Data Distribution Strategies for High-Resolution Displays

Han Chen, Yuqun Chen, Adam Finkelstein,
Thomas Funkhouser, Kai Li, Zhiyan Liu,
Rudrajit Samanta, and Grant Wallace

Princeton University

Abstract

Large-scale and high-resolution displays are increasingly being used for next-generation interactive 3D graphics applications, including large-scale data visualization, immersive virtual environments, and collaborative design. These systems must include a very high-performance and scalable 3D rendering subsystem in order to generate high-resolution images at real-time frame rates.

We are investigating how to build such a system using only inexpensive commodity components in a PC cluster. The main challenge is to develop scalable algorithms to partition and distribute rendering tasks effectively under the bandwidth, processing, and storage constraints of a distributed system. In this paper, we compare three different approaches that differ in the type of data transmitted from client to display servers: control, primitives, or pixels. For each approach, we describe our initial experiments with a working prototype system driving a multi-projector display wall with a PC cluster. We find that different approaches are suitable for different system architectures, with the best choice depending on the communication bandwidth, storage capacity, and processing power of the clients and display servers.

1 Introduction

We are moving into a new era of computing in which interactions with data are available across time and space with *ubiquitous display devices*. The reasons for this era are simple: rapid improvements in CPU performance, storage density, and network bandwidth, and the commoditization of display devices. The traditional way of using computers and networks has been to spend most of the CPU cycles on solving technical problems and manage business transactions. Nowadays, more and more CPU cycles and network bandwidth are spent on interactions including creating sophisticated content, transforming information, and presenting information for humans to browse and visualize.

Preprint submitted to Elsevier Science

Within the next decade, new generation displays, such as Light Emitting Plastics (LEP) and Organic Light Emitting Devices (OLED), will become commercially viable commodity components. These devices will be very inexpensive and they will be attached to walls, windows, furniture and arbitrarily shaped objects. They might literally cover the surfaces of an entire room, a floor, or even an entire building. Their deployment will introduce an interesting new technical problem: "how should we architect computer systems to drive so many pixels?"

Consider the case of a building covered by "digital wallpaper." A typical building has approximately 100 million square inches of surface area on the walls, ceilings, and floors, which could accommodate around 500 gigapixels of display imagery (at 72 dpi). If each image is updated 30 times per second, then the display system must be capable of updating 15 terapixels per second, or 500,000 times more pixels per second than current display devices. Fortunately, in most reasonable scenarios, not all the displays need to be updated at full resolution, or at video frame rates, all at once. Instead, a few displays showing feedback to user interaction will require fast refresh rates, while most other displays can be updated less frequently and/or at lower resolution. Meanwhile, the displays not visible to any user can be left blank. The challenge is to design a rendering system powerful enough and flexible enough to drive a large number of pixels spread over multiple displays in a dynamic environment.

The traditional approach to high-performance and high-resolution rendering is to use a large, tightly-integrated graphics computer connected to multiple video outputs. The Power Wall at the University of Minnesota and the Infinite Wall at the University of Illinois at Chicago are examples, each driven by an SGI Onyx2 with multiple InfiniteReality graphics pipelines. The main drawback of this approach is that it is very expensive, sometimes costing millions of dollars.

In the Scalable Display Wall Project at Princeton University [10], we take a different approach. Rather than relying upon a tightly integrated graphics subsystem, we combine multiple commodity processors connected by a network to construct a parallel rendering system capable of driving multiple displays with scalable rendering performance and resolution. The main theme of this approach is that inexpensive and high-performance systems can be built using a multiplicity of commodity parts. Our current system, shown in Figure 1, comprises 24 projectors arranged in a 6 by 4 grid to form a seamless image over an 18' by 8.5' rear projection screen. The resolution of each projector is 1024 by 768, and thus the cumulative resolution of the entire wall is 6K by 3K (18M pixels). Each projector is driven by a commodity PC connected to a network (Ethernet and/or Myrinet [3]). The system also contains PCs for tracking user input, synthesizing sounds for 14 speakers, and running applications. The total cost of the system is around $200K.

As compared to traditional rendering systems, our architecture has several advantages. First, we use commodity hardware components, and thus the system is less

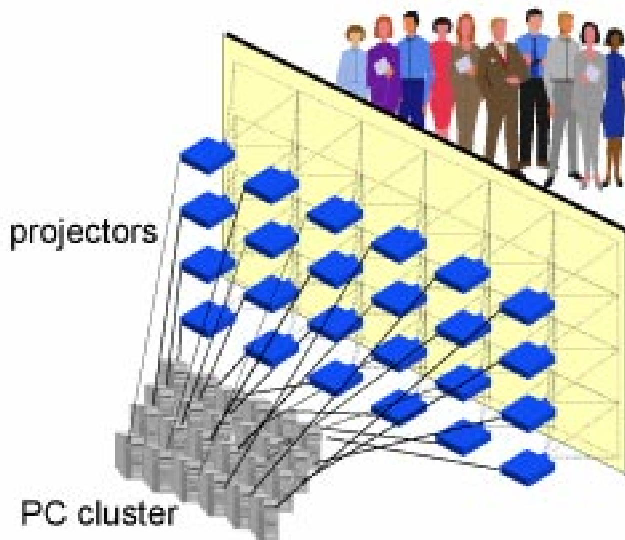


Fig. 1. Princeton Scalable Display Wall, shown schematically above.

expensive, more flexible, and tracks technology better than other systems with custom hardware. Second, we use a network for communication among processors, which provides modularity, flexibility, and scalability. For instance, heterogeneous processors and display devices can be added to the system independently, while the communication capacity of the system scales at the rate of commodity networking improvements. Finally, the images displayed are composed of multiple sub-images corresponding to frame buffers of different computers. This characteristic provides a natural image-parallel decomposition of the rendering computation, enabling very high-resolution displays. The primary challenge is to construct rendering strategies that work effectively within the processing, storage, and bandwidth characteristics of commodity components.

In this paper, we investigate research issues in using multiple commodity com-

ponents to construct a high-performance system driving multiple networked display devices. The goal of our study is to characterize the processing, storage, and communication requirements of several possible system architectures. We consider options ranging from a network of "smart displays" (where each display device has considerable local processing, storage, and communication capabilities) to a network of servers attached to remote "dumb displays" (where each display has only a small buffer and a low-power processor). In the former case, processing is distributed, and the main research issue is synchronization. In the latter case, processing is centralized, and the research issue is efficient distribution of data.

2 Taxonomy of Data Distribution Strategies

Perhaps the most critical systems problem for ubiquitous, heterogeneous, high-resolution display environments will be the transmission problem: how do we get the bits to the pixels? A simple taxonomy of solutions to this problem segments various approaches with respect to what form of data is transmitted between components of the system: control, primitives, or pixels. We are investigating these approaches in the context of the Display Wall Project, and have implemented a number of applications in each category, as described in Sections 3–5. But first, this section provides a brief overview.

Conventional applications running on a single desktop workstation generally behave as follows. The user controls the application through some form of user interface (e.g. the mouse). Control events from the user interface cause the application to generate new display primitives (e.g. new text appears in some window, or triangles representing a new view of a 3D object). The graphics subsystem rasterizes these primitives into a frame buffer. Finally, the frame buffer appears on the display.

For such an application to run across multiple networked commodity components, we must consider how to distribute data among the components and how to synchronize execution. One possible programming model is client-server. For instance, each user could interact with a client while servers manage the displays. In this scenario, there are three stages at which the client-server communication may occur (as shown in Figure 2), leading to three different forms of data being transmitted over a network:

- **Control (synchronized execution):** A copy of the application runs on each display server. The client handles user-interface events, and sends control information (for example synchronization events, or changes in view) to each display server. In this model, the network requirements are typically minimal. However the storage and processing demands at the server are significant because the bulk of the application as well as the graphics rendering pipeline runs there.
- **Primitives:** In this model, the user interface and application live on the client

side. The application sends 2D or 3D graphics primitives over the network to the display server, which rasterizes and displays them. In this scheme, network requirements depend on scene complexity. At the client side, the processing and storage demands are those of a conventional application. At *each* server machine, the rendering requirement is that of the full scene, unless some form of view-dependent culling is performed. Ideally, a load balancing scheme would distribute the rendering load at each server machine.

- **Pixels:** Here the application and full rendering pipeline run at the client side. The client ships pixels (typically compressed as JPEG images or MPEG streams) to the server, and thus the network demand is proportional to display resolution. At the display, the server simply decodes the pixels and thus requires little storage and processing power. On the other side, if a single client workstation runs an application that drives the entire high-resolution display, the rendering demands on the client are huge for dynamic applications. However, if many such applications each cover a portion of the wall, the client load is manageable.

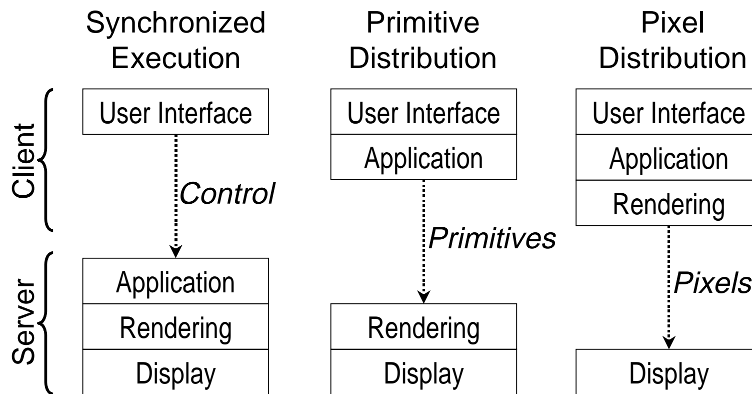


Fig. 2. Network transmits data at three different stages in the rendering pipeline.

In the following sections, we describe our initial experiments with these three approaches. Our experiments are based on a working prototype system driving a multi-projector display wall. We expect that different approaches are suitable for different system architectures, with the best choice depending on the communication bandwidth, storage capacity, and processing power of the clients and display servers. Our goal is to characterize the processing, storage, and communication requirements of each approach leading to an understanding of how to construct efficient and scalable display systems.

3 Synchronized Execution Model

One method of running an application on a tiled display wall is to use a synchronized program execution model. In this model, a duplicate instance of the application is run on each server. The only difference between the instances being run

comes from the environment information, such as which server in the tiled display they represent. The motivation for this model is to minimize communication over the network. Only control messages need to be sent over the network. These are messages such as synchronization events and user input, and tend to require little bandwidth.

In the synchronization model a synchronization boundary is established such that within this boundary all the instances assume identical behavior. We've experimented with having this boundary at both the system level and at the application level. With the synchronization boundary at the system level, each server produces identical graphics primitives and the graphics accelerator is used to do tile specific culling such that only the primitives that fall within a server's screen area are rendered (see figure 3). This technique is especially useful if source code for the application is not available. If the synchronization boundary is moved to the application level, more optimizations are possible. A view-dependent software layer can restrict itself to generating tile specific primitives (rather than generating all the primitives for the scene). An example of the second scenario is a scene graph render program that organizes the scene data in a hierarchy of objects. Given a tile-specific view frustum, the program can remove the objects that fall completely outside the frustum. Such an approach can improve processing performance as well as data transfer performance.

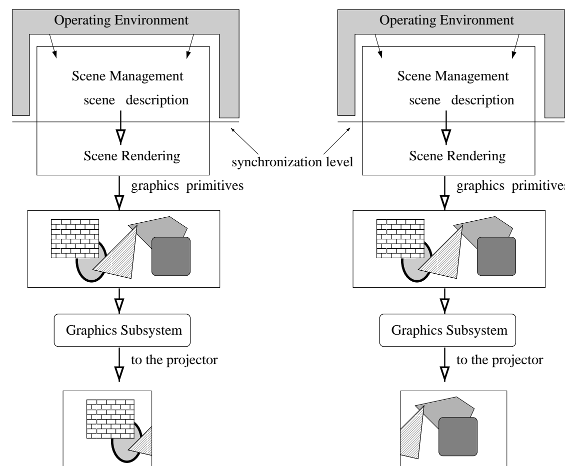


Fig. 3. Synchronized application execution with tile specific clipping

We've implemented a synchronization framework for both the system and application level [5]. The framework consists of barrier synchronizations that are set up on certain function calls. One server acts as a coordinator and broadcasts the result of the function call to all other servers. At the system level we use DLL replacement to intercept and synchronize on frame buffer swaps, timer calls and I/O operations. At the application level we've implemented a simple API including calls such as SynchronizeResult() which performs a barrier synchronization and distribution of results.

Using the synchronization framework, we've collected performance information on several applications including Cars (from WorldUp Toolkit, figure 4), Atlantis (Silicon Graphics demo), and Isoview (isosurface visualization tool). Our results show that the synchronization communication overhead is small, less than 500 bytes per frame. For immediate mode 3D graphics applications we've achieved speedups of between 1.2 and 4.2 as compared to a client-server approach (such as primitive distribution, see section 4 table 1). The amount of speedup is affected by the number of primitives and the per-primitive computation time. In general the synchronization model performs well when there are many primitives and the per-primitive computation time is low. When per primitive computation time is high, there is often little or no speedup because a client server approach can overlap network communication with computation. Our experiments have also shown that application level synchronization perform significantly better than system level synchronization. In the Atlantis application we were able to get an overhead reduction of 50% in the computation of shark poses and graphics primitives.

Fig. 4. Cars application running in synchronized execution mode

In general we've found that synchronized execution can be an efficient model for programming a tiled display. This is especially true when the computational abilities of the render servers (including processor and memory) outpace the available network bandwidth. This is often the case today when utilizing commodity PCs and networking components.

4 Primitive Distribution Model

A second option for distributing graphics applications is to distribute the graphics primitives over the network. In such a scenario, one or more clients run the applications which perform all the user interface and processing tasks. Then 2D or 3D primitives are sent over the network to the displays. The servers are required to be able to decode and render this stream of 2D or 3D primitives. This model

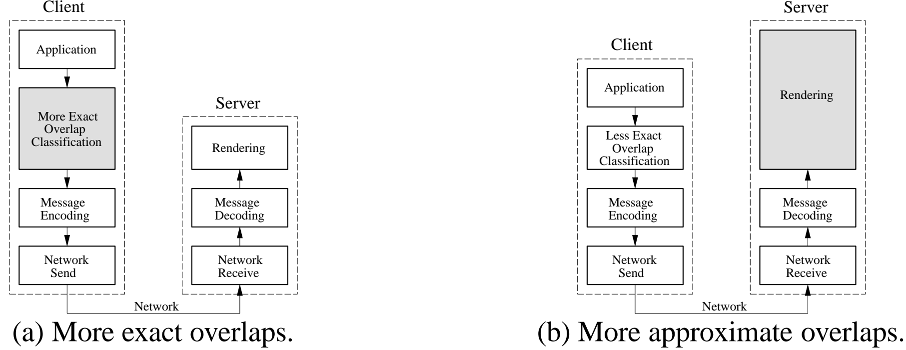
allows the application to run in a single location, avoiding redundant computation and distributing only the graphics rendering workload.

In order to make programming using this model easier, we can implement techniques that allow programs designed for a single desktop to run unmodified. Two approaches have been implemented to test this model : a *Virtual Display Driver* (VDD) for 2D primitives and a *Distributed OpenGL* (DGL) layer for 3D primitives. The VDD is implemented as a device driver for Windows 2000. It reports to the operating system the existence of a large (virtual) display. Standard Windows applications may then run and display to this device. As Windows requests the display driver to render 2D primitives, the VDD records these requests and broadcasts them to the servers. The DGL layer works in a similar way but is implemented as a dynamically linked library that replaces the original OpenGL library. In both cases the servers run a program that receives and decodes the incoming stream of 2D (in the VDD case) or 3D (in the DGL case) commands and passes them to the local graphics hardware.

Previous approaches have been implemented for both 2D and 3D cases, but they have generally been aimed at the single display, multiple clients case. One of the most commonly used 2D systems is the X-Window protocol [12]. In order to transfer 3D primitives as well, the GLS [7] was introduced. The WireGL system from Stanford [4] is another protocol for distributing OpenGL calls to a network of render servers. In WireGL, several optimizations are introduced to reduce the number of state changes required to be transmitted over the network while keeping the current state on each of the servers accurate and up to date.

In order to reduce both bandwidth and rendering overheads, we can choose to cull primitives by their screen space extent. In the interception layers, we can check which regions of the tiled display a particular primitive overlaps and then send network messages only to those servers instead of broadcasting the graphics commands. In order to do this, the client must transform the stream of primitives to screen space coordinates and then check against the tiles assigned to each server. Considering the current rendering rates of commodity graphics accelerators, a single client cannot keep up with a cluster of servers. In order to make the client sort primitives interactively, we perform amortized transformations by grouping a number of consecutive primitives in the sequence. If we group too few primitives, the client will be the bottleneck. On the other hand if we group too many, the servers will be required to do a large amount of redundant work due to the high primitive-tile overlap. These two situations are shown schematically in Figure 5. The ideal point in this tradeoff depends on a number of factors such as the primitive size and client speed. Our system is able to dynamically choose an ammortization factor that tries to create the best balance possible between client overhead and server rendering overhead.

To evaluate the performance of the DGL layer we have tested a number of appli-



(a) More exact overlaps. (b) More approximate overlaps.

Fig. 5. More approximate overlap classifications shift the processing burden from the client to the servers.

cation binaries compared to the System Synchronization and Application Synchronization setups (Figure 6 shows the Isoview Application). The times, bandwidth and memory requirements are shown in Table 1. As can be seen, the data transferred over the network and frame times are generally worse in the DGL case. However, since the application is not run at every server, it saves processing time and has significantly less memory requirements on the servers for most cases.

Applcation	Parameter	Dist GL	System Sync	App Sync
Cars	Polygons (K)	200	200	-
	Data size (KB)	99.2	0.47	-
	Frame time (ms)	370	325	-
	Memory (MB)	4	73	-
Atlantis	Polygons (K)	94.6	94.6	94.6
	Data size (KB)	3300	0.24	0.19
	Frame time (ms)	147.6	86.2	57.9
	Memory (MB)	10	6	6
Isoview	Polygons (K)	645.2	645.2	645.2
	Data size (KB)	46500	0.82	0.47
	Frame time (ms)	20163	16825	4798
	Memory (MB)	4	100	100

Table 1
Comparison of bandwidth, memory requirements and frame times for applications using Distributed GL, System-level synchronization and Application-level synchronization. Information shown is per frame, averaged over a number of frames.

Primitive based approaches are useful in a number of situations. First, these approaches are useful for applications which require a significant amount of processing time which cannot easily be parallelized. Second, these approaches allow us to run application binaries without modifications. Also, this scheme lets us have fairly simple display servers (they need to only decode a command stream and render 2D or 3D primitives) which is useful for making the display system cost-effective and easy to maintain. While bandwidth requirements are higher than the case where the application is run in parallel among the servers, distributing primitives helps to

Fig. 6. Isoview Displaying Visible Woman Dataset Using DGL Layer

avoid redundant computations by the application.

5 Pixel Distribution Model

The third option is to send pixels to the displays. The main motivations for this approach are its generality and its simplicity. Since each display requires only a image/video decoder, its processing and storage needs are simplified (as compared to the cases where control or primitive data are sent to the display), and the communication requirements are more predictable (they are related to the resolution of the display rather than the complexity of the underlying primitive data).

The challenge is to distribute and decode the large bandwidth of pixel data required for high-resolution displays. In this section, we describe the results of our investigation with a parallel MPEG decoder using commodity PC components. The difficult question we address is how to partition and distribute data among the PCs. MPEG-2 video stream defines a hierarchy of syntactical elements – *Sequence*, *Group of Picture* (GOP), *Picture*, *Slice*, and *Macroblock* [8], and any of them could be chosen as the unit of work for parallelization. Unfortunately, none of them suffices in itself for decoding ultra-high-resolution video streams in a PC cluster based tiled display system. In a coarse granularity parallelization [2,9] (e.g., sequence, GOP, picture, or slice), the splitting cost is very low, but the communication cost is high, because decoding servers need to communicate with each other for both referencing previous frames and redistributing decoded pixels. In a fine granularity parallelism (e.g., macroblock level), the communication cost is low and distributed, but the splitting cost is high, because MPEG-2 video stream does not provide byte-aligned start code for macroblocks. The splitter becomes a bottleneck when the number of servers increases. There are also ways to parallelize a decoder functionally [1], however, these methods only work well for a shared memory SMP, and do not scale.

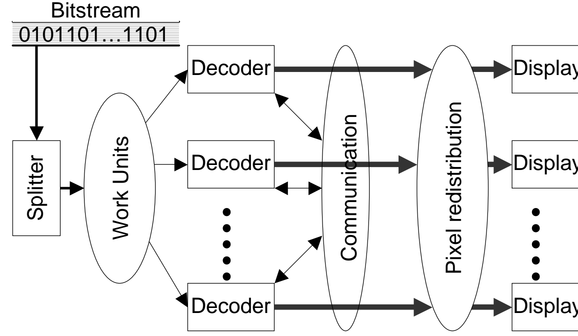


Fig. 7. A Generalized Parallel Decoder for PC Cluster

We use a hierarchical decoding system to achieve scalable high resolution decoding. It consists of one or two levels of splitters and a set of decoding servers(see Figure 7). The splitters divide the input stream into macroblocks and send them to the decoders, while the decoders decompress the pixels and display them. For relatively low-resolution video streams (such as DVD or HDTV), a single macroblock-level splitter is adequate. For high-resolution videos, the system uses a root splitter to split an input video stream at picture level and pass pictures to multiple second-level splitters, which split pictures at macroblock level to feed decoding servers. We call the decoding method 1-k-(m,n) system for a hierarchy of one single root picture splitter, k second-level macroblock splitters, and $m \times n$ decoding servers in a tiled $m \times n$-projector display wall system. When only one splitter is used, we call it a 1-(m,n) system.

Results of experiments with this distributed MPEG decoder are shown in Table 5. We used 8 MPEG-2 video streams resolution ranging from 720×480 to 3,840×2,800 to test the performance and scalability of the system. Each stream contained 240 frames. (Figure 8 is a screenshot of the parallel decoder showing stream sis30.) The number of second-level splitters was chosen in order to keep the decoders running at their maximum speed. From the numbers in the rightmost column, we can see that the two-level system achieves a near linear acceleration in the rate at which pixels can be decoded, and thus it scales very well. The bandwidth requirements of each splitter and decoder are also well within the range of current commodity networks. For instance, in the experiment with the 1-4-(4,4) system decoding stream orion100, the send and receive bandwidths ranged from 4 to 5 MB/s for each splitter and from 0.6 to 4.5 MB/s for each decoder.

Overall, we observe that the bandwidth required for pixel data is far more than for control data but less than for primitive data. The per-server communication and processing requirements are almost constant regardless of the overall resolution. The overall performance of the system is able to scale as the network switching capability increases. However, the processing demand for video encoding on the client are problematic. Currently, distributing pixel data is most appropriate for pre-captured content.

Fig. 8. Parallel MPEG-2 Decoder Showing “Seen In Shadow”

Table 2
Performance of the Hierarchical Decoder

Stream Name	Screen Resolution	Config	# Nodes	Frames/ Sec	Mpixels/ Sec
Matrix	720×480	1-(1,1)	2	105.0	36.3
Fish	1,280×720	1-(2,1)	3	87.4	80.5
CBS	1,920×1,080	1-2-(2,2)	7	74.0	154
SIS30	2,000×1,500	1-2-(3,2)	9	60.2	181
Orion40	2,880×1,440	1-2-(3,2)	9	45.3	188
Orion60	2,880×2,160	1-3-(3,3)	13	43.0	268
Orion80	3,840×2,160	1-3-(4,3)	16	38.9	323
Orion100	3,840×2,800	1-4-(4,4)	21	38.9	418

6 Conclusion and Future Work

As we move to an era in which all surfaces in the work environment may potentially be high-resolution *display* media, a critical problem emerges: how can we transmit the data to those display surfaces. This paper investigates protocols for transmitting data from three levels of the graphics rendering pipeline: control, primitives, and pixels. Not surprisingly, transmission of different forms of data places different demands on the system. We hope that the observations and data present in this paper help future designers of high-resolution display systems in choosing data distribution strategies suitable for their system architectures.

This work suggests several areas for future research:

Compression: In Section 5, we describe a scheme for parallel software decom-

pression and rendering of an MPEG stream. In our experiments, the MPEG streams were all pre-compressed offline and saved to a disk. However, in order to use this display environment for remote computing, we need to also provide the accompanying dynamic compression scheme, and incorporate it into the rendering pipeline. One simple method might be to read the frame buffers of the rendering machines in real time by intercepting the DVI (digital) output from their graphics cards and compressing and transmitting this data. When the source of the pixels is 2D or 3D primitive rendering, there may be further opportunities for efficient compression, either based on compression of primitives (e.g., [6]) or based on primitive-guided pixel compression (e.g., [13]).

Load balancing: An interesting issue for future work is dynamic allocation of rendering processors to display surfaces. In this paper, we have only considered scenarios in which one rendering processor is dedicated to each display device. But, then, if the rendering load is not uniformly distributed over all display devices (e.g., all the primitives reside in one small region of one wall), or if we have more graphics processors available than there are display devices, this simple, static allocation does not achieve optimal performance. In related work, we have developed dynamic load balancing algorithms for PC clusters driving a display wall [11]. However, further work is required to develop effective load balancing methods for remote applications.

Other data paths: In this paper, we focus on transmission of data from an application to a display. However, other data paths face similar challenges. For example, for applications whose data sets are larger than fit in memory, we must consider the path from disk to application memory. This work is likely to involve parallel disk farms, caching schemes, and data migration. As another example, we could consider paths in which video data tracking user input is transmitted from the cameras to the application (e.g., for teleconferencing). This data path replaces the computer graphics pipeline with the computer vision pipeline, yet faces many similar challenges.

References

[1] A. Bala, D. Shah, U. Feng, and D. K. Panda. Experience with software MPEG-2 video decompression on an smp pc. In *Proceedings of the 1998 ICPP Workshop on Architectural and OS Support for Multimedia Applications/Flexible Communication Systems/Wireless Networks and Mobile Computing*, pages 29–36, 1998.

[2] A. Bilas, J. Fritts, and J. P. Singh. Real-time parallel MPEG-2 decoding in software. In *Proceedings of the 11th International Parallel Processing Symposium*, Apr 1997.

[3] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local area network. *IEEE MICRO*, 15(1):29–36, February 1995.

[4] Ian Buck, Greg Humphreys, and Pat Hanrahan. Tracking graphics state for networked rendering. In *Eurographics/SIGGRAPH workshop on Graphics hardware*, pages 87–98, August 2000.

[5] Yuqun Chen, Han Chen, Douglas W. Clark, Zhiyan Liu, Grant Wallace, and Kai Li. Software environments for cluster-based display systems. In *First IEEE/ACM International Symposium on Cluster Computing and the Grid*, May 2001.

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

[7] C. Dunwoody. The OpenGL stream codec : A specification.

[8] S. Eckart and C. E. Fogg. ISO/IEC MPEG-2 software video codec. In *Proc. Digital Video Compression: Algorithms and Technologies 1995*, pages 100–109. SPIE, 1995.

[9] M. K. Kwong, P. T. Peter Tang, and B. Lin. A real time MPEG software decoder using a portable message-passing library. *Mathematics and Computer Science Division ANL, Preprint MCS-P506-0395*, Apr 1995.

[10] Kai Li, Han Chen, Yuqun Chen, Douglas W. Clark, Perry Cook, Stefanos Damianakis, Georg Essl, Adam Finkelstein, Thomas Funkhouser, Timothy Housel, Allison Klein, Zhiyan Liu, Emil Praun, Rudrajit Samanta, Ben Shedd, Jaswinder Pal Singh, George Tzanetakis, and Jiannan Zheng. Early experiences and challenges in building and using a scalable display wall system. *IEEE Computer Graphics and Applications*, pages 29–37, July 2000.

[11] Rudrajit Samanta, Jiannan Zheng, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh. Load balancing for multi-projector rendering systems. In *SIGGRAPH '99. Proceedings 1999 Eurographics/SIGGRAPH workshop on Graphics hardware, Aug. 8–9, 1999, Los Angeles, CA*, pages 107–116. ACM Press, 1999.

[12] Robet W. Scheifler and James Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[13] Dan S. Wallach, Sharma Kunapalli, and Michael F. Cohen. Accelerated MPEG compression of dynamic polygonal scenes. *Proceedings of SIGGRAPH 94*, pages 193–197, July 1994.