Design Considerations for a Multi-Projector Display Rendering Cluster

David Gotz

Department of Computer Science
University of North Carolina at Chapel Hill

Abstract

High-resolution, multi-projector displays are often built using standard consumer electronics. Before designing a PC-cluster rendering system to drive these displays, a number of issues must be addressed. This paper provides an overview of these issues. The issues addressed include the type of parallel rendering algorithm, load balancing, network latency, and network overhead. The goal of this paper is to provide a general overview of the design space and highlight the major tradeoffs between different designs.

1 Introduction

Over the last few years, computers have become faster and more powerful by almost all measurements. Computer processor speeds have increased, memory has become cheaper, and hard drive sizes have grown. However, there is one computer component that has remained largely unchanged. Displays, such as standard computer monitors, have remained at a resolution of around one million pixels.

To address these concerns, many research groups have been building multi-projector systems that combine readily available consumer products into single, unified displays that render at far higher resolution than traditional computer monitors [4, 12, 6, 5, 2, 3]. An important part of any multi-projector display is the underlying rendering system. Some systems use large supercomputers while other systems use a distributed computer cluster. Due to the large monetary cost associated with a powerful supercomputer, cheaper PC-based computer clusters are a more affordable option.

This paper will address some of the important design considerations associated with PC-cluster rendering for multi-projector display systems. Three issues will be discussed in detail:

- **Sorting in the Rendering Pipeline** : Sorting is an important part of any parallel rendering system. This paper surveys various sorting methods and discusses their applicability to a multi-projector display cluster rendering system.
- **Load Balancing** : Load balancing algorithms are designed to optimize the performance of parallel systems by distributing work as evenly as possible across each node in the system. This paper discusses strategies for load balancing within the rendering framework.
- **Network Performance** : We analyze the impact of various design decisions on network performance. We utilize the LogGP network model to highlight specific network performance concerns.

The rest of this paper is organized as follows: Section 2 provides some background knowledge on multi-projector displays, sorting classifications, load balancing algorithms, and the LogGP network model. Section 3 presents a rendering algorithm for multi-projector displays that is used as a basis for discussion. Section 4 discusses how sorting, load balancing, and network performance apply to cluster rendering systems for multi-projector displays. Section 5 provides a few design suggestions based on the discussion in Section 4. Section 6 addresses the issue of system scalability, and is followed by a brief conclusion.

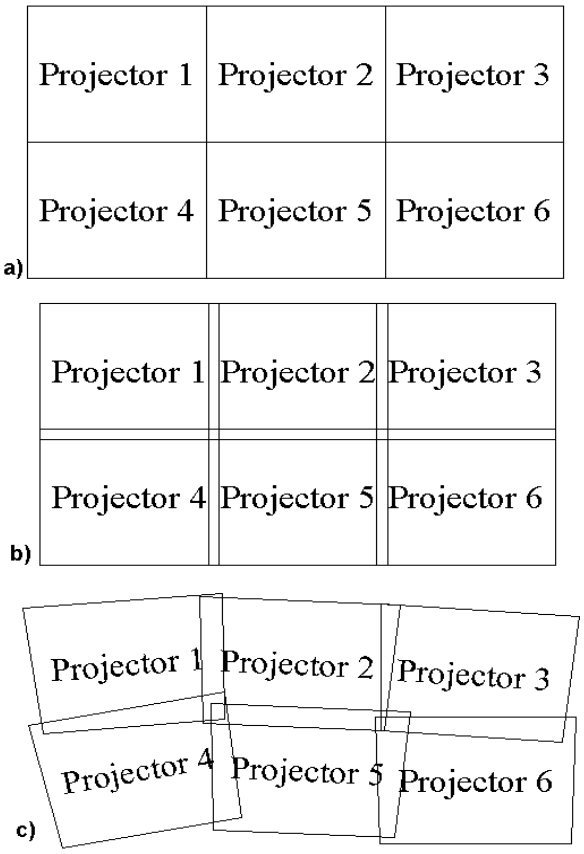

Figure 1: The three display classes: a) Abutted, b) Regular Overlap, c) Rough Overlap.

2 Background

2.1 Multi-Projector Displays

As mentioned in the introduction, a number of research groups have investigated multi-projector displays. All of these displays create a single high-resolution display by combining a collection of lower resolution projectors. Most of these displays fall into one of three broad classes: 1) *abutted*, 2) *regular overlap*, and 3) *rough overlap*.

2.1.1 Abutted Displays

Abutted displays are the first class of multi-projector displays. Abutted displays require that all projectors in the display be carefully aligned so that no pixels overlap. Abutted systems are fairly common and are used in everything from sports stadium scoreboards to trade show exhibits. Some examples of abutted displays are the CAVE [2], Office of Real Soon Now" [1], and the display wall system at Lawrence Livermore National Laboratory [12]. Figure 1(a) depicts an abutted display.

2.1.2 Regular Overlap Displays

A second class of multi-projector displays requires projectors to be carefully aligned so that there is some controlled overlap between projectors. The projectors are required to have precise geometric relationships that ensure regularity between overlap regions. The overlap regions are used to blend imagery across projector boundaries. This is done to help hide both photometric and geometric discontinuities at the boundaries. Princeton's Scalable Display Wall [6] and Stanford's Interactive Mural [5] are both examples of regular overlap displays. Figure 1(b) depicts a display configuration with regular overlap.

2.1.3 Rough Overlap Displays

The third class of display systems is the most complex because it allows rough overlap regions between projectors. The only requirement is that projectors actually overlap. This means that overlap regions can be of arbitrary shape and size. UNC's PixelFlex system is an example of this type of display [3]. Figure 1(c) shows a display with rough overlap regions.

2.2 Sorting Classification

An inherent step in parallel rendering algorithms is sorting the data and assigning it to individual processors. The same is true for distributed rendering. Once sorted, the data can be intelligently distributed to individual machines for processing. There are three broad classes of parallel rendering algorithms, each performing the sort at a different stage in the rendering pipeline. These three classes: 1) *sort-first*, 2) *sort-middle*, and 3) *sort-last* [8].

2.2.1 Sort-First

Sort-first algorithms are designed to distribute *world-space* primitives as early in the pipeline as possible. Before starting, regions of screen space are assigned to each processor. In addition, each primitive is assigned to a processor in some arbitrary manner. Then, during rendering, the processors perform the minimal amount of work that determines the screen-space location of the primitive. Typically, the algorithm will compute the screen-space projection of the primitive's bounding box. This operation is known as *pre-transformation*. Once the pre-transformation is applied, the primitive is distributed to the appropriate processor or processors.

2.2.2 Sort-Middle

While sort-first algorithms distribute world-space primitives, sort-middle algorithms are designed to distribute *screen-space* primitives. Regions of screen space are assigned to individual processors in the same manner as sort-first algorithms. However, once rendering begins, sort-middle algorithms compute the actual screen-space coordinates of each primitive. This contrasts with sort-first algorithms which don't compute the actual primitive coordinates, but instead use bounding box coordinates. Once the screen-space coordinates are known, the primitives are distributed to the appropriate processor or processors. In sort-middle, the sort occurs at the natural split between geometry processing and rasterization.

2.2.3 Sort-Last

The third and final class of parallel rendering algorithms is sort-last. Unlike sort-first and sort-middle algorithms, sort-last algorithms don't distribute primitives at all. Instead, pixels are distributed following the rasterization stage. For each pixel, the computed data is sent to the appropriate processor or processors where depth values are compared for visibility determination.

2.3 Load Balancing

When distributing computational tasks across multiple processors, load imbalances between processors could be detrimental to the system's overall performance. The performance penalty is a result from inefficient allocation of the system's resources. Load balancing algorithms attempt to redistribute tasks across processors in order to achieve a more balanced load distribution. This paper will utilize the load balancing model presented by Willebeek *et al* [13]. In this model, there are four stages in the load balancing process: 1) *processor load evaluation*, 2) *profitability determination*, 3) *task migration*, and 4) *task selection*.

2.3.1 Processor Load Evaluation

The first stage in the general load balancing model is processor load evaluation. In this stage, a load value is estimated for each node. The evaluation at each node is made independently. The values that result from this stage are used as input in the subsequent load balancing stages.

2.3.2 Profitability Determination

Once load evaluation has been performed on all nodes, the system must perform load balancing profitability determination. In this stage, the system calculates an *imbalance factor*. The imbalance factor is a function of the load evaluation scores obtained in the first stage. This factor is a measure of the degree of imbalance in the system's current state. The imbalance factor is then compared to the overhead associated with correcting the imbalance. If the system determines that correcting the load imbalance will be profitable, load balancing is initiated.

2.3.3 Task Migration

Once load balancing is initiated, task migration occurs. Task migration starts by determining which overworked nodes can hand off work and which under-worked nodes can take on additional work. This is accomplished by analyzing the results from the load evaluation stage. This stage concludes when the system notifies the sources of the quantity of tasks to be migrated and the destination node for each migration.

2.3.4 Task Selection

The fourth and final stage in load balancing is task selection. Once a source has been notified how much work it should hand off and to which node it should send the tasks, the source must decide which tasks to send. In this stage, the source should intelligently decide which tasks are most appropriate for the given destination. Appropriate tasks are ones that would require the least overhead to transfer and tasks that are contextually appropriate for the destination node.

2.4 The LogGP Network Model

When discussing how various design decisions impact network performance, it is useful to rely upon a network model such as the *LogGP model* proposed by Martin *et al* [7]. The LogGP model attempts to describe a network's impact on distributed systems in an implementation-neutral manner. The model defines five major parameters that describe overall network performance:

- **Network Medium Latency,** L : the time spent sending a message from the source to the destination. L accounts only for the time spent in transit through the network. Time spent in the source or destination's processor is not included in L.
- **Network Overhead,** o : refers to the time spent by the processor in sending or receiving a message. During the time spent on o, the processor cannot engage in any other activity.
- **Gap,** g : the minimum time interval between consecutive message transmissions or receptions. If one source sends (receives) a message at time t, the source must wait until $t + g$ before sending (receiving) a second message. The time spent waiting for g is the time needed for a message to get through the system's bandwidth bottleneck.
- **Bulk Gap,** G : a second measure of gap. Most machines have different mechanisms for bulk message transfer than for short messages. The g gap parameter applies to short messages while the G gap parameter applies to bulk messages. G refers to the time-per-byte spent transmitting bulk messages. This is the reciprocal of the bulk transfer bandwidth.
- **Number of Processors,** P : the number of processors in the system.

L, o, g, and G are measures of time. For example, the time spent sending a single packet from one machine to another is $L+2o$. Both the source processor and the receiver processor are utilized for o seconds each. The LogGP model assumes that the network capacity is bounded such that at most $\lceil L/g \rceil$ messages can be in transit at any one time. The LogGP model assumes that if the maximum network capacity is reached, messages are stalled until room on the network is available for the message.

3 A Rendering Algorithm

Before attempting to design a distributed rendering cluster, it is important to understand the rendering algorithm without the additional concerns of cluster computing. This paper presents one particular algorithm designed for multi-projector displays with rough overlap. This section ignores all issues related to the distributed nature of cluster computing.

There are two main stages in the rendering algorithm. The first stage is known as *Geometric Registration* and is pre-computed. The second stage is the *Rendering Loop* and occurs in real-time.

3.1 Geometric Registration

Before the rendering process begins, a number of parameters that describe the geometric properties of the display must be determined. Once measured, the parameters are used by the rendering algorithm to blend the projectors together to form a unified display.

The first goal of the geometric registration process is to determine each projector's relationship to a global coordinate system. This is done by computing a 3×3 matrix for each projector that maps each independent pixel space coordinate system into a single unified coordinate system. We refer to this matrix as a *collineation matrix*. A collineation matrix maps one two-dimensional space into another via an affine transformation. In order to incorporate the collineation matrix into a standard computer graphics pipeline, the matrix is expanded to a 4×4 representation. This technique was used by Raskar [10] to render corrected imagery for roughly aligned projectors.

The second goal of the geometric registration process is overlap estimation. Overlap regions can be computed by using the collineation matrices determined in the first part of the registration process. We would then like to attenuate pixel intensity values in the overlap regions to hide the seams. A two-dimensional lookup table can be built for each projector that stores a per-pixel attenuation factor. This table is called an *alpha map*. To achieve effective blending in the overlap regions, the alpha map is calculated so that the attenuation factors of overlapping pixels sum to one.

3.2 Rendering Loop

In the rendering loop, the images for each projector are rendered independently. The 4×4 collineation matrix corresponding to the appropriate projector is appended to the front of the matrix stack used in the rendering loop. The rendering pipeline then proceeds as normal. At the end of the pipeline, the alpha mask generated during the overlap estimation stage is applied to the final imagery. When these images are projected onto the display surface, the result is a unified high-resolution display built from multiple roughly aligned projectors.

4 Major Cluster Design Factors

When designing a rendering cluster that uses the algorithm presented in Section 3, there are a number of issues that must be explored. These design factors include the location of sorting in the rendering pipeline, assigning roles to individual machines, load balancing, and the impact of design decisions on network and communication performance. In the following sections, each of these issues will be discussed in detail.

4.1 Sorting in the Rendering Pipeline

Section 2.2 presented three types of sorting algorithms. In this section, we take a closer look at all three algorithms and how they would apply to cluster rendering for multi-projector displays. At first glance, both sort-first and sort-middle appear to be viable options. Conversely, sort-last algorithms appear to be ill suited for high-resolution systems. This is because sort-last systems must transmit pixel data across the network and multi-projector displays typically have a very large number of pixels. This is especially true as the display system scales upwards towards higher and higher resolution.

One important factor when choosing a sorting algorithm is the availability of consumer level systems. The low price for fully integrated high-powered graphics systems makes sort-first an attractive option. Implementing sort-middle with these consumer products could prove difficult.

However, it is still useful to explore the computational costs of each approach to determine which option is most applicable. The following sections (4.1.1, 4.1.2, and 4.1.3) build upon the cost comparison presented by Molnar *et al* [8].

4.1.1 The Cost of Sort-First

Sort-first algorithms incur an additional cost due to the overhead of the pre-transformation stage. This cost is proportional to the number of primitives being rendered. The next overhead cost is due to *bucketization*. Bucketization refers to the process of determining

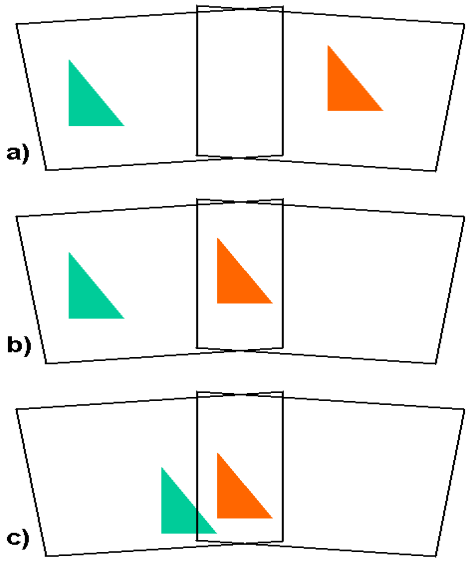

Figure 2: (a) Both the green and orange triangles fall in a single region. The overlap factor is 1. (b) The green triangle falls in one region while the orange triangle falls in two regions. The overlap factor is 1.5. (c) Both triangles fall in two regions. The overlap factor is 2.

to which processor each primitive should be forwarded. The cost of bucketization is proportional to both the number of raw primitives and the *overlap factor*. The overlap factor is a measure of how many regions are concerned about an average primitive. Figure 2 illustrates the overlap factor concept.

The cost of bucketization is related to the overlap factor because primitives falling within overlapping regions must be redistributed to more than one processor. It is important to note here that unlike the Molnar *et al* analysis, the overlap factor for both regular and rough overlapping displays may not approach one [8]. This is because the screen regions may actually overlap. In Molnar's analysis, the regions were assumed to have abutted relationships. The overlap factor is therefore a significant factor in our analysis.

Following bucketization, the primitives must be distributed across the network. This cost is proportional to the number of raw primitives, the overlap factor, and the fraction of raw primitives that must be redistributed between processors. This fraction is very important in deciding which algorithm to use. To help reduce the redistribution fraction, the system would be able to take advantage of frame-to-frame coherence. The coherence comes from the fact that a raw primitive assigned to one processor is likely to be assigned to the same processor in the next frame. Exploiting frame-to-frame coherence can drive down the redistribution fraction, greatly reducing the cost of redistribution. Once the primitives have been redistributed, each processor handles its assigned workload.

4.1.2 The Cost of Sort-Middle

Sort-middle algorithms don't perform any pre-transform work. Instead they calculate the real screen space coordinates for each primitive. This work is done in sort-first algorithms as well. The difference is that in sort-first algorithms, it takes place after redistribution. Following the calculation of screen space coordinates, many sort-middle algorithms perform tessellation of raw primitives into display primitives. The ratio of display primitives to raw primitives is known as the *tessellation ratio*, T. Sort-middle algorithms perform both bucketization and redistribution of display primitives instead of raw primitives. As a result, the costs of these two stages of rendering are T times greater than for sort-first. Following redistribution, each processor handles its assigned workload.

4.1.3 Cost Comparison

The cost differences between sort-first and sort-middle algorithms occur in three stages of the rending pipeline. The first difference occurs in the pre-transformation stage. Sort-first incurs a penalty proportional to the number of raw primitives while sort-middle does not need to perform this step and incurs no cost at all. Note that pre-transformation costs are incurred only on the processor performing the calculations. This means that as the system scales up to more projectors and higher resolution, there will not be any increase in network utilization due to the pre-transformation stage. No additional bandwidth will be needed.

The other two differences are due to the tessellation ratio. For sort-first, tessellation is not performed until after primitive distribution. For sort-middle, the cost of both bucketization and redistribution increases proportionally to the tessellation ratio, T. Bucketization costs are associated purely with computation, while redistribution costs impact both computation and communication. While the added cost of bucketization are similar to the pre-transformation costs of sort-first algorithms, the additional communication needs for sort-middle redistribution will translate into more traffic on the network. The increased traffic will be short-message transfers of primitives. As discussed in Section 4.4, this would place increased pressure on the network's short message gap time, g.

When designing a PC cluster for interactive rendering, all of these factors must be analyzed. If the pre-transformation costs are less than the added cost of tessellation, then sort-first is the more appropriate option. Otherwise, sort-middle may be more appropriate. Recall that when comparing the costs of the two options, it is important to differentiate between computation cost and network cost.

4.2 Role Assignment

Rendering algorithms call for the completion of a number of specialized tasks. As a result, a logical design decision would be to assign some specific roles to each PC in a cluster. One PC should be assigned to each projector. Additional PCs could be used to further divide the work load, but would require additional data to be transmitted across the network at the end of the rendering pipeline.

In addition, one or more PCs should be in charge of distributing primitives across the system. There could be one master machine in the cluster in charge of synchronization for the other PCs and other tasks that may require a single point of control. However, the design must be careful when assigning roles to the master PC to avoid bottlenecks. If at all possible, algorithms should be designed to avoid a central control point. This is particularly important as the scale of the system increases.

Machines may perform one, many, or all of the roles in the system, but it is important to determine what the roles are and how the cluster resources should be allocated. Figure 3 depicts a typical arrangement.

4.3 Load Balancing

Regardless of how effectively resources are allocated by role assignment, the load on each machine during run time will fluctuate and create imbalances. Imbalances in load might negatively affect the clusters performance. Section 2.3 presented a four phase load balancing model. This section will explore how the model can be applied to the distributed rendering pipeline.

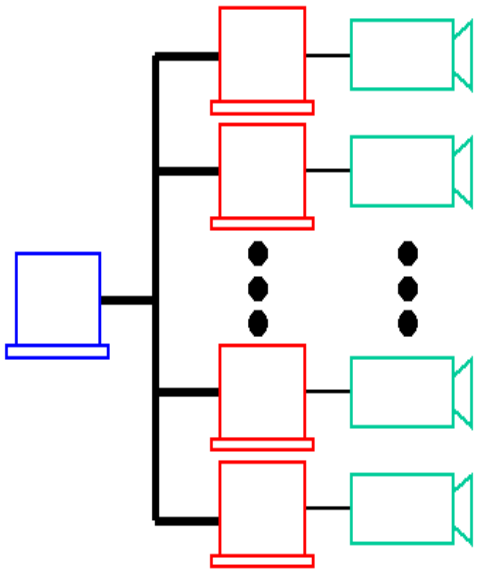

Figure 3: This figure shows a typical cluster arrangement. The blue machine represents the application host and control PC. The red machines are the individual PCs assigned to each projector. The projectors are represented in green. The thick black line represents a high speed point-to-point network.

4.3.1 The Load Balancing Task

The generic load balancing model redistributes *tasks* between processors to balance the computational load. Before analyzing the load balancing model any further, it is important to discuss what a task is in the context of a distributed rendering cluster. Section 2.2 stated that both sort-first and sort-middle assigned regions of screen-space to each processor. These regions were fixed, and for multi-projector display systems, they could corresponded to the region covered by the processor's projector. Because the regions assigned to each processor are fixed, the load on each processor is dependent on the geometric complexity of the screen-space region assigned to it.

To facilitate load balancing, it is desirable to allow the regions associated with each processor to change with time. However, this could result in pixel data targeted for one projector being computed at a processor bound to a different projector. To solve this problem, a small number of pixels would then need to be transmitted over the network at the end of the pipeline.

With this new approach, the system can divide screen space up into regions smaller than the regions associated with any single projector. The system could start with each of the smaller regions associated with the processor that will project that area of screen space. The system could then be capable of redistributing these small regions of screen space to other nodes in the system to accomplish load balancing. The balancing comes at the additional cost of forwarding the final pixel data across the network to the appropriate node. Because bulk data transfer mechanisms could be used to distribute the pixel data, the additional cost would likely rely heavily on the bulk message gap, G. The balancing might also cause a slight increase in primitive distribution by disrupting the frame-to-frame coherence exploited by the sorting algorithm. The cost of this increase would rely on the short message gap, g.

Using this approach, the task that is being distributed for load balancing is the work associated with a small area of screen space. The Princeton Scalable Display Wall uses a similar method for redistributing work in a rendering cluster [11].

4.3.2 Folding Load Balancing Into The Rendering Pipeline

Given the computational costs of load balancing, it appears at first glance that it might not be appropriate for a graphics rendering system designed for real-time frame rates. However, it is often possible to take advantage of frame-to-frame coherence in order to implement a low cost load balancing algorithm that is fully compatible with the rendering algorithm outlined in previous sections.

There are two main sources of computation in the rendering pipeline. This first source is the geometry processing. The second source is rasterization. A rough measure of load evaluation for the geometry stage is the number of primitives at a node. A rough measure of rasterization load is the number of primitives multiplied by the average amount of screen-space area for a primitive. Calculation of this load measure requires that the entire rendering pipeline finish to completion. However, we would like to have the load estimation before we start processing primitives in order to balance the load. Luckily, frame-to-frame coherence can be exploited at this stage. Load evaluation can be performed at the end of the pipeline and the results can be used to balance the next pass through the pipeline.

The profitability determination stage can be inserted at the very start of the pipeline. Before initiating the rendering process, the system should compare the load evaluations with the cost of balancing. The cost is a measurement of the time required to redistribute tasks as well as the network resources needed by the redistribution. If load balancing is not beneficial, the system should continue on as normal. If load balancing is beneficial, the task migration stage begins.

During task migration, nodes with the highest load should be notified that they are considered sources for load balancing. However, the sources should not just blindly send tasks to under-loaded nodes. Doing this would create a fragmented region of screen space to be assigned to each node. Because of the substantial overlap factor, this would create a large amount of replicated work, increasing the total amount of computation required by the system.

A more sophisticated task migration strategy would take advantage of the spatial relationships between the projectors associated with each node. If a source processor is overloaded, it will push tasks towards neighbors that are less loaded than the source. These neighbors will perform similar operations on the next iteration. This strategy slows down the responsiveness of the load balancing operation, but greatly reduces fragmentation in screen space. This technique of *context sensitive load balancing* is shown in Figure 4. It is similar to the sender initiated diffusion technique outlined by Willebeek-LeMair and Reeves [13].

Once the sources and destinations are determined, the fourth and final stage of load balancing takes place. During this stage, individual tasks are selected for redistribution. In the context sensitive migration strategy outlined above, there are a number of small screen-space regions that fall on the border between the source and destination nodes. During task selection, the best region for redistribution must be selected. In the context of the rendering algorithms discussed in this paper, the evaluation should be based on the load disparity present in the system, the load evaluation in each region, and the amount of overlap present in each region.

After load balancing has taken place, the rendering system continues executing the standard sort-first or sort-middle pipeline. At the end of the pipeline, pixels that were computed away from the default node must be transmitted across the network. The cost of the pixel redistribution is very high and must be incorporated into the profitability determination. If a large number of pixels must be transmitted at the end of the pipeline, the system may achieve better performance in an unbalanced state.

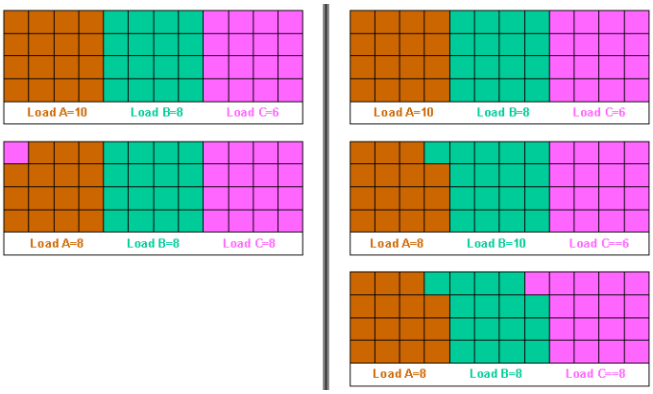

Figure 4: The image to the left depicts the spatial fragmentation that occurs with a simple load balancing algorithm. The image to the left shows that a context sensitive algorithm can reduce fragmentation at the added cost of slower balancing performance.

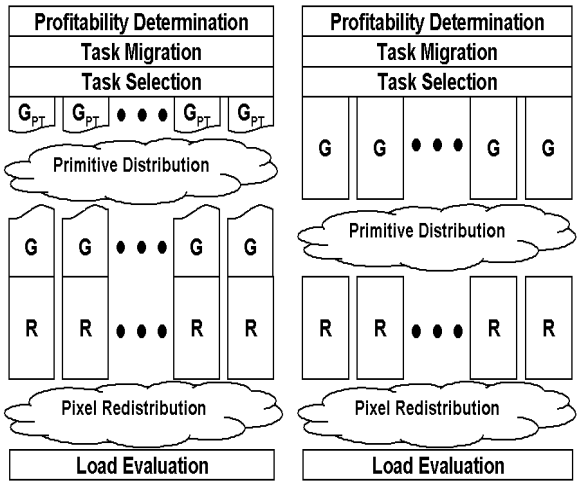

Figure 5: The left figure shows a sort-first pipeline. The right figure depicts a sort-middle pipeline. In this figure, G represents the graphics processing stage. G_{PT} represents the pre-transformation stage needed in sort-first pipelines. R represents the rasterization stage.

4.3.3 Impact of Sorting on Load Balancing

The is an important relationship between load balancing and the sorting algorithm that depends upon where in the pipeline each operation takes place. As shown in Figure 5, load evaluation occurs at the very end of the pipeline. The remaining stages of load balancing occur at the start of the pipeline. Recall that as task selection occurs, screen-space regions have been reassigned to different machines to balance the load.

However, no geometry data gets reassigned until the primitive distribution stage. This creates a slight delay in the effectiveness of the load balancing operation. The delay occurs between task selection and primitive distribution. As Figure 5 shows, the length of this delay depends on the sorting algorithm. Sort-first algorithms must wait for the pre-transformation stage to complete, while sort-middle algorithms must wait for the entire geometry processing stage to complete. Because sort-first algorithms distribute primitives earlier in the pipeline, the benefit of load balancing is felt earlier.

4.4 Communication and Network Topology

When designing a PC rendering cluster, there are a number of options concerning network infrastructure and the overall topology. Options for the infrastructure include Myranet, Ethernet, and Gigabit Ethernet. Network protocols need to be chosen for communication. These protocols may or may not provide reliable delivery and or quality of service guarantees. Network topology is a critical issue as well. Should the network be a broadcast system? Should it provide point-to-point communication? What parameters of the network are most critical to the cluster's performance?

4.4.1 Point-to-Point Network

Due to the nature of both task and pixel redistribution, it seems that a point-to-point network would be most appropriate. This is because both tasks and pixels are sent from a specific source to a specific destination. Most communication will therefore be in the form of point-to-point messages.

However, incorporating broadcast capabilities might prove useful in synchronizing all of the nodes in the cluster at critical points in the rendering pipeline. Reliable delivery also seems important for maintaining a high performance level. Lost or missing data would likely cause problems and slow down the rendering process.

4.4.2 Applying the LogGP Model to a Rendering Cluster

To get a better understanding of the network requirements needed for a cluster rendering system, we can use the LogGP model of distributed network performance described in Section 2.4. However, before we can utilize the model, it is important to analyze the rendering system's properties.

A PC rendering cluster capable of interactive frame rates will be a highly synchronized system since interactive applications are often required to render multiple frames per second. In addition, the rendering system in this paper will require multiple synchronization points per frame. For example, the system must synchronize after redistributing primitives, after redistributing load balancing tasks, and when finished rendering the final pixel data.

This high degree of synchronization will weigh heavily on the network design. Experiments performed at the University of California at Berkeley have shown that synchronized applications are likely to be dependent on total round trip times [7]. This implies that a rendering cluster will be sensitive to L, the network medium latency. Therefore, a high-speed network will be critical to the cluster's performance.

For both sort-first and sort-middle systems, the redistribution stage of the pipeline will likely use short messages to perform the redistribution of tasks. This is because the items being distributed are individual primitives. This implies that the short message gap time, g, is important. Similarly, the bulk transfer gap, G, will be important at the end of the pipeline because bulk transfers will likely be used to transfer raw pixel data across the network. Pixel data will need to be sent back to the appropriate hosts after they have been computed elsewhere to achieve load balancing. This implies that the required bandwidth will be proportional to the amount of data redistribution (due to both both sorting and load balancing) expected from the rendering cluster during typical operation.

As the display supported by the cluster scales up to higher and higher resolutions, the bulk transfer gap, G, will become more important as the number of raw pixels redistributed over the network increases. The number of pixels to be redistributed increases for both regular and rough overlapping displays because higher resolution displays use more projectors. This results in a larger number of pixels in overlap regions.

The analysis above might imply that L, g, and G are the most important factors in network design. Indeed, the network should

be designed to optimize all three factors. However, the Berkeley study showed that above all else, applications are most sensitive to network overhead, o. Even lightly communicating applications will exhibit a factor of 3-5 slowdown on systems with network overhead values typical of current LAN communication stacks [7]. Highly communicating applications such as the rendering cluster will show far worse effects.

In order to limit network overhead, o, as much as possible, the rendering cluster should be designed with a messaging system that allows applications to bypass the LAN communication stack. For example, MPI, or Message Passing Interface, allows applications to send messages across a local area network without accruing the overhead of the communication stack [9].

Improvements to both gap, g, and network overhead, o, will provide improvements to cluster performance. The Berkeley studies showed that almost all applications show a linear dependence on both parameters. This implies that reduction of network overhead and gap will result in a similar improvement in overall cluster performance. The same studies showed that improvements due to lower bulk transfer gap, G, and network medium latency, L, were more difficult to quantify. They showed that improvements from reduced G and L are more closely related to the application's design.

5 Cluster Design Suggestions

After exploring the cluster design factors presented in Section 4, a few guidelines become clear that should be followed when designing a distributed PC render cluster for roughly aligned projector displays. First, a hybrid rendering approach should be used. The hybrid algorithm should combine sort-first or sort-middle for primitive redistribution with pixel redistribution at the end of the rendering pipeline. The sort-first or sort-middle segment of the algorithm facilitates distribution of raw or display primitives to the host responsible for rendering the appropriate region of screen space. The pixel redistribution segment of the algorithm facilitates redistribution of the pixel data back to the host that is attached to the appropriate projector after load balancing. Choosing sort-first over sort-middle will make integrating standard graphics hardware an easier process. Figure 5 illustrates the two distribution stages in the hybrid pipeline.

Second, load balancing should be implemented if typical uses of the cluster will create grossly uneven workloads such as geometry-bound bottlenecks. Although frame-to-frame coherence can be exploited to improve load balancing, the cost associated with transferring tasks and pixels is still very high. Care should be taken that the benefits of load balancing outweigh the cost and that it actually improves performance.

The third guideline is that when designing the communication portions of the cluster, message passing libraries that bypass the typical LAN communication stack should be used to minimize the cost of network overhead. Likewise, networks should be designed to minimize both gap values. The will allow primitive and pixel redistribution to perform at higher rates. It is also important to recognize that the high degree of synchronization inherent in load balancing and the rendering pipeline are likely to make network latency a very important parameter to system performance.

6 System Scalability

One of the largest benefits to using a PC cluster to render for multi-projector displays is the relatively cheap cost of increasing the system's scale. If the rendering were performed on a large supercomputer, scaling up the system would require a new machine with a large financial cost. Using a PC cluster to render the imagery allows the system to scale up by adding an additional PC to the cluster. This allows new projectors to be added to the display at low cost. However, it is important to understand how scaling the system affects the system's performance.

The area impacted most heavily by an increase in the number of hosts is the network. An increase in hosts translates into increased network traffic. More tasks will be distributed and more pixel data will be transferred at the end of the pipeline. This increased traffic will increase bandwidth usage. The other network parameters (L, G, g, and o) will not be directly affected. It is important to insure that the underlying communication medium supplies enough bandwidth for the scale of the system.

The load balancing algorithm will also be affected. Processor load evaluation occurs independently at each host and remains the same. Likewise, task selection is independent to scale and remains the same. This is because task selection in the render cluster context depends only on spatial neighbors. However, load balancing profitability determination and task migration become more complicated as the number of inputs into the decisions increases.

The rendering algorithm should remain relatively immune to the scale of the rendering system. The major parameters to the rendering algorithm's performance relate to the geometry being rendered. This includes the number of primitives and the tessellation ratio. However, the one area that is affected by additional hosts is the overlap factor. As more hosts are added to the system, the screen-space regions become smaller and smaller with respect to the geometry if all other factors such as geometry and field-of-view remain constant. This means that the effective size of primitives in pixel space will be larger and they will be more likely to fall on overlap regions. Increasing the overlap factor results in duplication of work at various nodes. This trend towards an increasing overlap factor can be partially overcome by merging all screen-space regions on a single host into one region. This will reduce overlap factors within each host to the minimum possible. However, the overlap factor will still increase as the scale of the system increases, albeit at a slower rate.

7 Conclusion

Designing a rendering cluster that provides interactive graphics for multi-projector displays is a complicated task. There are a number of design decisions that must be made when building such a cluster. This paper has presented a few of these issues and explored many of the options faced during the design of a rendering cluster.

One major issue addressed is the sorting process needed within the rendering pipeline. Depending on the final cluster architecture, either sort-first or sort-middle algorithms may be appropriate. Another major issue is load balancing. This paper discussed a context sensitive load balancing approach and provided an overview on how the technique could be used in conjunction with a parallel rendering algorithm such as sort-first or sort-middle. The last major issue covered in this paper was network performance. This paper provided a description of how the redistribution requirements associated with both sorting and load balancing impact network performance.

The rendering algorithm's sorting process, load balancing, and network communication are all important factors that weigh heavily on the system's overall performance. When designing a PC-based rendering cluster, they must all be explored during the design process.

References

[1] Gary Bishop and Greg Welch. Working in the Office of the "Real Soon Now". *IEEE Computer Graphics and Applications*, 20(4):76–78, 2000.

[2] Carolina Cruz-Neira, Daniel J. Sandin, and Thomas A. DeFanti. Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE. *Computer Graphics*, 27(Annual Conference Series):135–142, 1993.

[3] David Gotz. The Office of the Future DisplayWall System. UNC-CH Comp238 Class Report.

[4] Mark Hereld, Ivan R. Judson, and Rick L. Stevens. Introduction to Building Projection-based Tiled Display Systems. *IEEE Computer Graphics and Applications*, 20(4):22–28, 2000.

[5] Greg Humphreys and Pat Hanrahan. A Distributed Graphics System for Large Tiled Displays. In *IEEE Visualization 1999*, San Fransisco, October 1999. citeseer.nj.nec.com/241717.html.

[6] 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. Building and Using A Scalable Display Wall System. *IEEE Computer Graphics and Applications*, 20(4):29–37, 2000.

[7] Richard P. Martin, Amin M. Vahdat, David E. Culler, and Thomas E. Anderson. Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[8] Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32, 1994.

[9] MPI. The MPI-2.0 Standard. http://www.mpi-forum.org/.

[10] Ramesh Raskar. Immersive Planar Display using Roughly Aligned Projectors. In *IEEE VR 2000*, New Brunswich, NJ, USA, March 2000.

[11] Rudrajit Samanta, Jiannan Zheng, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh. Load Balancing for Multi-Projector Rendering Systems. In *ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, August 1999.

[12] Daniel R. Schikore, Richard A. Fischer, Randall Frank, Ross Gaunt, John Hobson, and Brad Whitlock. High-Resolution Multiprojector Display Walls. *IEEE Computer Graphics and Applications*, 20(4):38–44, 2000.

[13] Marc H. Willebeek-LeMair and Anthony P. Reeves. Strategies for Dynamic Load Balancing on Highly Parallel Computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):979–993, 1993.