# Chromium: A Stream-Processing Framework for Interactive Rendering on Clusters

Greg Humphreys* Mike Houston* Ren Ng* Randall Frank† Sean Ahern† Peter D. Kirchner‡
James T. Klosowski‡

*Stanford University †Lawrence Livermore National Laboratory ‡IBM T.J. Watson Research Center

## Abstract

We describe Chromium, a system for manipulating streams of graphics API commands on clusters of workstations. Chromium's stream filters can be arranged to create sort-first and sort-last parallel graphics architectures that, in many cases, support the same applications while using only commodity graphics accelerators. In addition, these stream filters can be extended programmatically, allowing the user to customize the stream transformations performed by nodes in a cluster. Because our stream processing mechanism is completely general, any cluster-parallel rendering algorithm can be either implemented on top of or embedded in Chromium. In this paper, we give examples of real-world applications that use Chromium to achieve good scalability on clusters of workstations, and describe other potential uses of this stream processing technology. By completely abstracting the underlying graphics architecture, network topology, and API command processing semantics, we allow a variety of applications to run in different environments.

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

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

## 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 slow 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 largely been 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 utilize more easily 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, can be used as the underlying mechanism for any cluster-graphics algorithm by having the algorithm use OpenGL to move geometry and imagery across a network as required. In addition, existing OpenGL applications can use a cluster with very few modifications, because Chromium provides an industry-standard graphics API that virtualizes the disjoint rendering resources present in a cluster. In some cases, the application does not even need to be recompiled. Compatibility with existing applications may 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 consider only 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 [33], which insulates applications from the details of the underlying implementations of a common API.

## 2 Background and Related Work

### 2.1 Cluster Graphics

Clusters have long been used for parallelizing traditionally non-interactive graphics tasks such as ray-tracing, radiosity [5, 25], and volume rendering [6]. Other cluster-parallel rendering efforts have largely concentrated on exploiting inter-frame parallelism rather than trying to make each individual frame run faster [20]. 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 [9]. Although the system described in that paper ran on an SGI InfiniteReality, 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 [7]. 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 [29, 31]. 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 [30]. 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 [28]. Although these papers present an excellent study of differing data-management strategies in a clustered environment, they all provide *algorithms* rather than *mechanisms*. Applying 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 [8]. 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 [10]. WireGL can use either software-based image reassembly or custom hardware such as Lightning-2 [32] 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.

### 2.2 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 [4], multimedia, rendering of remotely stored 3D models [27], database queries [2], and theoretical computer science [16]. 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 a potentially infinite 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 this 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[1]. Owens et al. implemented an OpenGL-based polygon renderer on Imagine, a programmable stream processor [17]. Using Imagine, they achieved performance that is competitive with custom hardware while 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 [14]. 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 System Architecture

The overall design of Chromium was influenced by Stanford's WireGL system [8]. Although the sort-first architecture implemented by WireGL is fairly restrictive, one critical aspect of the design led directly to Chromium: The wire protocol used to move image tiles from the servers to the compositor is the same as the networked-OpenGL protocol used to move geometry from the clients to the servers. In effect, WireGL's servers themselves become clients of a second parallel rendering application, which uses imagery as its fundamental drawing primitive. This means that the compositing node is not special; in fact, it is just another instance of the same network server executing OpenGL commands and resolving ordering constraints on behalf of some parallel client.

If we consider a sequence of OpenGL commands to be a stream, WireGL provides three main stream "filters". First, it can sort a serial stream into tiles. Next, it can dispatch a stream to a local implementation of OpenGL. Finally, WireGL can read back a framebuffer and generate a new stream of image-drawing commands. In WireGL, these stream transformations can only be realized at specific nodes in the cluster (e.g., an application's stream can only be sorted). To arrive at Chromium's design, we realized that it would be useful to perform other transformations on API streams, and it would also be necessary to arrange cluster nodes in a more generic topology than WireGL's many-to-many-to-few arrangement.

### 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 actually divided into two parts: a *transformation* portion and a *serialization* portion.

[1] Because most graphics API's 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 [18, 22], and extensions to the graphics pipeline have been proposed to further generalize its computational expressiveness [12].

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 independent 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 extensions to the OpenGL API that provide barriers and semaphores, as proposed by Igehy et al [10]. 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 similar to the one described by Buck et al [3]. This state representation allows for 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*. 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.

### 3.2 OpenGL Stream Processing

Stream transformations are performed by OpenGL "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 1. The client loads the `tilesort` SPU, which incorporates all of the sort-first stream processing logic from WireGL. 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, giving identical semantics and similar performance to the tiled display system described by Humphreys et al [7]. Notice that in figure 1, the graph edges originate from the `tilesort` SPU, not the application itself. This convention is used because the SPU in fact manages its own network resources, originates connections to servers, and generates traffic.

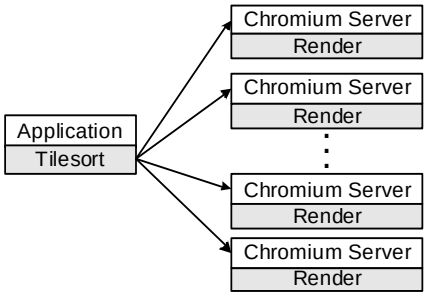



Figure 1: A simple Chromium configuration. In this example, a serial application is made to run on a tiled display using a sort-first stream processor called `tilesort`.

### 3.3 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 style" 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; it is straightforward for the client to 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 API's, they can change their own exported interface on the fly and 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. When the wireframe SPU wishes to set the drawing mode to wireframe during initialization, it calls the `passthrough` SPU's implementation of `glPolygonMode`.

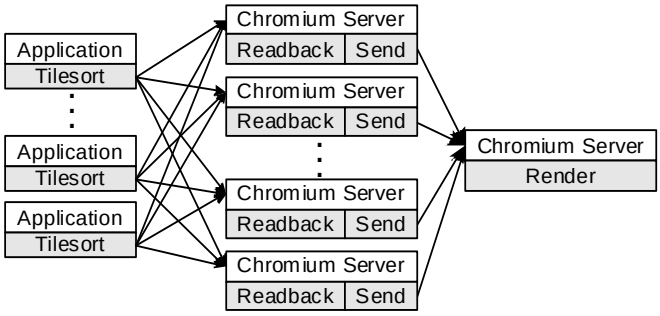



Figure 2: Chromium configured as a complete WireGL replacement. A parallel application drives a tiled display using the sort-first logic in the `tilesort` SPU. Imagery is then read back from the servers managing those tiles and sent to a final compositing server for display.

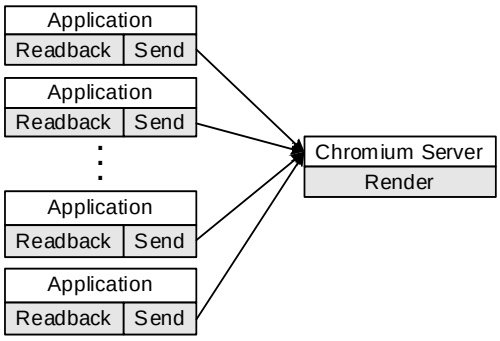



Figure 3: Another possible Chromium configuration. In this example, nodes in a parallel application render their portion of the scene directly to their local hardware. The color and depth buffers are then read back and transmitted to a final compositing server, where they are combined to produce the final image.

### 3.5 Provided Tools and SPUs

Chromium provides four libraries that encapsulate frequently performed stream operations. The first is a stream packing library. 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, as described in section 4.3. We use a very similar encoding method to the one described by Buck et al [3]. It incurs almost no wasted space, retains natural argument alignment, and allows a group of command "opcodes" and their arguments to be sent with a single call to the networking library.

Second, we provide a stream unpacking library. This library decodes an already serialized representation of a sequence of commands and dispatches those commands to a given SPU. This library is primarily used by Chromium's network server to handle incoming network traffic, but it can also be used by SPUs that need to locally buffer a portion of a stream in order to perform more global analysis or make multiple passes over that portion.

The third is a point-to-point connection-based networking abstraction. This library abstracts the details of the underlying transport mechanism; we have implemented this API on top of TCP/IP and Myrinet. In addition, the library can be used by SPUs and applications to communicate with each other along channels other than those implied by the configuration graph described in section 3.1. This out-of-band communication allows complex compositing SPUs to be built, such as the one described in section 4.1.

Finally, Chromium includes a complete OpenGL state tracker. In addition to maintaining the entire OpenGL state, this library can efficiently compute the difference between two graphics contexts, generating a call to a given SPU for every discrepancy found. This efficient context differencing operation is due to a hierarchical representation described by Buck et al [3].

In addition to these support libraries, Chromium provides a number of SPUs that can be used as is or extended to realize the desired stream transformation. There are too many SPUs to list here; a complete list can be found in the Chromium documentation at `http://chromium.sourceforge.net`.

### 3.6 Realizing Parallel Rendering Architectures

We now present two examples of parallel rendering architectures that can be realized using Chromium. As described by Molnar et al., parallel rendering architectures can be classified according to the point in the graphics pipeline at which data is "sorted" from an object-parallel distribution to an image-parallel distribution [15].

The first configuration, shown in figure 2, shows a sort-first graphics architecture that functions identically to WireGL. As in figure 1, we use the `tilesort` SPU to sort the streams into tiles. Each intermediate server serializes its incoming streams and passes the result to the `readback` SPU. The `readback` SPU inherits from the `render` SPU using the mechanism described in section 3.4, so the streams are rendered on the locally housed graphics hardware. However, the `readback` SPU provides its own implementation of `SwapBuffers`, so at the end of the frame it extracts the framebuffer and uses `glDrawPixels` to pass the pixel data to another SPU. In the figure, each pixel array is passed to a `send` SPU, which transmits the data to a final server for tile reassembly. Each `readback` SPU is configured at startup to know where its tiles should end up in the final display; these coordinates are passed to the `send` SPU using `glRasterPos`. The `readback` SPU also uses Igehy's parallel graphics synchronization extensions [10] to ensure that the tiles all arrive at their destination before the final rendering server displays its results. This final tile reassembly step could also be performed using custom hardware such as Lightning-2 [32].

A dramatically different architecture is shown in figure 3. In this figure, the `readback` SPU is loaded directly by the applications. Recall that the `readback` SPU dispatches all of 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. In addition, when doing depth compositing in Chromium, it can be beneficial to write a special SPU to perform the composite in software, because compositing depth images in OpenGL requires using the stencil buffer in a way that is quite slow on many architectures. A more advanced (and practical) Chromium-based sort-last architecture is presented in section 4.1.

Because Chromium provides a virtual graphics pipeline with a parallel interface, the application in figure 3 could be run unmodified on the architecture in figure 2 simply by specifying a different configuration DAG. The architectures may provide different semantics (e.g., the sort-last architecture cannot guarantee ordering constraints), but the application need not be aware of the change.

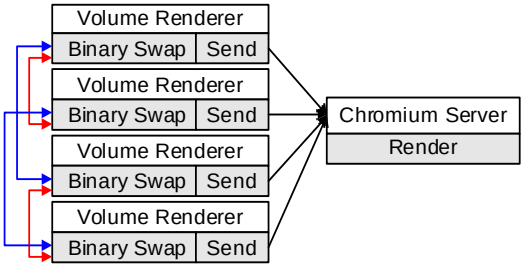



Figure 4: Configuration used for a four-node version of our cluster-parallel volume rendering system. Each client renders its local portion of the volume using local graphics hardware. Next, the volumes are composited using the `binaryswap` SPU. The SPUs use out-of-band communication to exchange partial framebuffers until each SPU contains one quarter of the final image. These partial images are then sent to a single server for display.

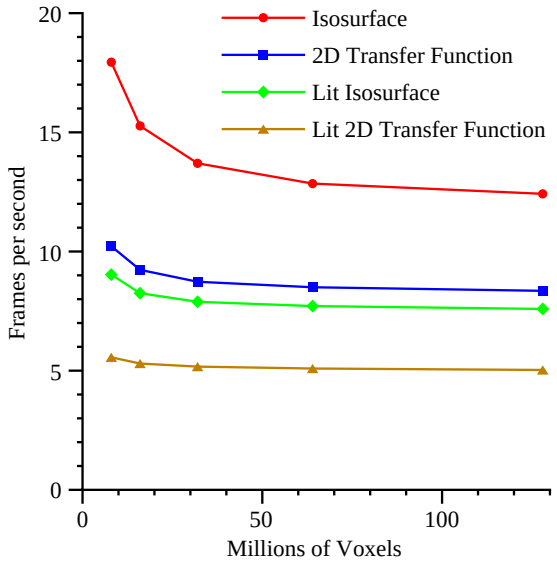



Figure 5: Performance of our volume renderer as larger volumes are used. In this graph, each node renders a 256×256×128 subvolume to a 1024×256 window. The data points correspond to a cluster of 1, 2, 4, 8, and 16 nodes. At 16 nodes, we are rendering two copies of the full 256×256×1024 dataset.

## 4 Results

In this section, we present three different Chromium usage scenarios: a parallel volume renderer used to interactively explore a large volumetric dataset, the reintegration of an application's graphics stream into its original user interface on a high-resolution display device, and a stream transformation to achieve a non-photorealistic drawing style.

### 4.1 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 [22], 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.

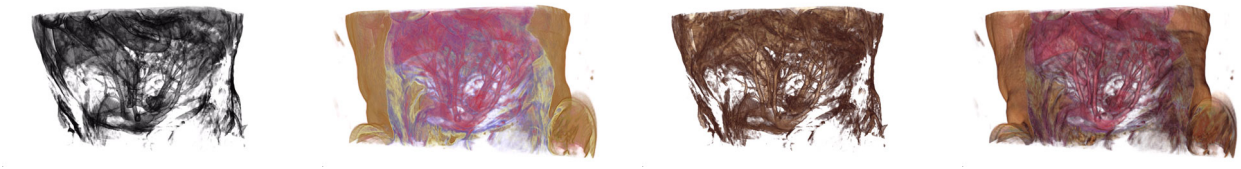
Because we store our volumes as textures, the maximum size of the volume that can be rendered is limited by the amount of available texture memory. In practice, on a single GeForce3 with 64 MB of texture memory, the largest volume that can be rendered with the shader described above is 256×256×128. In addition, the speed of volume rendering with 3D textures is limited by the fill rate of our graphics accelerator. While the theoretical fill rate of the GeForce3 is 800 Mpix/sec, complex fragment processing greatly decreases the attainable performance. Depending on the complexity of 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 [11]. 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. This communication uses Chromium's connection-based networking abstraction, described in section 3.5. The SPUs then composite the image they received with their local framebuffer. 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 [21], the sequence of pairings is chosen carefully so that blending is performed in the correct order with respect to the viewpoint.

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. The dataset is a 256×256×1024 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 1 shows the four shaders we used to vary the achievable per-node performance.

Figure 4 shows the Chromium communication graph for a cluster of four nodes. 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 5 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 256×256×1024 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 256×256×128 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 $\frac{15}{16}$



Isosurface | 2D Transfer Function | Lit Isosurface | Lit 2D Transfer Function

| Shader | 3D textures | 2D (dependent) textures | Register Combiners | Single-Node Fill Rate (Mpix/sec) |
|---|---|---|---|---|
| Isosurface | 1 | 0 | 6 | 190 |
| 2D Transfer Function | 1 | 1 | 4 | 98 |
| Lit Isosurface | 2 | 0 | 8 | 78 |
| Lit 2D Transfer Function | 2 | 1 | 8 | 42 |

Table 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.

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 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.

### 4.2 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 can interfere with the productive use of some applications. To address this problem, we have implemented the `integration` SPU to reincorporate 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 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 [32], or a network-attached parallel framebuffer such as IBM's Scalable Graphics Engine (SGE) [19]. 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 8. 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[2].

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 (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 6 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 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

[2]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.

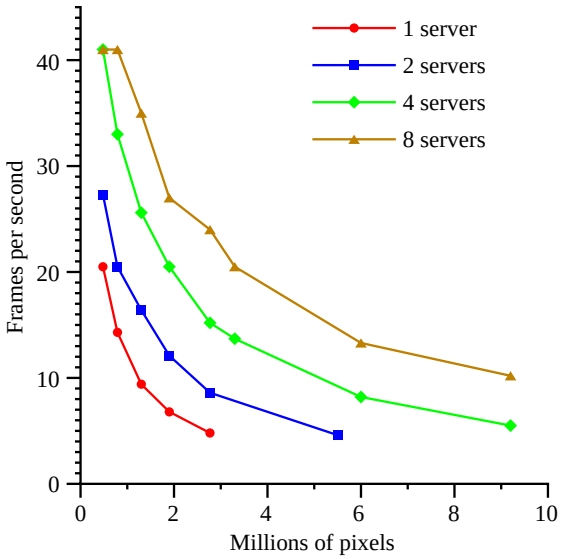



Figure 6: Performance of the GLUT "atlantis" demo using the `integration` SPU to drive the T221 display at different resolutions. Each curve shows the relationship between performance and resolution for a given number of rendering servers. For smaller windows, the SPU becomes limited by the vertical 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.

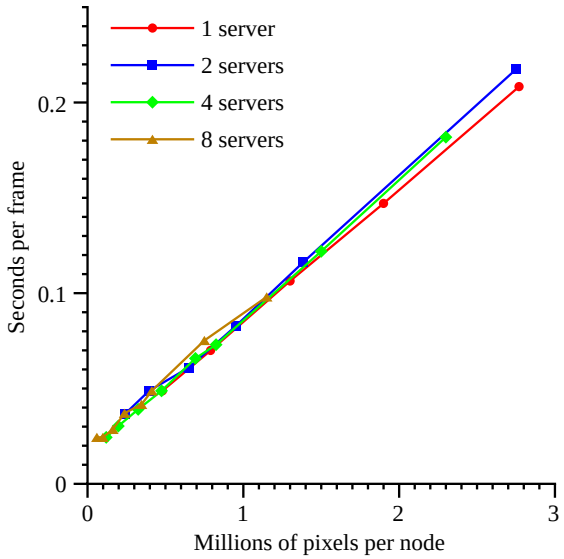



Figure 7: We have replotted the data from figure 6 to show *seconds per frame* versus pixels *per node*, to show per-node throughput. The coincidence of the four curves shows that there is insignificant overhead to doubling the number of rendering nodes, so linear speedup is achieved until the monitor refresh rate becomes the limiting performance factor.

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 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 under-performing 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 by Humphreys et al. when evaluating WireGL on a cluster of SMP nodes [8].

### 4.3 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 [13, 26]. More recently, Raskar has shown that non-trivial NPR styles can be achieved with no model analysis using either standard graphics pipeline tricks [24] or slight extensions to modern programmable graphics hardware [23].

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 [14], 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 [1].

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. 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 9, and the visual results are shown in figure 10. The performance impact of this SPU is shown in figure 11.

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 before being packed. This behavior is frequently overly cautious; most state queries are attempts to deter-

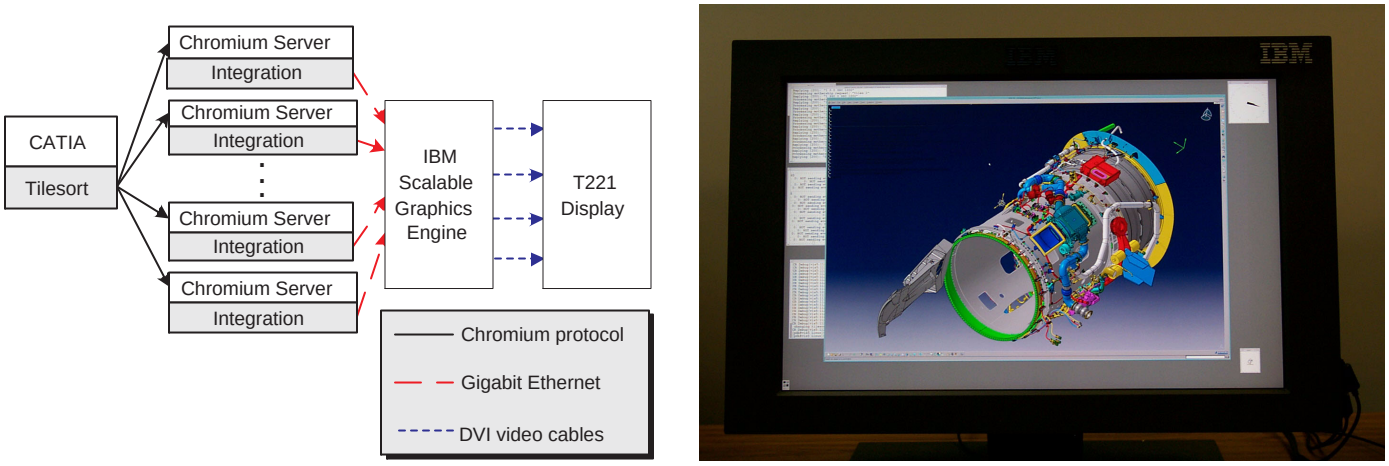



Figure 8: Configuration used to drive IBM's 3840×2400 T221 display using Chromium. The commercial CAD package CATIA is used to create a tiled rendering of a jet engine nacelle (model courtesy of Goodrich Aerostructures). The tiles are then re-integrated into the application's original user interface, 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.

mine 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` 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 10.

It should be noted that the `hiddenline` SPU as presented requires potentially infinite storage, since it buffers the entire frame, and therefore cannot be considered a true stream processor. There are two possible solutions to this problem. One is to perform primitive assembly in the `hiddenline` SPU, drawing each stylized primitive separately. This technique does satisfy our resource constraints (extremely large polygonal primitives can be split into smaller ones), but would 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 state changes.

A better solution to this problem is to use multiple cluster nodes, as shown in figure 12. Rather than buffering the entire frame, we 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 SPUs for each of these rendering styles, we would inject the appropriate OpenGL 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 3.6. 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.

```
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 9: 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.

## 5 Discussion and Future Work

In their seminal paper on virtual graphics, Voorhies, Kirk and Lathrop note that providing a level of abstraction between an applica-

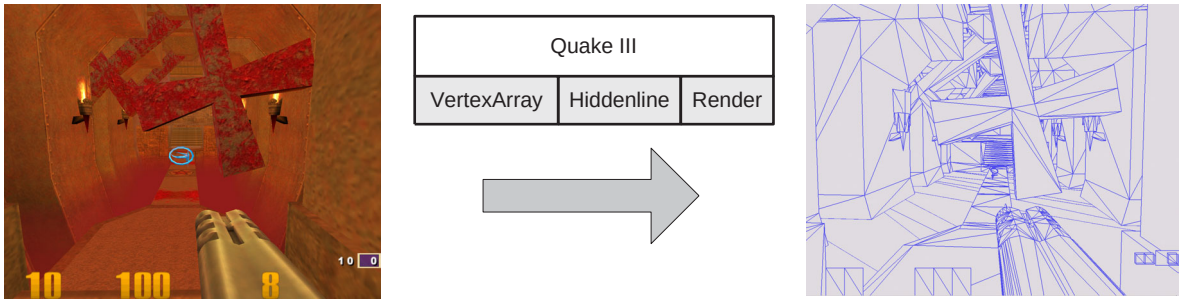



Figure 10: Drawing style enabled by the `hiddenline` SPU. After uses of vertex arrays are filtered out, the SPU records the entire frame, and plays it back twice to achieve a hidden-line effect. No high-level knowledge of the model is required.

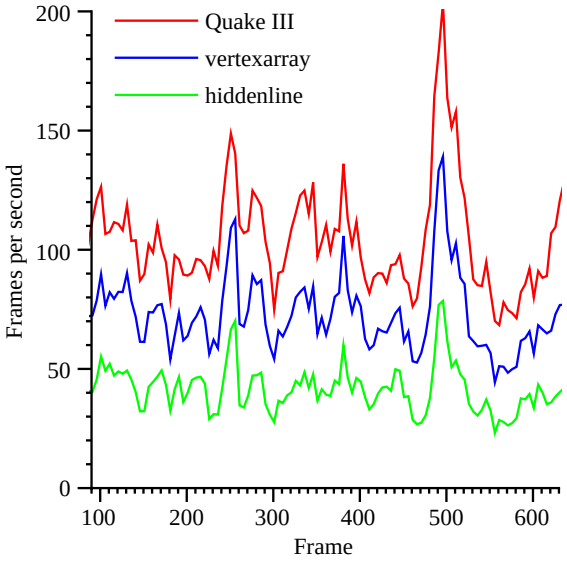



Figure 11: Performance of Quake III running a prerecorded demo. The first 90 frames are devoted to an introductory splash screen and are not shown here. The red curve shows the performance achieved by the application alone. The blue curve shows the same demo using just the `vertexarray` SPU, and the 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.

tion and the graphics hardware "allows for cleaner software design, higher performance, and effective concurrent use of the display" [33]. 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 4.3 adds only approximately 250 lines of code to Chromium's SPU template.

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.

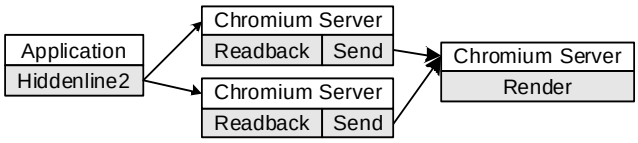



Figure 12: A different usage model for achieving a hidden-line drawing style. In this example, the filled polygon stream and the wireframe stream are sent to two different rendering servers and the resulting images are depth composited. This way, no single SPU needs to buffer the entire frame, and the system requires only finite resources.

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.

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.

## 6 Conclusions

We have described Chromium, a flexible framework for manipulating streams of graphics API commands on clusters of workstations. Chromium's stream processors can be configured to provide a sort-first parallel rendering architecture with a parallel interface, or a sort-last architecture capable of handling most of the same applications. Chromium's flexibility makes it an ideal launching point for new research in parallel rendering systems, particularly those that target clusters of commodity hardware. In addition, it is likely that Chromium's stream-processing model can be applied to other problems in visualization and computer illustration.

## Acknowledgments

The authors would like to thank Brian Paul and Alan Hourihaine for their tireless efforts to make Chromium more robust. Allan Johnson, Gary Cofer, Sally Gewalt, and Laurence Hedlund from the Duke Center for In Vivo Microscopy (an NIH/NCRR National Resource) provided the dataset for our volume renderer. Kekoa Proudfoot and Bill Mark provided assistance with the implementation of a volume renderer on top of the Stanford Real Time Shading Language. Finally, we would like to especially thank all the WireGL and Chromium users for their continued support. This work was funded by DOE contract B504665, and was also performed under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48 (UCRL-JC-146802).

## References

[1] *Advanced Graphics Progamming Techniques Using OpenGL*. SIGGRAPH 1998 Course Notes.

[2] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *SIGMOD Record*, pages 109–120, September 2001.

[3] Ian Buck, Greg Humphreys, and Pat Hanrahan. Tracking graphics state for networked rendering. *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 87–95, August 2000.

[4] Corrina Cortes, Kathleen Fisher, Daryl Pregibon, Anne Rodgers, and Frederick Smith. Hancock: A language for extracting signatures from data streams. *Proceedings of 2000 ACM SIGKDD International Conference on Knowledge and Data Mining*, pages 9–17, August 2000.

[5] Thomas Funkhouser. Coarse-grained parallelism for hierarchical radiosity using group iterative methods. *Proceedings of SIGGRAPH 96*, pages 343–352, August 1996.

[6] Christopher Giertsen and Johnny Peterson. Parallel volume rendering on a network of workstations. *IEEE Computer Graphics and Applications*, pages 16–23, November 1993.

[7] Greg Humphreys, Ian Buck, Matthew Eldridge, and Pat Hanrahan. Distributed rendering for scalable displays. *IEEE Supercomputing 2000*, October 2000.

[8] Greg Humphreys, Matthew Eldridge, Ian Buck, Gordon Stoll, Matthew Everett, and Pat Hanrahan. WireGL: A scalable graphics system for clusters. *Proceedings of SIGGRAPH 2001*, pages 129–140, August 2001.

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

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

[11] Kwan-Liu Ma, James Painter, Charles Hansen, and Michael Krogh. Parallel volume rendering using binary-swap image compositing. *IEEE Computer Graphics and Applications*, pages 59–68, July 1994.

[12] William Mark and Kekoa Proudfoot. The F-buffer: A rasterization order FIFO buffer for multi-pass rendering. *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 57–64, August 2001.

[13] Lee Markosian, Michael Kowalski, Samuel Trychin, Lubomir Bourdev, Daniel Goldstein, and John Hughes. Real-time nonphotorealistic rendering. *Proceedings of SIGGRAPH 1997*, pages 415–420.

[14] Alex Mohr and Michael Gleicher. Non-invasive, interactive, stylized rendering. *ACM Symposium on Interactive 3D Graphics*, pages 175–178, March 2001.

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

[16] Liadan O'Callaghan, Nina Mishra, Adam Meyerson, Sudipto Guha, and Rajeev Motwani. Streaming-data algorithms for high-quality clustering. To appear in *Proceedings of IEEE International Conference on Data Engineering*, March 2002.

[17] John Owens, William Dally, Ujval Kapasi, Scott Rixner, Peter Mattson, and Ben Mowery. Polygon rendering on a stream architecture. *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 23–32, August 2000.

[18] Mark Peercy, Marc Olano, John Airey, and Jeffrey Ungar. Interactive multi-pass programmable shading. *Proceedings of SIGGRAPH 2000*, pages 425–432, August 2000.

[19] Kenneth Perrine and Donald Jones. Parallel graphics and interactivity with the scaleable graphics engine. *IEEE Supercomputing 2001*, November 2001.

[20] Pixar animation studios. *PhotoRealistic RenderMan Toolkit*. 1998.

[21] Thomas Porter and Tom Duff. Compositing digital images. *Proceedings of SIGGRAPH 84*, pages 253–259, July 1984.

[22] Kekoa Proudfoot, William Mark, Svetoslav Tzvetkov, and Pat Hanrahan. A real time procedural shading system for programmable graphics hardware. *Proceedings of SIGGRAPH 2001*, pages 159–170, August 2001.

[23] Ramesh Raskar. Hardware support for non-photorealistic rendering. *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 41–46, August 2001.

[24] Ramesh Raskar and Michael Cohen. Image precision silhouette edges. *ACM Symposium on Interactive 3D Graphics*, pages 135–140, April 1999.

[25] Rodney Recker, David George, and Donald Greenberg. Acceleration techniques for progressive refinement radiosity. *ACM Symposium on Interactive 3D Graphics*, pages 59–66, 1990.

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

[27] Szymon Rusinkiewicz and Marc Levoy. Streaming QSplat: A viewer for networked visualization of large, dense models. *ACM Symposium on Interactive 3D Graphics*, pages 63–68, 2001.

[28] Rudrajit Samanta, Thomas Funkhouser, and Kai Li. Parallel rendering with k-way replication. *IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, October 2001.

[29] Rudrajit Samanta, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh. Sort-first parallel rendering with a cluster of PCs. *SIGGRAPH 2000 Technical Sketch*, August 2000.

[30] Rudrajit Samanta, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh. Hybrid sort-first and sort-last parallel rendering with a cluster of PCs. *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 97–108, August 2000.

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

[32] 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. *Proceedings of SIGGRAPH 2001*, pages 141–148, August 2001.

[33] Douglas Voorhies, David Kirk, and Olin Lathrop. Virtual graphics. *Proceedings of SIGGRAPH 88*, pages 247–253, August 1988.