WireGL: A Scalable Graphics System for Clusters

Greg Humphreys* Matthew Eldridge* Ian Buck* Gordon Stoll† Matthew Everett* Pat Hanrahan*

*Stanford University †Intel Corporation

Abstract

We describe WireGL, a system for scalable interactive rendering on a cluster of workstations. WireGL provides the familiar OpenGL API to each node in a cluster, virtualizing multiple graphics accelerators into a sort-first parallel renderer with a parallel interface. We also describe techniques for reassembling an output image from a set of tiles distributed over a cluster. Using flexible display management, WireGL can drive a variety of output devices, from standalone displays to tiled display walls. By combining the power of virtual graphics, the familiarity and ordered semantics of OpenGL, and the scalability of clusters, we are able to create time-varying visualizations that sustain rendering performance over 70,000,000 triangles per second at interactive refresh rates using 16 compute nodes and 16 rendering nodes.

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

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

1 Introduction

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 [14] and solutions to fluid dynamics problems with several hundred million data points per frame over thousands of frames [8, 21]. 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 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 designed and implemented WireGL, a software system that unifies 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 block diagram of WireGL's major components is shown in figure 1. WireGL provides a virtualized interface to the graphics hardware through the OpenGL API. OpenGL provides immediate-mode semantics, so we support visualizations of time-varying data that would be inconvenient to express with a retained-mode interface or in a scene graph.

In addition, 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. WireGL extends 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 [5]. WireGL allows a flexible assignment of tiles to graphics accelerators, recombining these tiles using DVI-based tile reassembly hardware called Lightning-2 [27]. 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, WireGL 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.

2 Design Issues and Related Work

Designing a parallel graphics system involves a number of tradeoffs and choices. In this section, we present some of the most crucial issues facing parallel graphics system designers.

2.1 Commodity Parts and Work Granularity

Parallel graphics architectures can usually be classified according to the point in the graphics pipeline at which data are redistributed [16]. This redistribution, or "sorting" step is the transition from object parallelism to image parallelism, and the location of this sort has significant implications for the architecture's communication needs. When building a new hardware architecture, the design of the communication infrastructure is flexible, and can be engineered to meet the requirements of the system.

The SGI InfiniteReality is a sort-middle architecture which uses bus-based broadcast communication to distribute primitives [18].

*{humper|eldridge|ianbuck|meverett|hanrahan}@graphics.stanford.edu
†gordon.stoll@intel.com

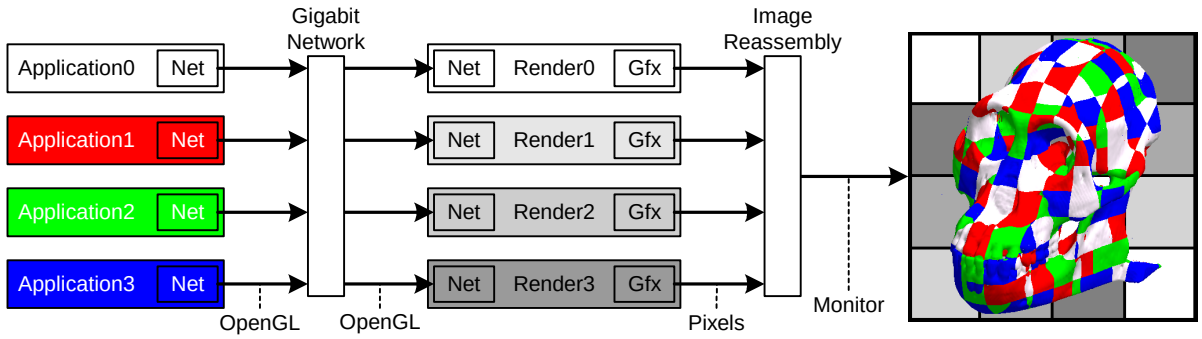

Figure 1: WireGL is comprised of application nodes, rendering nodes, and a display. In this example, each application node is performing isosurface extraction in parallel and rendering its data using the OpenGL API. Each application node is responsible for the correspondingly colored portions of the 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.

To overcome the difficulties encountered in load-balancing image-parallel data, it uses a fine interleaving of tiles, which works well because of the available high-bandwidth broadcast bus. PixelPlanes 5 is a sort-middle architecture with large tiles, which uses a ring network to distribute primitives from a retained-mode scene description [7].

Because such systems do not use commodity building blocks, they must be repeatedly redesigned or rebuilt in order to continue to scale as faster semiconductor technology is developed. WireGL chooses instead to unify multiple unmodified commodity graphics accelerators housed in cluster nodes. This decision has the advantage that we can upgrade the graphics cards or the network at any time without redesigning the system.

However, the choice of cluster network will greatly affect the overall performance and scalability of the resulting system. On PC clusters today, high-speed networks tend to be in the 100-200 megabyte per second range. These networks are an order of magnitude slower than that of a high-end SMP like the SGI Origin 3000, and yet another order of magnitude slower than custom on-chip networks. Although PC cluster networks are not as efficient as more custom solutions, we can still use them to provide scalable graphics performance. As high-speed commodity networks improve in bandwidth and robustness, WireGL will be able to provide better scalability in larger clusters, as well as higher peak performance.

Using commodity parts restricts our choices about communication and work granularity because we cannot modify the individual graphics accelerators. As shown in figure 2, there are only two points in the graphics pipeline where we can introduce communication: immediately after the application stage, and immediately before the final display stage. Communication after the application stage provides a redistribution of primitives to remote graphics accelerators based on those primitives' screen-space extent, which is a traditional sort-first graphics architecture. By introducing communication at the very end of the graphics pipeline, the final image can be recombined from multiple framebuffers. Although WireGL uses this stage to perform tile reassembly, communication at the end of the pipeline can also be used for image composition-based renderers.

For remote use of unmodified graphics components, GLR [13] and SGI's "Vizserver" product [26] transmit a stream of compressed images from the framebuffer of a graphics supercomputer to a low-end desktop. Image compression and streaming technology is an attractive approach to rendering at a distance, although it is not the best approach when the eventual display is local to the rendering hardware.

Although WireGL is a sort-first renderer, sort-last architectures also use a final image recombination step to produce a single image from a fragmented framebuffer. PixelFlow uses image-composition to drive a single display from a parallel host [17]. The Hewlett-Packard visualize *fx* architecture uses a custom network to composite the results of multiple graphics accelerators [4]. Sony's GSCube combines the outputs of multiple Playstation2 graphics systems using a custom network, and supports both sort-first and image composition modes of operation. The GSCube is a particularly interesting architecture because it leverages consumer technology to produce a scalable rendering technology.

To perform image reassembly on clusters, Compaq Research has developed a system called Sepia for performing image composition using ServerNet-II networking technology [9]. Blanke et al. describe the Metabuffer, a system for performing sort-last parallel rendering on a cluster using DVI to scan out color and depth [1]. The Metabuffer is similar to Lightning-2 [27], the DVI-based image reassembly network that we use to drive displays with our cluster. 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.

2.2 Flexible Application Support

Many applications visualize the results of a simulation as those results are calculated. In this case, the simulation usually generates data more slowly than the graphics system can accept it. Such an application is referred to as *compute-limited*. There are many compute-limited visualization applications that scale by generating geometry in parallel and communicating that geometry over a network to a single display server. This geometry communication is almost always done with custom networking code, using a custom wire protocol.

Other applications, however, make intensive use of the graphics hardware, and a single client may effectively occupy many servers. Such an application is called *graphics-limited*. For example, volume rendering with 3D textures requires high fill rates while using few primitives. In this case, a single client may submit commands to multiple servers and keep them all busy because the rendering time of each individual primitive is so large.

Many applications are limited by the rate at which they can issue geometry to the graphics system. Such an application is *interface-*

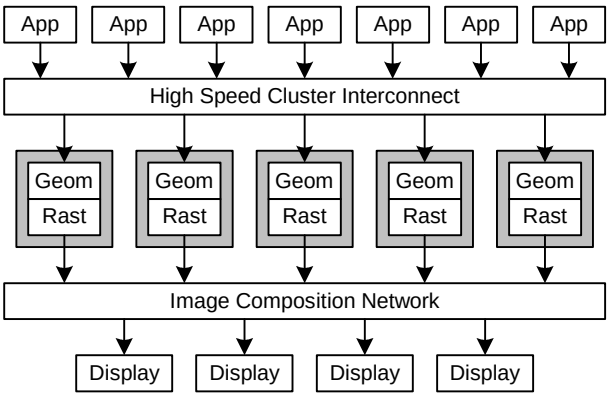

Figure 2: Communication in WireGL. Each graphics pipeline is a standalone graphics accelerator, so we cannot introduce communication between its stages. Notice that the number of application nodes, graphics pipelines, and final displays can all be changed independently according to the application's needs.

limited. For example, visualizations of large geometric data sets that have been computed off-line will tend to be interface limited. Interface limitation is the usual argument for using display lists, compiled vertex arrays, or other retained-mode interfaces. Another way to alleviate the interface bottleneck is to allow multiple processors to issue graphics commands in parallel.

Finally, some applications are not limited by performance, but they cannot effectively visualize their data due to a lack of display resolution. Such an application is called *resolution-limited*. This is typical of many scientific applications where it is important to view all the data at a macroscopic level to get an overview of a dataset, and also to examine microscopic details to fully understand the data. Such an application requires the combined resolution of multiple graphics accelerators and a high-resolution tiled display. One example of this type of display is IBM's Bertha, a 3840×2560 LCD display driven by four DVI inputs.

WireGL does not place any restrictions on the number of clients or servers. For compute-limited applications it is desirable to have more clients than servers, for graphics-limited applications it is better to have more servers than clients, and for interface-limited applications it is most effective to have an approximately equal number of clients and servers. WireGL also works well in a heterogeneous environment where the servers and the clients may be running different operating systems on different hardware.

2.3 Programming Interface

Graphics API's 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 [22]. 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 [23, 24].

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. Any scene graph library that uses OpenGL for rendering can run on top of WireGL. In addition, if the scene graph has bounding-box information about primitive groups, that information can be provided to WireGL through the OpenGL hinting mechanism to speed up geometry sorting.

WireGL 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 recompiling it. If that application is graphics-limited, WireGL can provide an immediate speedup. 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. Portions of WireGL were first described by Humphreys et al. [10]. In that paper, we described our techniques for sorting OpenGL streams to tile servers in order to transparently support large displays. 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 [25].

Many applications, however, are not graphics-limited and must be parallelized to achieve speedup. Using WireGL, 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 ordering requirements, achieving parallelism is straightforward.

For applications with more complex ordering requirements, WireGL implements extensions to OpenGL that were first proposed by Igehy, Stoll and Hanrahan [12]. Their simulations showed that scalable applications could easily be written using their extensions, results that were further verified by the Pomegranate simulations [6]. These extensions add traditional synchronization primitives (barriers and semaphores) to the graphics library. WireGL is the first implementation of this API in a hardware-accelerated (that is, not simulated) architecture.

Although OpenGL is an immediate-mode API, some OpenGL features like display lists and texture objects allow data to be stored by the graphics system and reused. WireGL supports this by storing those data on the server, so that users who want to replicate data across the nodes of the cluster can do so. In addition, texture objects can optionally be shared between multiple clients, which means that they can be specified once at the start of the application and do not need to be duplicated per-client. It would also be easy to allow similar sharing of display lists between clients, although we have not implemented this feature.

3 WireGL

A WireGL based rendering system consists of one or more clients submitting OpenGL commands simultaneously to one or more graphics servers, called *pipeservers*. The pipeservers are organized as a sort-first parallel graphics pipeline [19], and together they render a single output image. Each pipeserver 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 4. A high-level view of the system is shown in figure 1. In that figure, each rendering node is a pipeserver. WireGL virtualizes this architecture, providing a single conceptual graphics pipeline to the clients.

3.1 Client Implementation

This section provides an overview of WireGL's sort-first client implementation. Interested readers should refer to Humphreys et al. [10] for a more complete description of our sort-first system, the protocol efficiency, and display size scalability results. The state tracking system is described in detail in Buck, Humphreys, and Hanrahan [3].

The WireGL client library is implemented as a replacement for the system's OpenGL library on Windows, Linux, or IRIX. As the application makes calls to the OpenGL API, WireGL classifies each call into one of three categories: geometry, state, or special. Special commands, such as `SwapBuffers`, `glFinish`, and `glClear`, require individual treatment, and will not be described here.

Geometry commands are those that legally appear between a `glBegin`/`glEnd` pair, as well as commands that can generate fragments on their own, such as `glDrawPixels`. These commands are packed immediately into a global "geometry buffer". This buffer contains a copy of the arguments to the function, as well as an opcode. Each opcode is encoded in a single byte, and opcodes and data are packed into separate portions of the buffer which grow in opposite directions. This representation allows the buffer to retain each argument's memory alignment, minimizes the space overhead of the opcodes, and keeps opcodes and data contiguous in memory so that they can be sent with a single call to the networking library. Some commands that can appear legally between a `glBegin`/`glEnd` pair do not generate fragments, such as `glNormal3f`. These commands are still packed immediately into the buffer, but their state effects are also recorded. Our geometry packing code has been carefully engineered, and achieves a maximum packing performance of over 20 million vertices per second (the exact computer configuration used to perform these experiments is described in section 5).

As each vertex is specified, WireGL maintains an object-space bounding box. Each incremental update to the bounding box requires only six conditional moves, which can be implemented efficiently using a SIMD instruction set such as the Pentium III's. When geometry is sent to the servers, this bounding box is transformed into screen space, and the set of overlapped screen tiles is computed. This set is used to compute the servers that need to receive the geometry buffer. Because geometry sorting is done on groups of primitives, the overhead of bounding box transformation and extent intersection is amortized over many vertices.

State commands are those that directly affect the graphics state, such as `glRotatef`, `glBlendFunc`, or `glTexImage2D`. The effects of state commands are recorded into a graphics context data structure. Each element of state has *n* bits associated with it indicating whether that state element is out of sync with each of *n* servers. When a state command is executed, the bits are all set to 1, indicating that each server might need a new copy of that element. The OpenGL state is represented as a hierarchy, roughly mirroring the layout described in the OpenGL specification [20]. For example, `GL_LIGHT0`'s diffuse color is a member of `GL_LIGHT0`'s state, which is an element of the lighting state. Each non-leaf node in the hierarchy also has a vector of *n* synchronization bits which reflect the logical OR of all its children. We have shown that this representation allows for very efficient computation of the difference between two contexts [3].

Either of two circumstances can trigger the transmission of the geometry buffer. First, if the buffer fills up, it must be flushed to make room for subsequent commands. Second, if a state command is called while the geometry buffer is not empty, the geometry buffer must be flushed before the state command is recorded, since OpenGL has strict ordering semantics. However, we cannot send the geometry buffer to the overlapped servers immediately, because they might not have the correct OpenGL state. We must prepend a packed representation of the application's state before transmitting any geometry. To do this, the client library keeps a copy of each server's graphics state. Using our efficient context differencing operation, the commands needed to bring the server up to date with the application are placed in that server's outgoing network buffer. The global geometry buffer can then be copied after the state differences. By updating state lazily and bucketing geometry, we keep network traffic to a minimum.

This behavior has an important implication for the granularity of work in WireGL. Sorting individual primitives in software would be too expensive, but grouping too many primitives may result in excessive overlap and inefficient network usage. Assuming that a state call is made before a network buffer fills, WireGL's work granularity is that of groups of primitive blocks, or multiple `glBegin`/`glEnd` pairs. The optimal granularity of work will be a balance between screen-space coherency and the expense of bounding-box transformation.

It would be impractical to transform each primitive separately, but it is not always beneficial to coalesce the maximum number of primitive blocks, as this may result in partial network broadcasts if the geometry is not spatially coherent and requires a large screen-space bounding box. WireGL currently has no automatic mechanism for determining the best time to bucket geometry. Applications that are aware of their bucketing needs can optionally force a sort after a specified number of primitive blocks.

When running a parallel application, each client node behaves in the manner described above, performing a sort-first distribution of geometry and state to all pipeservers. This means that each pipeserver must be prepared to handle multiple asynchronous incoming streams of work, each with its own associated graphics context. 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, WireGL adds barriers and semaphores to the OpenGL API, as proposed by Igehy et al. [12].

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, as the same ordering restrictions must be observed by all servers, and we wish to avoid a central oracle making global scheduling decisions.

When running a parallel application, WireGL does not change the semantics of any commands, even those with global effects. For example, `SwapBuffers` marks the end of the frame and causes a buffer swap to be executed by all servers. Therefore, it is important that only one client execute `SwapBuffers` per frame. Also, a parallel application with no intra-frame ordering dependencies will still need two barriers per frame. To ensure that the framebuffer clear happens before any drawing, a barrier must follow the call to `glClear`. Similarly, all nodes must have completely submitted their data 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 3. More complex usage examples can be found in Igehy's original paper [12].

3.2 Pipeserver Implementation

A pipeserver maintains a queue of pending commands for each connected client. When new 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. A pipeserver continues

```
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 3: A minimal parallel display routine. 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.

executing 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 semaphore or barrier they are waiting on. The pipeserver's queue structures are shown in figure 4.

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 [3], which is slow enough that it would limit the amount of intra-frame parallelism achievable in WireGL.

To overcome this limitation, each pipeserver uses the same state tracking library as the client to maintain the state of each client in software. Just as an extremely efficient context differencing operation is the key to lazy state update between the client and the server, it is also effective for performing context switching on the server. Since nodes in a parallel application are collaborating to produce a single image, they will typically have similar graphics states, and performing context switching with our hierarchical representation has a cost proportional to the contexts' disparity. We have measured this hierarchical approach as being able to switch contexts almost 200,000 times per second for contexts that differ in current color and transformation matrix, and over 5 million times per second for identical contexts [3].

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. 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, the pipeserver's scheduling algorithm is not a significant factor in an application's performance.

Since each pipeserver may manage more than one tile, it may be necessary to render a block of geometry more than once. The arrangement of tiles in the local framebuffer is described in section 4.1. The client library inserts the 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 points that are larger than one pixel, 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.

3.3 Network

We use a connection-based network abstraction to support multiple network types such as TCP/IP and Myrinet. Our abstraction provides a credit-based flow control mechanism to prevent servers from exhausting their memory resources when they cannot keep up with the clients. Flow control is particularly important when 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.

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 such as the results of `glReadPixels` to the clients. More importantly, WireGL supports the `glFinish` call so that applications can determine when the commands they have issued have been fully executed. This is available 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.

4 Display Management

To form a seamless output image, tiles must be extracted from the framebuffers of the pipeservers 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

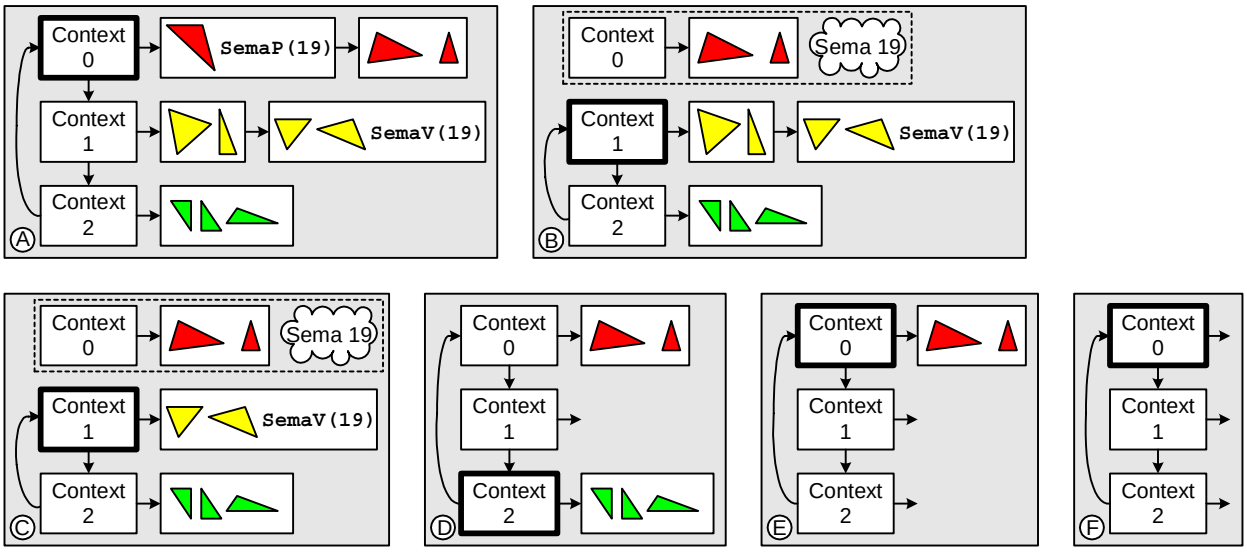

Figure 4: Inside a pipeserver. Runnable contexts will be serviced in a round-robin fashion. Graphics commands being issued by a context's application can be appended to the end of a work queue at any time, until the client consumes its allotted server-side buffer space. Blocks A-F show sequential timesteps as the pipeserver decodes command blocks; the currently executing context is shown with a heavy outline. In timestep A, the pipeserver 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.

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 most straightforward way to reassemble the image after scan-out is to allow each pipeserver to drive a single locally-attached display. These displays can then be abutted to form a large logical output space. This arrangement constrains each pipeserver 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 construct.

4.1 Display Reassembly in Hardware

For our experiments with hardware display assembly, we use the Lightning-2 system [27]. 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 by the application 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 waits for this notification before executing a 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. 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 Bertha or a multi-projector display wall without tearing artifacts. This in turn enables stereo rendering on tiled displays.

Each pipeserver 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. 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 can 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 be able to adjust the screen tiling on the fly to meet the application's needs automatically.

4.2 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, WireGL has a mode called the "visualization server". In this mode, all pipeservers read the color contents of their managed tiles at the end of each frame. Those images are then sent over the cluster's interconnect to a separate

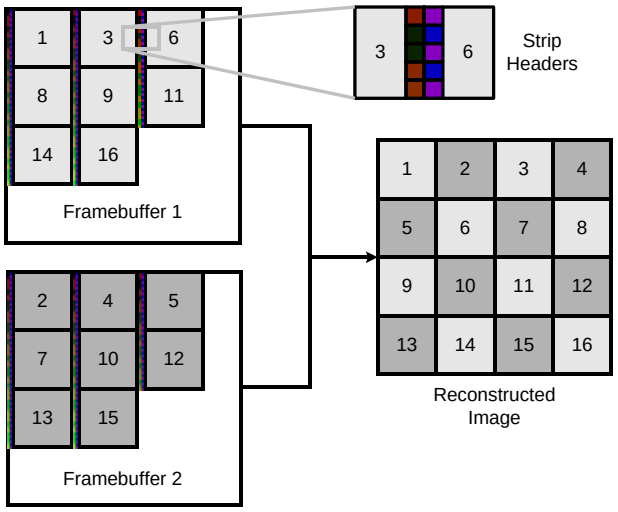

Figure 5: Allocating multiple tiles to a single accelerator with Lightning-2. In the zoomed-in region, the two-pixel wide strip headers are clearly visible.

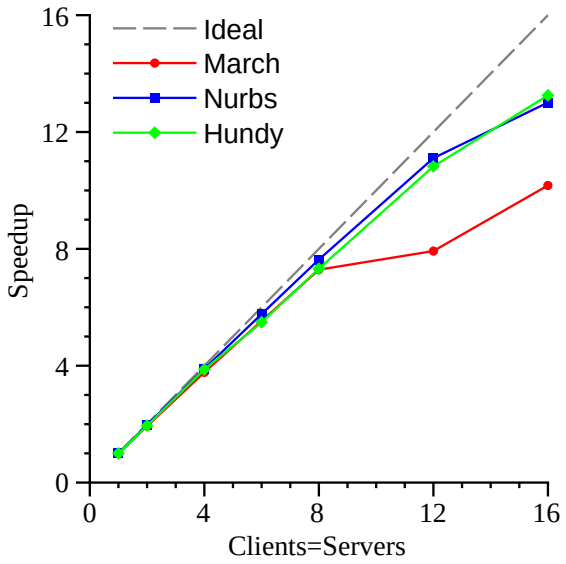

Figure 6: Speedup for March, Nurbs, and Hundy using up to 16 pipeservers. With 16 clients and 16 servers, Hundy achieves 83% efficiency, Nurbs achieves 81% efficiency, and March achieves 64% efficiency.

compositing server for reassembly, with the same protocol used by the clients to send geometry to the pipeservers. In effect, each pipeserver becomes a client in a parallel image-drawing application. The compositing server is simply another WireGL pipeserver accepting `glDrawPixels` commands and parallel API synchronization directives.

The primary drawback of this pure software 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 section 5.3.

5 Performance and Scalability

The cluster used for all our experiments, called "Chromium", 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 [2] 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.

When rendering locally, each node can draw 21.4 million unlit points per second using an immediate mode interface (i.e., without display lists or vertex arrays). WireGL's maximum packing rate (the speed at which WireGL can construct network buffers) is 21.8 million vertices per second. When using WireGL to render remotely from one client to one server, we achieve a maximum rate of 7.5 million points per second. Since each point occupies 13 bytes (three floats plus an opcode byte), this represents a network bandwidth of 93 MB/sec, which is close to the 101 MB/sec we have measured when repeatedly resending the same packet after creation.

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 5.4.

5.1 Applications

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

- March is a parallel implementation of the marching cubes volume rendering algorithm [15]. 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.

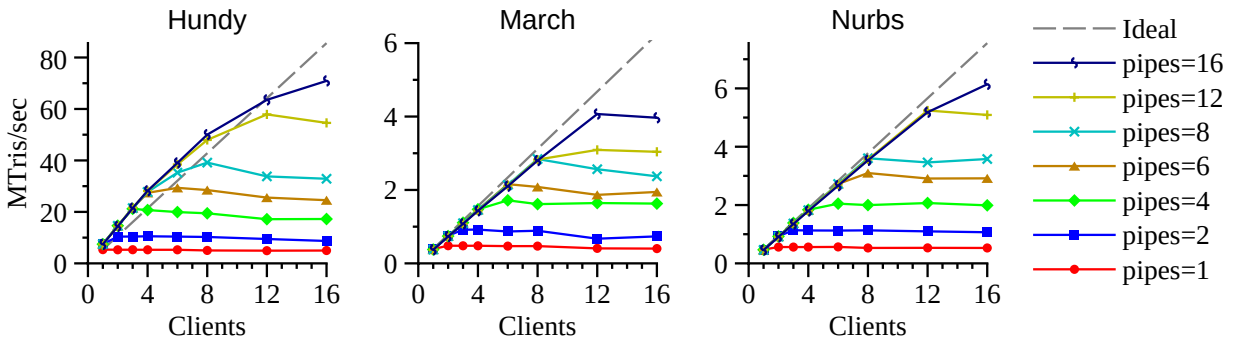

Figure 7: Scaling interface-limited applications. For each application, the number of clients and servers is varied. Hundy uses a tile size of 100×100, and achieves a peak rendering performance of 71 million tris/sec at a rate of 17.7 fps. Nurbs uses a tile size of 100×100, and achieves a peak rendering 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.

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 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 pipeservers is shown in figure 6.

5.2 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 7. In these graphs, the tile size is chosen empirically, and Lightning-2 reconstructs a final 1600×1200 output image.[1] Each curve represents a different number of pipeservers, 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 superlinear speedup; this is because Hundy generates a large amount of network traffic per second. This traffic is spread 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 pipeserver. The results of this experiment are shown in figure 8. For each test, the application scales excellently until it reaches the interface limit of the single pipeserver or the size of the cluster.

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

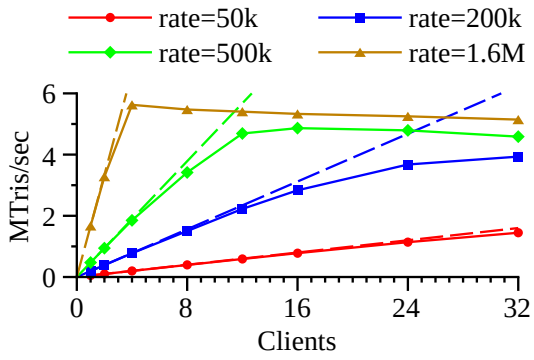

Figure 8: Scaling a compute-limited application with a single pipeserver. For each curve, Hundy's issue rate has been restricted. We achieve excellent scalability up to either the pipeserver interface limit, or the full 32 nodes of our cluster.

The results shown in figures 7 and 8 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.

5.3 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 sec-

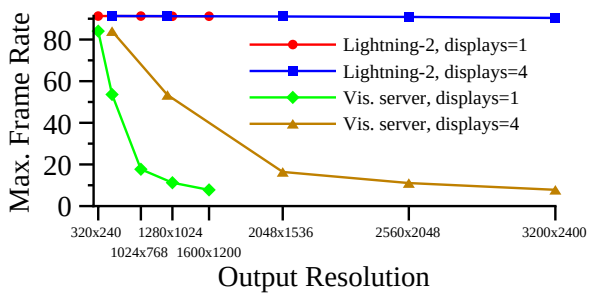

Figure 9: Maximum framerate achievable using Lightning-2 or the visualization server. As the image size increases, the expense of reading and writing blocks of pixels to the framebuffer quickly limits the visualization server to non-interactive framerates.

tion 3.1, only one node in a parallel application calls `SwapBuffers` for each frame. In each experiment, 12 pipeservers are used. The results are shown in figure 9. The "displays=4" curves are representative of a tiled display wall or a multi-input display such as IBM's Bertha.

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.

5.4 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 pipeserver is a lower bound on the total amount of network traffic for any number of pipeservers, 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 10, 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 6. However, as cluster size increases, the increasing load imbalance will eventually limit Nurbs' scalability. Nonetheless, WireGL provides excellent scalability up to 16 pipeservers, which makes it a 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 pipeservers 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.

To fully understand our scalability results, we have measured the achievable send and 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 11. 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 Discussion and Future Work

The real power of WireGL derives from its flexibility. Because WireGL is based on commodity parts, it is easy and inexpensive to build a parallel rendering system with a cluster. 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 like WireGL on small to medium-sized clusters.

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 WireGL's performance on a 16 to 32 node cluster will improve dramatically with the introduction of new server-area networking technology such as

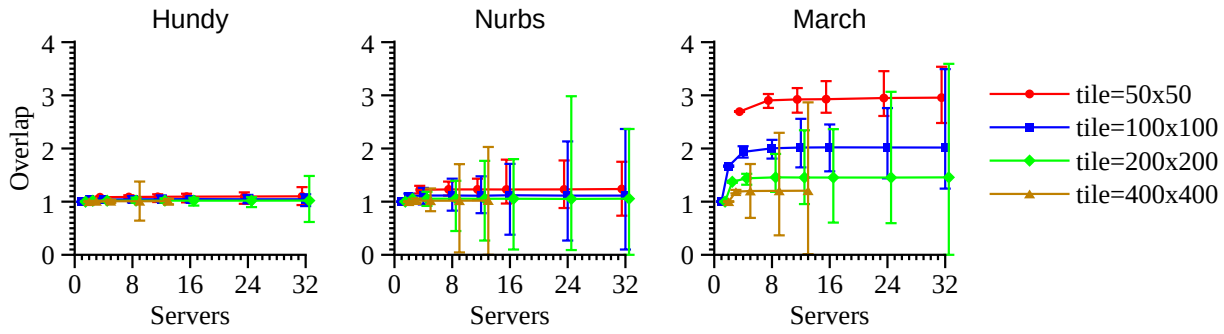

Figure 10: Overlap factor and load imbalance with various tile sizes on a 1600×1200 display. The height of each curve indicates the overlap factor, while the size of the error bars is proportional to the load imbalance. Increasing the tile size decreases the total amount of network traffic, but at the 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.

InfiniBand. To achieve peak performance today, it is necessary to perform image reassembly after scan-out. Our Lightning-2 implementation is a large custom piece of hardware, but a smaller version could be built very cheaply and would enable the construction of a small, self-contained cluster that could act as a standalone graphics subsystem for a larger cluster.

6.1 Scalability Limits

We have demonstrated that WireGL's sort-first approach to parallel rendering on clusters provides excellent scalability for a variety of applications with a configuration of up to 16-pipeservers 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.

6.2 Texture Management

WireGL'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.

WireGL's naive approach to parallel texture management can be a limitation for some applications. More work needs to be done in this area, and we are beginning to investigate new texture management strategies. One approach being considered will leverage our recent work in parallel texture caching [11].

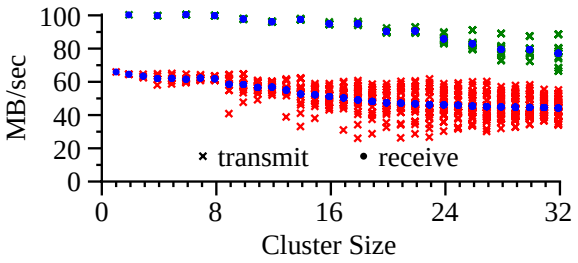

Figure 11: Transmit and receive bandwidth for Myrinet with all-to-all communication. For each cluster size, the observed send and receive bandwidth is plotted for all nodes. The top dataset represents a partitioned *n*-to-*n* run, where sources and sinks are not run on the same nodes. The bottom dataset 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.

6.3 Latency

There are two main sources of latency in WireGL: 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. WireGL'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 pipeserver 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.

6.4 Consistency Model

In their paper on the parallel API, Igehy, Stoll and Hanrahan defined the concept of sequential consistency for parallel graphics systems. The graphics system presented in their paper provides command-sequential consistency, which means that each OpenGL command is considered to be an atomic operation. WireGL provides a weaker form of consistency called fragment-sequential consistency. In this consistency model, only operations on the framebuffer are considered to be atomic. When considered in isolation, each tile in WireGL is command-sequentially consistent, but the final image is not. If two clients each draw a triangle without any explicit ordering or depth buffering, WireGL may show one or the other on top on a per-tile basis. Igehy notes that any graphics system that supports the parallel API should provide at least fragment-sequential consistency. Parallel applications that must always produce exactly the same final image can achieve this in one of two ways: they can use depth buffering, or they may express their ordering requirements through the use of the parallel API. WireGL provides both of these capabilities, and we have not found any application that both produces deterministic images and also relies on the stronger command-sequential consistency model.

6.5 Future Work

The main future direction of the WireGL project is to add additional flexibility. The current system is suitable for many applications, but some parallel rendering tasks require a more flexible configuration. When considering the visualization server configuration of WireGL, it is clear that each node in the cluster is acting as an OpenGL stream processor. The application is a stream source, generating multiple streams to a number of rendering tiles. The intermediate pipeservers accept an incoming stream of geometry and generate a new outgoing stream of imagery. The final compositing pipeserver accepts multiple imagery streams and generates a final image for display.

Because the system is closed (that is, each stream is in exactly the same format), it is easy to imagine that other useful stream processing configurations could be constructed. The next version of our cluster rendering software will allow the user to describe an arbitrary directed graph of graphics stream processing units. Stream processors will be written using a standardized interface so that new stream processors can easily be created and plugged into the cluster rendering framework. This will provide researchers with a framework for testing their own cluster rendering algorithms, be they sort-last, sort-first, retained-mode, or extremely specialized. We will be developing parallel applications for volume rendering, interactive exploration of unstructured grid data, terrain flythroughs, as well as parallelized versions of commonly used visualization packages such as VTK, all targeted at this new common cluster rendering framework.

Another promising application of this new technology is transparent support for CAVEs or arrays of casually aligned projectors. A version of WireGL has already been adapted to allow unmodified applications to run in a CAVE, and we have seen a demonstration of WireGL used for a tiled display consisting of off-axis projectors. In addition, we have nearly completed a Microsoft DirectShow backend for the Visualization Server to leverage the latest technology in streaming video codecs for rendering at a distance. Our latest results allow us to deliver the rendering power of our cluster at 640×480 across a 100 megabit network, and recent codec advances will allow us to use even slower networks for scalable remote visualization. A version of the system described in this paper has already been developed to perform sort-last parallel rendering, using Lightning-2 to perform depth compositing on the pixel chain.

7 Conclusions

We have described WireGL, a scalable graphics system for clusters of workstations. By integrating parallel submission into our sort-first parallel renderer, we are able to achieve scalable rendering performance for a variety of application types. WireGL allows users to build a graphics system capable of handling demanding real-time, immediate mode tasks at a fraction of the cost of a traditional graphics supercomputer. Alternately, it is possible to realize much higher performance on a cluster of workstations for the same price.

WireGL is a more flexible graphics system than an internally parallel standalone graphics accelerator. By leveraging commodity parts, the building blocks of WireGL can be easily upgraded as technology improves. WireGL enjoys the economies of scale of off-the-shelf parts, providing excellent price performance. In addition, algorithm or system designers can use WireGL as a base for experimentation with parallel rendering. WireGL's flexibility and scalable performance make it an attractive system for real-time rendering on clusters.

Acknowledgments

The authors would like to thank the entire Lightning-2 team for their efforts, without which much of this work would be impossible. John Gerth and Chris Niederauer were instrumental in procuring, constructing, and setting up our cluster. Randy Frank, Dino Pavlakos and Brian Wylie provided valuable guidance in designing the cluster and selecting equipment. Nick Triantos and Andrew Ritger provided timely updates to NVIDIA Linux drivers, often providing top-of-tree driver builds. Bob Felderman and Glenn Brown from Myricom were very helpful in tracking down our Myrinet bugs (performance and otherwise). Maureen Stone and François Guimbretière have been very patient users of WireGL as it was maturing. This work was sponsored by the DOE VIEWS program (contract B504665) and the DARPA DIS program (contract DABT63-95-C-0085-P00006).

References

[1] W. Blanke, C. Bajaj, D. Fussel, and X. Zhang. The Metabuffer: A Scalable Multiresolution Multidisplay 3-D Graphics System Using Commodity Rendering Engines. TR2000-16, University of Texas at Austin, February 2000.

[2] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro*, pages 29–36, February 1995.

[3] I. Buck, G. Humphreys, and P. Hanrahan. Tracking Graphics State for Networked Rendering. *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, August 2000.

[4] R. Cunniff. visualize *fx* Graphics Scalable Architecture. *Proceedings of Eurographics/SIGGRAPH Hot3D*, pages 29–38, August 2000.

[5] Digital Visual Interface Specification. http://www.ddwg.org.

[6] M. Eldridge, H. Igehy, and P. Hanrahan. Pomegranate: A Fully Scalable Graphics Architecture. *Proceedings of SIGGRAPH 2000*, pages 443–454, July 2000.

[7] H. Fuchs, J. Poulton, J. Eyles, T. Greer, H. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, and L. Israel.

Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories. *Proceedings of SIGGRAPH 89*, pages 79–88, July 1989.

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

[9] A. Heirich and L. Moll. Scalable Distributed Visualization Using Off-the-Shelf Components. *IEEE Parallel Visualization and Graphics Symposium*, pages 55–59, October 1999.

[10] G. Humphreys, I. Buck, M. Eldridge, and P. Hanrahan. Distributed Rendering for Scalable Displays. *IEEE Supercomputing 2000*, October 2000.

[11] H. Igehy, M. Eldridge, and P. Hanrahan. Parallel Texture Caching. *Proceedings of SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 95–106, August 1999.

[12] H. Igehy, G. Stoll, and P. Hanrahan. The Design of a Parallel Graphics Interface. *Proceedings of SIGGRAPH 98*, pages 141–150, July 1998.

[13] M. Kilgard. GLR, an OpenGL Render Server Facility. *Proceedings of X Technical Conference*, February 1996.

[14] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, and D. Fulk. The Digital Michelangelo Project: 3D Scanning of Large Statues. *Proceedings of SIGGRAPH 2000*, pages 131–144, July 2000.

[15] W. E. Lorensen and H. E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *Proceedings of SIGGRAPH 87*, pages 163–169, July 1987.

[16] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Algorithms*, pages 23–32, July 1994.

[17] S. Molnar, J. Eyles, and J. Poulton. PixelFlow: High-Speed Rendering Using Image Composition. *Proceedings of SIGGRAPH 92*, pages 231–240, August 1992.

[18] J. Montrym, D. Baum, D. Dignam, and C. Migdal. InfiniteReality: A Real-Time Graphics System. *Proceedings of SIGGRAPH 97*, pages 293–302, August 1997.

[19] C. Mueller. The Sort-First Rendering Architecture for High-Performance Graphics. *1995 Symposium on Interactive 3D Graphics*, pages 75–84, April 1995.

[20] OpenGL Specifications. http://www.opengl.org/Documentation/Specs.html.

[21] W. C. Reynolds and M. Fatica. Stanford Center for Integrated Turbulence Simulations. *IEEE Computing in Science and Engineering*, pages 54–63, April 2000.

[22] J. Rohlf and J. Helman. IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics. In *Proceedings of SIGGRAPH 94*, pages 381–395, July 1994.

[23] R. Samanta, T. Funkhouser, K. Li, and J. P. Singh. Sort-First Parallel Rendering with a Cluster of PCs. *SIGGRAPH 2000 Technical Sketch*, August 2000.

[24] R. Samanta, J. Zheng, T. Funkhouser, K. Li, and J. P. Singh. Load Balancing for Multi-Projector Rendering Systems. *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 107–116, August 1999.

[25] SGI multipipe. http://www.sgi.com/software/multipipe/.

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

[27] G. Stoll, M. Eldridge, D. Patterson, A. Webb, S. Berman, R. Levy, C. Caywood, M. Taveira, S. Hunt, and P. Hanrahan. Lightning-2: A High-Performance Display Subsystem for PC Clusters. *Proceedings of SIGGRAPH 2001*, August 2001.