A Distributed Graphics System for Large Tiled Displays

Greg Humphreys* Pat Hanrahan†

Computer Science Department
Stanford University

Abstract

Recent interest in large displays has led to renewed development of tiled displays, which are comprised of several individual displays arranged in an array and used as one large logical display. Stanford's "Interactive Mural" is an example of such a display, using an overlapping four by two array of projectors that back-project onto a diffuse screen to form a 6' by 2' display area with a resolution of over 60 dpi. Writing software to make effective use of the large display space is a challenge because normal window system interaction metaphors break down. One promising approach is to switch to immersive applications; another approach, the one we are investigating, is to emulate office, conference room or studio environments which use the space to display a collection of visual material to support group activities.

In this paper we describe a virtual graphics system that is designed to support multiple simultaneous rendering streams from both local and remote sites. The system abstracts the physical number of computers, graphics subsystems and projectors used to create the display. We provide performance measurements to show that the system scales well and thus supports a variety of different hardware configurations. The system is also interesting because it uses transparent "layers," instead of windows, to manage the screen.

1 Introduction

Very large displays are an exciting new area of research because they have the potential to truly change the way people interact with computers. Large, high resolution displays may be hung on walls and built into tables to create "smart spaces" that allow new methods of collaboration, visualization, and interaction. One advantage of these displays is the increased resolution, which is particularly important for scientific visualization applications where phenomena may be explored at many levels of detail. Another advantage is the extra physical display space, which may be used to surround a user or group with imagery. The power of immersion is exemplified by spatially-immersive displays such as the CAVE[13]. The same technology may be used in a normal office or conference room environment to support decision making and other group activities.

The biggest challenge of using such systems is picking the user interface metaphor. Although one might be tempted to use a traditional window and mouse based environment, it is not clear that it is the most appropriate design for this environment. Designing visualizations for large displays differs from designing for desktop monitors due to a variety of factors, including size, resolution, brightness, contrast and orientation. Designing the interface is equally cumbersome, since large displays are difficult to use with a tethered mouse or keyboard. Group applications and tasks are also very different from single-user applications.

In order to investigate these emerging types of applications, we have built a tiled, back-projected display called the "Interactive Mural". It is constructed from an array of 8 projectors connected to a high performance graphics system. Each projector has a resolution of 1024x768 and outputs 900 ANSI lumens. Our display is of modest size (6 feet wide by 2 feet high), but high resolution (3796 by 1436) and very bright. Unlike the CAVE, it is bright enough to be used in ordinary office lighting without dimming the environment. A photo of our system in use is shown in our color plate.

The primary input device is a laser pointer stylus, which can be used to either draw on the screen directly or point at objects from a distance. The physical environment is roughly the size of a wide whiteboard, giving it a "walk up and touch it" look and feel. There is enough room for two or more people to work in front of the system. Although similar physically to a computer whiteboard, the Mural is designed to support interactive visualization of large heterogeneous databases, not just drawing and markup [15]. A diagram of the physical construction of our display is shown in figure 1.

In this paper we describe the virtual graphics system that we built for this display. We had several goals for the system:

- The low-level graphics system is designed to support flexible configurations of small physical displays. A single large logical display is created by overlapping multiple projectors and feathering along the seams. Although technology exists for aligning and feathering images at arbitrary angles [20, 24], we only support rectangular tiling patterns. We do allow the individual displays to be rotated by 90 degrees.

- The system is designed to be scalable, supporting variable numbers of computers, graphics cards, and video output ports. Although in this paper we describe our current system for a distributed shared memory machine with two graphics pipes, we were also able to implement our design using a PC cluster configuration.

- The graphics system is designed as a distributed server to support multiple remote clients. Each client is a different application and is typically running on a different machine. Unlike the X window system, which typically has a single server per display, our server is distributed since the graphics system may be partitioned across multiple machines. The system was also designed to support simultaneous rendering from multiple streams.

- The basic unit of screen real estate is a layer rather than a window. Layers behave logically much like windows, but they

*humper@graphics.stanford.edu
†hanrahan@graphics.stanford.edu

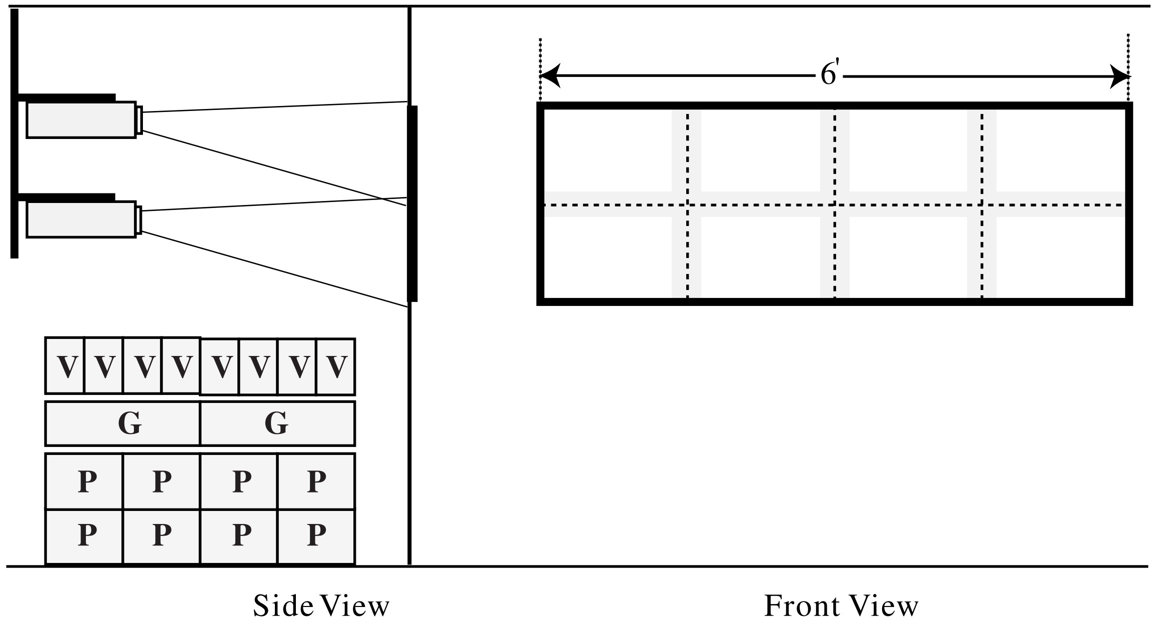

Figure 1: Basic layout of the Interactive Mural. An 8-processor [P] shared memory computer (SGI Origin) connected to two graphics systems [G] (SGI Infinite Reality), each with 8-way video outputs (SGI DG-8) drive a bank of 8 LCD projectors [V] (NEC 1030). Each projector has a resolution of 1024 by 768 and outputs 900 ANSI lumens; their projected images overlap by approximately 100 pixels. The total display size is 6 ft. by 2 ft. and has a resolution of 3796 by 1436.

are composited using alpha blending. Layers also support geometric transformations, and they are not normally nested like windows.

We will present a system that virtualizes the distributed graphics resources controlling the display and provides an interface for multiple remote applications to share the display at once. The main challenges facing an architect of such a system are efficient network utilization and overcoming the graphics context switching overhead. We address these challenges using structured graphics caching and OpenGL proxies.

2 Related Work

There are many ways to virtualize a tiled display system in software. MacOS and, more recently, Microsoft Windows have support for extending the desktop onto multiple monitors connected to a single computer system. In these cases, the device driver takes care of managing the screen real estate and the distributed framebuffer memory. Similarly, the latest release of the X Window system contains the XINERAMA extension[9], which allows multiple displays to be combined into one large virtual desktop. The Silicon Graphics InfiniteReality[17] system allows a single framebuffer to drive multiple displays through one large logical X server.

A more general approach is taken by DEXON Systems' DXVirtualWall[5], which provides extremely large tiled displays that run either X Windows or Microsoft Windows. Clients connect to a display proxy that broadcasts the display protocol to multiple display servers, each of which offsets the coordinates to display its own small portion of the larger desktop. Another use for protocol proxies of this nature is to duplicate a single display across several remote displays, as in Brown University's XmX[10] project. These proxy-based systems do not currently support any high performance 3D graphics API such as OpenGL or Direct3D, and do not handle overlapping displays. Additionally, as the number of displays gets very large, the number of concurrent redraws exerts increasing pressure on the network, causing performance to suffer.

IRIS Performer[22] provides an API for managing multiple graphics pipes within a single application. However, Performer is designed for a single application driving the entire display, and most compelling Performer demos are full screen, immersive walk-through applications. Running multiple Performer applications simultaneously incurs great context switching overhead, resulting in pronounced performance degradation. In addition, Performer is designed around hierarchically defined "scene graphs"; arbitrary OpenGL applications do not receive much of the benefit of Performer. Finally, Performer makes little attempt to virtualize the configuration of a multiple-pipe system; applications need to be aware of the number of pipes available to them and use them explicitly.

In the area of remote graphics, the X Window System[18] has provided a remote graphics abstraction for many years. This system has a heavily optimized network usage model, and forms the basis for much of our system's design. GLX[16] is the dominant API and protocol used for rendering OpenGL remotely over a network. GLX provides a seamless way to display 3D graphics on a remote workstation. However, GLX has no underlying support for sending the same stream of commands to multiple displays. GLX's wire protocol for OpenGL is very compact, and our own wire protocol is almost identical.

The University of Minnesota's PowerWall[7] uses the output of multiple graphics supercomputers to drive multiple outputs, creating a large tiled display for high resolution video playback and immersive applications. The University of Illinois at Chicago extended this system to support stereo display and user tracking in the InfinityWall[14] system. Unlike our system, neither of these displays overlaps its projectors. These systems are designed to facilitate a single full-screen application, which is often an immersive virtual reality system. More expensive custom hardware solutions are available as well. Panoram Technologies[3] has a suite of hardware devices designed to create large tiled displays. Their "Integrator", a special analog feathering box, blends the outputs of individual projectors. Because of the increasing performance of graphics systems, it is now practical to perform this blending step in software.

Finally, projects similar to ours are currently underway at Princeton University[2] and UNC Chapel Hill[19].

3 Design Goals and Implications

Our main goal while designing this system was to create a graphics system that effectively virtualized the distributed nature of our display, without sacrificing familiarity or performance. We also wanted to provide a layered graphics abstraction which facilitates the construction of many kinds of applications, and allows for natural manipulation of inherently layered data such as maps or animations. In addition, it was important to use a portable graphics API as well as a traditional input processing paradigm so that existing applications could use the display with a minimum of effort. Furthermore, a crucial element of our design was the ability to facilitate distributed, remote visualization. Rather than thinking of large display spaces as an opportunity for immersive applications, we instead designed and tuned our system for graphical information sharing, collaboration, and visualization. Finally, we wanted our system to allow for different tiling configurations, and to effectively virtualize the tiled display by eliminating the seams between projectors both logically and visually.

3.1 Layers

An emerging metaphor in a large number of graphics systems is the use of layers as a primitive. For example, drawing and image editing software such as Macromedia Freehand and Adobe Photoshop is based on layers. Video editing, special effects systems, and computer animation systems all use layers. Finally, game machines have always used layers (a.k.a. sprites); the Microsoft Talisman architecture [26] is a modern example of such a system. More importantly, the design of interfaces is changing rapidly in this direction, as exemplified by the look and feel of a web page versus a traditional widget-based application. For these reasons we think layers, not conventional windows, provide the best foundation for the class of applications we are building.

In some ways layers are like traditional windows; they can be moved and resized, shown and hidden. They are rectangular and have a stacking order with respect to each other. Each layer is represented as an RGBA image, and the stack of layers is combined using alpha compositing to create the final displayed image. Each layer is a fully independent entity which can be placed or scaled arbitrarily, without regard to the position or size of any other layers.

Using layers instead of windows has several consequences. First, because the visible pixels may be the result of a blend between many layers, redrawing any layer in a stack means that all the layers in the stack must be recomposited to form the final image. In order to schedule redrawing, we require that each layer has "frame semantics"; that is, we require that an application indicate when it is finished drawing the layer. This is similar to the explicit buffer swap that is performed with a double-buffered application. Finally, because of the transparent nature of layers, they must be composited in back-to-front order, requiring the drawing commands associated with each layer to be serialized.

Our system also allows an application to associate arbitrary named data with a layer, so that applications can communicate with each other using the Mural as an intermediary. This is similar to the concept of interclient communications in X Windows[23]. This capability allows applications that are not on the same machine to communicate without creating direct connections between them. It also allows us to easily build user interface tools like magic lenses[11]. For example, an application might associate a display list describing a complex 3D object with a layer. When another layer is moved in front of that layer, it would be notified of the overlap and could draw the 3D object in a different style.

3.2 OpenGL and Input

It is important that the graphics API supported by our display be widely used, so that applications can be developed using a stable and mature graphics library that is familiar to most graphics programmers. We therefore support OpenGL as the low-level graphics API. Although there are several graphics libraries that could have served our purposes, we feel that OpenGL best allows us to support visually demanding, high performance 2D and 3D applications. Furthermore, the portability of OpenGL gives us flexibility in choosing the systems that drive our projectors; in particular, we want to support both UNIX and Windows operating systems.

Choosing OpenGL as the only method of drawing imposes certain restrictions on the types of data that can be easily displayed. There is no direct way to display an existing X or Microsoft Windows application on our display. We have solved this problem by creating a client for the VNC system from AT&T Research[8], which allows us to display a Windows or X desktop in a layer. The VNC client runs as a normal remote application, not as an extension to the server. Another problem with using OpenGL is that it provides no support for rendering fonts. We have created our own system for rendering text from vector font descriptions, based on the `gltt` library[21]. One final addition we have made to the graphics library is direct support for video playback — the system is capable of either decoding multiple MPEG streams or playing back numbered frames from disk.

The sensing and integration of new input devices (such as laser pointers, gesture recognition, head tracking, etc) into our system is a separate topic of research. The major complexity is that the input devices and their data form a distributed sensor network that must be fused as well as transported across a network. We also need to support input from multiple users. However, the interface to these input devices for the application programmer is the classic input loop paradigm and/or registered callbacks.

3.3 Distributed, Scalable Graphics

An important goal of our system is to allow multiple applications to share the display space without overloading any one system component. Therefore, we find it natural to allow applications to run on remote computers. To achieve this, it was necessary to separate the Mural into a true client/server architecture, and to define a wire protocol to transport the entire API and callback mechanism between the Mural server and the Mural applications.

Remote graphics is a key element for collaboration at a distance. We want remote sites to be able to use our system to visualize their datasets and share visual information, using our system as the central point of their collaboration. We must provide a framework for high-performance remote rendering, and make every effort to efficiently use the networking resources between the display server and the remote site controlling the visualization application.

3.4 Support for Tiling

In order for the system to appear as one seamless display, it is necessary to geometrically and radiometrically calibrate the bank of projectors. First, we need to compensate for the fact that the overlapping area between projectors will appear brighter than the non-overlapping area. Second, seams in the display will be visible unless we address the varying color temperature and overall brightness between projectors.

To handle the overlap, we draw "feathering" polygons that cover the overlapping regions. These polygons are texture-mapped with a one dimensional alpha-only image that modulates their opacity from fully opaque to fully transparent. Adequately handling the differing color characteristics between projectors requires a color calibration procedure that is beyond the scope of this paper.

The use of software feathering imposes frame semantics on applications that use the display, just as did the layer abstraction presented earlier in this section. In order to properly blend overlapping projectors together, the system must know when all applications have finished drawing their frames to the screen so that the feathering polygons can be applied as the last operation in each frame.

4 System Architecture

The main challenges facing the architect of a large-scale display server are utilizing the network efficiently and minimizing graphics context switches. Graphics context switches occur when two different applications want to use the graphics hardware simultaneously, and the hardware needs to continually swap their respective graphics contexts in and out. This is an extremely expensive operation and should be avoided whenever possible. We define an OpenGL wire protocol which overcomes this problem by letting our server render all graphics commands on behalf of each client, using a single context for all clients.

Our graphics system is implemented as a networked server which runs on a Silicon Graphics workstation. We create one thread per pipe to manage and render all graphics commands, one "master" thread to oversee redraw dispatching and event redirection, and one client thread per connected application to dispatch other Mural-related commands like "move layer", "resize layer", etc. Applications render using our implementation of the OpenGL API, which sends a stream of commands over the network to the pipe threads for rendering. A block diagram of this system is shown in figure 2.

Because of the compositing behavior of layers, it is often necessary to redraw the entire screen even though only one layer is changing. This is a major difference from normal window systems, and many of the choices made in our implementation reflect the difficulty of achieving acceptable performance in light of this constraint. Our system maintains a per-layer display list so that non-changing layers can be redrawn without generating any network traffic. X servers provide a rarely used option called "backing store" that lets the server maintain a bitmap for a window so redraws of that window do not require a network round-trip. Our approach is similar, but more structured: we maintain a list of drawing commands for each layer, so they can be played back on the server without re-requesting them from the application. This also means that simply moving a layer will not generate any new draw events or network traffic, even though other layers may be partially obscured or exposed by the operation. This allows for extremely efficient use of network resources, as well as a very responsive system even when multiple complex data sets are being displayed and manipulated simultaneously.

We also provide an equivalent for most of the traditional operations performed on a window system, such as moving layers, resizing layers, and requesting input. Although this is a new display space management API, it is still somewhat similar to managed windows in a traditional graphical user environment.

The internal structure of our design is very similar to a multithreaded X server[18]. The key differences are that structured backing store is a crucial element to the design, and that several different screens are virtualized in the same server (X allows for multiple screens on a single server, but it names each one individually). The most important similarity is our heavy reliance on well-designed wire protocols and the emphasis on efficient use of network resources.

4.1 The OpenGL Protocol

Because of the distributed nature of our system, we need to provide an implementation of the OpenGL API that will work remotely over a network. To achieve this, we define a wire protocol for OpenGL. This protocol is used in conjunction with our own implementation of the OpenGL API, which we call the Mural Client Graphics Library (MCGL). The MCGL protocol is very similar to the GLX protocol[6], with a few minor changes. The decision to implement our own protocol rather than port GLX was mainly due to the fact that GLX is heavily tied to the X windows model of displays and contexts, and we wanted to extract the essence of the protocol without worrying about features like pixmaps, contexts, visuals, etc.

In our system, an application makes calls to MCGL, which sends packets to the Mural server's pipe threads. The pipe threads then execute the OpenGL commands on the application's behalf. MCGL is a replacement for the standard OpenGL shared libraries, which gives us extra flexibility in dealing with pre-built applications.

4.2 Graphics Pipe Threads

The Mural server contains one thread for each graphics pipe controlled by the system. The main purpose of these pipe threads is to receive OpenGL commands via the wire protocol described above, and to execute those commands on behalf of the client application. Each pipe receives the MCGL stream, decodes it, and compiles it into a display list for the appropriate layer. Then, for each projector that the layer intersects, the pipe thread sets up an appropriate viewport and scissor rectangle to clip the drawing to the layer's extent on that projector, and finally calls the display list. The use of display lists is crucial to the success of our system. Display lists alleviate the overhead of decoding the wire protocol more than once if the layer overlaps more than one projector. Although the sophisticated display controllers in the InfiniteReality graphics board would allow us to render the entire scene once and scan out overlapping regions directly to the projectors, this would make software blending between the edges impossible.

Certain OpenGL commands cannot normally be executed inside a display list, like the creation of additional display lists. Because all of our drawing commands are being compiled into display lists by the server, most of these restrictions are imposed on all drawing commands made by the application. We provide two workarounds for this restriction. When a layer is first created, the application receives a special "setup" event which allows the client to set up any initial state that should persist throughout the lifetime of that layer, as well as perform one-time operations such as pre-loading textures into texture memory. In addition, we allow for the creation of display lists within a draw callback by first stopping the compilation of the current display list. We then create the user's new display list, and finally, create a third display list for the frame's remaining OpenGL commands. This requires us to maintain an array of display lists per layer, rather than a single one. This has the useful side effect of being able to "append" to the list of drawing commands rather than replace them during the next draw callback. This feature allows the user to treat the display as a single buffered device, relieving some of the restrictions of frame semantics.

If there is more than one graphics pipe then we broadcast the graphics commands to all the pipes at once. Each pipe then decodes and executes the commands in parallel for each projector that the layer overlaps. A simple rectangle intersection test lets the pipe thread decide whether or not to draw a layer on each projector.

The pipe threads also maintain mirror copies of each layer's graphics state. Because there is only one graphics context for each

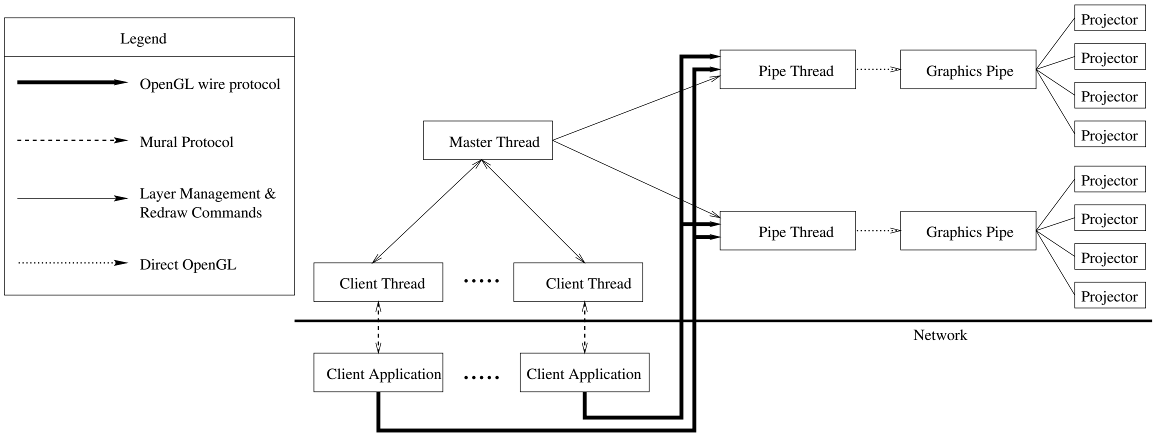

Figure 2: Block diagram of our current system architecture. Each application may be running on a different remote site. Notice that graphics commands do not go directly to the graphics pipes, but rather through the pipe thread "proxies", to avoid context switching the graphics pipe. The pipe thread will cache these graphics commands for use when a redraw is required.

pipe, it is necessary for us to record all the OpenGL state changes that are made by a layer and reset the graphics state so that it does not appear to change between draw callbacks. This relieves application programmers from having to set up the graphics state each time they receive a draw callback and is much cheaper than an actual context switch. The only element of the state that is not restored is the current texture map — we recommend that application programmers use the texture object API to name their textures. This also allows them to share textures between applications easily.

Finally, the pipe threads are responsible for "feathering" the overlapping areas between projectors. Once all the layers on the system are drawn, a polygon is drawn on the overlapping region whose transparency is modulated from fully transparent to fully opaque. Since this is done on two projectors that overlap the same space, the additive effect of the projectors causes the image to appear with a constant brightness throughout. This is, of course, limited by our ability to radiometrically calibrate the projectors.

The pipe threads themselves contain very little actual logic; they simply respond to commands sent to them by outside sources. In addition to making OpenGL calls on behalf of client applications, the pipe threads must also be told when to clear the display, when to execute a buffer swap, when to draw the feathering polygons, and which client application to listen to for OpenGL commands. These controlling operations are sent to the pipe thread by a single master thread that oversees the entire display system.

4.3 The Master Thread

The main controller thread in the system is responsible for tasks such as layer management and dispatching events. Layer management is achieved by maintaining a database of layer positions, sizes, owners, associated display lists, and graphics states. Any thread in the system can query any of these values so it can perform its task. For instance, when a pipe thread needs to know how to set its viewport, it queries the database maintained by the master thread for the layer's position and size.

The master thread is also responsible for noticing when a redraw is required and sending redraw events to applications. For example, say one of the applications has moved a layer's position, and the screen must be updated. If there are several layers on the system at the time of a redraw, and two of those layers have had explicit redraw event requests, the master thread must send one draw event to each of the two layers' creators. Because our display is double buffered to allow for smooth animation in a layer, it is necessary to redraw all the layers on the screen any time the display needs to be updated.

The master thread also computes the stacking order of all visible layers, and sends messages to the pipe threads instructing them where to listen for MCGL packets, and in what order to draw the layers. However, the master thread is not responsible for actually updating the layer database when, for example, a layer has moved. Rather than having the master thread listen to all connected applications for commands and processing them in a serial manner, tasks that are performed on a per-client basis are handled by a separate per-client thread.

4.4 Client Threads

Each Mural client has a separate dedicated thread inside the server. Because these clients may be running on different machines on the network, the layer management API (functions such as move layer, resize layer, request input, etc.) must have its own associated packets, which are decoded by the application's associated client thread. These packets are modeled after the packets used in the X protocol. They have a similar packet structure to those described in the *X Protocol Reference Manual*[18].

The other responsibility of the client threads is to notify the master thread that an operation which requires a redraw has taken place. Operations such as a layer move or a redisplay request will cause the entire display to be redrawn.

5 Performance and Scalability

In order to evaluate how effectively we have achieved our goals, we took several sets of detailed performance measurements. Our tests were run using a Silicon Graphics Onyx2 with eight R10000 processors and two InfiniteReality graphics pipelines. Our results demonstrate several critical aspects of our system:

- Our remote OpenGL implementation makes efficient use of network resources and performs well with the addition of multiple graphics pipes.
- Through the judicious use of display lists, applications can overcome the limitations of indirect rendering and take full advantage of the graphics hardware.

	local	100 Mbit	10 Mbit
X window (GLX)	860	110	21
1 Projector	197	170	24
4 Projectors, 2 pipes	180	153	20
4 Projectors, 1 pipe	150	130	17

Table 1: Thousands of triangles per second for various remote graphics configurations. These triangles cover approximately 5 pixels each, each with a different color. The local X window measurements are using direct rendering to write directly to the graphics hardware. In the other two cases, the X window is using GLX to render remotely over a network. The disparity between the local X window rates and all other measurements illustrates the large overhead associated with remote rendering and the need for display lists. All numbers are an average of 100 runs of the test to help eliminate discrepancies from our shared network.

- Our strategy for caching each layer's drawing commands lets us create a large number of layers while still maintaining good performance.
- Avoiding graphics context switching is a key component to effective sharing of display resources.
- Software feathering is a cheap alternative to hardware blending in eliminating seams in an overlapping display.

5.1 Remote Triangle Rate and Network Utilization

Our first test measures the total triangle rate for applications that do not use display lists. To measure this, we wrote a simple application that generates a large number of small triangles, each with a different flat color, and measured the total time it took to render these OpenGL primitives. A single triangle in this scheme requires 3 bytes of color data plus 36 bytes of geometry data. Each triangle has an average area of under 5 pixels. Table 1 shows the triangle rate measurements calculated by this application running against different configurations of the Mural, as well as running in a window in a normal workstation environment.

Because we compile the stream of graphics commands into a display list and execute that list multiple times per pipe (once per projector), the overhead of having more than one projector is usually dwarfed by the overhead of an application transmitting lots of OpenGL commands, since packing, transmitting, and unpacking that command stream is expensive.

We also measured total network utilization in our triangle rate experiment. We discovered that we spent about 5 seconds transmitting 40 megabytes over a network capable of a peak bandwidth of 100 megabits. This 5 seconds includes not only transmission time, but also overhead of decoding the TCP/IP protocol. This corresponds to a network utilization of about 64 percent. These tests were conducted on an uncontrolled shared network which was carrying unknown amounts of other traffic at the time.

With these data, we can approximate the optimal number of projectors per graphics pipe for immediate mode rendering. Assuming that we have a 100 megabit network and that it takes 36 bytes to describe a triangle, we can transmit 233,000 triangles per second over this network. Since our measurements also indicate that our graphics hardware can render 860,000 triangles per second, this means that our four projector system is right around the crossing point of network utilization versus graphics performance. If we updated our system to use gigabit networking, we would once again be graphics limited.

In practice, many of our triangles will probably be clipped outside the viewport since they will not fall on all four projectors. Also, we can only transmit this number of triangles if they all have the same color, normals, and texture coordinates. Any interesting scene will vary these parameters, and we will be able to transmit many fewer triangles than in our idealized tests. Therefore, we still have a long way to go before graphics hardware becomes the limiting factor. Although gigabit networking is now available, graphics performance is increasing much faster than networking bandwidth, and we expect that our system will continue to be network limited. This makes clear the need for the use of display lists or server-side caching of display commands.

5.2 Graphics Hardware Utilization

Our next test measures the maximum utilization of the graphics hardware using display lists. To measure this, we used the Data Explorer portion of the `viewperf`[4] performance measurement system. This test visualizes a set of particle traces through a vector flow field. The object being visualized contains about 100,000 triangles, and the visualization uses two lights and no texture mapping. More information about this particular benchmark can be found at the SPEC web site[1].

A graph of the rendering time per frame is shown in figure 3. When rendering directly to the graphics hardware in an X window, the application achieved a maximum frame time of 0.024 sec/frame, compared to 0.026 running on one projector of the Mural. Using two pipes to control two projectors, the frame time was still 0.026, clearly showing that the Mural can achieve almost the maximum frame rate if display lists are used properly.

When a layer overlaps three projectors, it must overlap two projectors on one pipe and one on the other. Its rendering time is limited by the time it takes one pipe to render to two projectors, which should be exactly the same time as it takes two pipes to render to four projectors. This relationship is clearly shown in the measurements for three and four projector layers on a two pipe system in figure 3. The fact that a layer overlapping n projectors can be rendered on two pipes as fast as a layer overlapping half as many projectors on one pipe is indicative of our efficient use of multiple graphics pipes.

5.3 Multiple Layers

In this section, we measure the refresh rate of our display as a function of the number of layers currently being displayed. These measurements were taken using two different configurations to illustrate the need for server-side display caching. In the first configuration, display commands are cached in a display list as described above. In the second configuration no caching is performed, requiring each application to retransmit its graphics data over the network on each frame. Each layer contained 100 random triangles, each having an average area of 150 pixels. Finally, a dummy layer was created which contained a single semi-transparent triangle of the same size. This dummy layer was then moved across the screen, forcing the entire display to redraw itself repeatedly. The display server and the client application which created these layers were run on the same machine. A graph of frame time measurements is shown in figure 4.

Without caching the display commands, creating 1000 simple layers brings the refresh rate of the display to one frame every two seconds, while display caching lets the display refresh over six times faster. As can be clearly seen from the graph, the disparity in refresh times grows much larger with the number of layers, so display caching becomes even more important as the number of layers increases.

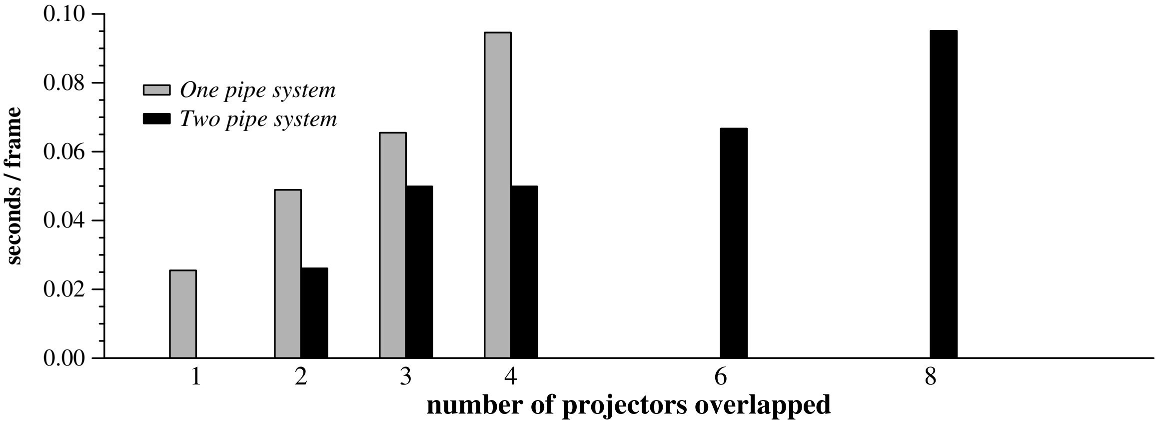

Figure 3: Frame times for the Data Explorer `viewperf` benchmark. The linear scalability of our system with respect to multiple projectors can clearly be seen. In addition, the results for the two pipe system on n projectors line up with the results for the one pipe system on half as many projectors (rounded up), demonstrating efficient utilization of multiple graphics accelerators.

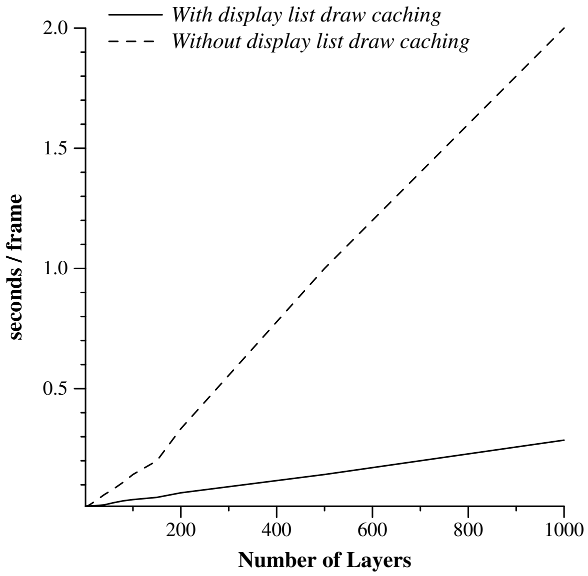

Figure 4: Frame time measurements for multiple layers on a constantly redrawing display. As the number of layers increases, caching the display commands becomes more and more important for maintaining interactivity. We have not yet discovered the cause of the small kink at 150 layers.

5.4 Context Switching

By limiting context switching overhead we are able to achieve an over 40-fold increase in performance. By creating two applications that tried to bind and unbind from a single context as fast as possible (without drawing anything at all), we determined that we could bind to the context almost exactly 100 times per second (i.e., each of our two applications was bound to the context 50 times in a second) on an InfiniteReality. With 1000 layers, the test from the last section would have a 10 second context switching overhead per frame. This is clearly not acceptable for an interactive system, and highlights the need for our OpenGL wire protocol to work around this graphics system limitation.

5.5 Cost of Feathering

Finally, we have measured the cost of our software approach for feathering the seams between overlapping projectors. The actual cost of feathering is simply that of drawing at most four textured polygons per projector. These polygons are big, however, and the fill rate of the graphics hardware should be the limiting factor. We found that we spent an average of 127 microseconds drawing the feathering polygons, which is less than 0.5% of the rendering time when running a rendering limited application such as the `viewperf` benchmark test. This is an acceptable penalty compared to the expense of custom hardware blending solutions, and will become even more attractive as hardware graphics performance increases in the future.

6 Discussion and Future Work

The system described in this paper is the third generation of our tiled display design. The evolution of our system has mainly been motivated by scalability concerns and other practical issues that we discovered along the way.

Our first implementation took an image based approach to the problem. Applications rendered into off-screen buffers, and the Mural server simply drew the images as textured rectangles representing layers. This approach was very simple to implement, and worked well, but suffered from two major drawbacks. First, hardware accelerated off-screen buffers (called “pbuffers”) are an extremely scarce resource. Once the system ran out of pbuffers, we had to fall back on non-hardware accelerated `GLXPixmaps`, which are both slow and not guaranteed to render the exact same pixels as their pbuffer equivalents. Second, we needed to switch the graphics contexts for the two pipes between the applications and the Mural server, a very expensive operation. This approach did have the nice property that the Mural server had a copy of each layer's bitmap, so redrawing layers that were not animating was very cheap.

Our next implementation allowed applications to draw directly to the screen. Each application would bind directly to the context controlling the projectors and use direct rendering to draw itself. This approach was a little more complex to implement, but it removed the reliance on pbuffers, which meant that all layers could use hardware accelerated rendering. However, the context switching overhead, while reduced in this implementation, still crippled the responsiveness of the Mural with only a few applications running. In this implementation, there was also no way for the Mural server to re-render a layer that was not animating; that burden was shifted back to the applications, which saturated our network very quickly.

Finally, we arrived at the solution described in this paper. Although the remote rendering overhead can be a limitation, we have overcome the more serious problem of context switching overhead. The main drawback of the system we have described is that the scalability is limited by the number of graphics pipes that can be attached to one computer.

In order to continue to scale our system to larger displays, we have recently extended this system to support a network of workstations, with one workstation for each projector. In this configuration, the "pipe threads" in figure 2 become separate "pipe servers" that can communicate with the "master thread" over a network. The new version of the system uses eight PC's driving eight projectors and a ninth computer running the Mural server, all tied together on a dedicated high speed network. Because of the excellent scalability of broadcast networking on a local area network[25], such a system should be cost-effective and scale well.

To address the remote rendering overhead, we will design a plug-in architecture for the Mural server, so that speed critical rendering routines can be built into the server and render directly to the screen. On the SGI version of the system, these plugins can keep their scenes and other data consistent using a simple shared memory model, but this is a much more challenging task on the PC version. We will be investigating methods for simplifying data management and consistency in distributed remote rendering. Speed critical applications (such as applications using real time head tracking) may need to manage a large scene database and properly handle updates to that database in a consistent, fast, and scalable way.

The system described in this paper has been in use for over a year, and many applications have been built using it. Some applications have been designed mainly to exercise and improve the graphics library, such as a Quicktime VR[12] viewer and a volume rendering application. The more ambitious projects, however, are more focused on new interaction techniques and new ways of using the large display space. For example, we have prototyped a construction planning scenario where our display is shared by contractors and building planners to effectively visualize and plan a change in schedule. We are also developing higher level toolkits to ease the creation of applications that use multiple large displays as well as a variety of input devices and techniques. The main direction of our group's research is towards new interaction paradigms for large displays and smart spaces.

Acknowledgments

Thanks to Terry Winograd for much of the overall vision of the interactive workspaces project at Stanford; Terry also suggested that we use layers instead of windows. François Guimbretière designed and led the physical construction of the Mural. Richard Salvador and François also developed the calibration tools for controlling the feathering of the overlapping projectors. Thanks to Diane Tang for her work on input and networking issues, Tamara Munzner for her work on input and application development, and Mary Baker and Brad Johanson for their help with networking and general distributed computing issues. Thanks also to Cindy Chen and James Davis for building the laser pointer tracking system, and to Richard for integrating it into the system. Kekoa Proudfoot provided valuable datasets and advice on performance analysis, and Henry Berg and Karen Butler put a lot of hard work into the construction of the Mural. Sandy Napel kindly provided the radiology concept image. This work was supported in part by grants from Intel, Interval, and DARPA.

References

[1] Data Explorer Viewset Description. http://www.spec.org/gpc/opc.static/dx.html.

[2] Immersive Interactive System. http://www.cs.princeton.edu/omnimedia/index.html.

[3] Panoram Technologies' Integrator 3. http://www.panoramtech.com/integrator3.htm.

[4] The OpenGL Performance Characterization Project. http://www.spec.org/gpc/opc.static/.

[5] DEXON Systems Ltd. http://www.dexonsystems.com/jdxvir.html.

[6] GLX Specification: OpenGL Graphics with the X Window System. http://toolbox.sgi.com/TasteOfDT/documents/OpenGL/.

[7] PowerWall. http://www.lcse.umn.edu/research/powerwall/powerwall.html.

[8] Virtual Network Computing. http://www.uk.research.att.com/vnc/.

[9] X11R6.4 Release Notes. http://www.x.org/r6.4doc/relnotes/relnotes.htm.

[10] XmX. http://www.cs.brown.edu/software/xmx/.

[11] E. Bier, M. Stone, K. Pier, W. Buxton, and T. DeRose. Toolglass and Magic Lenses: The See-Through Interface. In *Computer Graphics (SIGGRAPH 93 Proceedings)*, 1993.

[12] S. E. Chen. QuickTime VR — An Image-Based Approach to Virtual Environment Navigation. In *Computer Graphics (SIGGRAPH 95 Proceedings)*, 1995.

[13] C. Cruz-Neira, D. Sandin, and T. DeFanti. Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE. In *Computer Graphics (SIGGRAPH 93 Proceedings)*, 1993.

[14] M. Czernuszenko, D. Pape, D. Sandin, T. DeFanti, G. Dawe, and M. Brown. The ImmersaDesk and InfinityWall Projection-Based Virtual Reality Displays. In *Computer Graphics*, May 1997.

[15] S. Elrod, R. Bruce, R. Gold, D. Goldberg, F. Halasz, W. Janssen, D. Lee, K. McCall, E. Pedersen, K. Pier, J. Tang, and B. Welch. Liveboard: A Large Interactive Display Supporting Group Meetings, Presentations and Remote Collaboration. In *Proceedings of the Conference on Computer Human Interaction (CHI)*, May 1992.

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

[17] J. Montrym, D. Baum, D. Dignam, and C. Migdal. InfiniteReality: A Real-Time Graphics System. In *Computer Graphics (SIGGRAPH 97 Proceedings)*, 1997.

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

[19] R. Raskar, G. Welch, M. Cutts, A. Lake, L. Stesin, and H. Fuchs. The Office of the Future: A Unified Approach to Image-Based Modeling and Spatially Immersive Displays. In *Computer Graphics (SIGGRAPH 98 Proceedings)*, 1998.

[20] R. Raskar, G. Welch, and H. Fuchs. Seamless Projection Overlaps using Image Warping and Intensity Blending. In *Proceedings of the Fourth International Conference on Virtual Systems and Multimedia*, 1998.

[21] S. Rehel. The GLTT Graphics Library. http://services.worldnet.net/˜rehel/gltt/gltt.html.

[22] J. Rohlf and J. Helman. IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics. In *Computer Graphics (SIGGRAPH 94 Proceedings)*, 1994.

[23] D. Rosenthal. Inter-Client Communication Conventions Manual. Web version by Christophe Tronche: http://tronche.com/gui/x/icccm/.

[24] R. Surati. *Scalable Self-Calibrating Display Technology for Seamless Large-Scale Displays*. PhD thesis, Massachusetts Institute of Technology, 1999.

[25] A. Tannenbaum. *Computer Networks*. Prentice Hall, 1996.

[26] J. Torborg and J. Kajiya. Talisman: Commodity Realtime 3D Graphics for the PC. In *Computer Graphics (SIGGRAPH 96 Proceedings)*, 1996.