to achieve a different rendering style.

One example of such a SPU is a "wireframe" filter. This SPU issues a `glPolygonMode` call to its downstream SPU at startup to set the drawing mode to wireframe. It then passes all OpenGL calls directly through except `glPolygonMode`, which it discards, preventing the application from resetting the drawing mode. Note that Chromium does not require a stream to be rendered on a different node from where it originated; the client can load the `render` SPU as part of its chain. In this way, an application's drawing style can be modified while it runs directly on the node's graphics hardware, without any network traffic.

SPU chains are always initialized in back-to-front order, starting with the final SPU in the chain. At initialization, a SPU must return a list of all the functions that it implements. A SPU that wants to pass a function call through to the SPU immediately downstream can return the downstream SPU's function pointer as its own. Because there is no indirection in this model, passing OpenGL calls through multiple SPUs does not incur any performance overhead. Such function pointer copying is common in Chromium; as long as SPUs copy and change OpenGL function tables using only our provided APIs, they can change their own exported interface on the fly and we will automatically propagate those changes throughout the node.

3.4 SPU Inheritance

A SPU need not export a complete OpenGL interface. Instead, SPUs benefit from a single-inheritance model in which any functions not implemented by a SPU can be obtained from a "parent", or "super" SPU. The SPU most commonly inherited from is the `passthrough` SPU, which passes all of its calls to the next SPU in its node's chain. The wireframe drawing SPU mentioned in the previous section would likely be implemented this way—it would implement only `glPolygonMode`, and rely on the `passthrough` SPU to handle all other OpenGL functions. At initialization, each SPU is given a dispatch table for its parent. For example, when the wireframe SPU wishes to set the drawing mode to wireframe during initialization, it calls the `passthrough` SPU's implementation of `glPolygonMode`.

Another common use of SPU inheritance is per-frame computation. It is often useful to take an action exactly once per frame, and this can be accomplished by overriding a parent SPU's implementation of `SwapBuffers`. Simple SPUs can use this ability to track performance or statistics, but more complex SPUs can extract the rendered frame and manipulate the resulting image. One such SPU is described in detail in section 6.1.

3.5 Stream Serialization

Because inter-node communcation is implemented as remote invocations of OpenGL commands, Chromium provides a library for efficiently encoding and decoding graphics API commands. This library takes a sequence of commands and produces a serialized encoding of the commands and their arguments. Although this library is normally used to prepare commands for network transmission, it can also be used to buffer a group of commands for later analysis or playback. Chromium's stream packing library can also reverse the byte order of the encoded stream, allowing for communication between computers of

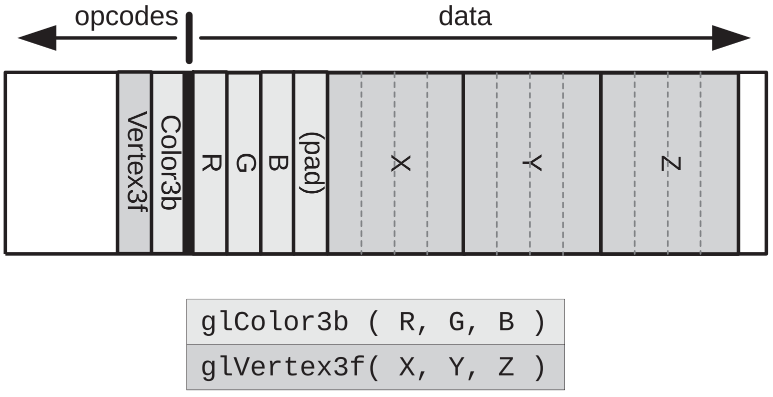


Figure 3.2: A packed network buffer. Each thin rectangle is one byte. The data are packed in ascending order and the opcodes in reverse order. A header applied to the network buffer before transmission encodes the location of the split between opcodes and data. Only the shaded area is actually transmitted.

differing endianness.

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, although to a lesser extent.

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 [Intel, 1999]. On other architectures, where misaligned reads are not allowed at all, reading unaligned data would require manual shift and mask operations.

In order to make the network protocol as efficient as possible, we first eliminate redundant OpenGL commands, such as `glVertex3f` and `glVertex3fv`. This greatly reduces the number of OpenGL commands that the wire protocol needs to encode[2]. Even with this reduction, however, there are still more than 256 OpenGL functions to encode, so we manually identify enough infrequent commands (e.g., certain OpenGL extensions) to be grouped together into a single "extend" call. This design also allows us to easily grow the wire protocol without worrying about adding too many functions.

Because we now have fewer than 256 functions to encode, the wire protocol can use 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. The "extend" opcode places a secondary opcode in the data payload to disambiguate the exact call to be made.

Using a single byte opcode introduces a wrinkle into the protocol design, however. Because we wish to maintain the natural alignment of each argument, interleaving the 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. Rather than send two separate buffers, however, we pack the opcodes *backwards* in the same network buffer as the data, as shown in figure 3.2. This way, the data remain aligned and the buffer remains contiguous so it can be sent with only one call to the networking library. Allocated network buffers are split $\frac{1}{5}$ of the way from their

[2]We are deliberately vague about the exact number of OpenGL commands representable by our wire protocol because the number changes frequently. Each time a new extension is added to Chromium or the OpenGL specification changes, we add new functionality to the wire protocol.

beginning to provide 20% space for opcodes and 80% space for data. The transition from a dual-buffer scheme to this backwards-packing design resulted in a 20% increase in peak network transmission rate for vertices over a 100 megabit ethernet network.

In order to be cautious about buffer overflow, OpenGL functions that take no arguments (e.g., `glEnd` or `glPopMatrix`) use 32 bits of padding to guard against the pathological case when the buffer is filled entirely with such functions. This way, we only have to check for overflow at one end of the buffer. When the buffer fills, a user-specified function is called. Typically this function will cause the buffer to be sent over a network, although the buffer can also be analyzed immediately or stored on a chain of buffers if, for example, an entire frame worth of commands needs to be analyzed.

This simple command representation allows for very fast packing and unpacking of graphics commands. In most cases, packing simply requires copying the function arguments to the data buffer and writing the opcode. Using Linux on an 800 MhZ Pentium III with 256 MB of RDRAM, Chromium can pack over 21.5 million vertices per second. 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 corresponding 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, requiring no data copies to be made.

3.6 Network Abstraction

Chromium provides a connection-based network abstraction to support multiple network types such as TCP/IP and Myrinet. We also provide specialized network interfaces such as an ideal network model that discards all traffic, allowing application load balance to be easily measured, and also a file-trace network model that allows streams to be recorded to disk for later playback. In addition to enabling inter-node communcation as specified in the configuration DAG, this network layer can be used by applications or by the SPUs themselves for out-of-band communication. An example of such an out-of-band application is binary-swap compositing for sort-last parallel rendering, described in section 6.1.

In this abstraction, each server/client pair is joined by a connection. By making buffer allocation the responsibility of the network layer, we allow a zero-copy send. For example, the client packs OpenGL commands directly into network buffers, and the Myrinet network layer sends them over the network using DMA. In order for this to work, these buffers must be pinned (locked and unpageable), which is done by the implementation of our network abstraction for Myrinet. Receiving data on our network operates in a similar manner: the network layer allocates (possibly pinned) buffers, allowing a zero-copy receive.

The connection is completely symmetric, which means that the servers can return data (e.g., the results of `glReadPixels`) to the clients. To enable these semantics, we provide a synchronization mechanism between clients and servers: The client can insert a "write-back" opcode into the graphics stream at any time. When the writeback opcode is decoded by a server, a message is sent back to the client indicating that the server has received and decoded all commands up to and including the writeback. When using the `tilesort` SPU, the `glFinish` call will wait for writeback notification from all servers before proceeding. This functionality is important so that applications that need to synchronize their output with some external input source can make sure the graphics system's internal buffering is not causing their output to lag behind the input. The user can optionally enable an implicit `glFinish`-like synchronization on each `SwapBuffers` call, which ensures that no client will ever get more than one frame ahead of the servers.

Our network abstraction includes an internal flow control mechanism that ensures that an endpoint is not overrun with data. For example, although client applications do not stall when they issue Parallel API commands, they do restrict the order of execution possible at the servers. Thus, a server can receive graphics commands that it is not currently allowed to execute. The network layer uses a credit-based flow control scheme to prevent each client from consuming more than a fixed amount of memory on each server[3]. Each client has a number of send credits per server, which is initialized to the maximum number of outstanding bytes per server. These credits are decremented each time a send is performed, and a send blocks when the client has no more credits. When the server processes a buffer and returns the buffer to the network layer, the appropriate client is credited, allowing it to send more data. In order to avoid sending a message to a client node each time work is received, credits are aggregated by the server and returned to client nodes periodically. Flow control is particularly important when the Parallel API causes a context is blocked, since additional commands may come in from the client at any time even though the server cannot drain a blocked context's command queue.

3.7 State Tracking

Graphics APIs like OpenGL are typically *stateful*. Each state element controls the manner in which future geometric primitives will be interpreted and rendered. The state contains attributes such as the current transformation matrix, the current color, and so on. On a workstation with hardware graphics acceleration, the graphics hardware keeps track of most or all of the current state. However, in order to properly implement a remote protocol for such an API, it is necessary for the client to keep track of some of the state in software. For instance, OpenGL allows the programmer to specify certain alignment and

[3]A previous scheme used on-off flow control. Network congestion could result in an “off” message being substantially delayed, with the disadvantage that a great deal of additional data might arrive in the meantime. This means that on-off flow control cannot make any guarantees about the amount of resources that a message source could potentially consume at the message sink, which is unacceptable.

offset properties for pixel arrays that are to be used as textures. The client library must unpack these textures in order to send the correct data to the server. Therefore, the client library must track these offsets and alignments as the application is running.

Chromium includes a complete OpenGL state tracker, based on the one described by Buck, Humphreys and Hanrahan [Buck *et al.*, 2000]. The ability to maintain the graphics state greatly improves the performance and flexibility of our networked rendering system. Our original Interactive Mural graphics system [Humphreys and Hanrahan, 1999] was a straightforward RPC-style network protocol for OpenGL, similar to the GLX protocol [Segal and Akeley, 1999]. Although this approach met our functionality goals, extending it to support high performance remote rendering proved difficult. Because each command simply created a packet representation of its parameters, we could not gain any semantic knowledge of the application's graphics state or the primitives it was drawing in order to use the network more efficiently.

In OpenGL, almost all commands that do not generate fragments are commands to manipulate the graphics state. `glRotatef`, `glPixelStorei`, and `glFogf` are examples of these commands. Our state tracker allows the effects of these functions to be recorded into a "graphics context." A graphics context is a complete encapsulation of the configuration of the graphics pipeline, along with all the texture memory currently in use. The context is made up of individual state elements, such as the current blending mode, the current transformation matrix, or the enable/disable state of lighting.

Our graphics contexts are arranged in a hierarchy. At the top level, we have broken the state into 19 categories: transformation, pixel, current, viewport, fog, texture, lists, client, buffer, hint, lighting, line, polygon, scissor, stencil, evaluators, imaging, selection, and extensions. These categories closely follow the ones laid out in table 6.5 of the OpenGL 1.2.1

specification [Segal and Akeley, 1999], although we have collapsed some of the more similar categories (e.g., color buffers and depth buffers are collapsed into "buffer" state). Further levels of the hierarchy are present where appropriate; for example, each of OpenGL's lights has a complete set of lighting parameters associated with it. The benefit of arranging the state hierarchically will become apparent when we discuss lazy state updates in section 4.2, and also soft context switching in section 5.2.

3.8 Server Implementation

A Chromium server maintains a queue of pending commands for each connected client. When commands arrive over the network, they are placed on the end of their client's queue. These queues are stored in a circular "run queue" of contexts. Each server executes a client's commands until it runs out of work or the context "blocks" on a barrier or semaphore operation. Blocked contexts are placed on wait queues associated with the blocking semaphore or barrier. The server's queue structures are shown in figure 3.3.

Because each client has an associated graphics context, a context switch must be performed each time a client's stream blocks. Although all modern graphics accelerators can switch contexts fast enough to support several concurrent windows, hardware context switching is still slow enough to discourage fine-grained sharing of the graphics hardware. When programmatically forced to switch contexts, the fastest modern accelerators achieve a rate of approximately 12,000 times per second [Buck *et al.*, 2000], which is slow enough that it would limit the amount of intra-frame parallelism achievable in Chromium. See section 5.2 for more details about this soft context switching mechanism.

In practice, when a context blocks, the servers often have a choice of many potentially runnable contexts. Because a parallel application will almost always enter a barrier immediately before the end of the frame, it is unlikely that one context will become starved.

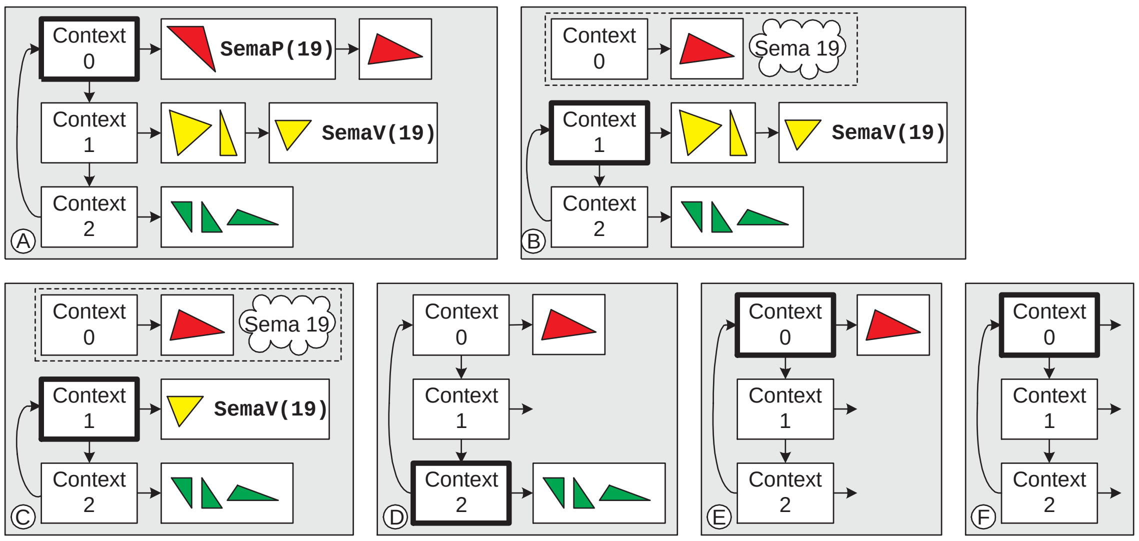


Figure 3.3: Inside a Chromium server. Runnable contexts will be serviced in a round-robin fashion. Graphics commands being issued by a context's application can be appended to the end of a work queue at any time, until the client consumes its allotted server-side buffer space. Blocks A-F show sequential timesteps as the server decodes command blocks; the currently executing context is shown with a heavy outline. In timestep A, the server encounters the `SemaP` operation in context 0, which blocks the context and removes it from the run queue. In timestep C, context 1's `SemaV` command will unblock context 0 and place it back on the run queue.

Therefore, in choosing a scheduling algorithm, the main concerns are the expense of the context switch itself as well as the amount of useful work that can be done before the next context switch.

In practice, we have found that a simple round-robin scheduler works well, for two reasons. First, clients participating in the visualization of a large dataset are likely to have similar contexts, making the expense of context switching low and uniform. Also, since we cannot know when a stream is going to block, we can only estimate the time to the next context switch by using the amount of work queued for a particular context. Moreover, any large disparity in the amount of work queued for a particular context is most likely the result of an application-level load imbalance. This load imbalance, not context switching overhead, will certainly be the main performance limitation of the application. In general, because of the low cost of context switching, and because we need to complete execution of all contexts before the end of the frame, we have not found the server's scheduling algorithm to be a significant factor in an application's performance. Note that if more than one context is unblocked by a Parallel API command, the servers may not make the same choices about which context to run next. This is not a problem, however, because any more strict ordering could be explicitly expressed through the use of additional Parallel API synchronization commands.

3.9 Provided SPUs

Chromium provides a number of SPUs that can be used or extended to realize the desired stream transformation. Eleven of Chromium's most useful SPUs are shown in table 3.1, and new specialized SPUs are frequently added to the repository. Some of these SPUs will be described in detail in the following chapters.

SPU	Description
`error`	Prints a fatal error when any OpenGL function is called. This SPU is implicitly the parent of any SPU that does not specify otherwise. This way, SPUs that accidentally fail to implement a necessary function will emit a meaningful error message rather than crashing or failing in other ways that can be difficult to debug.
`framerate`	Measures the frame rate of the stream as it passes by. This is useful for measuring the performance at various places in a communication graph without instrumenting each SPU separately.
`nop`	Silently discards all OpenGL calls, which can be a used to replace the `render` SPU to factor out rendering time when measuring performance.
`passthrough`	Passes all functions to the next SPU in a node's local chain. Many SPUs will inherit from the `passthrough` SPU, allowing them to modify only a small subset of the entire OpenGL API.
`print`	Dumps a human-readable log of all OpenGL calls and their arguments to a file. SPU debugging often involves liberal application of the `print` SPU and some post-processing of the resulting logs.
`readback`	Transforms a stream of commands into a single image. The `readback` SPU can be configured to extract color and/or depth, and also to include commands that position the resulting image at an arbitrary offset using `glRasterPos`.
`render`	Passes all OpenGL calls directly to the graphics hardware, producing an image in a window. Typically the `render` SPU is used at every node that does not generate any new streams.
`saveframe`	Renders a stream and saves an image file for each frame. This can be used to make frame-by-frame movies of any OpenGL application.
`send`	Transmits a serialized representation of the entire stream to a server. Functions that query the OpenGL state or have a non-`void` return type will require a round-trip message using this SPU.
`tilesort`	Sorts a single stream into tiles, and sends specialized streams to multiple servers managing those tiles.
`vertexarray`	Removes uses of OpenGL vertex arrays by converting those calls into sequences of standard OpenGL functions. This can be useful for SPUs that need to guarantee that the data provided to their functions will persist over time.

Table 3.1: SPUs provided by Chromium. These SPUs can be used by any node in a cluster, and can be extended and combined to perform different stream transformations.

4. Scaling Display Resolution

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 [PowerWall, 1994], CAVE [Cruz-Neira *et al.*, 1993], and Stanford's Interactive Mural [Humphreys and Hanrahan, 1999] support resolutions well beyond the common desktop. This chapter describes a Chromium configuration that decouples output resolution from rendering performance. Because large datasets are typically very intricate, the ability to interactively render them on such a display can provide additional insight into their structure because the viewer can achieve an overall view of the data while still examining their finest detail.

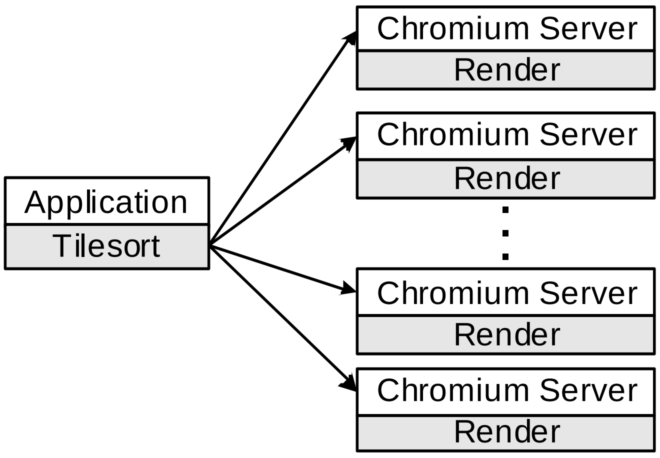


Figure 4.1: Diagram of the system architecture. The `tilesort` SPU intercepts the application's calls to the graphics hardware. It then distributes the rendering to multiple rendering servers. These servers are connected to projectors which display the final output.

Previous attempts to solve the resolution challenge have faced a number of limitations. One method is to provide a parallel computer (e.g., SGI Origin) 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.

In short, our Chromium configuration meets the following goals:

- 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, each of which costs a few hundred dollars and offers performance comparable to or faster than high-end workstation graphics.

- Support an immediate-mode API 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.

Some of these goals are partially or wholly met by the design of Chromium itself. For example, Chromium provides support for the OpenGL API, which meets the third goal, and it is designed specifically to support a networked cluster of workstations, which enables the second goal. The main challenge in enabling scalable display resolution is understanding exactly how to manipulate streams of OpenGL to achieve the goal of scalable display resolution.

The stream processor we will use to achieve scalable resolution is called `tilesort`. A diagram of a typical Chromium configuration when using the `tilesort` SPU is shown in figure 4.1. Each server has its own graphics accelerator, and the output of that accelerator is connected to a display (typically a projector). Those displays are then abutted in a rectangular array to form a common screen.

Referring back to Molnar's sorting classification [Molnar *et al.*, 1994], described in section 2.2, and recalling that sort-middle architectures are impractical to realize with clusters of commodity components, it is clear that the architecture realized by the configuration in figure 4.1 is a sort-first parallel rendering architecture. The screen has been divided into tiles, and each tile has its own full graphics pipeline associated with it. Primitives (or groups of primitives) from the application are sorted to the appropriate pipeline for transformation and rasterization.

In many internally parallel standalone graphics accelerators, geometric primitives are sorted using broadcast communication. This way, primitives are assigned to pipelines according to some distribution algorithm, and the pipelines merely snoop a shared bus to receive work. However, in a cluster, high-speed interconnects are point-to-point switched networks, and do not support an efficient hardware supported broadcast. Therefore, we must 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 lazy state update. 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.

4.1 Geometry 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[1].

[1]Another attractive approach might be to forward network traffic from one server to the next, creating a virtual ring network. This adds latency to the overall system, but that latency should be well hidden by the streaming nature of our application. In fact, we did construct a ring-network based version of WireGL, and experimentally verified that it worked quite well, although its advantage over the one-to-many approach taken by `tilesort` was minimal. Although the ring-based software design is much simpler, we chose to keep the stream specialization approach because it was straightforward to add a parallel interface to the architecture, as described in chapter 5.

Because the `tilesort` SPU (hereinafter referred to simply as `tilesort`) 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. `tilesort` 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, `tilesort` has no advance knowledge of the scene being rendered.

The OpenGL API explicitly maintains three different coordinate systems: object, camera, and screen[2]. Transformations between these coordinate systems are represented as two separate 4×4 matrices, allowing any linear transformation. Vertices specified by an application are assumed to be in object coordinates, and are transformed first to camera coordinates, and then to screen coordinates, where they are rasterized.

As the application makes OpenGL calls, `tilesort` packs the commands into a geometry buffer using the packing library described in section 3.5, and tracks their 3D bounding box. The bounding box represents the extent of the primitives in their local (object) 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, `tilesort` transforms that bounding box to screen coordinates to compute the extent of the buffer's geometry on the eventual display (note that in order to perform this transformation, we must maintain the two transformation matrices mentioned in the previous paragraph.

[2]Although the names of these coordinate systems tend to imply their relationship to each other, OpenGL does not enforce this interpretation. Geometry specified in object coordinates is transformed first to camera coordinates and then to screen coordinates, but no geometric computation is performed in camera space. Because our application is only concerned with the object to screen transformation, no semantic interpretation of the three coordinate systems is necessary. In fact, many OpenGL implementations maintain the product of the two transformation matrices explicitly, bypassing the intermediate camera coordinate system entirely.

This is accomplished using Chromium's state tracking system.). For each server whose output area is overlapped by the transformed bounding box, `tilesort` copies the packed geometry opcodes and data into that server's outgoing network buffer. 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, `tilesort` 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 more than one matrix multiplication. Instead, we transform the bounding box only when the packed geometry buffer is flushed, which amortizes the transformation cost over many primitives. The geometry buffer is flushed either when it is completely filled, or when the state tracking system indicates that a flush is necessary, as explained in the next section.

The success of this algorithm relies in part on the spatial locality of successive primitives. Many large scale visualization applications (e.g. volume rendering) exhibit excellent spatial locality. These applications tend to issue many small, spatially coherent primitives. Because of the small primitive size, the resulting transformed bounding box is much smaller than the screen extent of a tile 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.

4.2 Lazy State Update

A common technique for improving the performance of any system is to alter the order of the commands executed while maintaining the semantics mandated by the programming interface. One straightforward example of this is instruction re-ordering in processor design: the processor is free to execute instructions in whatever order it determines would be most efficient as long as the program's behavior does not change. A similar technique is one we call *lazy state update*, or just *lazy update*, where each state command is postponed until the last possible moment in the hope that the system can determine that the operation is unnecessary.

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 [Torborg, 1987]. 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.

`tilesort` solves this problem by tracking not only the entire OpenGL state of the application, but also the complete state of each of the servers. When a state-changing call is made by the application, `tilesort` merely updates the data structure containing the application's graphics state instead of packing the call into its network representation, as shown in figure 4.5. Whenever such a command is called, `tilesort` must first determine if

```
send_geometry( server ) {
  compute difference between application's context
    and server's context
  send state commands to update server's context
  update client's copy of server's context
  clear server's dirty bits
  send geometry commands
}
```

Figure 4.2: Pseudo-code for `tilesort`'s buffer flush algorithm. By not encoding state until geometry has an effect on the output image, `tilesort` saves unnecessary network traffic.

any geometry has been packed but not yet sent. If this is the case, it must update the servers' states and flush the geometry command buffer before recording the state change. Pseudo-code for the server update algorithm is shown in figure 4.2. For each server managing an overlapped tile, `tilesort` computes the differences between the application's state and that server's state, and sends the minimal set of updates to the server to synchronize it with the application. Once these differences are sent, the packed geometry can follow. The buffer management scheme used to achieve this behavior is shown in figure 4.6.

This algorithm has the advantage that state is only sent when the geometry it affects is also sent. Therefore, if a block of geometry falls outside the viewing frustum, its associated state will never consume network resources. Lazy update is particularly advantageous when using many large texture maps. Certain applications like *Quake III: Arena* make numerous calls to `glTexSubImage2D` in order to update small details on surfaces in the scene. This usage pattern is especially expensive when rendering remotely, and updating textures lazily can provide a substantial performance gain.

```
glBlendFunc( src,dst ) {
  if (arguments generate error) return
  update application context with src,dst
  set all blend function dirty bits
  set all fragment state dirty bits
}
```

Figure 4.3: Pseudocode for the `glBlendFunc` function. Notice the use of multiple dirty bits in accordance with our hierarchical state change tracking scheme.

4.3 Computing Context Differences

We associate *n* "dirty" bits with each state element, where *n* is the number of rendering servers. When `tilesort` tracks a state command, all bits are set to 1, indicating that the application's context is possibly out of sync with the remote context on all servers. Note that we do not check whether or not the user has set the state element to its current value. This does not mean that calling a state command repeatedly will cause multiple packets to be transmitted; instead we simply track the latest value for that element of the state.

Most applications change only a very small subset of the entire OpenGL state between geometry blocks. We therefore maintain a *hierarchy* of these dirty-bit vectors. This way, we can quickly find the elements of the state that have changed without re-examining the entire graphics state. For example, we have a bit-vector for the diffuse color of OpenGL's `LIGHT0`, a bit-vector for all state pertaining to `LIGHT0`, and a bit-vector for all OpenGL lighting state. Because context differences are computed very frequently when using the `tilesort` SPU, it is crucial that these differences be located quickly.

As an example, pseudocode for the state-altering `glBlendFunc` function is shown in figure 4.3. Pseudocode for the hierarchical bit tests to update the fragment state is shown in figure 4.4.

The dirty bits provide a fast mechanism for detecting which elements of the OpenGL

```
update_fragment_state() {
  if (fragment bit set) {
    if (alphafunc bit set && ...)
            ...
    if (blendfunc bit set &&
        application blendfunc != server blendfunc) {
      generate glBlendFunc packet
      server blendfunc = application blendfunc
    }
    clear blendfunc bit
    if (clearaccum bit set && ...)
            ...
  }
  clear fragment bit
}
```

Figure 4.4: A portion of the code to update a server's fragment state. If no fragment state changes had been made, we would not check any of the fragment state elements, including `glBlendFunc`.

state need to be updated. Furthermore, the update routine performs a comparison of the application and server values before generating a state command. This prevents redundant state commands from being transmitted on the network (for instance, if a user sets up the entire OpenGL state every frame).

As a further optimization, some care is taken to only send "relevant" state updates. Although not shown in figure 4.4, the user may have changed the `glBlendFunc` parameters since the last time `tilesort` updated a particular server's blend function settings, but if blending is now disabled, those settings do not need to be sent to the server. Instead, the blend function's dirty bit is left marked, but the bit for the entire fragment state is cleared. This way, the parameters will be sent if blending is enabled in the future, but in the meantime the fragment state will be skipped entirely when computing context differences.

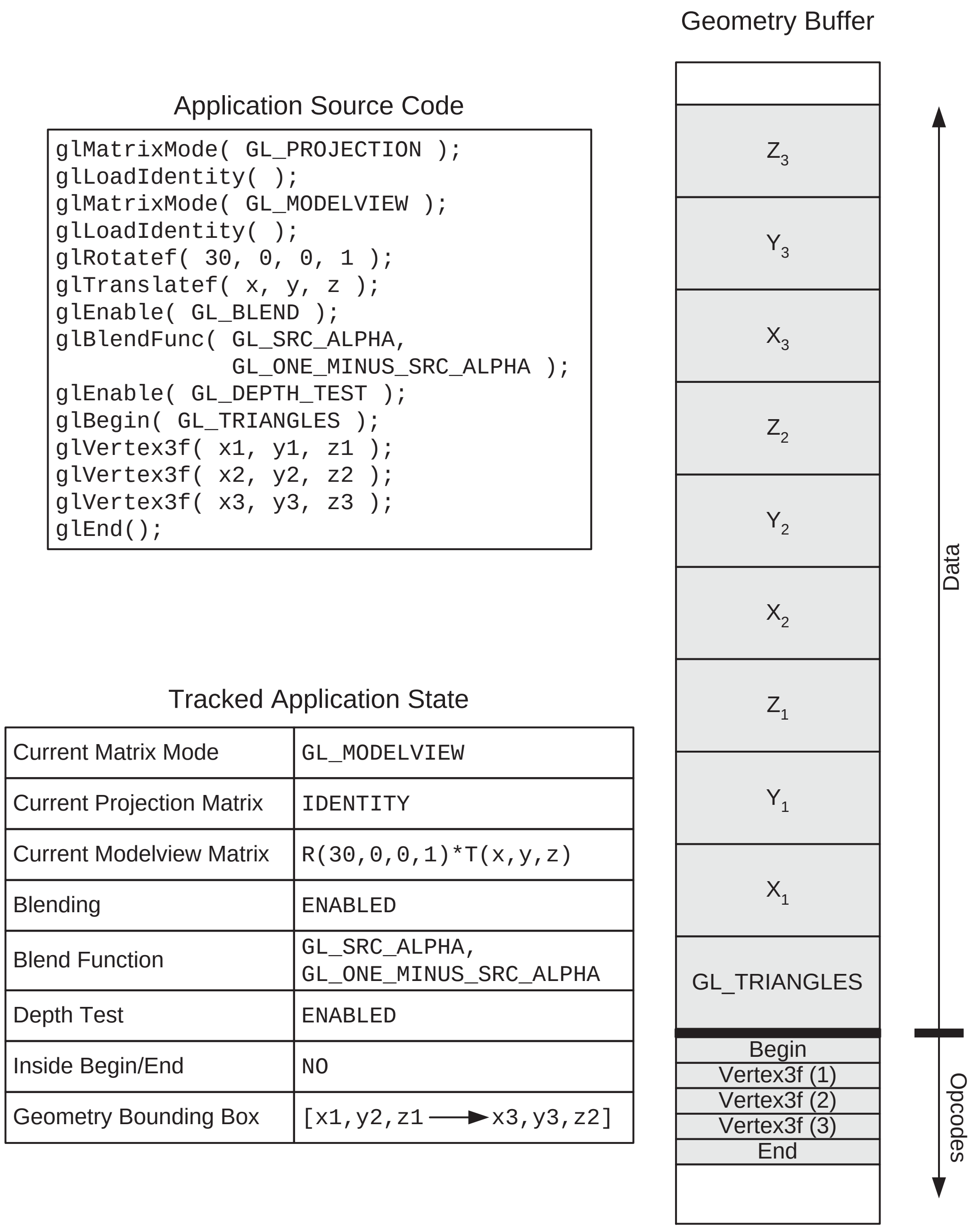
Some state commands are cumulative. For example, when `glRotatef` is called by the programmer, the top of the current matrix stack is multiplied by the implied rotation matrix. When tracking the transformation state, we perform these matrix multiplications in software. Since we always have the current transformation matrix available, we can collapse a series of transformation calls into a single `glLoadMatrix` packet. This is in contrast to more straightforward state commands like `glBlendFunc`, where the state is updated by sending the original parameters across the network.

Note that our context differencing implementation uses SPU dispatch tables to handle each discovered difference. In figure 4.4, the statement "`generate glBlendFunc packet`" is performed by simply calling the `glBlendFunc` pointer of a supplied SPU dispatch table. In `tilesort`, this function pointer points to the code that packs the command arguments into our network buffers. Because the state differencing system simply calls a supplied function for any difference found, it can be used for other tasks, such as soft context switching, described in section 5.2.

4.4 API Performance

The overall performance of the `tilesort` SPU is directly related to the speed at which it can process OpenGL commands. 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", which 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.

Table 4.1 shows the results of these tests. On the ideal network, `tilesort` is capable



Application Source Code

```
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();
```

Tracked Application State

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 → x3,y3,z2]

Figure 4.5: 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.

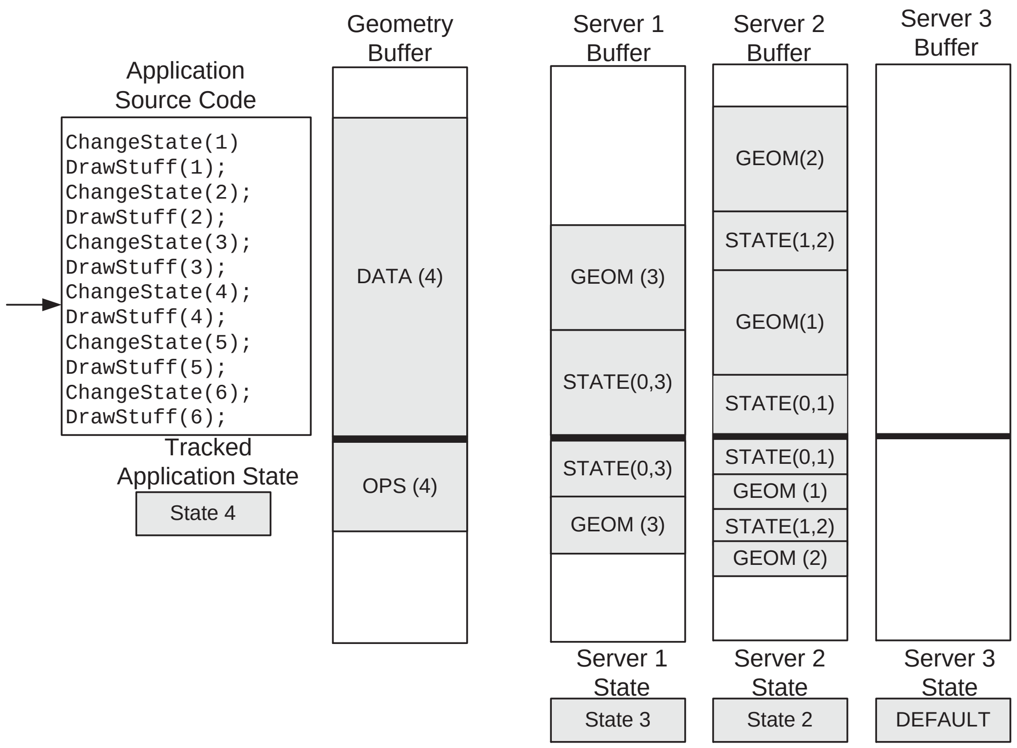


Figure 4.6: A snapshot of WireGL's buffer management scheme. Geometry commands are packed immediately into the geometry buffer. State commands record their effect into the client's tracked state structure. When the geometry buffer is flushed, the bounding box of the geometry buffer is used to determine 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

Table 4.1: Geometry rates for `tilesort`. `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.

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 `tilesort` implementation.

4.5 Scalability Results

To demonstrate scalability, we tested `tilesort` 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.

- `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 aggressive visibility culling, and its extensive use of OpenGL's feature set stresses our implementation.

Each application was run in six different tiled display configurations: 1x1, 2x1, 2x2, 4x2, 4x4, and 8x4, shown in figure 4.7. 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 4.7(a,b), clearly demonstrate `tilesort`'s scalability. As we increase the output size of the display, `tilesort` 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 `tilesort` when rendering to a 1x1 tiled display because it does not incur the overhead of state tracking and maintaining the bounding box. However, as

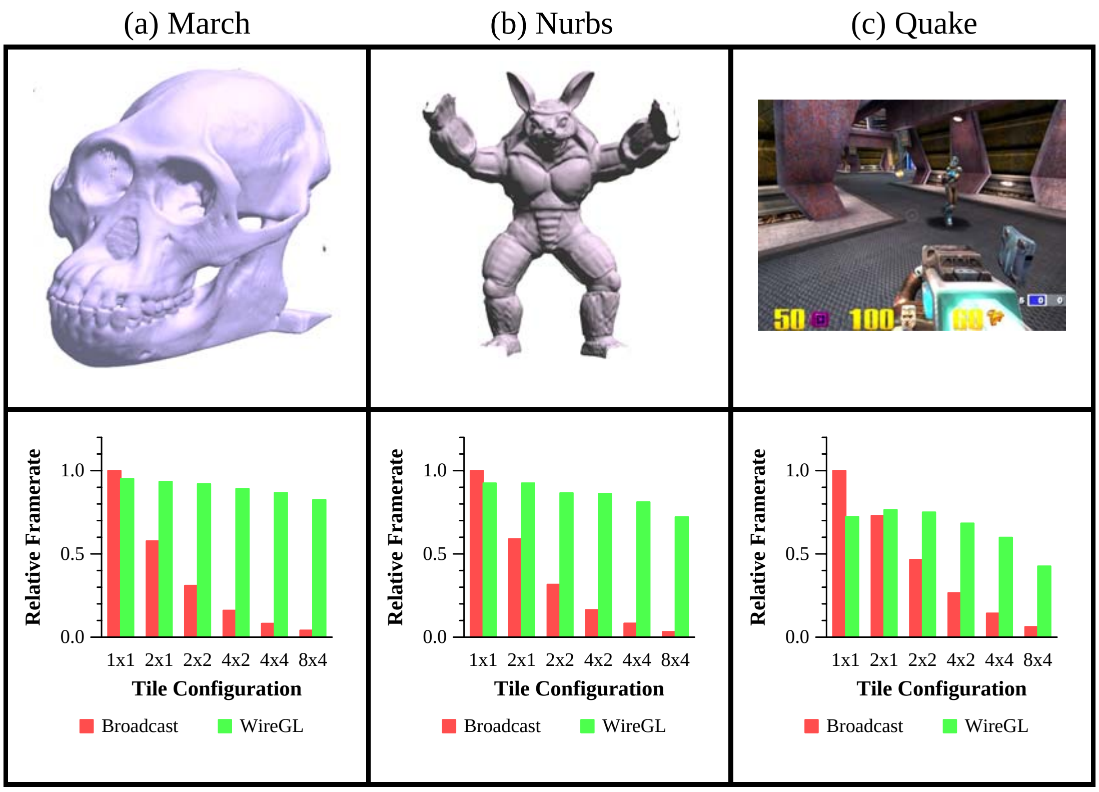


Figure 4.7: Frame rate comparisons between `tilesort` and broadcast. The broadcast frame rate falls off quickly as expected, while `tilesort`'s frame rate falls much more slowly. The more rapid falloff of `Quake` is due to larger primitive sizes, which need to be transmitted to many servers. The actual 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.

soon as we add a second display `tilesort` 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`, `tilesort`'s rate for 32 rendering servers only decreases by 13% from the single server configuration, compared to broadcast, which runs 25 times slower. `tilesort`'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 `tilesort` is proportional to the multiply transmitted geometry for each application.

`Quake`, shown in figure 4.7(c) does not scale as well as the other two examples, but `tilesort` still has a large advantage over broadcast. `Quake` makes use of complex textures rather than incorporating additional geometry to represent detail, and therefore issues many large polygons. These polygons reduce the network efficiency of `Quake` because they must be sent to more than one server. This effect 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 improve performance when using graphics hardware, it can be detrimental to `tilesort`'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 4.7(c) clearly demonstrates that our system can still efficiently manage `Quake` rendering and is superior to basic broadcasting.

5. Scaling Rendering Rate I: Sort-First

Although the techniques presented in the previous chapter provide scalable display resolution, there is a bottleneck at the client node[1]. Because the network interface is typically slower than the local interface between a CPU and the graphics subsystem, `tilesort` exacerbates the existing interface-limited nature of high performance graphics applications.

To overcome this interface limitation, it is necessary to extend the system to support a parallel interface. That is, multiple processes, each with its own graphics context, should be able to submit commands to the graphics system in parallel. In this chapter, we present

[1]In some cases, inefficiently written serial applications can run faster using `tilesort` than they can run locally. This surprising result is due to `tilesort`'s parsimonious attitude towards sending state commands. Repeatedly setting state elements to their current value can, in some cases, be more expensive than transmitting the command over a network. By removing these redundant commands, the application can run faster over a network than it can locally. Such an effect was demonstrated by NCSA using the WireGL system [Humphreys *et al.*, 2000] at SuperComputing 2000.

a Chromium configuration that uses multiple `tilesort` SPUs to create a sort-first architecture with such a parallel interface. This configuration is semantically very similar to the complete WireGL system, and the results in this chapter were originally presented in SIGGRAPH 2001 [Humphreys *et al.*, 2001]. Because they are so similar, we will refer to a Chromium configuration involving multiple `tilesort` SPUs and one or more rendering servers as "WireGL", pointing out differences as they occur.

A WireGL based rendering system consists of one or more Chromium clients, each using the `tilesort` SPU to submit OpenGL commands simultaneously to one or more Chromium servers. Recall that each `tilesort` SPU provides a sort-first tiled architecture; together they provide a parallel interface to that architecture, allowing nodes in a parallel application to collaborate to render a single output image. Each server typically has its own graphics accelerator and a high-speed network connecting it to all clients. The output image is divided into tiles, which are partitioned over the servers, each server potentially managing multiple tiles. The assembly of the final output display from the tiles is described in section 5.4. A high-level view of the system is shown in figure 5.1.

5.1 Overview

Despite recent advances in accelerator technology, many real-time graphics applications still cannot run at acceptable rates. As processing and memory capabilities continue to increase, so do the sizes of data being visualized. Today we can construct laser range scans comprised of billions of polygons [Levoy *et al.*, 2000] and solutions to fluid dynamics problems with several hundred million data points per frame over thousands of frames [Heermann, 1998; Reynolds and Fatica, April 2000]. Because of memory constraints and lack of graphics power, visualizations of this magnitude are difficult or impossible to perform on even the most powerful workstations. Therefore, the need for a scalable

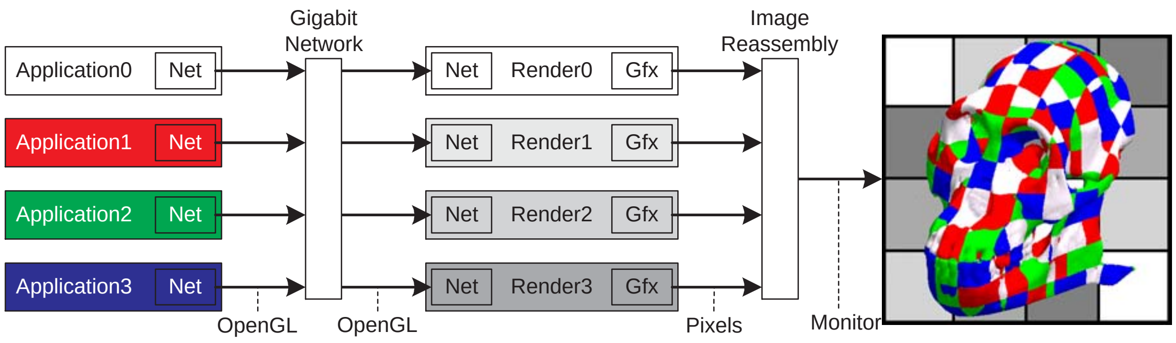


Figure 5.1: In this example, each application node is performing isosurface extraction in parallel and rendering its data using the OpenGL API. Each application node is responsible for the correspondingly colored portions of the object. In the configuration shown, the display is divided into 16 tiles, each of which is managed by the correspondingly shaded rendering node. These tiles are reassembled to a single monitor after they are scanned out of the graphics accelerators.

graphics system is clear.

The necessary components for scalable graphics on clusters of PC's have matured sufficiently to allow exploration of clusters as a reasonable alternative to multiprocessor servers for high-end visualization. In addition to graphics accelerators and processor power, memory and I/O controllers have reached a level of sophistication that permits high-speed memory, network, disk, and graphics I/O to all occur simultaneously, and high-speed general purpose networks are now fast enough to handle the demanding task of routing streams of graphics primitives.

To take advantage of these opportunities, we have created a Chromium configuration called WireGL to unify the rendering power of a collection of graphics accelerators in cluster nodes, treating each separate framebuffer as part of a single tiled display. A high-level diagram of WireGL is shown in figure 5.1.

In addition to supporting tiled displays, WireGL provides a parallel interface to the virtualized graphics system, so each node in a parallel application can issue graphics commands directly. This helps applications overcome one of the most common performance-limiting factors in modern graphics systems: the interface bottleneck. We extend the OpenGL API to allow the simultaneous streams of graphics commands to obey ordering constraints imposed by the programmer.

Another recent development is the introduction of the Digital Visual Interface (DVI) standard for digital scan-out of the framebuffer [DVI Specification, 1998]. We allow a flexible assignment of tiles to graphics accelerators, recombining these tiles using experimental DVI-based tile reassembly hardware called Lightning-2 [Stoll *et al.*, 2001]. In the absence of image composition hardware, WireGL can also perform the final image reassembly in software, using the general purpose cluster interconnect. Because of this flexible assignment of tiles to accelerators, we can deliver the combined rendering power of a cluster to any display, be it a multi-projector wall-sized display or a single monitor. By decoupling the number of graphics accelerators from the number of displays and allowing a flexible partitioning of the output image among the accelerators, image reassembly gives applications control over their graphics load balancing needs.

By providing a common virtualized interface for high performance rendering and screen space tiling, WireGL can provide a reasonable rendering load balance for many applications assuming a good distribution of geometric primitives over the entire output space. The burden of ensuring a fair distribution of computational work among cluster nodes still lies with the application designer. In addition, WireGL can expose the details of the screen tiling to a more savvy application to allow the application to make higher-level decisions about work distribution. Application nodes may perform their own fast culling, or provide bounding information through the use of OpenGL hints to further accelerate WireGL. In general, however, the tiling of the screen and the number of remote rendering engines is hidden from the application.

The strengths of WireGL can be summarized as follows:

- Handle large, time-varying data sets through an immediate-mode API
- Virtualize the existence of multiple disjoint accelerators to provide flexible output configurations
- Provide fast software context switching to support multiple simultaneous rendering clients
- Provide ordering for multiple simultaneous contexts without forcing the client nodes to block
- Provide scalability in output resolution and overall input rendering rate

5.2 Soft Context Switching

Before we describe the details of this architecture, it is important to understand how Chromium's network servers handle the multiple incoming graphics streams that result from any architecture with a parallel interface. As was described in section 3.1, Chromium servers act as serializers for multiple graphics streams. Because each incoming stream of graphics commands has its own associated graphics context, it is necessary to insert the proper commands into the serialized stream to ensure that when we switch from one stream to another, the graphics state has been restored. Because stream serialization makes no assumptions about the nature of the stream transformation to follow, we must perform this context switch in software. In this section, we describe the mechanism by which these context switches are accomplished. Note that it is frequently the case that the serialized stream is about to be rendered by locally attached graphics hardware, which itself supports multiple contexts. This section also shows that soft context switching is still a good idea even when there is underlying hardware support.

Traditional graphics accelerators are optimized to move data in one direction: from the host to the screen. Therefore, any operation that requires a reversal of the pipeline is typically very expensive. In particular, allowing multiple applications to share a single display requires the accelerator to switch graphics contexts rapidly; a process usually handled by the operating system and graphics driver. Unless the graphics card supports multiple contexts in hardware, this operation will require that the entire pipeline be flushed and the state registers be read back over the system bus. In fact, even cards that support multiple hardware contexts, such as the NVIDIA GeForce, are forced to read back state from the card when an application thread explicitly changes contexts due to constraints of the window system in which they operate.

Performing OpenGL state tracking in software alleviates the need for reading back from the graphics card. We can use the same efficient context differencing operation described in section 4.2 to perform a software, or "soft", context switch. By performing this switch in software, the graphics card needs only to maintain a single hardware context which never needs to be interrupted.

The model for soft context switching is only slightly different from the virtual graphics contexts used by the `tilesort` SPU for remote rendering. Each dirty bit now represents an input stream's context's relationship to the state of the serialized output stream. If a bit is set, the output context may not have the same value as the indicated input context. When we perform a soft context switch, we examine the bits hierarchically as before. If a particular state element is out of date, we insert a command to update the output context, and set all the other contexts' bits to one. Pseudocode for a portion of this algorithm is shown in figure 5.2.

```
switch_fragment_state( context ) {
  if (fragment bit set) {
    if (alphafunc bit_set && ...)
            ...
    if (blendfunc bit set &&
        input blendfunc != output blendfunc) {
      call system glBlendFunc
      set all other blendfunc bits
      set all other fragment bits
    }
    clear blendfunc bit
    if (clearaccum bit set && ...)
            ...
  }
  clear fragment bit
}
```

Figure 5.2: A portion of the code to switch contexts in software using our hierarchical bit-vector representation. If no streams are changing the blend function, no `glBlendFunc` calls will be issued. Also, if no streams change any of the fragment state, many tests can be skipped.

Once the soft context switch is completed, the output stream's context will exactly match the context of the current input stream. In generating a `glBlendFunc` command, the system may have overwritten the blend function with respect to the context of any or all other streams. We therefore must re-evaluate the validity of the blend function when we switch again. Also, the value equality test plays an important role in soft context switching. By checking the actual values, we prevent unnecessary state commands from being generated when multiple input contexts have the same values. This optimization is important because the input streams are usually collaborating to produce a single image, and will therefore tend to have very similar graphics contexts at all times.

An added benefit of performing this soft context switching is the ability to insert a level of indirection between each stream's texture object and display list identifiers and the identifiers used in the serialized stream. This allows multiple streams to load textures and display lists without worrying about conflicts.

The speed of soft context switching is proportional to the differences between the currently active context and the context to which we are switching. If they are exactly identical, the operations require 18 bit tests and one assignment. To measure the performance of soft context switching, we wrote a simple application that created one window and many contexts, and manually switched the window's rendering context as quickly as possible. To evaluate the quality of our implementation, we modified our OpenGL implementation to simply track state and pass all commands directly to the hardware. The results for this experiment are shown in table 5.1.

Graphics card	Processor	Identical	Varying
SGI InfiniteReality	195 MHz	719	697
SGI Cobalt	500 MHz	2,239	2,101
NVIDIA GeForce	733 MHz	11,919	5,968
Chromium	733 MHz	5,817,152	191,699

Table 5.1: Context switching rates for various OpenGL implementations. For the "Identical" column, we are context switching between contexts with no differences. For the "Varying" column, we change the matrix stack and current color for each context before switching. The InfiniteReality is hosted in an 8 processor MIPS R10000 based Silicon Graphics Onyx2; all other hosts use a single Intel Pentium III processor. The lower performance of the InfiniteReality is largely due to the expense of flushing its deep command buffers, a problem faced by more complex graphics accelerators.

Clearly, Chromium will run more slowly when the contexts are varying; the varying context column in table 5.1 requires a software matrix multiplication as well as calls to `glLoadMatrix` and `glColor3f`, plus drawing a triangle. Still, this application runs 38 times faster with Chromium than without it, on the same hardware.

5.3 Parallel OpenGL Extensions

When running a parallel application with WireGL, each client node behaves in the manner described in chapter 4, performing a sort-first distribution of geometry and state to all servers. OpenGL guarantees that commands from a serial context will appear to execute in the order they are issued. When multiple OpenGL contexts render to a single image, this restriction must be relaxed because the graphics commands are being issued in parallel. To provide ordering control for parallel rendering, Chromium adds barriers and semaphores to the OpenGL API, as proposed by Igehy et al. [1998].

The key advantage of these synchronization primitives is that they do not block the application. Instead, the primitives are encoded into the graphics stream, and their implied ordering is obeyed by the graphics system when a context switch occurs. A graphics context may enter a barrier at any time by calling `glBarrierExec(name)`. Semaphores can be acquired and released with `glSemaphoreP(name)` and `glSemaphoreV(name)`, respectively. Note that these ordering commands must be broadcast by `tilesort`, as the same ordering restrictions must be observed by all servers, and we wish to avoid a central oracle making global scheduling decisions.

`tilesort` can change the semantics of commands with global effects. For example, `SwapBuffers` marks the end of the frame and causes a buffer swap to be executed by all servers. This is different from the original implementation of WireGL; there, it was important that only one client execute `SwapBuffers` per frame [Humphreys *et al.*, 2001]. Also, a parallel application with no intra-frame ordering dependencies needed two barriers per frame: To ensure that the framebuffer clear happens before any drawing, a barrier must follow the call to `glClear`; and all nodes must have completely submitted their data

```
Display() {
    if (my_thread_id == 0) // I am the master
        glClear( ... );
    glBarrierExec( global_barrier );
    DrawFrame();
    glBarrierExec( global_barrier );
    if (my_thread_id == 0) // I am the master
        glSwapBuffers();
}
```

Figure 5.3: A minimal parallel display routine from the original WireGL design [Humphreys *et al.*, 2001]. Although the geometry itself has no intra-frame ordering dependencies, the imposition of frame semantics requires barriers following the framebuffer clear and preceding the buffer swap to ensure that the entire frame is visible.

for the current frame before swapping buffers, so another barrier must precede the call to `SwapBuffers`. Pseudocode for this minimal usage is shown in figure 5.3. More complex usage examples can be found in Igehy's original paper [1998].

In Chromium, however, the requirement of extra barriers and selective issuing of `glClear` and `SwapBuffers` is removed. Instead, we allow the user to optionally alter the semantics of `glClear` and `SwapBuffers` so that the barriers are implicit. In addition, the user can configure the servers to only execute `glClear` or `SwapBuffers` from a single client. At first glance, this ability may not seem very useful, but we will see that it is crucial to achieving a truly virtualized parallel graphics architecture.

Note that we must be careful when selectively executing commands from clients. In effect, we are assuming that each client application has the same basic OpenGL usage pattern as the one shown in figure 5.3. In particular, we are assuming that applications tend not to call `glClear` in the middle of a frame. This assumption is not always valid; for example, each instance of the `readback` SPU (described in section 5.6) needs to clear the remote OpenGL stencil buffer to achieve proper depth compositing. In an attempt to accommodate such usage, we only ignore extra calls to `glClear` for the color and depth buffer. However, it is easy to write applications for which this assumption is not valid. For those applications, modifications similar to those shown in figure 5.3 are still possible.

5.4 Display Management

Since each server may manage more than one tile, it may be necessary for a server to render a block of geometry more than once. The arrangement of tiles in the local framebuffer is described in section 5.5. `tilesort` inserts the screen-space bounding box for each block of geometry between the geometry itself and its preceding state commands. Each server compares this bounding box against the extents of the tiles managed by that server. For each intersection found, a translate and scale matrix is prepended to the current transformation matrix, positioning the resulting geometry with respect to the intersected tile's portion of the final output. Because of the semantics of OpenGL rasterization, this technique can lead to seaming artifacts for anti-aliased or wide lines and points. Unfortunately, not all OpenGL implementations adhere to the same rules regarding clipping of wide lines and fat points, so this problem is difficult to address in general.

Calls to `glViewport` and `glScissor` are then issued to restrict the drawing to the tile's extent in the server's local framebuffer, and finally the geometry opcodes are decoded and executed. Because the geometry block also includes vertex attribute state, the graphics state may have changed by the end of the geometry block. However, the client will insert commands to restore the vertex attribute state at the beginning of the geometry buffer. Therefore, if the geometry overlaps more than one tile, the vertex attribute state will always be properly restored before the geometry is re-executed.

To form a seamless output image, tiles must be extracted from the framebuffers of the servers and reassembled to drive a display device. We provide two ways to perform this reassembly. For highest performance, the images may be reassembled after being scanned out of the graphics accelerator. If this is not possible, the tiles can be extracted from the framebuffer over the host bus interface and distributed over a general purpose network, often the same one used for distributing geometry commands.

Of course, the easiest way to reassemble the image is to allow each server to drive a single locally-attached display. These displays can then be abutted to form a large logical output space. This arrangement constrains each server to manage exactly one tile that is precisely the size of its local framebuffer. This limits WireGL's ability to provide an application with flexible load balancing support, but makes the final display simple to manage.

5.5 Display Reassembly in Hardware

For our experiments with hardware display assembly, we use the Lightning-2 system [Stoll *et al.*, 2001]. A high-level view of a Lightning-2 system is shown in figure 5.4. Each Lightning-2 board accepts 4 DVI inputs from graphics accelerators and emits up to 8 DVI outputs to displays. Multiple Lightning-2 boards can be connected in a column via a "pixel bus" to provide more total inputs. Multiple columns can also be chained by repeating the DVI inputs, providing more DVI outputs. An arbitrary number of accelerators and displays may be connected in such a two-dimensional mesh, and pixel data from any accelerator may be redirected to any location on any output display. Routing information is drawn into the framebuffer in the form of two-pixel-wide (48 bit) "strip headers". Each header specifies the destination of a one-pixel-high, arbitrarily wide strip of pixels following the packet header in the frame buffer. Lightning-2 can drive a variable number of displays, including a single monitor.

Each input to Lightning-2 usually contributes to multiple output displays, so Lightning-2 must observe a full output frame from *each* input before it may swap, introducing exactly one frame of latency. However, almost no currently available graphics accelerators have external synchronization capabilities. For this reason, Lightning-2 provides a per-host back-channel using the host's serial port. When Lightning-2 has accepted an entire frame from all inputs, it then notifies all input hosts simultaneously that it is ready for the next frame. WireGL's servers wait for this notification before executing the client's `SwapBuffers` command. Because the framebuffer scan-out happens in parallel with the next frame's rendering, Lightning-2 will usually be ready to accept the new frame before the host is done rendering it, unless the application runs at a faster rate than the eventual monitor's refresh rate. In this case, the application will be limited to the display's refresh rate, which is often a desirable property anyway. Lightning-2 can also lock groups of outputs to swap together. Having synchronized outputs allows Lightning-2 to drive tiled display devices such as IBM's T221 or a multi-projector display wall without tearing artifacts. This in turn enables stereo rendering on tiled displays.

Each of WireGL's servers reserves space for its assigned tiles in its local framebuffer in a left-to-right, top-to-bottom pattern, leaving two-pixel-wide gaps between tiles, as shown in figure 5.5. A fixed pattern of strip headers is drawn into the gaps to route the tiles to their correct destination in the display space. Because Lightning-2 routes portions of a single horizontal scanline, non-uniform decompositions of the screen such as octrees or KD-trees could easily be accomplished using WireGL and Lightning-2. In general, each application will have different tiling needs which should be determined experimentally. In the future, we would like to use a heuristic to adjust the screen tiling on the fly to try to meet the application's needs automatically.

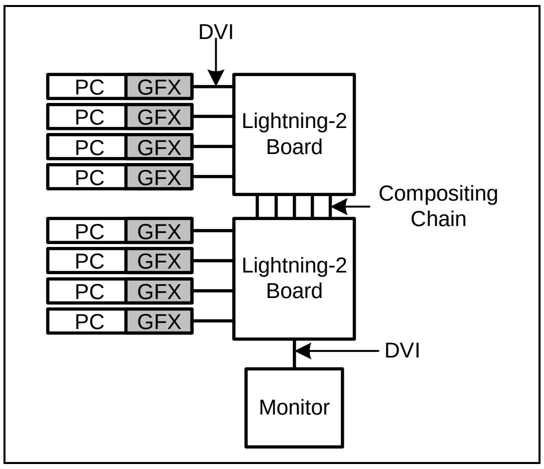


Figure 5.4: A sample Lighning-2 configuration. Each Lightning-2 board takes four DVI inputs and produces eight DVI outputs. Boards can be chained in one dimension to provide more inputs, and in the other dimension to provide more outputs. In this example, eight graphics accelerators are driving a single monitor. By decoupling the number of graphics accelerators from the total number of displays, and allowing each accelerator to manage tiles much smaller than the actual size of each individual display, we greatly improve load balancing.

5.6 Display Reassembly in Software

Without special hardware to support image reassembly, the final rendered image must be read out of each local framebuffer and redistributed over a network. This network can be the same one used to distribute graphics commands, or it could be a separate dedicated network for image reassembly.

To provide this functionality, we introduce the `readback` SPU. This SPU derives from the `render` SPU using the inheritance model described in section 3.4. The `readback` SPU renders all commands to the local graphics hardware as before, but at the end of the frame it reads back each tile and passes that tile to a downstream SPU using `glDrawPixels`. The `readback` SPU either behaves in tile reassembly mode (as described in this section), or in image-compositing mode, described in chapter 6. In tile reassembly mode, the `readback` SPU uses `glRasterPos` to position the rectangular tiles on the eventual output device.

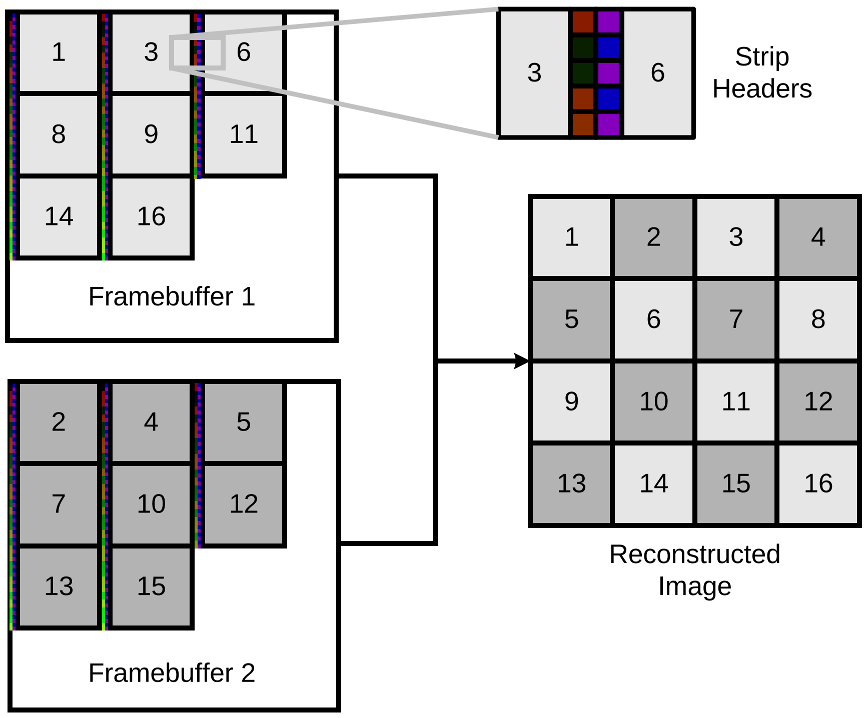


Figure 5.5: Allocating multiple tiles to a single accelerator with Lightning-2. In the zoomed-in region, the two-pixel wide strip headers are clearly visible.

Typically, these images will be sent over a network to a single server to be displayed by a `render` SPU. In effect, each of WireGL's servers becomes a client in a parallel image-drawing application. A diagram of this architecture is shown in figure 5.6.

The primary drawback of this approach is its potential impact on performance. Pixel data must be read out of the local framebuffer, transferred over the internal network of the cluster, and written back to a framebuffer for display. Even with the limited bandwidth available on modern cluster networks, image drawing bandwidth will tend to be the limiting factor for applications that can update at high framerates. As networks and graphics cards improve and can carry more pixel data along with the geometry data, this technique may become more attractive, but it cannot currently sustain high frame rates, as we will show in

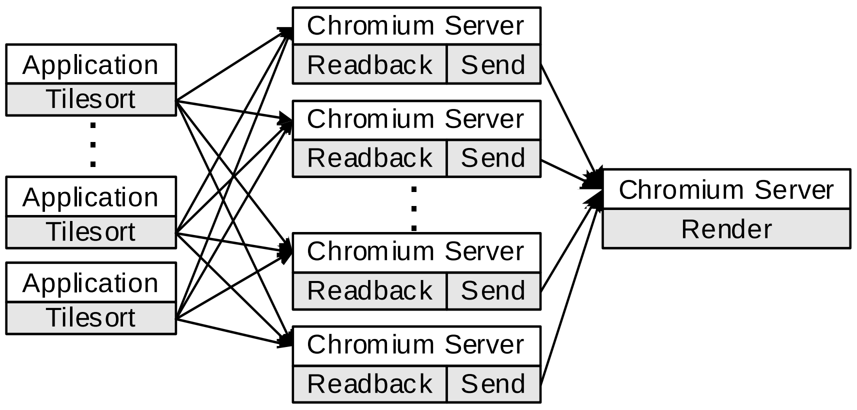


Figure 5.6: A complete WireGL system with software tile reassembly. A parallel application drives a tiled display using the sort-first logic in the `tilesort` SPU. Imagery is then read back from the servers managing those tiles and sent to a final compositing server for display.

section 5.7.

5.7 Performance and Scalability

The cluster used for our experiments consists of 32 Compaq SP750 workstations. Each node has two 800 MHz Intel Pentium III Xeon processors, 256 megabytes of RDRAM, and an NVIDIA Quadro2 Pro graphics adapter. The SP750 uses the Intel 840 chipset to control its I/O and memory channels, including a 64-bit, 66 MHz PCI bus, an AGP4x slot, and dual-channel RDRAM. Each SP750 is running RedHat Linux 7.0 with NVIDIA's 0.9-769 OpenGL drivers.

Each node has a Myricom high-speed network adapter [Boden *et al.*, 1995] 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.

For our experiments with parallel applications, we partition the cluster into 16 computation nodes and 16 visualization nodes. This is done because our network does not perform well when senders and receivers are running on the same host, as shown in section 5.7.

Applications

We have analyzed WireGL's performance and scalability with three applications:

- `March` is a parallel implementation of the marching cubes volume rendering algorithm [Lorensen and Cline, 1987]. A $200 \times 200 \times 200$ volume is divided into subvolumes of size $4 \times 4 \times 4$ which are processed in parallel by a number of isosurface extraction and rendering processes. `March` draws independent triangles (three vertices per triangle) with per-vertex normal information. `March` extracts and renders 385,492 lit triangles per frame at a rate of 374,000 tris/sec on a single node. Our graphics accelerators can render 2.9 million lit, independent triangles with vertex normals per second.

- `Nurbs` is a parallel patch evaluator that uses multiple processors to subdivide curved surfaces and tessellate them for submission to the graphics hardware. For our tests, `Nurbs` tessellates and renders 413,100 lit, stripped triangles per frame with vertex normals, at a rate of 467,000 tris/sec on a single node.

- `Hundy` is a parallel application that renders a set of unorganized triangle strips. Each strip is assigned a color, but no lighting is used. `Hundy` is representative of many scientific visualization applications where the data are computed off-line and the visualization can be decomposed almost arbitrarily. Each processor is responsible for its own portion of the scene database. Each frame of `Hundy` renders 4 million triangles, at a rate of 7.45 million tris/sec. On a single node, `Hundy` is completely

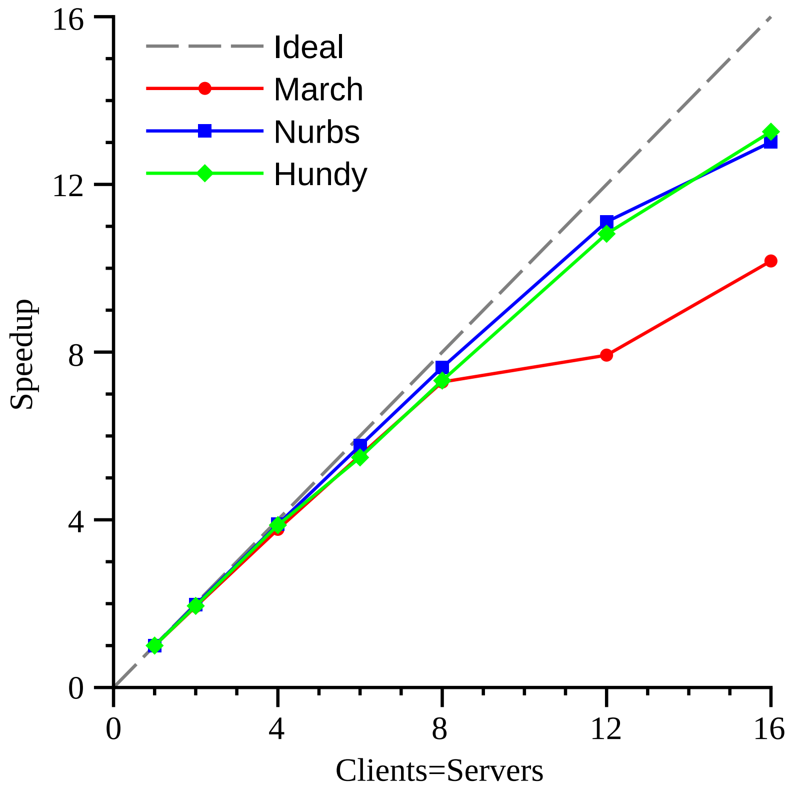


Figure 5.7: Speedup for `March`, `Nurbs`, and `Hundy` using up to 16 servers. With 16 clients and 16 servers, `Hundy` achieves 83% efficiency, `Nurbs` achieves 81% efficiency, and `March` achieves 64% efficiency.

limited by the interface to the graphics system; it cannot submit its data fast enough to keep the graphics system busy.

Scaling `March`, `Nurbs`, and `Hundy` using a single system is a challenging problem. Although other useful applications could be written that pose less of a challenge for WireGL, the applications we have chosen stress our implementation. Each application has very different load balancing behavior, requires immediate mode semantics, and generates a large amount of network traffic per frame. The speedup for these applications using 16 servers is shown in figure 5.7.

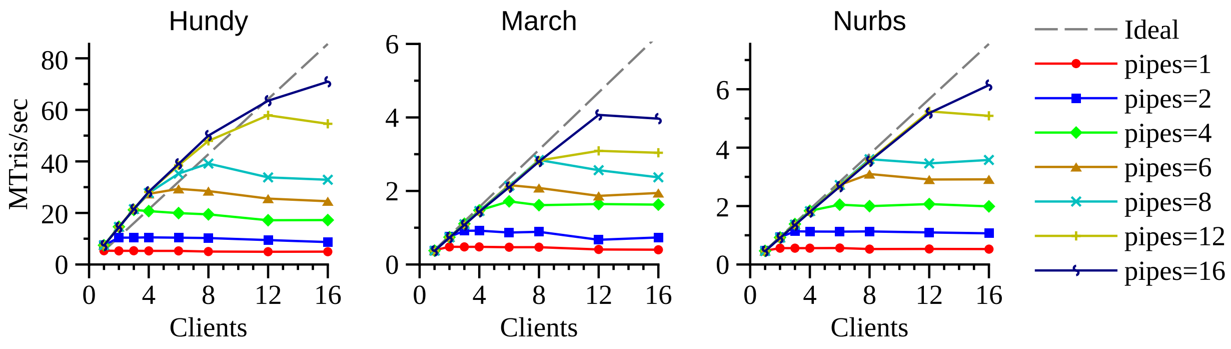


Figure 5.8: Scaling interface-limited applications. For each application, the number of clients and servers is varied. `Hundy` uses a tile size of 100×100, and achieves a peak rendering performance of 71 million tris/sec at a rate of 17.7 fps. `Nurbs` uses a tile size of 100×100, and achieves a peak rendering performance of 6.1 million tris/sec at a rate of 14.9 fps. `March` uses a tile size of 200×200, and achieves a peak rendering performance of 4 million tris/sec at a rate of 10.6 fps. For each run, the display is a single 1600×1200 monitor. As the number of clients surpasses the number of servers, the performance of the application once again becomes limited by the interface.

Parallel Interface

To scale any interface-limited application, it is necessary to allow parallel submission of graphics primitives. To demonstrate this, we have run our applications in a number of different configurations, shown in figure 5.8. In these graphs, the tile size is chosen empirically, and Lightning-2 reconstructs a final 1600×1200 output image.[2] Each curve represents a different number of servers, from 1 to 16. As the number of clients grows greater than the number of servers, the performance flattens out, demonstrating that such a configuration is once again limited by the interface.

Some of `Hundy`'s performance measurements show a super-linear speedup; this is because `Hundy` generates a large amount of network traffic per second. This traffic is spread

[2]Currently, Lightning-2 supports input resolutions up to 1280×1024, so for one server we bypass Lightning-2 and drive the display directly.

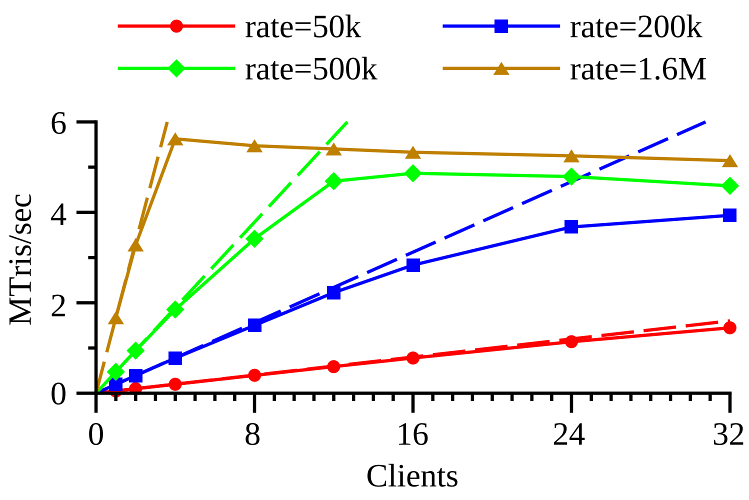


Figure 5.9: Scaling a compute-limited application with a single server. For each curve, `Hundy`'s issue rate has been restricted. We achieve excellent scalability up to either the server interface limit, or the full 32 nodes of our cluster.

uniformly over all the servers, and when the number of servers is greater than the number of clients, each path in the network is less fully utilized. Essentially, this shows that Hundy's performance is very sensitive to the behavior of our network under high load.

WireGL's approach provides scalable rendering to applications with a variety of graphics performance needs. To measure scalability with a compute-limited application, we have artificially limited `Hundy`'s geometry issue rate. The number of submitting clients is then varied while only using one server. The results of this experiment are shown in figure 5.9. For each test, the application scales well until it reaches the interface limit of the single server or the size of the cluster.

The results shown in figures 5.8 and 5.9 demonstrate WireGL's flexibility. Interface-limited applications can be scaled by adding servers and clients, while compute-limited applications can be scaled by adding clients only.

Hardware vs. Software Image Reassembly

The overhead of performing software image reassembly can quickly dominate the performance of an application as the output image size grows. Each node in our cluster has a pixel read performance of 28 million pixels/sec, and a pixel write performance of 64 million pixels/sec. If we can transmit 100 MB/sec of image data into a display node, this implies a maximum performance of 33 million pixels/sec for the visualization server. In practice, we achieve approximately half this rate in all-to-one communication, yielding a maximum frame rate of approximately 8 Hz at a resolution of 1600×1200.

To measure the overhead of the visualization server versus Lightning-2, we wrote a simple serial application that calls `SwapBuffers` repeatedly. The performance of this application represents an upper bound on the achievable framerate of any application. A serial application is a fair test because, as described in section 5.3, only one node in a parallel application calls `SwapBuffers` for each frame. In each experiment, 12 servers are used. The results are shown in figure 5.10. The "displays=4" curves are representative of a tiled display wall or a multi-input display such as IBM's T221.

This graph demonstrates that hardware supported image reassembly is necessary to maintain high framerates for most output image sizes. Lightning-2 is able to maintain a constant refresh rate of 90 Hz for any image size ranging from 320×240 to 3200×2400. The visualization server provides a maximum refresh rate of 8 Hz for a 1600×1200 image, which is approximately 46 MB/sec of network traffic. This is consistent with the measured bandwidth of our network under high fan-in congestion.

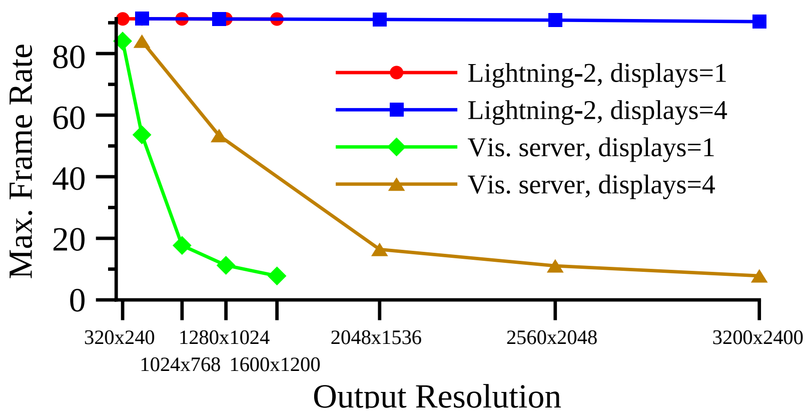


Figure 5.10: Maximum framerate achievable using Lightning-2 or the visualization server. As the image size increases, the expense of reading and writing blocks of pixels to the framebuffer quickly limits the visualization server to non-interactive framerates.

Load Balance

When evaluating a scalable graphics application, there are two different kinds of load balancing to consider. First, there is application-level load balance, or the amount of computation performed by each client node. This type of load balancing cannot be addressed by WireGL; it is the responsibility of the application writer to distribute work evenly among the application nodes in the cluster.

To evaluate application-level load balance, we measured the speedup of our applications in a full 32-node configuration without a network (i.e., discarding packets). In this configuration, `March` achieved 85% efficiency, `Nurbs` 98% efficiency, and `Hundy` 96% efficiency. From these results, we conclude that each application has a good distribution of work across client nodes.

The other type of load balancing is graphics work. For most applications, the interface to a single rendering server quickly becomes a bottleneck, and it is necessary to distribute the rendering work across multiple servers. However, the rendering work required to generate an output image is typically not uniformly distributed in screen space. Thus, the tiling of the output image introduces a potential load imbalance, which may in turn create a load imbalance on the network as well.

Because the triangles in our test applications are uniformly small, the server-side load balance can be reasonably measured by the total number of bytes sent to each server. For each application, the total incoming traffic when using one server is a lower bound on the total amount of network traffic for any number of servers, since adding servers will result in some redundant communication. The overlap factor is the ratio of total traffic received by all servers to this lower bound, and the load imbalance is the ratio of the maximum traffic received by any server to the average traffic. In figure 5.11, the height of each curve shows the overlap factor. The error bars indicate the overlap if each server received the maximum or minimum traffic received by any server. The load imbalance is therefore the ratio of the maximum shown to the observed overlap factor for that number of servers.

As expected, the choice of tile size affects the load balance and the overlap factor. For smaller tiles, there is less variance in the total number of bytes received, resulting in a better load balance, but the overall average data transmitted has increased due to overlap. As the tiles get larger, the overlap is smaller, but longer error bars indicate a poorer load balance. At a tile size of 100×100, `Nurbs` has a load imbalance of 1.53 on 16 servers, while at 32 servers the load imbalance increases to 2.13. The load imbalance will continue to increase as the number of servers increases. Currently, `Nurbs` is sufficiently compute-limited that its load imbalance is not exposed in the speedup curve shown in figure 5.7. However, as cluster size increases, the increasing load imbalance will eventually limit `Nurbs`' scalability. Nonetheless, WireGL provides excellent scalability up to 16 servers, which makes it a

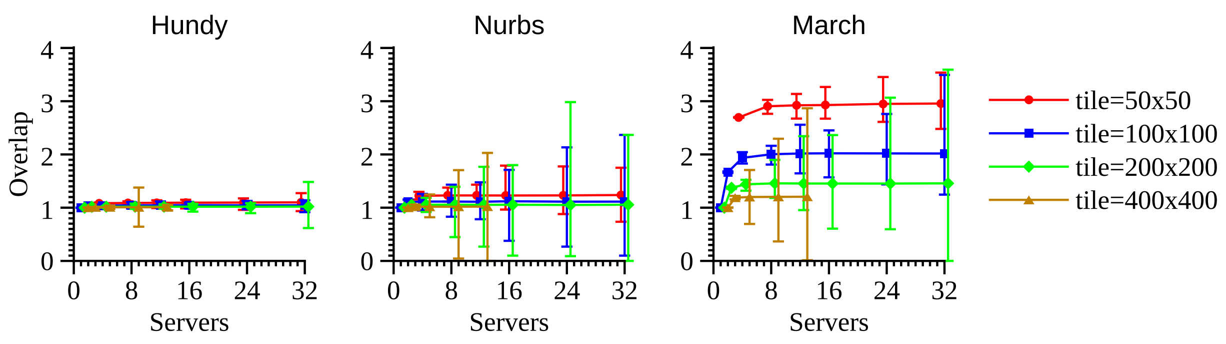


Figure 5.11: Overlap factor and load imbalance with various tile sizes on a 1600×1200 display. The height of each curve indicates the overlap factor, while the size of the error bars is proportional to the load imbalance. Increasing the tile size decreases the total amount of network traffic, but at the expense of load balance. Note that with a 400×400 tile size, only 12 total tiles are needed to cover the display, so no more than 12 servers can contribute to the final image.

useful solution for many applications on many current cluster configurations.

To verify our assumption that the server load balance can be reasonably measured by simply counting network traffic, we ran all our measurements in a mode where the servers discarded incoming traffic rather than decoding it. The performance measurements in this mode were almost identical to the measurements when graphics commands were actually executed. This demonstrates that the performance of interface-limited applications will largely be determined by the scalability of the network under heavy all-to-all communication, and not by the execution of the graphics commands. As networks improve, this effect will be reduced, although it is difficult to predict if future network technologies will ever be fast enough to handle the increasing performance of remote graphics cards or the increasing ability of CPU's to generate geometry.

5.8 Network Characteristics

To fully understand our scalability results, we have measured the achievable send and

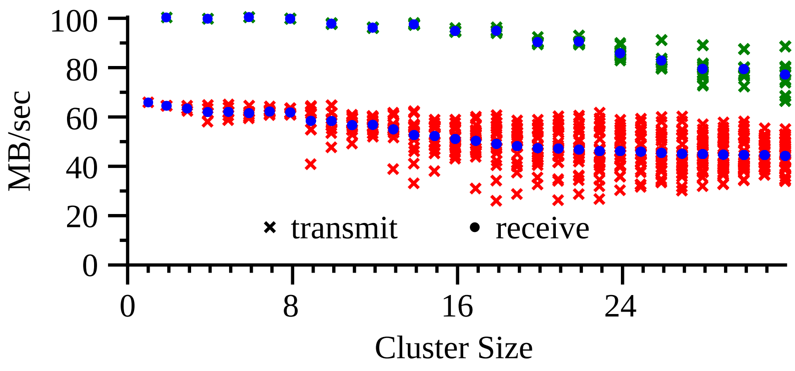


Figure 5.12: Transmit and receive bandwidth for Myrinet with all-to-all communication. For each cluster size, the observed send and receive bandwidth is plotted for all nodes. The top dataset represents a partitioned *n*-to-*n* run, where sources and sinks are not run on the same nodes. The bottom dataset is an unpartitioned run of all *n* nodes. Partitioning the cluster results in much higher bandwidth in general, as well as less transmit bandwidth variance.

receive bandwidths of our network when performing all-to-all communication. We performed this test in a partitioned configuration, in which sources and sinks run on different cluster nodes, and an unpartitioned configuration where sources and sinks run on the same cluster nodes. This test was performed with a WireGL-independent program in which each source node sends fixed-size network packets to all sink nodes in a round-robin pattern. The results are shown in figure 5.12. The partitioned dataset, shown with green crosses, achieves much higher overall performance, and has much less transmit bandwidth variance. For example, in an unpartitioned 18-way test, the transmit bandwidth ranges from 26.02 to 60.75 to MB/sec, while a partitioned run using 9 clients and 9 servers had bandwidths ranging from 93.92 to 96.96 MB/sec. It is interesting to note that any individual node will observe a very stable transmit bandwidth over the lifetime of its run. That is, the node achieving 26 MB/sec will always achieve 26 MB/sec, although varying the number of nodes will change which nodes perform poorly.

6. Scaling Rendering Rate II: Sort-Last

The sort-first architecture described in chapter 5 has certain drawbacks and fundamental limitations. Sort-first architectures are notoriously hard to load-balance, since it is difficult for a particular application node to know in advance how much work it will generate for any one given graphics pipeline. In addition, the amount of parallelism available in screen space is limited, because as we make the tiles smaller relative to the total display size, we aggravate the overlap factor. In addition, the sort-first architecture requires geometry to be moved over the network each frame, which can cause underutilization of graphics resources for applications with very high per-node performance.

A dramatically different architecture is shown in figure 6.1. In this figure, the `readback` SPU is loaded directly by the applications. Recall that the `readback` SPU dispatches all of

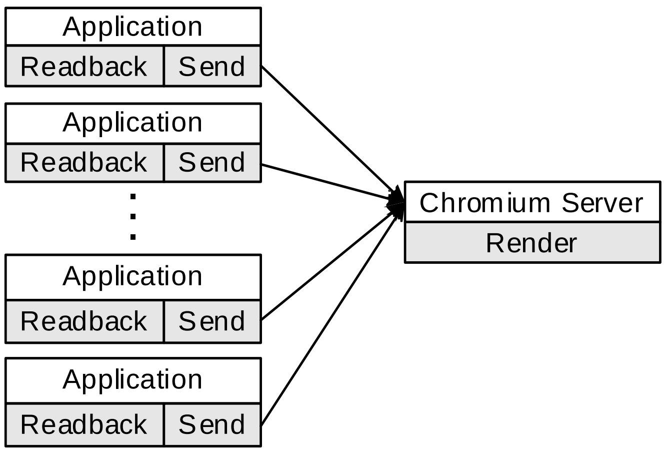


Figure 6.1: Chromium configured as a sort-last graphics pipeline. In this example, nodes in a parallel application render their portion of the scene directly to their local hardware. The color and depth buffers are then read back and transmitted to a final compositing server, where they are combined to produce the final image.

the OpenGL API directly to the underlying graphics hardware, so the application running in this configuration benefits from the full performance of local 3D acceleration. In this case, the `readback` SPU is configured to extract both the color and depth buffers, sending them both to a final compositing server along with the appropriate OpenGL commands to perform a depth composite. In contrast to WireGL, this is a sort-last architecture. In practice, having many full framebuffers arriving at a single display server would be a severe bottleneck, so this architecture is rarely used as shown. A more advanced (and practical) Chromium-based sort-last architecture is presented below.

Because Chromium provides a virtual graphics pipeline with a parallel interface, the parallel application in figure 6.1 could be run unmodified on the sort-first architecture from chapter 5 simply by specifying a different configuration DAG. The architectures may provide slightly different semantics (e.g., the sort-last architecture cannot guarantee ordering constraints between the clients), but the application need not be aware of the change.

6.1 Parallel Volume Rendering

Our volume rendering application uses 3D textures to store volumes and renders them with view-aligned slicing polygons, composited from back to front. Using Stanford's Real-Time Shading Language [Proudfoot *et al.*, 2001], we can implement different classification and shading schemes using the latest programmable graphics hardware, such as NVIDIA's GeForce3. Small shaders can easily exhaust these cards' resources; for example, a shader that implements a simple 2D transfer function and a specular shading model requires two 3D texture lookups, one 2D texture lookup (dependent on one of the 3D lookups), and all eight register combiners.

Because we store our volumes as textures, the maximum size of the volume that can be rendered is limited by the amount of available texture memory. In practice, on a single GeForce3 with 64 MB of texture memory, the largest volume that can be rendered with the shader described above is $256\times256\times128$. In addition, the speed of volume rendering with 3D textures is limited by the fill rate of our graphics accelerator. While the advertised fill rate of the GeForce3 is 800 Mpix/sec, complex fragment processing greatly impacts the attainable performance. Depending on the shader being used, we achieve between 42 and 190 Mpix/sec, or roughly 5% to 24% of the GeForce3's theoretical peak fill rate.

Both of these limitations can be mitigated by parallelizing the rendering across a cluster. We first divide the volume among the nodes in our cluster. Each node renders its subvolume on locally housed graphics hardware using the `binaryswap` SPU, which composites the resulting framebuffers using the "binary swap" technique described by Ma et al [Ma *et al.*, 1994]. In this technique, rendering nodes are first grouped into pairs. Each node sends one half of its image to its counterpart, and receives the other half of its counterpart's image. The SPUs composite the image they receive from their peers with their local framebuffer.

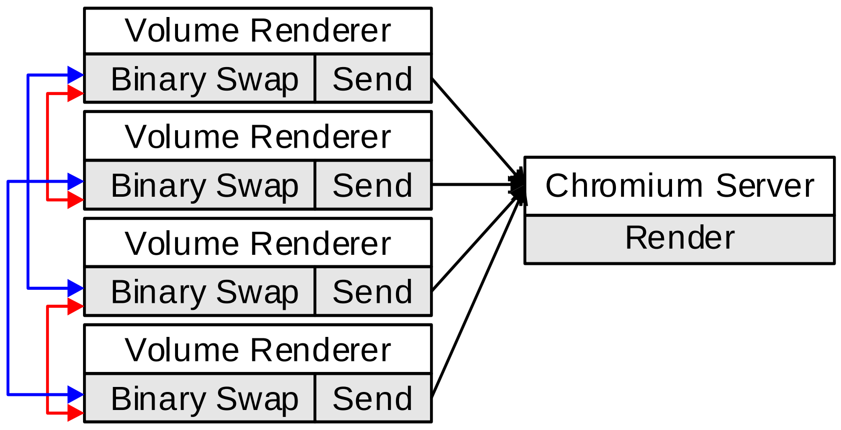


Figure 6.2: Configuration used for a four-node version of our cluster-parallel volume rendering system. Each client renders its local portion of the volume using local graphics hardware. Next, the volumes are composited using the `binaryswap` SPU. The SPUs use out-of-band communication to exchange partial framebuffers until each SPU contains one quarter of the final image. These partial images are then sent to a single server for display.

This newly composited sub-region of the image is then split in half, a different pairing is chosen, and the process repeats. If there are n nodes in our cluster, after $\log(n)$ steps each node will have completely composited $\frac{1}{n}$ of the total image. Because we are compositing transparent images using Porter and Duff's "over" operator [Porter and Duff, 1984], the sequence of pairings is chosen carefully so that blending is performed in the correct order with respect to the viewpoint. The Chromium configuration used to realize this architecture is shown in figure 6.2. The communication shown in red and blue uses Chromium's connection-based networking abstraction, described in section 3.9.

6.2 Performance and Scalability

Our scalability experiments were conducted on a cluster of sixteen nodes, each running RedHat Linux 7.2. The nodes contain an 800 MHz Pentium III Xeon, a GeForce3 with 64 MB of video memory, 256 MB of main memory, and a Myrinet network with a maximum bandwidth of approximately 100 MB/sec (bidirectional). The dataset is a $256\times256\times1024$ magnetic resonance scan of a mouse. All of our renderings are performed in a window of size 1024×256, ensuring that each voxel is sampled exactly once. Table 6.1 shows the four shaders we used to vary the achievable per-node performance. Note that a minimum of eight nodes is required to render the full mouse volume, because each node in our cluster has only 64 MB of texture memory.

Figure 6.3 shows the performance of our volume renderer as the size of the volume is scaled. In this experiment, we rendered a portion of the mouse dataset on each node in our cluster. The initial drop in performance is due to the additional framebuffer reads required, but because the binary swap algorithm keeps all the nodes busy while compositing, the graph flattens out, and we sustain nearly constant performance as the size of the volume is repeatedly doubled. At 16 nodes, we render two copies of the full $256\times256\times1024$ volume at a rate between 643 MVox/sec and 1.59 GVox/sec, depending on the shader used.

If we instead fix the size of the volume and parallelize the rendering, we quickly become limited by our pixel readback and network performance. When rendering a single $256\times256\times128$ volume split across multiple nodes, the rendering rate rapidly becomes negligible. When creating a 1024×256 image, our volume renderer's performance converges to approximately 22 frames per second. Because the parallel image compositing and final transmission for display happen sequentially, we can analyze this performance as follows: With 16 nodes, each node eventually extracts and sends $\frac{15}{16}$ of its framebuffer, requiring four bytes per pixel. The final transmission sends only $\frac{1}{16}$ of a framebuffer at three bytes per pixel, but because all of these framebuffer portions arrive at the same node, we must consider the aggregate incoming bandwidth at that node, which is a full framebuffer at three bytes per pixel. This adds up to 1.69 MB/frame, or 37.1 MB/sec. This measurement

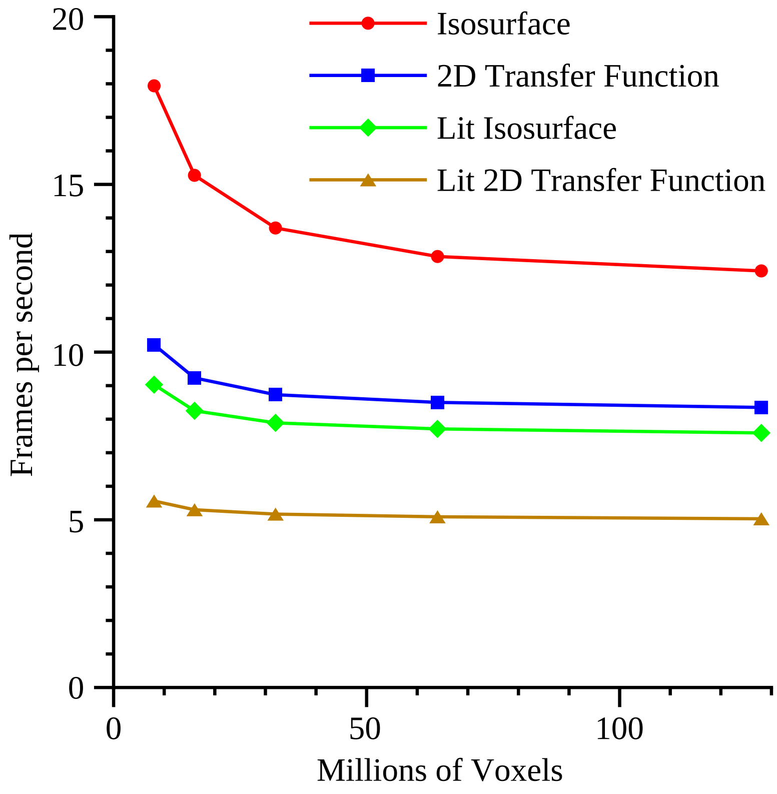
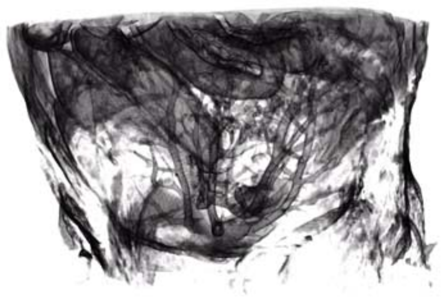
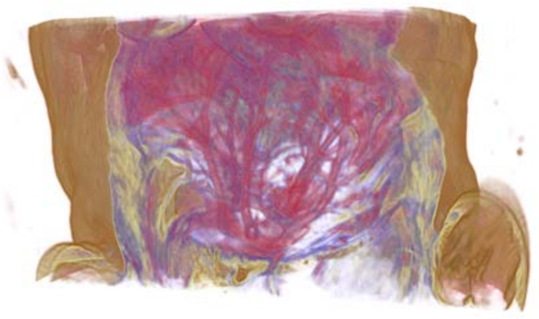


Figure 6.3: Performance of our volume renderer as larger volumes are used. In this graph, each node renders a 256×256×128 subvolume to a 1024×256 window. The data points correspond to a cluster of 1, 2, 4, 8, and 16 nodes. At 16 nodes, we are rendering two copies of the full 256×256×1024 dataset.

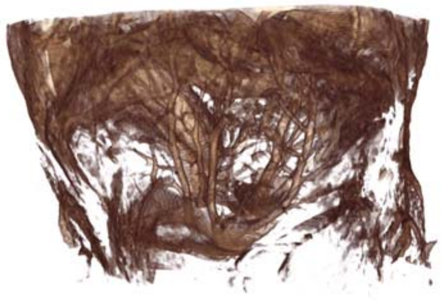
is close to our measured RGBA readback performance of the GeForce3, which is clearly the limiting factor for the `binaryswap` SPU, since our network can sustain 100 MB/sec. Future improvements in pixel readback rate and network bandwidth would result in higher framerates, as would an alpha-compositing mode for a post-scanout compositing system such as Lightning-2.



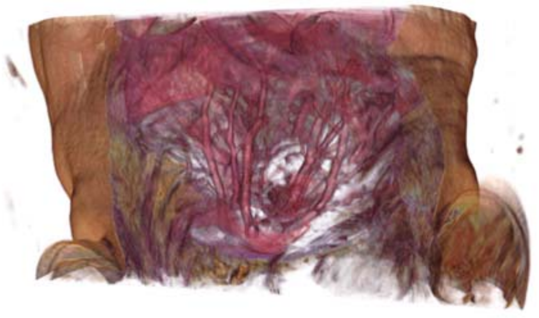
Isosurface



2D Transfer Function



Lit Isosurface



Lit 2D Transfer Function

Shader	3D textures	2D textures	Combiners	Per-Node Fill Rate
Isosurface	1	0	6	190 Mpix/sec
2D Transfer Function	1	1	4	98 Mpix/sec
Lit Isosurface	2	0	8	78 Mpix/sec
Lit 2D Transfer Function	2	1	8	42 Mpix/sec

Table 6.1: Shaders used in our volume rendering experiments. The lit 2D transfer function shader exhausts the resources of a GeForce3. Mouse dataset courtesy of the Duke Center for In Vivo Microscopy.

7. Other Chromium Applications

In this chapter, we describe two other uses of Chromium. These applications are not directly related to scalability, but demonstrate the power of non-invasive manipulation of graphics streams.

7.1 Integration With an Existing User Interface

Normally, when Chromium intercepts an application's graphics commands, that application's graphics window will be blank, with the rendering appearing in one or more separate windows, potentially distributed across multiple remote computers. Because the interface is now separated from the visualization, this behavior can interfere with the productive use of some applications. To address this problem, we have implemented an "`integration`" SPU that reincorporates remotely rendered tiles into the application's user interface. This way, users can apply a standard user interface to a parallel client.

This manipulation can also be useful for serial applications. Even though the net effect

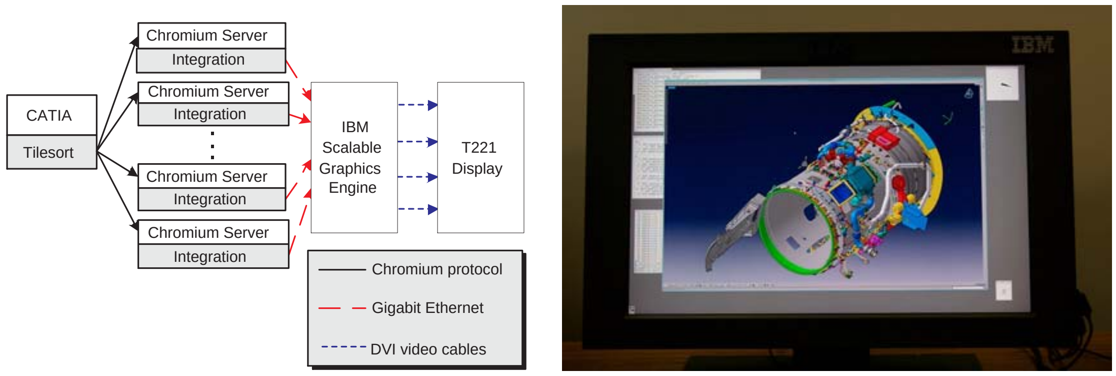


Figure 7.1: Configuration used to drive IBM's 3840×2400 T221 display using Chromium. The commercial CAD package CATIA is used to create a tiled rendering of a jet engine nacelle (model courtesy of Goodrich Aerostructures). The tiles are then re-integrated into the application's original user interface, allowing CATIA to be used as designed, despite the distribution of its graphics workload on a cluster. Due to the capacity and range of gigabit ethernet, all of the computational and 3D graphics hardware can be remote from the eventual display.

is a null transformation on the application's stream, it can aid in driving high resolution displays. For our experiments, we use the IBM T221, a 3840×2400 LCD. Few graphics cards can drive this display directly, and those that can do not have sufficient scanout bandwidth to do so at a high refresh rate. The T221 can be driven by up to four separate synchronized digital video inputs, so we can achieve higher bandwidth to the display using a cluster and special hardware such as Lightning-2 [Stoll *et al.*, 2001], or a network-attached parallel framebuffer such as IBM's Scalable Graphics Engine (SGE) [Perrine and Jones, 2001]. The SGE supports up to 16 one-gigabit ethernet inputs, can double buffer up to 16 million pixels, and can drive up to eight displays. In our tests, we used the SGE to supply four synchronized DVI outputs that collectively drive the T221 at its highest resolution. An X-Windows server for the SGE provides a standard user interface for this configuration.

The `integration` SPU is conceptually similar to the `readback` SPU in that it inherits almost all of its functionality from the `render` SPU. To extract the color information from the framebuffer, the `integration` SPU implements its own `SwapBuffers` handler, which uses the SGE to display those pixels on the T221. The configuration graph used to conduct this experiment is shown in figure 7.1. The application's graphics stream is sorted into tiles managed by multiple Chromium servers, each of which dispatches its tile's stream to the `integration` SPU. The integration SPU places the resulting pixels into X regions by *tunneling*, meaning that the pixels are transferred to the SGE's framebuffer without the involvement of the X server that manages the display. Because the SGE supports multiple simultaneous writes to the framebuffer, this technique does not unnecessarily serialize tile placement. Note that the number of tiles sent to the SGE is independent of the number of the SGE's outputs, so we use an 8-node cluster to drive the four outputs at interactive rates.

The `integration` SPU must also properly handle changes to the size of the application's rendering area. When an application window is resized, it will typically call `glViewport` to reset its drawing area. Accordingly, the `integration` SPU overrides the `render` SPU's implementation of `glViewport` to detect these changes, and adjusts the size of the render tiles if necessary. Because the `tilesort` SPU sorts based on a logical decomposition of the screen, it does not need to be notified of this change[1].

Although the `integration` SPU enables functionality that is not otherwise possible, it is still important that it not impede interactivity. For our performance experiments, we used a cluster of eight nodes running RedHat Linux 7.1, each with two 866MHz Pentium III Xeon CPUs, 1GB of RDRAM, NVIDIA Quadro graphics, and both gigabit ethernet and Myrinet 2000 networking. One of our cluster nodes runs the SGE's X-windows server in addition to the Chromium server. We successfully tested applications ranging from trivial

[1]Our example application uses only geometric primitives. In order for pixel-based primitives to be rendered correctly, the `tilesort` SPU would need to be notified when the window size changes. Alternately, the tilesort SPU could be configured to broadcast all `glDrawPixels` calls.

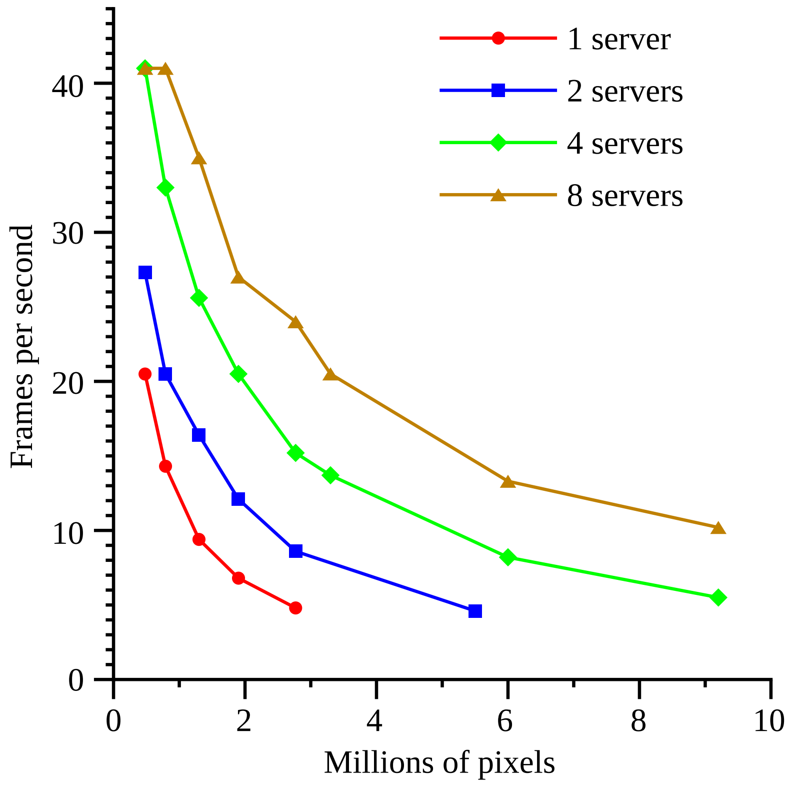


Figure 7.2: Performance of the GLUT "atlantis" demo using the `integration` SPU to drive the T221 display at different resolutions. Each curve shows the relationship between performance and resolution for a given number of rendering servers. For smaller windows, the SPU becomes limited by the vertical refresh rate of the display (41 Hz). As the resolution approaches 3840×2400 (9.2 million pixels), a small 8-server configuration still achieves interactive refresh rates.

(simple demos from the GLUT library) to a medium-complexity scientific visualization application (OpenDX) to a closed-source, high-complexity CAD package (CATIA).

The graph shown in figure 7.2 shows the average frame rate as we scale the display resolution of the T221 from 800×600 to 3840×2400. Four curves are shown, corresponding to a cluster of 1, 2, 4, and 8 nodes. Because we want to measure only the performance impact of the `integration` SPU, we rendered only small amounts of geometry (approximately 5000 vertices per frame) using the GLUT `atlantis` demo. This demo runs at a

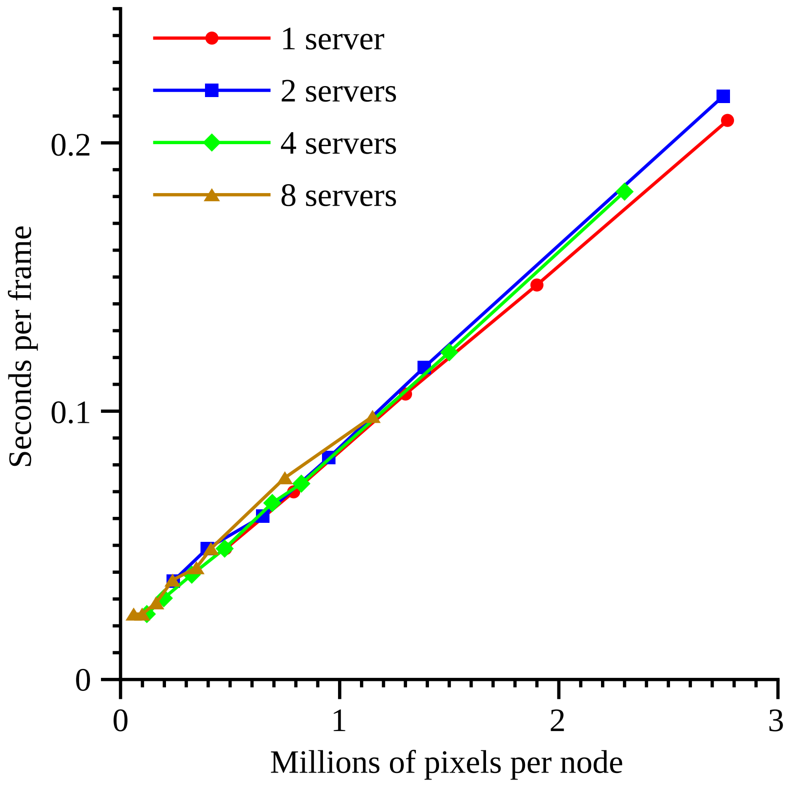


Figure 7.3: We have replotted the data from figure 7.2 to show *seconds per frame* versus pixels *per node*, to show per-node throughput. The coincidence of the four curves shows that there is insignificant overhead to doubling the number of rendering nodes, so linear speedup is achieved until the monitor refresh rate becomes the limiting performance factor.

much greater rate than the refresh rate of the display, so its effect on performance is minimal compared to the expense of extracting and transmitting tiles.

The maximum frame rate achieved using 4 or 8 nodes is 41 Hz, which is exactly the vertical refresh rate of the T221. Because the SGE requires hardware synchronization to the refresh rate, no higher frame rate can be achieved. For a given fixed resolution, the `integration` SPU achieves the expected performance increase as more rendering nodes are used, because this application is completely limited by the speed at which we can redistribute pixels. Figure 7.3 shows this phenomenon more clearly. In this graph, the same data are plotted showing seconds per frame rather than frames per second. In addition, the data have been normalized by the number of nodes used, so the quantity being measured is the pixel throughput per node. The coincidence of the four curves shows that there is no penalty associated with adding rendering nodes, so linear speedup is achieved until the display's refresh rate becomes the limiting factor. The rate at which each node can read back pixels and send them to the SGE is given by the slope of the line, which is approximately 12 MPix/second/node, or 48 MB/second/node. Extrapolating to a very small image size, the system overhead is approximately 15 milliseconds, which indicates that the maximum system response rate of the `integration` SPU is approximately 70 Hz (in the absence of monitor refresh rate limitations).

The measurements presented here give a worst-case scenario for the `integration` SPU, in which it is responsible for almost 100% of the overhead in the system. We are able to demonstrate frame rates exceeding 40Hz using only 8 nodes, and achieve an interactive 10 Hz even with each node supplying over one million pixels per frame. In addition, if measured independently, pixel readback rate and the SGE transfer rate can both provide bandwidths exceeding 23 Mpix/sec, nearly twice what they achieve when measured together. This leads us to believe that the system I/O bus or memory subsystem is underperforming when these two tasks are being performed simultaneously, an effect that will likely be eliminated with the introduction of new I/O subsystems designed specifically for high-end servers. This is a similar contention effect to that observed in section 5.8.

7.2 Stylized Drawing

For a long time, research on non-photorealistic, or "stylized", rendering focused on non-interactive, batch-mode techniques. In recent years, however, there has been considerable interest in real-time stylized rendering. Early interactive NPR systems required *a priori* knowledge of the model and its connectivity [Markosian *et al.*, 1997; Rossignac and van Emmerik, 1992]. More recently, Raskar has shown that non-trivial NPR styles can be achieved with no model analysis using either standard graphics pipeline tricks [Raskar and Cohen, 1999] or slight extensions to modern programmable graphics hardware [Raskar, 2001].

We have developed a simple stylized rendering filter that creates a flat-shaded hidden-line drawing style. Our approach is similar to that taken by Mohr and Gleicher [2001], although we show a technique that requires only finite storage. Hidden line drawing in OpenGL is a straightforward multi-pass technique, accomplished by first rasterizing all polygons to the depth buffer, and then re-rasterizing the polygon edges. The polygon depth values are offset using `glPolygonOffset` to reduce aliasing artifacts [SIGGRAPH Course Notes, 1998].

Achieving this effect in Chromium can be accomplished with a single SPU. The "`hiddenline`" SPU packs each graphics command into a buffer as if they were being prepared for network transport. This has the effect of recording the entire frame into local memory (a solution requiring only finite storage is presented below). Instead of actually sending them to a server, we instead decode the commands twice at the end of each frame, once as polygons and once as lines, achieving our desired style. The code required to achieve this transformation is shown in figure 7.4, and the visual results are shown in figure 7.5. The performance impact of this SPU is shown in figure 7.7.

There are three interesting notes regarding the actual implementation of a `hiddenline` SPU. First, the application may generate state queries that need to be satisfied immediately and not recorded. In order to do this, the entire graphics state is maintained using our state tracking library, and any function that might affect the state is passed to the state tracker

```
void hiddenline_SwapBuffers( void )
{
  /* Draw filled polygons */
  super.Clear( color and depth );
  super.PolygonOffset( 1.5f, 0.000001f );
  super.PolygonMode( GL_FRONT_AND_BACK, GL_FILL );
  super.Color3f( poly_r, poly_g, poly_b );
  PlaybackFrame( modified_child_dispatch );

  /* Draw outlined polygons */
  super.PolygonMode( GL_FRONT_AND_BACK, GL_LINE );
  super.Color3f( line_r, line_g, line_b );
  PlaybackFrame( modified_child_dispatch );

  super.SwapBuffers();
}
```

Figure 7.4: End-of-frame logic for a simple hidden-line style SPU. The entire frame is played back twice, once as depth-offset filled polygons, and once as lines. We modify the downstream SPU's dispatch table to discard calls that would affect our drawing style, such as texture enabling and color changes.

before being packed. This behavior is frequently overly cautious; most state queries are attempts to determine some fundamental limit of the graphics system (such as the maximum size of a texture), rather than querying state that was set by the application itself. Robust implementations of style filters like the `hiddenline` SPU would likely benefit from the ability to disable full state tracking.

Second, the SPU does not play back the exact calls made in the frame. Because we want to draw all polygons in the same color (and similarly for lines), the application must be prevented from enabling texturing, changing the current color, turning on lighting, changing the polygon draw style, enabling blending, changing the line width, disabling the depth test, or disabling writes to the depth buffer. To accomplish this, a new OpenGL dispatch table is built, containing mostly functions from the SPU immediately following the `hiddenline`

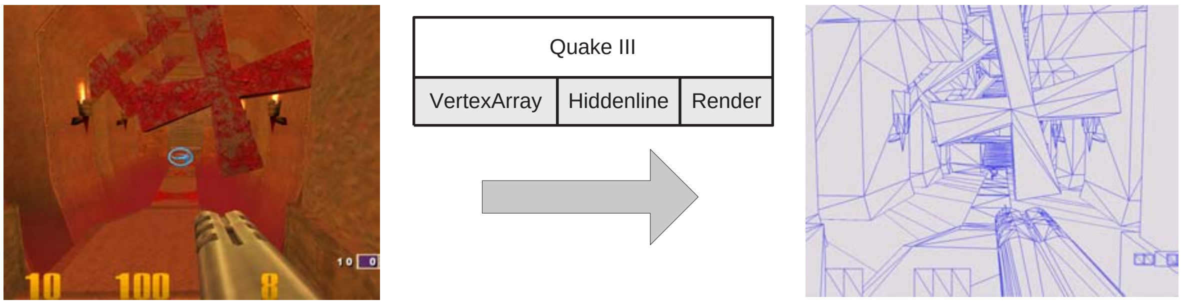


Figure 7.5: Drawing style enabled by the `hiddenline` SPU. After uses of vertex arrays are filtered out, the SPU records the entire frame, and plays it back twice to achieve a hidden-line effect. No high-level knowledge of the model is required.

SPU in its chain, but with our own versions of `glEnable`, `glDisable`, `glDepthMask`, `glPolygonMode`, `glLineWidth`, and all the `glColor` variants, which enforce these rules. Applications which rely on complex uses of these functions may not function properly using this SPU.

Finally, some care must be taken to properly handle vertex arrays. Because the semantics of vertex arrays allow for the data buffer to be changed (or discarded) after it is referenced, we cannot store vertex array calls verbatim and expect them to decode properly later in the frame. Instead, we transform uses of vertex arrays back into sequences of separate OpenGL calls. Although this could be done by the `hiddenline` SPU itself, we have found this transformation to be useful in other situations, so we have implemented the vertex array filtering in a separate "`vertexarray`" SPU. This SPU appears immediately before the `hiddenline` SPU in figure 7.5.

The `hiddenline` SPU as presented requires potentially infinite storage, since it buffers the entire frame, and therefore cannot be considered a true stream processor. One possible solution to this problem is to perform primitive assembly in the `hiddenline` SPU, drawing each stylized primitive separately. This technique is more difficult to implement, but does

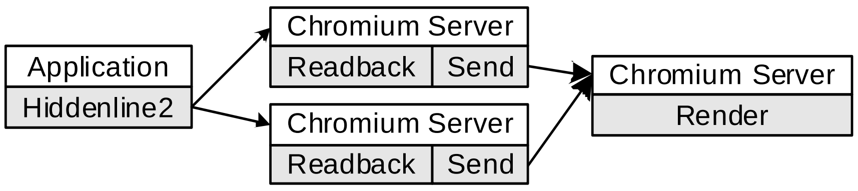


Figure 7.6: A different usage model for achieving a hidden-line drawing style. In this example, the filled polygon stream and the wireframe stream are sent to two different rendering servers and the resulting images are depth composited. This way, no single SPU needs to buffer the entire frame, and the system requires only finite resources.

satisfy our resource constraints. It will also result in a significant performance penalty for applications with a high frame rate, due to the overhead of software primitive assembly as well as the frequent added state changes.

A better solution to this problem is to use more than one node in our cluster, as shown in figure 7.6. Rather than buffering the entire frame, we could send the entire stream verbatim to two servers, one rendering the incoming stream as depth-offset polygons, the other as lines. Instead of writing two new (trivial) SPUs for each of these rendering styles, we would inject the appropriate `glPolygonMode` and `glPolygonOffset` calls into the streams before transmission. We then use the `readback` and `send` SPUs to combine the two renderings using a depth-compositing network, as described in section 6.1. Note that we could more economically use our resources by rendering depth-offset polygons locally and forwarding the stream to a single line-rendering node (or vice versa), thereby requiring only three nodes instead of four, although this would require a more complex implementation.

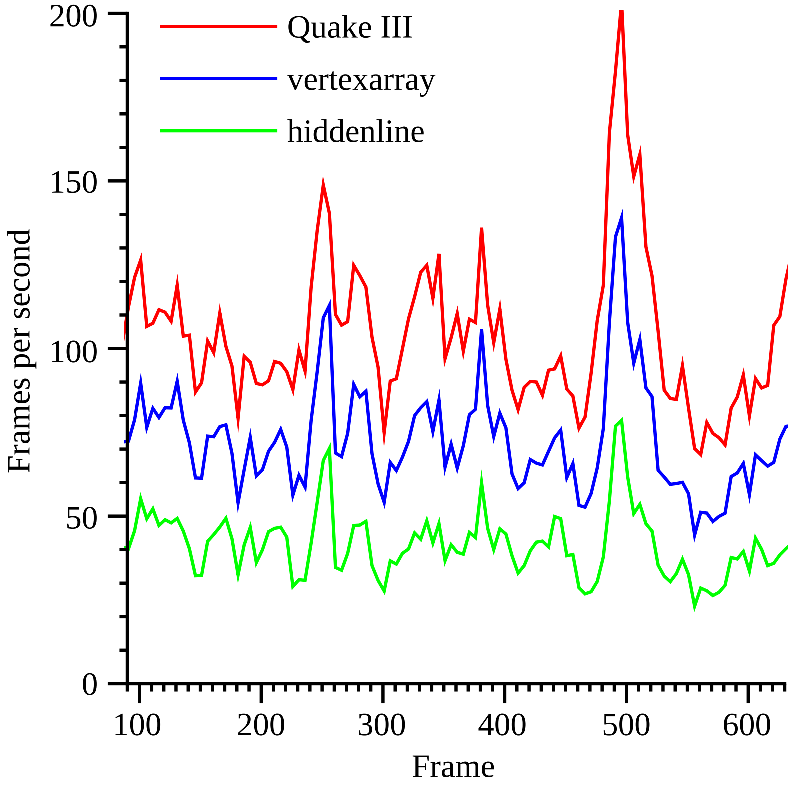


Figure 7.7: Performance of Quake III running a prerecorded demo. The first 90 frames are devoted to an introductory splash screen and are not shown here. The red curve shows the performance achieved by the application alone. The blue curve shows the same demo using just the `vertexarray` SPU, and the green curve gives the performance of the demo rendering with a hidden-line style. Despite more than a 2:1 reduction in speed, the demo still runs at approximately 40-50 frames per second.

8. Discussion and Future Work

In their seminal paper on virtual graphics, Voorhies, Kirk and Lathrop note that providing a level of abstraction between an application and the graphics hardware "allows for cleaner software design, higher performance, and effective concurrent use of the display" [Voorhies *et al.*, 1988]. We believe that the power and implications of these observations have not yet been fully explored. Chromium provides a compelling mechanism with which to further investigate the potential of virtual graphics. Because Chromium provides a complete graphics API (many of the key SPUs such as `tilesort`, `send`, and `render` pass almost all of the OpenGL conformance tests), it is no longer necessary to write custom applications to test new ideas in graphics API processing. Also, the barrier to entry is quite low; for example, the `hiddenline` SPU described in section 7.2 adds only approximately 250 lines of code to Chromium's SPU template.

We believe that providing increased performance or scalability within the constraints of a familiar operating environment or programming API is critical for adoption of a new system. Often the benefits of moving to an unfamiliar system or API are overshadowed by the difficult learning curve or the expensive process of porting software, resulting in significant inertia. It follows that performance and scalability should be augmented while remaining as backwards-compatible as possible.

Of course, it is sometimes necessary to slightly redefine semantics in order to continue to scale. The Parallel API is a good example of this. Existing serial applications do not parallelize automatically with Chromium, but if the application's visualization domain is easy to decompose, the transformation to a scalable parallel application using Chromium is both straightforward and unimposing. As more flexibility and performance is desired, "Chromium-aware" applications can use novel features of the graphics system directly, usually through the use of OpenGL hints or Chromium-specific extensions to OpenGL.

The real power of Chromium derives from its flexibility. Because a Chromium-based cluster is based on commodity parts, it is easy and inexpensive to build a parallel rendering system. Although there can be a tradeoff between using commodity parts and parallel efficiency, the ability to reconfigure the system to meet an application's load balancing and resource needs is a large advantage for commodity-based parallel rendering solutions.

Because the techniques used to provide scalability are independent of specific graphics adapters and networking technology, any component in our system may be upgraded at any time to obtain better performance. In particular, we believe that Chromium's performance on a 16 to 32 node cluster will improve dramatically with the introduction of new server-area networking technology such as InfiniBand.

8.1 Sort-First Scalability Limits

We have demonstrated that `tilesort`'s sort-first approach to parallel rendering on clusters provides excellent scalability for a variety of applications with a configuration of up to 16-servers and 16-clients. Our experiments indicate that the system would scale well in a 32-server, 32-client setup if the cluster were bigger, or if the network had better support for all-to-all communication. However, there is a limit to the amount of screen-space parallelism available at any given output size. This limit will prevent a sort-first approach from scaling to much bigger configurations, such as clusters of 128 nodes or more. For clusters that large, the tile size becomes small enough that it is very difficult to provide a good load balance for any non-trivial application without introducing a prohibitively high overlap factor. One possible solution to this problem would be to provide dynamic screen tiling, either automatically (using frame-coherent heuristics) or with application support. We believe alternate architectures such as sort-last image composition would scale better on larger clusters, but this will likely come at the cost of ordered semantics.

8.2 Texture Management

`tilesort`'s client implementation treats texture data as state elements, and lazily updates it to servers as needed. In the worst case, this will result in each texture being replicated on every server node in the system. This replication is a direct consequence of our desire to use commodity graphics accelerators in our cluster; it is not possible to introduce a stage of communication to remotely access texture memory. `tilesort`'s naive approach to parallel texture management can be a limitation for some applications. More work needs to be done in this area. One possible approach would leverage Igehy et al's work in parallel texture caching [Igehy *et al.*, 1999].

Textures for sort-last applications are more complex. Because Chromium does not provide any communication between application nodes, it is not possible for nodes in a parallel application to share textures. This means that not only does the texture itself need to be (potentially) replicated across all the nodes, but the application must manage this replication itself. Chromium would benefit from an architecture-independent parallel texture declaration and management library, particularly for memory-intensive 3D textures used in volume rendering.

8.3 `tilesort` Latency

There are two main sources of latency in `tilesort`: the display reassembly stage, and the buffering of commands on the client. When using Lightning-2, display reassembly will add exactly one frame of latency. While single-frame latency is usually acceptable for interactive applications, it can be a problem for certain virtual reality applications. The overhead of using software image reassembly will usually be much higher (on the order of 50-100 milliseconds), although it will vary with the image size.

The latency due to command buffering will depend on the size of the network buffers. `tilesort`'s default buffer size is 128KB, which we can fill with geometry in half a millisecond, given our packing rate of 20 MTris/sec (recall that a triangle occupies 13 bytes in our protocol). Additional latency can occur due to network transmission, although the latency of most high-speed cluster interconnects is less than 20 μs. Finally, since the server cannot process the buffer until it has been completely received, we incur slightly over one millisecond of additional latency for a 128KB buffer on a network with 100 MB/sec of bandwidth.

8.4 Sort-First Consistency Model

In their paper on the Parallel API, Igehy, Stoll and Hanrahan defined the concept of sequential consistency for parallel graphics systems [Igehy *et al.*, 1998]. They extended the notion of traditional sequential consistency [Lamport, 1979] by defining atomic operations in the graphics system. The graphics system presented in their paper, called Argus, provided command-sequential consistency, which means that each OpenGL command is considered to be an atomic operation. If two contexts each draw a triangle without any explicit ordering or depth buffering, a command-sequentially consistent architecture will produce one of two pictures, depending on which triangle ends up on top.

`tilesort` provides a weaker form of consistency called framebuffer-sequential consistency. In this consistency model, only operations on the framebuffer are considered to be atomic. The reason `tilesort` is only able to provide this consistency model is that there is no guarantee that servers will make the same scheduling decisions across tiles. Providing this guarantee across tiles would require a module in `tilesort` similar to Pomegranate's sequencer, which had global knowledge of all the ordering decisions to be made across multiple pipelines [Eldridge *et al.*, 2000]. Such a module would be impractical in our environment and limit the scalability of our cluster-based approach. Each tile in `tilesort`, considered in isolation, is command-sequentially consistent, but the final image is not. If two contexts draw a triangle without any explicit ordering or depth buffering, `tilesort` may show one or the other on top on a per-tile basis.

The OpenGL API does not require *any* form of sequential consistency. However, Igehy notes that any graphics system that supports the Parallel API should provide at least framebuffer-sequential consistency to guarantee reasonable behavior. Parallel applications that need to always produce the same final image (as opposed to our unordered example above) can achieve this in one of two ways. First, they can use depth buffering and draw opaque geometry. In order to properly support depth buffering for parallel programs, a graphics system must be at least framebuffer-sequentially consistent. Otherwise, a read-modify-write to the depth buffer would not be an atomic operation and incorrect images could result. Second, an application may express its ordering requirements through the use of the Parallel API. `tilesort` provides both of these capabilities, and we have not been able to conceive of an application that both produces deterministic images and also relies on the stronger command-sequential consistency model provided by Argus and Pomegranate.

8.5 Future Directions

In the future, we would like to see Chromium applied to new application domains, especially new ideas in scalable interactive graphics on clusters. Of particular interest is the problem of managing enormous time-varying datasets, both volumetric and polygonal. Today's time-varying volumetric datasets can easily exceed 30 terabytes in size. We intend to build a new parallel rendering application designed specifically for interactively visualizing these datasets on a cluster, using Chromium as the underlying transport, rendering, and compositing mechanism. A key research component of this new system will be properly balancing the allocation of I/O bandwidth between the rendering network, the compositing network (if any), and parallel disk access.

We are particularly interested in building infrastructure to support flexible remote graphics. We believe that a clean separation between a scalable graphics resource and the eventual display has the potential to change the way we use graphics every day. We are actively pursuing a new direction to make scalable cluster-based graphics appear as a remote, shared service akin to a network mounted filesystem.

We would also like to explore the possibilities afforded by non-invasive analysis of graphics API streams. It has already been shown that some non-photorealistic rendering styles can be achieved this way; we intend to apply stream transformations and analysis to other domains. One possibility is the automatic real-time generation of cutaway and exploded views of objects. We believe that by allowing such views, we can greatly enhance a user's ability to understand and visualize complex 3D spatial relationships between objects. Furthermore, graphics stream manipulation need not be restricted to new drawing styles. By visualizing the graphics state itself alongside a running program and its source code, an extremely useful debugging tool could be created. Such a tool could attempt to automatically answer one of the most challenging questions in computer graphics: "Why is my window black?". Tools that analyze the graphics stream rather than modify it could also be used for on-the-fly performance analysis.

Most of all, we hope that Chromium will be adopted as a common low-level mechanism for enabling new graphics algorithms, particularly for clusters. If this happens, research results in cluster graphics can more easily be applied to existing problems outside the original researcher's lab. Chromium is a completely open-source project that supports both Microsoft Windows and several variants of UNIX. It can be downloaded for free from `http://chromium.sourceforge.net`.

Bibliography

[Akeley and Jermoluk, 1988] Kurt Akeley and Tom Jermoluk. High-Performance Polygon Rendering. In *Proceedings of SIGGRAPH 88*, pages 239–246, August 1988.

[Akeley, 1993] Kurt Akeley. RealityEngine Graphics. In *Proceedings of SIGGRAPH 93*, pages 109–116, August 1993.

[Babu and Widom, 2001] Shivnath Babu and Jennifer Widom. Continuous Queries over Data Streams. *SIGMOD Record*, pages 109–120, September 2001.

[Blanke *et al.*, 2000] William Blanke, Chandrajit Bajaj, Donald Fussel, and Xiaoyu Zhang. The Metabuffer: A Scalable Multiresolution

Multidisplay 3-D Graphics System Using Commodity Rendering Engines. Tr2000-16, University of Texas at Austin, February 2000.

[Boden *et al.*, 1995] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[Buck *et al.*, 2000] Ian Buck, Greg Humphreys, and Pat Hanrahan. Tracking Graphics State for Networked Rendering. In *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 87–95, August 2000.

[Cortes *et al.*, 2000] Corrina Cortes, Kathleen Fisher, Daryl Pregibon, Anne Rodgers, and Frederick Smith. Hancock: A Language for Extracting Signatures from Data Streams. In *Proceedings of 2000 ACM SIGKDD International Conference on Knowledge and Data Mining*, pages 9–17, August 2000.

[Cox, 1995] Michael Cox. *Algorithms for Parallel Rendering*. PhD thesis, Princeton University, 1995.

[Cruz-Neira *et al.*, 1993] Caroline Cruz-Neira, Daniel Sandin, and Thomas DeFanti. Surround-Screen Projection Based Virtual Reality: The Design and Implementation of the CAVE. In *Proceedings of SIGGRAPH 1993*, pages 135–142, July 1993.

[Cunniff, 2000] Ross Cunniff. Visualize fx Graphics Scalable Architecture. In *Proceedings of Eurographics Hardware/SIGGRAPH Hot3D*, pages 29–38, August 2000.

[Czernuszenko *et al.*, 1997] Marek Czernuszenko, Dave Pape, Dan Sandin, Tom DeFanti, Greg Dawe, and Maxine Brown. The ImmersaDesk and InfinityWall Projection-Based Virtual Reality Displays. *Computer Graphics*, 31(2):46–49, May 1997.

[DVI Specification, 1998] Digital Video Interface Specification, 1998. `http://www.ddwg.org`.

[Eldridge *et al.*, 2000] Matthew Eldridge, Homan Igehy, and Pat Hanrahan. Pomegranate: A Fully Scalable Graphics Architecture. In *Proceedings of SIGGRAPH 2000*, pages 443–454, July 2000.

[Eldridge, 2001] Matthew Eldridge. *Designing Graphics Architectures around Scalability and Communication*. PhD thesis, Stanford University, 2001.

[Ellsworth *et al.*, 1990] David Ellsworth, Howard Good, and Brice Tebbs. Distributing Display Lists on a Multicomputer. In *Proceedings of ACM Symposium on Interactive 3D Graphics*, pages 147–154, 1990.

[Eyles *et al.*, 1997] John Eyles, Steven Molnar, John Poulton, Trey Greer, Anselmo Lastra, Nick England, and Lee Westover. PixelFlow: The Realization. In *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 57–68, August 1997.

[Funkhouser, 1996] Thomas Funkhouser. Coarse-Grained Parallelism for Hierarchical Radiosity Using Group Iterative Methods. In *Proceedings of SIGGRAPH 1996*, pages 343–352, August 1996.

[Giertsen and Peterson, 1993] Christopher Giertsen and Johnny Peterson. Parallel Volume Rendering on a Network of Workstations. *IEEE Computer Graphics and Applications*, 13(6):16–23, November 1993.

[Heermann, 1998] Philip D. Heermann. Production Visualization for the ASCI One TeraFLOPS Machine. In *Proceedings of IEEE Visualization 1998*, pages 459–462, 1998.

[Heirich and Moll, 1999] Alan Heirich and Laurent Moll. Scalable Distributed Visualization Using Off-the-Shelf Components. In *Proceedings of the Parallel Visualization and Graphics Symposium 1999*, October 1999.

[Humphreys and Hanrahan, 1999] Greg Humphreys and Pat Hanrahan. A Distributed Graphics System for Large Tiled Displays. In *Proceedings of IEEE Visualization 1999*, pages 215–224, October 1999.

[Humphreys *et al.*, 2000] Greg Humphreys, Ian Buck, Matthew Eldridge, and Pat Hanrahan. Distributed Rendering for Scalable Displays. In *Proceedings of Supercomputing 2000*, November 2000.

[Humphreys *et al.*, 2001] Greg Humphreys, Matthew Eldridge, Ian Buck, Matthew Everett, Gordon Stoll, and Pat Hanrahan. WireGL: A Scalable Graphics System for Clusters. In *Proceedings of SIGGRAPH 2001*, July 2001.

[Humphreys *et al.*, 2002] Greg Humphreys, Mike Houston, Ren Ng, Sean Ahern, Randall Frank, Peter Kirchner, and James T. Klosowski. Chromium: A Stream Processing Framework for Interactive Graphics on Clusters of Workstations. In *Proceedings of SIGGRAPH 2002*, July 2002.

[Igehy *et al.*, 1998] Homan Igehy, Gordon Stoll, and Pat Hanrahan. The Design of a Parallel Graphics Interface. In *Proceedings of SIGGRAPH 1998*, pages 141–150, July 1998.

[Igehy *et al.*, 1999] Homan Igehy, Matthew Eldridge, and Pat Hanrahan. Parallel Texture Caching. In *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 95–106, August 1999.

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

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

[Lamport, 1979] Leslie Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 28(9):241–248, 1979.

[Levoy *et al.*, 2000] Marc Levoy, Karri Pulli, Brian Curless, Szymon Rusinkiewicz, David Koller, Lucas Pereira, Matt Ginzton, Sean Anderson, James Davis, Jeremy Ginsberg, Jonathan Shade, and Duane Fulk. The Digital Michelangelo Project: 3D Scanning of Large Statues. In *Proceedings of SIGGRAPH 2000*, pages 131–144, July 2000.

[Lorensen and Cline, 1987] William E. Lorensen and Harvey E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In *Proceedings of SIGGRAPH 1987*, pages 163–169, July 1987.

[Ma *et al.*, 1994] Kwan-Liu Ma, James S. Painter, Charles D. Hansen, and Michael F. Krogh. Parallel Volume Rendering Using Binary-Swap Compositing. *IEEE Computer Graphics and Applications*, 14(4):59–68, July 1994.

[Mark and Proudfoot, 2001] William Mark and Kekoa Proudfoot. The F-Buffer: A Rasterization Order FIFO Buffer for Multi-Pass Rendering. In *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 57–64, August 2001.

[Markosian *et al.*, 1997] Lee Markosian, Michael Kowalski, Samuel Trychin, Lubomir Bourdev, Daniel Goldstein, and John Hughes. Real-Time Nonphotorealistic Rendering. In *Proceedings of SIGGRAPH 1997*, pages 415–420, 1997.

[Mohr and Gleicher, 2001] Alex Mohr and Michael Gleicher. Non-Invasive, Interactive, Stylized Rendering. In *ACM Symposium on Interactive 3D Graphics*, pages 175–178, March 2001.

[Moll *et al.*, 1999] Laurent Moll, Alan Heirich, and Mark Shand. Sepia: Scalable 3D Compositing Using PCI Pamette. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines 1999*, pages 146–155, April 1999.

[Molnar *et al.*, 1992] Steven Molnar, John Eyles, and John Poulton. PixelFlow: High-Speed Rendering Using Image Composition. In *Proceedings of SIGGRAPH 92*, pages 231–240, July 1992.

[Molnar *et al.*, 1994] Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32, July 1994.

[Montrym *et al.*, 1997] John S. Montrym, Daniel R. Baum, David L. Dignam, and Christopher J. Migdal. InfiniteReality: A Real-Time Graphics System. In *Proceedings of SIGGRAPH 97*, pages 293–302, August 1997.

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

[O'Callaghan *et al.*, 2002] Liadan O'Callaghan, Nina Mishra, Adam Meyerson, Sudipto Guha, and Rajeev Motwani. Streaming-Data Algorithms for High-Quality Clustering. In *Proceedings of IEEE International Conference on Data Engineering*, March 2002.

[Owens *et al.*, 2000] John D. Owens, William J. Dally, Ujval J. Kapasi, Scott Rixner, Peter Mattson, and Ben Mowery. Polygon Rendering on a Stream Architecture. In *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 23–32, August 2000.

[Peercy *et al.*, 2000] Mark Peercy, Marc Olano, John Airey, and Jeffrey Ungar. Interactive Multi-Pass Programmable Shading. In *Proceedings of SIGGRAPH 2000*, pages 425–432, August 2000.

[Perrine and Jones, 2001] Kenneth Perrine and Donald Jones. Parallel Graphics and Interactivity with the Scaleable Graphics Engine. In *Proceedings of IEEE Supercomputing 2001*, November 2001.

[Pixar, 1998] Pixar Animation Studios. *PhotoRealistic RenderMan Toolkit.*, 1998.

[Porter and Duff, 1984] Thomas Porter and Tom Duff. Compositing Digital Images. In *Proceedings of SIGGRAPH 1984,* pages 253–259, July 1984.

[PowerWall, 1994] PowerWall, 1994. `http://www.lcse.umn.edu/research/powerwall/powerwall.html`.

[Proudfoot *et al.*, 2001] Kekoa Proudfoot, William Mark, Svetoslav Tzvetkov, and Pat Hanrahan. A Real Time Procedural Shading System for Programmable Graphics Hardware. In *Proceedings of SIGGRAPH 2001*, pages 159–170, August 2001.

[Raskar and Cohen, 1999] Ramesh Raskar and Michael Cohen. Image Precision Silhouette Edges. In *ACM Symposium on Interactive 3D Graphics*, pages 135–140, April 1999.

[Raskar, 2001] Ramesh Raskar. Hardware Support for Non-Photorealistic Rendering. In *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 41–46, August 2001.

[Recker *et al.*, 1990] Rodney Recker, David George, and Donald Greenberg. Acceleration Techniques for Progressive Refinement Radiosity. In *ACM Symposium on Interactive 3D Graphics,* pages 59–66, 1990.

[Reynolds and Fatica, April 2000] William Reynolds and Massimiliano Fatica. Stanford Center for Integrated Turbulence Simulations. *IEEE Computing in Science and Engineering*, 2(2):54–63, April 2000.

[Rohlf and Helman, 1994] John Rohlf and James Helman. IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics. In *Proceedings of SIGGRAPH 94*, pages 381–394, July 1994.

[Rossignac and van Emmerik, 1992] Jareck Rossignac and Maarten van Emmerik. Hidden Contours on a Framebuffer. In *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, September 1992.

[Rusinkiewicz and Levoy, 2001] Szymon Rusinkiewicz and Marc Levoy. Streaming QSplat: A Viewer for Networked Visualization of Large, Dense Models. In *ACM Symposium on Interactive 3D Graphics*, pages 63–68, 2001.

[Samanta *et al.*, 1999] Rudrajit Samanta, Jiannan Zheng, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh. Load Balancing for Multi-Projector Rendering Systems. In *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 107–116, August 1999.

[Samanta *et al.*, 2000a] Rudrajit Samanta, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh. Sort-First Parallel Rendering with a

Cluster of PCs. In *SIGGRAPH 2000 Technical Sketch*, August 2000.

[Samanta *et al.*, 2000b] Rudrajit Samanta, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh. Hybrid Sort-First and Sort-Last Parallel Rendering with a Cluster of PCs. In *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 97–108, August 2000.

[Samanta *et al.*, 2001] Rudrajit Samanta, Thomas Funkhouser, and Kai Li. Parallel Rendering with K-Way Replication. In *IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, October 2001.

[Segal and Akeley, 1999] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 1.2.1)*. 1999. `ftp://ftp.sgi.com/opengl/doc/opengl1.2/`.

[SGI Multipipe, 2000] SGI Multipipe, 2000. `http://www.sgi.com/software/multipipe/`.

[SGI Vizserver, 1999] SGI Vizserver, 1999. `http://www.sgi.com/software/vizserver/`.

[SIGGRAPH Course Notes, 1998] *Advanced Graphics Programming Techniques Using OpenGL*. SIGGRAPH 1998 Course Notes., 1998.

[Stoll *et al.*, 2001] Gordon Stoll, Matthew Eldridge, Dan Patterson, Art Webb, Steven Berman, Richard Levy, Chris Caywood, Milton

Taveira, Stephen Hunt, and Pat Hanrahan. Lightning-2: A High-Performance Display Subsystem for PC Clusters. In *Proceedings of SIGGRAPH 2001*, July 2001.

[Torborg, 1987] Jay Torborg. A Parallel Processor Architecture for Graphics Arithmetic Operations. In *Proceedings of SIGGRAPH 1987*, pages 197–204, July 1987.

[Voorhies *et al.*, 1988] Douglas Voorhies, David Kirk, and Olin Lathrop. Virtual Graphics. In *Proceedings of SIGGRAPH 1988*, pages 247–253, August 1988.