Tracking Graphics State For Networked Rendering

Ian Buck   Greg Humphreys   Pat Hanrahan[*]

Stanford University

Abstract

As networks get faster, it becomes more feasible to render large data sets remotely. For example, it is useful to run large scientific simulations on remote compute servers but visualize the results of those simulations on one or more local displays. The WireGL project at Stanford is researching new techniques for rendering over a network. For many applications, we can render remotely over a gigabit network to a tiled display with little or no performance loss over running locally. One of the elements of WireGL that makes this performance possible is our ability to track the graphics state of a running application.

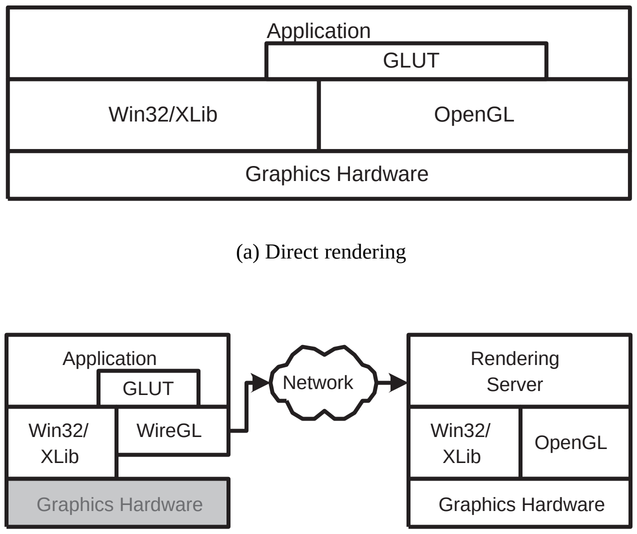
In this paper, we will describe our techniques for tracking state, as well as efficient algorithms for computing the difference between two graphics contexts. This fast differencing operation allows WireGL to transmit less state data over the network by updating server state lazily. It also allows our system to context switch between multiple graphics applications several million times per second without flushing the hardware accelerator. This results in substantial performance gains when sharing a remote display between multiple clients.

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: Remote Rendering, Networked Rendering, Graphics State

1 Introduction

It is often useful to remotely display the results of certain computation. Regardless of the speed of the network which connects the display and computation servers, some representation of graphics commands or images must be sent across the network in order to display the result at the remote node. The WireGL project at Stanford is researching the systems aspects of high performance remote rendering. A diagram of our remote rendering model is shown in figure 1. Our goal is to drive a remote display over a high speed network at least as fast as driving that display locally. In addition, we would like our techniques to easily scale, allowing us to render remotely to a cluster of workstations driving a tiled display[12]. One of the techniques WireGL uses to achieve the goal of high performance remote rendering is client side state tracking. Note that remote rendering often has a counter-intuitive definition of client/server computing. For this paper, a rendering *client* is the application making calls to the OpenGL library, and a rendering *server* is the process receiving and executing those commands.



(a) Direct rendering

(b) Networked rendering

Figure 1: Remote rendering using WireGL. Our OpenGL implementation is a replacement for the system's OpenGL library, allowing us to run an unmodified OpenGL application over a network.

Graphics APIs like OpenGL contain a large amount of state. The state contains attributes such as the current transformation matrix, the current color, and so on. On a workstation with hardware acceleration, the graphics hardware keeps track of most or all of the current state. However, in order to properly implement a remote protocol for these APIs, it is necessary for the client to keep track of some of the state in software. For instance, OpenGL allows the programmer to specify certain alignment and offset properties for pixel arrays that are to be used as textures. The client library must unpack these textures in order to send the correct data to the server. Therefore, the client library must track these offsets and alignments as the application is running.

The WireGL project extends this notion to keep track of the entire graphics state of a running application. The ability to maintain the graphics state greatly improves the performance and flexibility of our networked rendering system. We will present our OpenGL state tracking algorithms in detail, including an analysis of the OpenGL state commands and discussion of the acceleration

[*]{ianbuck|humper|hanrahan}@graphics.stanford.edu

structures used to facilitate state comparison operations. We will also describe some of the challenges of tracking state including display lists, vertex arrays, and texture management and how our system resolves these difficulties in the context of network rendering.

In addition, we will describe how state tracking can be used for network rendering to improve performance. *Lazy Update* is a method to postpone the transmission of state commands in order to minimize the number of commands that need to be transmitted. *Soft Context Switching* provides a context switch operation which is performed completely in software to allow many network streams to render efficiently to a single hardware context.

Finally, we will present several applications of this technology, including support for rendering to tiled displays, conserving network bandwidth, compressing geometry, and efficiently sharing a display between multiple independent applications.

2 Related Work

In the area of remote rendering, GLX and X windows stand out as the two most widely used solutions. GLX[1, 14] is a wire protocol for OpenGL that allows an application to transmit a simple packed representation of the OpenGL command parameters to a server to be executed on behalf of the client. GLX performs the minimum amount of state tracking required for correctness; it tracks pixel formats for packing and unpacking pixel data, as well as vertex array state. All other state commands are sent directly to the server. The design of GLX was motivated by a desire to interface with the X protocol, not by a need for high-speed remote rendering.

X windows provides remote 2D graphics capabilities. Previous versions of X were stateless (every drawing command carried with it all necessary state information), which was very inefficient for remote drawing. The latest version (R11) keeps all state information on the server. A single protocol request is used to change an arbitrary subset of the state at once[16]. X windows has the same model of network packet generation as GLX; each state command (e.g., `XSetForeground`) causes a state changing packet to be generated. Note that if multiple state changing commands are made in a row, the X client libraries will collapse these commands into a single protocol packet. WireGL takes this command collapsing technique one step further by waiting to generate any state commands until absolutely necessary, reducing unnecessary network traffic by collapsing multiple state elements and discarding unnecessary ones.

In the area of parallel geometry processing, Torborg[17] presents a solution whereby state commands are immediately broadcast to each geometry processor. This is practical because each processor is connected to a shared bus, therefore broadcasting the state is cheap. Our interconnect is not assumed to be a shared medium, so we avoided broadcasting commands to multiple servers in order to conserve network bandwidth. The RealityEngine[4] broadcasts infrequent commands (e.g., light model) to the geometry engines, but maintains a copy of frequent commands (e.g., color) near the host interface. When primitives are generated, copies of the state settings for these frequent commands are attached to the geometry data. This way, state elements that are assumed to be changing rapidly will not be broadcast needlessly. We employ a similar philosophy in our lazy state update but we make no distinction between “infrequent” and “frequent” commands.

For efficiently handling state updates, Michael Cox outlines a near optimal algorithm in his Ph.D. thesis[6] for state management in a parallel RenderMan implementation. His approach is conservative (it may send more data than absolutely necessary) because of the undecidability of computing state element equality in RenderMan. Because our system is focused on OpenGL, we do not have this restriction. In addition, we create a hierarchy to represent the changed state elements, so our cost to compute the state changes between an application and a rendering engine can be very fast if few state elements are changing. David Ellsworth et al.[9] describe a system which only sends relevant state elements to rendering nodes. Cox points out that Ellsworth’s system is restricted to supporting retained mode applications. In addition, Ellsworth’s algorithm still requires the broadcast of certain state elements (e.g., matrix transformations).

Finally, context switching is addressed by many papers on graphics hardware. The Apollo DN10000[18] supported multiple graphics contexts in hardware and could perform a context switch in 16 microseconds if the target graphics context was in its 6-context cache. Akeley and Jermoluk[5] identified the need for fast context switching in their paper on high performance polygon rendering. Despite apparent agreement in the hardware community on the need for fast context switching, most hardware implementations can switch contexts only a few thousand times per second. In many cases, these systems are limited by the design of the window system in which they must operate. WireGL’s efficient context differencing operation provides very fast context switching performance without the need for hardware support. This allows multiple applications to share a remote display with very little context switching penalty, a crucial feature for supporting parallel remote rendering.

3 State Tracking

Our original Interactive Mural graphics system[12] was a straightforward RPC-style network protocol for OpenGL, similar to the GLX protocol[1]. Although this approach met our functionality goals, extending it to support high performance remote rendering proved difficult. Because each command simply created a packet representation of its parameters, we could not gain any semantic knowledge of the application’s graphics state or the primitives it was drawing in order to use the network more efficiently.

For tracking state, we have separated API commands into three categories:

- *Primitives*: Any command that generates fragments but does not change any of the OpenGL state. Examples of these include `glVertex3f`, `glRectf`, `glArrayElement`, and `glDrawPixels`.
- *State*: Any command that directly affects the graphics state, e.g. `glRotatef`, `glBlendFunc`, `glColor3f`. Note that `glTexImage2D` falls into this category as well; our system allows for efficient texture management in a remote rendering implementation.
- *Special*: Everything else, i.e. `SwapBuffers`, `glFinish`, `glFlush`, and `glClear`. These commands are handled specially because they have no direct effect on the state, but rather have special interactions with our command buffers.

3.1 Primitive Commands

Since primitive commands do not modify state, these commands are packed immediately into a global command buffer. In addition to these commands, state commands which legally appear between `glBegin` and `glEnd` are also packed into the buffer and their affects on the state is recorded as described below. This buffer will eventually be sent to one or more rendering servers, depending on the system mode being used (see section 6.1 for a discussion of tiled rendering using WireGL).

The WireGL packet format has been greatly improved since its original design[12]. By collapsing redundant opcodes (e.g., `glVertex3f` and `glVertex3fv`) and eliminating opcodes that do not require network traffic (e.g., `glGet`), we have exactly 224 opcodes defined in our new protocol. This allows us to use a single byte opcode for each command. In addition, the length of most packets is implied by the opcode, so the length field has been eliminated. For variable length packets like glTexImage2D, the length appears as the first 32 bits of the data field. In order to retain alignment of arguments, we provide a separate opcode and data buffer to be sent to the servers. Therefore, a glVertex3f call will generate exactly 1 byte of header and 12 bytes of data. These packet format improvements also apply to state and special commands.

3.2 State Commands

Almost all commands that do not generate fragments are commands to manipulate the graphics state. glRotatef, glPixelStorei, and glFogf are examples of these commands. Our implementation of this class of functions merely records the state changes into a "virtual graphics context." The virtual context represents the running application's view of the current graphics state.

Each element of state has *n* "dirty" bits associated with it, where *n* is the number of rendering servers in our remote display configuration (recall that we allow multiple remote rendering servers for a single application). When the application executes a state command, all bits are set to 1, indicating that the virtual context is possibly out of sync with the physical context on all servers. Note that we do not check whether or not the user has set the state element to its current value. This does not mean that calling a state command repeatedly will cause multiple packets to be transmitted; instead it simply tracks the latest value for that element of the state.

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

Most applications change only a very small subset of the entire OpenGL state between geometry blocks. We therefore also maintain a hierarchy of dirty bit-vectors. This way, we can quickly get to the elements of the state that have changed without re-examining the entire graphics state. For example, we have a bit-vector for the diffuse color of OpenGL's LIGHT0, a bit-vector for all state pertaining to LIGHT0, and a bit-vector for all OpenGL lighting state. As we will explain in Section 4, the context differencing operation is a frequently executed part of the WireGL system, so it is imperative that it be as fast as possible.

The 18 categories we have chosen for state elements are: transformation, pixel, current, viewport, fog, texture, lists, client, buffer, hint, lighting, line, polygon, scissor, stencil, evaluators, imaging, and selection. These categories closely follow the ones laid out in table 6.5 of the OpenGL 1.2.1 specification[1], although we have collapsed some of the more similar categories (e.g., color buffers and depth buffers are collapsed into "buffer" state).

One optimization that can be enabled in our system is disabling the tracking of state commands inside of glBegin and glEnd. These state commands are used to specify vertex parameters and the application typically does not rely on the persistence of these values outside of the glBegin/glEnd pair. For example, it is unusual for an application to set the color *inside* a glBegin/glEnd pair and rely on that value persisting to future pairs (the application would usually set that color outside of the pair, since the color is not intrinsic to the geometry delimited by the glBegin/glEnd). This allows us to greatly optimize overall system performance without affecting the correctness of most applications. We allow the commands glColor3f, glNormal3f, glTexCoord2f, glIndexf, glMaterialf and their variants to be packed without updating the state. As a result, the system can processes these frequently executed commands much faster than with full state tracking. Note however if this optimization does produce artifacts, it can be disabled by the programmer at the expense of performance.

3.3 Display Lists

Display lists require special handling in WireGL. Because a display list may be executed at a later time when the graphics context is in an unknown state, we need to capture the entire list of commands verbatim rather than separating them into primitive and state commands. Therefore, when we encounter a glNewList command, all functions which are legal inside display lists are changed so that they pack their arguments into the global command buffer. By causing state commands to behave like primitive commands, we prevent list creation from affecting the current state. This way, the original definition of the display list as written by the programmer is sent to each rendering server which guaranties it to be resident on the server when the list is called.

This method allows our system to handle most display lists that appear in OpenGL programs, since they are typically used to encapsulate a block of geometry. However, OpenGL permits display lists to have side effects on the state after they are executed. While it is rare for an application to rely on such state changes, the algorithm described above does not properly handle such cases. We have designed an extension to WireGL to allow it to properly track state across display lists, even in the presence of inter-dependent list definitions, but it is not yet implemented. See section 7.2 for more details.

3.4 Vertex Arrays

OpenGL vertex arrays pose an interesting problem for efficient network rendering. Vertex arrays allow the OpenGL driver to retrieve the vertex, color, normal, and other attributes directly from the application's memory. This minimizes function call overhead and reduces the amount of data that has to be packed into command buffers by the driver. However, in remote rendering, the graphics card and the client application are separated by a network. One simple solution is convert a glArrayElement call into the equivalent glVertex3f, glNormal3f, glColor3f, or glTexCoord2f calls, which is the approach taken by GLX. While this technique produces the correct images, it negates all the advantages of using vertex arrays, since the vertex data are always sent over the network, regardless of whether an update is needed.

State tracking provides a much more efficient model for transmitting vertex arrays. WireGL maintains a local cache which contains the values of the array data from the last time the application referenced the geometry. As the application calls glArrayElement or glDrawArrays, the local cache is compared against the application data. If there is a difference, a dirty bit is set to indicate that the element needs to be updated on the server and the data are then copied into the local cache. This process is repeated for each of the enabled arrays. Finally, the element is added to a list for processing during lazy update (see section 4). Pseudocode for glArrayElement is shown in figure 2. When the state is transmitted the referenced array elements will be updated on the server.

```
glArrayElement ( element ) {
  for each enabled array {
    compute offset
    compare client array data with cache array data
    if (cache invalid) {
      copy client data to cache data
      set array dirty bit at offset
    }
  }
  copy element to element list
}
```

Figure 2: Pseudocode for `glArrayElement`. WireGL will only transmit used elements of the array. This caching strategy also permits the application to modify its vertex arrays in memory to change the geometry and WireGL will only transmit the modified regions of the data.

Note that if the application uses the common `glLockArrays` extension, WireGL knows the size of the vertex array and can depend on the application data to remain static while the array is locked. This simplifies the code and improves the speed at which `glArrayElement` calls can be made.

3.5 Texture Management

OpenGL textures can represent a large fraction of the network traffic in a remote rendering application. For example, a 512×512 32-bit texture with a full mipmap pyramid results in $1\frac{1}{3}$ megabytes of data. For texture intensive applications, the loading time for textures can be quite long and may seriously hinder performance.

By tracking texture state we can restrict network traffic to include only those textures which will actually be rendered on screen. However, keeping a complete copy of the texture state on the client is not a trivial task. Texture state in OpenGL can be divided into three levels:

1. *Global state*: The currently bound texture object number, texture coordinate generation state, and texture environment settings.

2. *Object state*: State associated with each texture object, such as filtering and coordinate wrapping.

3. *Mipmap state*: The actual images comprising the texture's mipmap pyramid.

Each of these levels is tracked along with associated dirty bits. When the client program makes a `glTexImage2D` call, WireGL copies the image data and sets all the dirty bits for that texture object's modified mipmap layer. During state update, only the texture which is currently bound will be sent over the network.

If regions of a texture map are modified by `glTexSubImage2D`, WireGL keeps track of the invalid rectangle for each mipmap level. Only the invalid regions of a texture will be transmitted.

3.6 Performance

Graphics drivers are highly tuned pieces of code that are designed to support applications rendering millions of primitives per second. Consequently, any computation placed in the implementation of the API must be well designed and optimized. For a networked rendering system, the time to create and transmit the command buffers will tend to be the limiting factor for system performance. However, we would like an application to be able to run remotely at least as fast as it can run locally.

In order to achieve this goal, we need to keep our API overhead to a minimum. Our routines to pack command buffers have been hand-optimized to maximize performance. To measure the impact of the state tracking system, we determined the percentage of each frame spent inside the state tracker for two applications: the OpenGL *atlantis* and *fire* demos. Both demos can be found in the standard GLUT distribution[3]. Our version of the *atlantis* demo has been modified to have 40 sharks instead of the default 10. The results of this experiment are shown in figure 3.

The *atlantis* demo puts more pressure on the state tracking system, since it makes repeated calls to modify the transformation stack. Each one of these calls incurs a software matrix multiply. When modeling the "ideal" network (i.e., infinite bandwidth and zero latency), state tracking for *atlantis* represents 20 percent of the total frame time. However, when run over a gigabit network (Myrinet), we see that the overhead has been reduced to less than 10 percent. When the network bandwidth drops to 100 megabits per second, the overhead of tracking state is negligible; it represents less than one percent of total frame time.

Our results are even better for the *fire* demo, which is a particle simulation. *fire* updates the position of its particles manually, which minimizes expensive calls that modify the matrix stack. As a result, state tracking represents less than 10 percent of the total frame time on the ideal network.

4 Lazy State Update

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

In a networked graphics system, conserving bandwidth is a top priority. Therefore, OpenGL API calls should be expressed in as few bytes as possible, so that WireGL can make better use of the available network resources. Because WireGL records the OpenGL state rather than generating packets for state commands immediately, it can defer decisions about what state elements to transmit until after the user has drawn some geometry.

Whenever the programmer makes a call to an OpenGL state command, WireGL must first determine if any geometry has been packed but not yet sent. If this is the case, it must update the servers' states and flush the global command buffer before recording the state change. Pseudo-code for WireGL's server update algorithm is shown in figure 4. The decision about which servers to update depends on the network rendering model; see section 6.1 for one possibility.

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

Figure 4: Pseudo-code for WireGL's buffer flush algorithm. By not encoding state until geometry has an effect on the output image, WireGL saves unnecessary network traffic.

This algorithm has the advantage that state is only sent when the geometry it affects is also sent. Therefore, if a block of geom-

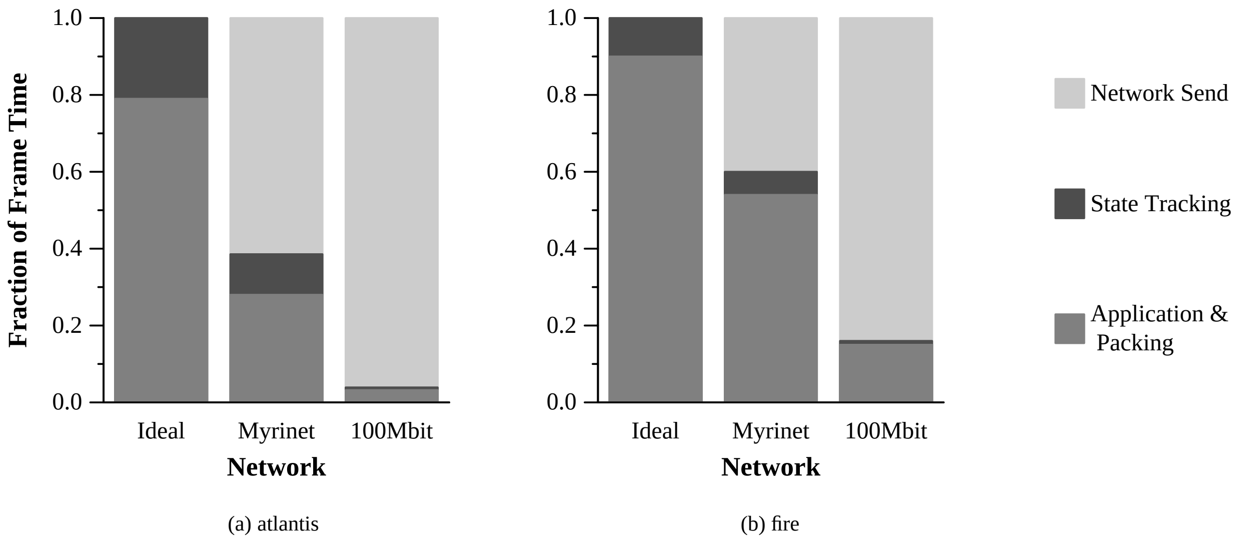


Figure 3: Frame time percentages for the OpenGL *atlantis* and *fire* demos. Time in each frame has been charged to three system components: network, state tracking, and application/packing. The network time is the amount of time spent waiting for data to be transmitted across the network. Note that the "ideal" network represents a network with infinite bandwidth, so no time is spent waiting for network transmission. Application/packing time is either time in application code or time spent packing command buffers for transmission. Any remaining time is charged to the state tracker. The state tracker is operating using the optimization described in section 3.2.

etry is culled by WireGL, its associated state will never consume network resources. As described in section 2, most OpenGL implementations simply broadcast the state to all geometry processing elements. Our approach is particularly advantageous when using many large texture maps. Certain applications like *Quake 3: Arena* make numerous calls to `glTexSubImage2D` in order to update small details on surfaces in the scene. This usage pattern is particularly expensive when rendering remotely, and updating textures lazily can provide a substantial performance gain (especially in tiled rendering mode).

4.1 Computing Context Differences

The hierarchical dirty bit representation allows WireGL to efficiently compute the difference between the application's virtual context and each server's context. Recall that each state command sets the dirty bit for each server. Pseudocode for `glBlendFunc` is shown in figure 5. Pseudocode for the hierarchical bit tests to update the fragment state is shown in figure 6. Other state categories are updated in a similar fashion.

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

Figure 5: Pseudocode for the `glBlendFunc` function. Notice the use of multiple dirty bits in accordance with our hierarchical state change tracking scheme.

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

Figure 6: A portion of the code to update a server's fragment state. If no fragment state changes had been made, we would not check any of the fragment state elements, including `glBlendFunc`.

The dirty bits provide a fast mechanism for detecting which elements of the OpenGL state need to be updated. Furthermore, the update routine performs a comparison of the application and server values before generating a state command. This prevents redundant state commands from being transmitted on the network (for instance, if a user sets up the entire OpenGL state every frame). Also, we perform error checking when each state command is called, in accordance with the OpenGL specification. Commands with erroneous parameters will not be sent over the network.

In addition, some care is taken to only send "relevant" state updates. Although not shown in figure 6, the user may have changed the `glBlendFunc` parameters since the last time WireGL updated a particular server's blend function settings, but if blending is disabled, those settings do not need to be sent to the server. Instead, the blend function dirty bit is left marked, but the bit for the entire fragment state is cleared. This way, the parameters will be sent if blending is enabled in the future, but in the meantime the fragment state will be skipped entirely when computing context differences.

Our context differencing implementation also maintains a level of indirection when generating the state commands. In figure 6, the statement `generate glBlendFunc packet` is performed by calling a function pointer associated with `glBlendFunc`. When rendering over the network, this function pointer points to the code that packs the command arguments into our network buffers. However, we can set this function pointer to the actual system implementation of `glBlendFunc`, which allows us to track the state of an application as it renders directly to the screen. We use this feature for soft context switching as described in section 5.

5 Soft Context Switching

Traditional graphics accelerators are optimized to move data in one direction: from the host to the screen. Therefore, any operation that requires a reversal of the pipeline is typically very expensive. In particular, allowing multiple applications to share a single display requires the accelerator to switch graphics contexts rapidly; a process usually handled by the operating system and graphics driver. Unless the graphics card supports multiple contexts in hardware, this operation will require that the entire pipeline be flushed and the state registers be read back over the system bus. In fact, even cards that support multiple hardware contexts, such as the NVIDIA GeForce, are forced to read back state from the card when an application thread explicitly changes contexts due to constraints of the window system in which they operate[2].

Performing OpenGL state tracking in software alleviates the need for reading back from the graphics card. We can use the same efficient context differencing operation described in section 4.1 to perform a software, or "soft", context switch. By performing this switch in software, the graphics card needs only to maintain a single hardware context which needs never to be interrupted.

The model for soft context switching is only slightly different from the virtual graphics contexts used for remote rendering. Each dirty bit now represents a local context's relationship to the actual state stored in hardware. If a bit is set, the hardware context may not have the same value as that context. When we perform a soft context switch, we examine the bits hierarchically as before. If a particular state element is out of date, we update the hardware context, and set all the other contexts' bits to one. Pseudocode for a portion of this algorithm is shown in figure 7.

Once the soft context switch is completed, the hardware's context will exactly match the target software context. In generating a `glBlendFunc` command, the system may have invalidated the graphics hardware with respect to all other soft contexts. We therefore must re-evaluate the validity of all other contexts' blend functions when we switch back. Also, the value equality test plays an important role in soft context switching. By checking the actual values, we prevent unnecessary state commands from being generated when multiple contexts have the same values.

An added benefit of performing soft context switching is the ability to insert a level of indirection between the application's texture object or display list numbers and the numbers actually sent to the hardware. This allows unrelated applications to share a single hardware graphics context without constantly reloading their texture objects and display lists.

The speed of soft context switching is proportional to the differences between the currently active context and the context to which we are switching. If they are exactly identical, the operations require 18 bit tests and one assignment. To measure the performance of soft context switching, we wrote a simple application that created one window and many contexts, and switched the window's rendering context as quickly as possible. The results for this experiment

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

Figure 7: A portion of the code to switch contexts in software using our hierarchical bit-vector representation. If no contexts are changing the blend function, no `glBlendFunc` calls will be issued. Also, if no contexts change any of the fragment state, many tests can be skipped.

are shown in table 1.

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

Table 1: Context switching rates for various OpenGL implementations. For the "Identical" column, we are context switching between contexts with no differences. For the "Varying" column, we change the matrix stack and current color for each context before drawing. The InfiniteReality is hosted in an 8 processor MIPS R10000 based Silicon Graphics Onyx2; all other hosts use a single Intel Pentium III processor. The lower performance of the InfiniteReality is largely due to the expense of flushing its deep command buffers, a problem faced by more complex graphics accelerators.

Clearly, WireGL will run more slowly when the contexts are varying; the varying context column in table 1 requires a software matrix multiplication as well as calls to `glLoadMatrix` and `glColor3f`, plus drawing a triangle. Still, this application runs 38 times faster with WireGL than without on the same hardware. For applications which need multiple contexts (such as the parallel submission scheme described in section 6.3) soft context switching has a potentially large performance advantage.

6 Applications

In this section we give a high level overview of three applications of our state tracking system: tiled rendering, geometry compression, and parallel graphics command submission. Each of these topics is a rich area of research in itself; we merely intend to give a flavor for the kinds of technology that a state tracking system enables.

6.1 Tiled Rendering

In tiled rendering, the screen is broken into a number of rectangular regions, called tiles. For our purposes, we assume that each tile is managed by a separate rendering server. Our state tracking system allows us to transparently parallelize an existing OpenGL application using a sort-first technique[7]. In sort-first rendering, primitives are transformed into screen space as early as possible, then sent to the rendering engine or engines that control that area of screen space.

We could simply broadcast the stream of graphics commands to all servers and rely on OpenGL to clip the geometry that falls off-screen. However, this places unnecessary strain on the network and does not scale well when throughput is limited by network bandwidth. Instead, WireGL generates a separate, distinct OpenGL stream to each server. As described in section 4, each stream consists of a block of state commands followed by a block of geometry commands. In order to efficiently perform the sort, we maintain the bounding box of geometry as it is issued by the running application. This prevents us from having to transform each vertex as it is issued, which would be very expensive.

In our current implementation we sort geometry whenever a state command is executed and geometry has already been packed. For most applications this allows us to bound and transform entire blocks of geometry at a time. For example, in the *atlantis* demo, state commands are only submitted between animals, which allows us to construct exactly one bounding box per animal. We also allow the programmer to manually place an upper bound on the number of vertices allowed in a single bounding box; we are looking into automatic techniques for determining this upper bound on the fly.

The basic sorting algorithm is as follows:

```
transform geometry's bounding box to screen space
for each server {
  if (box intersects the server's viewport) {
    send_geometry( server )
  }
}
```

where send_geometry is the method presented in figure 4. Note that tracking the bounding box can also avoid transmitting geometry that falls outside the application's viewing frustum.

This tiled rendering technique allows us to evaluate the performance of our lazy state update algorithm. The OpenGL *atlantis* demo sends 3,020 bytes of state commands per frame when rendering to a single remote server, compared to 375,223 bytes of primitive commands. Since the amount of state data is two orders of magnitude less than the amount of primitive data, it represents a small fraction of the network send time. However, when rendering to a 5x5 tiled display, broadcasting the state would cause WireGL to send 75,500 bytes of state commands. Without lazy update, the amount of state data sent could easily surpass the amount of primitive data as the size of the tiled display increases.

Using lazy update and bounding box tiling, the amount of state data sent to our 5x5 display is held to 4,530 bytes, a 16:1 savings over broadcasting state. Clearly, lazy state tracking is an essential component to the scalability of our tiled rendering system.

6.2 Compression

Using WireGL, we have implemented a geometry compression scheme that works on immediate mode streams. This scheme uses the screen space bounding box information to compute the derivatives of the object to screen space transformation, as in mipmapping[19]. Once these derivatives are known, we can compute the maximum object space movement that will result in a sub-pixel screen space movement. This allows adaptive quantization of the vertex data on a per-axis basis. Geometry that is farther away from the viewer will naturally receive more compression because of its perspective foreshortening.

Unlike off-line geometry compression techniques[8, 11], a real-time compression system has unique requirements, such as needing very fast compression as well as decompression. In previous work, the compression step has typically been performed in an off-line pre-process and requires *a priori* knowledge of the scene geometry. Since our compression scheme uses simple quantization based on derivatives of transformations that are already maintained by the state tracker, we are able to compress data very quickly. By running our compression system on a number of different applications, we have determined that any network that has less than 90 MB/sec of bandwidth can benefit from this technique.

The results of this system have been quite promising, and we have been able to achieve a 6:1 compression ratio for the *atlantis* demo with no loss of image fidelity.

6.3 Parallel Submission

In order to overcome the bottlenecks associated with sending graphics commands to the accelerators, recent research has focused on APIs for parallel issuing of graphics commands[13]. Although this work has mainly focused on overcoming system bus bottlenecks, the same technique allows multiple remote hosts to submit streams in parallel to a single rendering server over a high speed network.

In these parallel programs, each stream has a distinct associated OpenGL context, and explicit synchronization primitives are inserted into the command streams if the programmer wishes to express ordering constraints between contexts. Therefore, when the rendering server sees a synchronization primitive, it must quickly switch OpenGL contexts to begin executing commands from another stream until the synchronization primitive can be resolved.

In order for this technique to scale past a trivial number of submission nodes, context switching must be as lightweight as possible. Clearly the standard pipeline flush and read-back model will not be feasible, as the programmer may need to exert very fine grained control over the drawing order even within a single frame. For example, consider a parallel implementation of the marching cubes volume rendering algorithm[15]. Currently, the programmer must carefully choose the granularity of parallelism to avoid being context switch limited and still have acceptable load balancing behavior. For a 128^3 volume, the finest granularity of parallelism may force the graphics system to perform as many as 63 million context switches per second in order to refresh 10 times per second. This is an extreme example, but even assigning a 4x4x4 block of voxels as a unit of work will still result in slightly less than 1 million context switches per second. This level of context switching performance is not available in any hardware system today.

In WireGL the time to switch between two contexts depends on how often the application is changing the contexts. If neither context is changing a particular component of the state, this will become apparent with a small number of bit tests (see section 5). Most well-behaved applications set up state at the beginning of each frame (or even at startup) and issue large amounts of geometry over the course of each frame. Parallel marching cubes is an example of such an application; each processor submitting graphics commands initializes its graphics state at the beginning of its execution, and spends most of its time simply generating triangles to represent the isosurface of interest. This means that it will be extremely cheap to switch between any of the rendering contexts; 18 bit tests reveal that nothing has changed since the last time a context was active. This should allow us to achieve our full 5,800,000 context switches per second, as measured in section 5.

Although not all parallel applications are as well behaved as marching cubes, it is rare that we need to decode two completely unrelated streams in parallel with fine grained synchronization. Because the streams are contributing to a single image, their contexts will tend to differ only by a transformation matrix and possibly a few enabled flags. This makes soft context switching very effective. Using WireGL, programmers writing applications that wish to submit commands in parallel have much more freedom to choose the granularity of parallelism in their applications.

7 Future Work

Currently, WireGL is being deployed on a cluster of 40 workstations at Stanford, and will soon be deployed on larger systems outside our lab. We believe that our support for parallel submission and soft context switching should allow us to aggregate the performance of many independent graphics accelerators.

7.1 Mobile Contexts

Process migration has been a popular load balancing technique in the operating systems research community[10].

In order to migrate a process that is rendering 3D graphics, it is necessary to package up not only the program state, but also the graphics state, so that when the process is started in its new location it can continue rendering where it left off. With a software mirror of the graphics context, we can easily serialize the contents of that data structure and pass it along with the process state in order to move the application to a less loaded computer in our cluster.

In fact, by using a combination of digital video switching and chroma-keying we can easily migrate a process which is doing direct rendering to a user's workstation onto a more powerful server, without interrupting the user's session.

7.2 Display List Dependencies

Obviously we would like WireGL to be fully compliant with the OpenGL specification. Because we have been aggressive in conserving network bandwidth, the semantics of OpenGL display lists make compliance difficult. For example, consider the following snippet of code:

```
glNewList( 1, GL_COMPILE );
glRotatef( 30, 0, 0, 1 );
glCallList( 2 );
glEndList();

glNewList( 2, GL_COMPILE );
DrawTriangles();
glPopMatrix();
glEndList();

glMatrixMode( GL_MODELVIEW );
glPushMatrix();
glCallList( 1 );

glPushMatrix();
glRotatef( 130, 0, 0, 1 );
DrawTriangles();
glCallList( 2 );
```

Though pathological, this code is certainly legal OpenGL and has well defined semantics. Currently, WireGL will not realize that the matrix stack is empty at the end of this section of code. This is particularly unfortunate when using the tiled rendering scheme outlined in section 6.1, since tracking incorrect transformation matrices will result in geometry being clipped or drawn incorrectly.

In order to solve this problem, we propose to maintain an array of OpenGL commands that affect the state for each defined display list. Then, when a list is called, we re-execute those state affecting commands. Note that we would include `glCallList` in such a command array, so we can recurse through the all display list dependencies. Although we have not explored this space fully, we believe that there should generally be very few state affecting commands in a display list, since display lists are typically used to describe large blocks of geometry. Since this technique is only necessary if a program uses display lists with non-zero effects on the state, it could be turned off for many applications.

8 Summary and Conclusions

We have described a state tracking system designed to facilitate efficient networked rendering. The portions of our cluster rendering project that benefit from state tracking extend well beyond simply providing a fast response to application state queries. Having a software mirror of the graphics state allows WireGL to better manage network resources by transmitting only those commands that will affect the resulting rendered image. The benefits of lazy update are most visible when combined with our tiled rendering implementation. In many cases, WireGL can allow an application to render remotely to a tiled display as quickly as it can to a single display by quickly culling large blocks of geometry and not re-transmitting that geometry's associated state.

In addition, we have shown that performing context switching in software can be extremely efficient when combined with our hierarchical dirty bit representation for state changes. This allows us to support several independent remote streams very efficiently. Accepting multiple streams is important for sharing remote displays between applications and supporting parallel issuing of commands. As tiled displays become larger and more pervasive, the ability to efficiently share a display between multiple simultaneous users will become crucial.

In general, we believe that state tracking coupled with a hierarchical representation of state changes is an effective and efficient technique for improving the speed and robustness of a remote rendering system.

References

[1] OpenGL specifications. http://www.opengl.org/Documentation/Specs.html .

[2] Personal correspondence with Nick Triantos, NVIDIA Corporation.

[3] The OpenGL Utility Toolkit. http://reality.sgi.com/mjk/#glut .

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

[5] Kurt Akeley and Tom Jermoluk. High-performance polygon rendering. *Proceedings of SIGGRAPH 88*, pages 239–246, August 1988.

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

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

[8] Michael F. Deering. Geometry compression. *Proceedings of SIGGRAPH 95*, pages 13–20, August 1995.

[9] David Ellsworth, Howard Good, and Brice Tebbs. Distributing display lists on a multicomputer. *1990 Symposium on Interactive 3D Graphics*, pages 147–154, March 1990.

[10] M. Harchol-Balter and A. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Transactions on Computer Science*, pages 253–285, August 1997.

[11] Hugues Hoppe. Progressive meshes. *Proceedings of SIGGRAPH 96*, pages 99–108, August 1996.

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

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

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

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

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

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

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

[19] Lance Williams. Pyramidal parametrics. *Proceedings of SIGGRAPH 83*, pages 1–11, July 1983.