The Sort-First Architecture for Real-Time Image Generation

by

Carl Alden Mueller

A Dissertation submitted to the faculty of The University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill

2000

Approved by:

______________________ Anselmo Lastra	Advisor
______________________ Nick England	Reader
______________________ Henry Fuchs	
______________________ John Poulton	
______________________ Jan Prins	Reader

June 8, 2001 14:07 PM

CARL ALDEN MUELLER. The Sort-First Architecture for Real-Time Image Generation (under the direction of Anselmo Lastra).

ABSTRACT

The field of real-time computer graphics has been pushing hardware capability to its limits. There is demand to increase not only the complexity of the models that are displayed, but also the resolution of the images. To provide the ultimate power for interactive graphics, fully-parallelized systems have been developed which work on rendering multiple polygons at all major stages of the graphics pipeline. UNC researchers have devised a taxonomy of such architectures, naming the classes "sort-first," "sort-middle," and "sort-last" [MOLN91, MOLN94]. While the latter two have been well explored and developed, sort-first has not, despite the fact that it offers great promise for real-time rendering.

Sort-first offers an advantage over sort-middle in that it can take advantage of the frame-to-frame coherence that is inherent in interactive applications to reduce the communications bandwidth needed to send primitives among processors. Sort-first also offers an advantage over sort-last in that it does not require huge amounts of communication bandwidth to deal with pixel traffic. Based upon these advantages, our thesis states that sort-first is the architecture best suited for rapidly rendering very high-resolution images of complex model datasets. However, to support this thesis, we must show how sort-first can be implemented efficiently.

The main issues standing in the way of sort-first implementation are load balancing and management of a migrating primitive database. These issues are explored thoroughly in this dissertation. We demonstrate a new adaptive load-balancing method (applicable to sort-middle as well) which allows for efficient load distribution with low overhead. We also show two solution methods for managing a hierarchical database of migrating primitives. In addition we examine the communications requirements for sort-first, first by examining a design example and later by providing a comparison against sort-middle.

Our research shows that sort-first is a viable and very competitive architecture. Its strengths make it the ideal choice when very high-resolution rendering is needed for complex, retained-mode datasets. While this dissertation clears the way for sort-first implementation, there are still many opportunities for further exploration in this area.

ACKNOWLEDGEMENTS

I would like to give special thanks to my parents for providing constant support and encouragement during my graduate career. I would also like to thank my wife for having the patience to endure our long separation while I finished working on my dissertation.

I would also like to thank the many people who have helped me with my labors throughout the years. This includes my advisor, Anselmo Lastra, whose belief in my work was an important motivational factor. Thanks go to my office-mates over the years (Jonathan Cohen, Bill Yakowenko, Bill Mark, and Michael Bajura) both for putting up with me and providing help whenever I asked. Many thanks also go to David Ellsworth and Marc Olano for help with my papers and for providing many inspiring ideas.

There are many people, both in and out of UNC, whose presence has helped me in innumerable ways during my stay here. I thank you all for opening your hearts and minds to me.

TABLE OF CONTENTS

Page

1. Introduction

This is a dissertation on high-performance interactive computer graphics architectures. Decades of development in this area have yielded tremendous progress, yet there remain many challenges. If one wishes to render realistic images of a complex environment in real time, or if one wishes to interactively view a complex scientific dataset, there is only one class of architectures that is truly up to the task: fully-parallelized architectures. Yet even with such systems, certain challenges remain unanswered. One such challenge is the rendering of complex scenery using very high-resolution display devices. This dissertation addresses this challenge by exploring an architecture that has thus far received little attention. It is known as "sort-first."

The name of this architectural class comes from a taxonomy of fully-parallelized graphics architectures [MOLN94]. This taxonomy was developed around 1990 by UNC graphics researchers, and it includes the classes "sort-first," "sort-middle," and "sort-last." While the latter two classes have been well studied and developed into various working systems (see chapter 2), the former was thought to be too complex to be practical. At a glance, sort-first appears to have many difficulties that would make it difficult to implement efficiently.

Given these difficulties, and given the advances made with sort-middle and sort-last systems, there has been little impetus to pursue sort-first. However, as we shall see, sort-first does have some becoming characteristics. It can take advantage of the frame-to-frame coherence that is natural in interactive applications to greatly reduce system-wide communications needs. Sort-first can also take advantage of certain visibility-culling algorithms that would be difficult to implement efficiently with sort-middle or sort-last. For these and other reasons, sort-first appears to be well suited for the challenges mentioned: applications with large amounts of geometry to render to very high resolution display devices.

In this dissertation, we investigate the sort-first architecture and present solutions to enable its efficient implementation. We first examine sort-first in comparison to the other classes of fully-parallel graphics systems and show how sort-first has the potential to out-perform them for the applications mentioned above. We next look at load balancing, a potential pitfall for all parallel systems, and we demonstrate a new load-balancing algorithm that performs well for sort-first (and in fact could be applied to sort-middle as well).

We also examine several problems surrounding graphics database management for sort-first. Such management is not a simple issue with either sort-middle or sort-last, and it is made even more difficult with sort-first due to data migration. We describe two solution paths and examine their performance. After this we examine hardware considerations for sort-first implementation, and we describe specific considerations for implementing sort-first on an existing hardware architecture, UNC's PixelFlow [EYLE97].

The last part of this dissertation returns to the premise that sort-first is the best candidate architecture for high-resolution rendering of complex databases. Having already eliminated sort-last due to bandwidth considerations, we perform a detailed comparison between sort-first and sort-middle. The results show that while sort-middle offers slightly less computational overhead, sort-first's ability to take advantage of frame-to-frame coherence reduces its communications bandwidth requirements significantly compared to sort-middle, and thus sort-first is the architecture of choice.

1.1 Real-Time Computer Graphics

For applications such as vehicle simulation, architectural walk-through, computer-aided design, and scientific visualization, a real-time graphics system is necessary to provide an interactive visual output device. Given a scene consisting of many thousands of 3D primitives, the graphics system must be able to render views of the scene at (ideally) thirty times per second or faster. The output image may consist of anywhere from one quarter million pixels up to twenty million pixels or more.

Although there are various rendering algorithms available, nearly all high-end interactive graphics systems are based upon the standard "graphics pipeline" rendering algorithm [FOLE90]. This pipeline has two major steps. Starting with primitives (polygons) in object space, a geometric transformation step transforms the primitives into screen space. This is followed by a rasterization step to convert the primitives into a set of screen pixels. We finish with a set of appropriately colored pixels in the framebuffer.

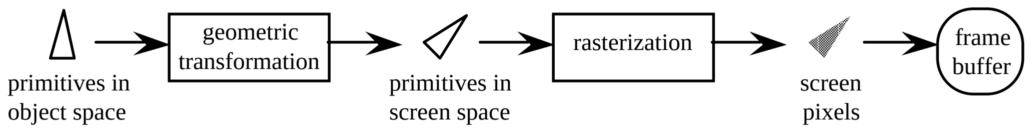


Figure 1.1 Traditional Graphics Pipeline

We note that rendering can be construed as a type of sorting problem: each pixel of the image may be considered a bin; each primitive from each object must have its contribution to the scene placed in the right set of bins (this fact was recognized in [SUTH74]).

1.2 Parallel Computer Graphics

To meet the growing demands of current applications, high-end graphics systems require parallel processing at both of the major pipeline stages. To develop such a “fully parallel” architecture, a designer must decide how the work is divided at each stage, how the results propagate from stage to stage, and how the partial results come together to form the final results. Given that rendering is a sorting problem, any initial subdivision of the work at the input stage will require a global exchange (a sort) at some point in time during the pipeline in order to construct the final image.

When one examines the graphics pipeline, one finds that there are ways to perform the sort by exchanging object-space primitives, screen-space primitives, or screen pixels. When UNC graphics researchers realized this fact, they introduced a taxonomy of fully parallel graphics architectures with branches known as “sort-first,” “sort-middle,” and “sort-last.” We will organize our exploration of the solution domain by considering each of these architectures in turn.

1.2.1 Sort-First

With sort-first, the work is partitioned using screen-space subdivision (fig. 1.2). In its ideal form, there is one computational node per region of the screen. The primitives are initially distributed among the various nodes. During rendering, the object-space primitives first undergo a “classification” step wherein they are routed to the correct nodes as soon as their screen-space positions are computed. Each node then completes the transformation and rasterization of the primitives in its region and sends the resulting pixels to the framebuffer. Thus once the primitives have been sorted, each node operates like serial graphics pipeline in order to render its portion of the screen. Note that after a frame has been rendered, the object-space primitives can remain where they were sent in order to reduce the communication needs for the next frame (assuming that the same set of primitives will be used again).

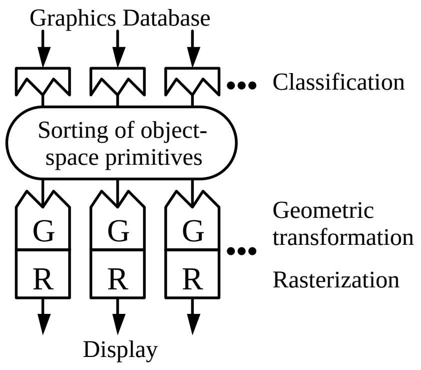


Figure 1.2 Sort-First Graphics Pipeline

1.2.2 Sort-Middle

In sort-middle, the nodes are logically split into two groups (fig. 1.3). One set of nodes is responsible for primitive transformation while the other set handles rasterization. The primitives are distributed among the transformation nodes, while the rasterization nodes are assigned different regions of the screen to render. Thus each transformation node performs all the transformation work for its primitives, then sends the resulting screen-space primitives to the appropriate set of rasterization nodes. There the primitives are rasterized and the resulting pixels are eventually sent to the framebuffer.

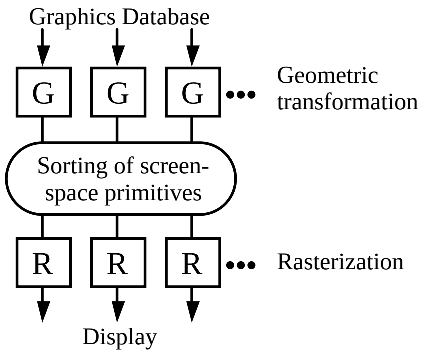


Figure 1.3 Sort-Middle Graphics Pipeline

1.2.3 Sort-Last

With sort-last, the primitives are again distributed equally among all of the nodes (fig. 1.4). Each node performs all of the transformation and rasterization for the primitives assigned to it. The result is a set of full-screen images, one from each node, each of which represents only a fraction of the primitives. A composition step is required to sort the pixels from these images to produce the complete image. During composition, the system examines the corresponding pixels from each image and passes on only the visible pixels.

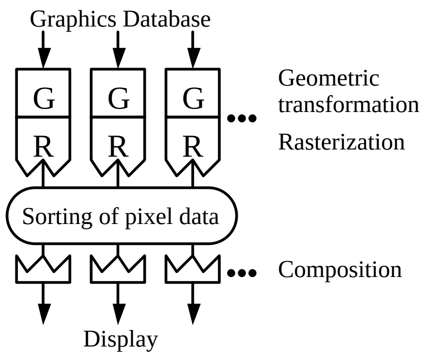


Figure 1.4 Sort-Last Graphics Pipeline

1.2.4 Comparison

We briefly examine the pros and cons of these architectures. Sort-last scales very well with respect to the number of primitives it can handle. This is because increasing the number of compute nodes does not significantly affect the amount of communication required for the sorting (composition) operation. However, this communication bandwidth is directly affected by the desired resolution and quality of the output image, and it can be become extremely high for large images. This is because in order for sort-last to produce high-quality (anti-aliased) images, each sample at every pixel must be composited. (Compositing only the final pixels from each board would lead to incorrectly sampled images.) For a 4000x4000 pixel, 16-sample image at 30 fps, the required bandwidth is close to 46 gigabytes per second. Thus composition network bandwidth can stand in the way of rendering very high-resolution images with sort-last.

Sort-middle and sort-first both send only “finished” pixels to the framebuffer, and thus they don't have the same communications problem at the back-end. However, sort-middle does require that for each image generated, every on-screen, transformed primitive must be sent from some transformation node to some rasterization node. For scenes with very large numbers of primitives, this can represent a very large amount of communication. In addition, this pattern of many-to-many communication does not scale well as the number of primitives increases.

With sort-first, the amount of primitive communication may be greatly reduced by taking advantage of frame-to-frame coherence. In interactive applications, we expect that one frame will be fairly similar to the next: the primitives will have only moved slightly in screen space. In sort-first, as a primitive moves across the screen, it will only need to be communicated to another node when it crosses the boundary into the other node's region. Taking advantage of coherence in this way does assume the use of "retained mode," wherein the primitive database is stored on the graphics system. With this communications optimization, the number of primitives being sent can be reduced to a small fraction of the number of on-screen primitives.

Thus sort-first appears to be the only architecture of the three that can deal well with both large numbers of primitives and large numbers of pixels. However, sort-first is not without its problems. As with any parallel system, load balancing is a major issue. How can the screen be divided in such a way as to provide each node with a similar amount of work, and yet keep the overhead associated with the subdivision down to a reasonable level? Another major concern is the management of a set of migrating primitives, i.e., the primitive database. How can the system keep track of this database, provide for the concerns of hierarchical database management, and yet still be efficient? Answering these questions forms the core of this thesis.

1.3 Thesis Statement and Contributions

The goal of this dissertation is to establish the following thesis:

> "Among the classes of highly-parallel graphics architectures, sort-first is the best choice to use for interactive graphics applications involving large retained-mode databases and very high-resolution displays."

The exact values for "large" and "very high-resolution" are relative, depending upon the capabilities of the hardware at any given point in time. At present, a "large" database is typically one million polygons or more, while "very high-resolution" suggests rendering over eight million pixels.

In order to demonstrate our thesis, we must accomplish two tasks. One, we must show that sort-first is a viable rendering architecture. This requires us to address the major difficulties in creating a sort-first implementation. Namely, these are the need to find an efficient load balancing algorithm for sort-first, and the need for an efficient database management scheme to handle sort-first's migrating primitives. Both of these are complex and interrelated issues which will be discussed in the following sections.

Second, we must show that among the various fully-parallel architectures, sort-first is best suited to the application domain suggested in the thesis statement. This task we approach in two parts. We first perform a bandwidth analysis, which rules out sort-last and suggests that sort-first has an advantage over sort-middle. Then, once we introduce a sort-first implementation, we perform a more detailed comparison between sort-first and sort-middle. As discussed below, we find sort-first has the advantage, and this establishes our thesis.

1.3.1 Efficient Load Balancing for Sort-First

Load balancing is critical for distributing the rendering workload among the processors and preventing a single overloaded processor from becoming a major bottleneck. With sort-first, load balancing means partitioning the screen and assigning the processing nodes to the various screen regions. Our research shows that static load-balancing methods are typically not well-suited to sort-first due to the extra overhead created when subdividing the screen into more regions than there are processors (which is necessary in static load-balancing to even out the processor load). We introduce a new load-balancing algorithm, known as "MAHD" (mesh-based adaptive hierarchical decomposition), which adaptively subdivides the screen based upon an estimation of the on-screen primitive distribution. The algorithm uses an efficient process to estimate the primitive distribution, and the screen subdivision process is simple as well. Since MAHD only assigns one region per processor, it does not suffer from the overhead that upsets the static load-balancing method. MAHD subdivides the screen using a kd-tree, and it works for an arbitrary number of processors (it is not limited to powers of two). We also show a simple constant-time algorithm for determining which regions of the kd-tree a given bounding-box lies in. This algorithm is necessary to avoid adding extra overhead when sorting primitives.

We examine many implementation details regarding the MAHD algorithm. We examine how the algorithm fits into the overall parallel rendering structure. This involves various data collection, communication, and synchronization issues. We also consider two forms of the algorithm: one using primitive distribution estimated from the current frame, and another "pipelined" version that uses information from the previous frame (in order to avoid synchronization delays). We find that both versions have no difficulty maintaining reasonable efficiency under normal circumstances.

Aside from the examination of the MAHD algorithm, this dissertation also discusses various practical issues related to sort-first load balancing. We look at how the system is affected by the use of primitives that are more complex than simple polygons. We also consider ways to render for multiple different viewpoints while still taking advantage of frame-to-frame coherence. Finally, we examine how rasterizer hardware design affects the rendering and load-balancing algorithms.

1.3.2 Efficient Database Management Algorithms

Maintaining a database consisting of a simple primitive list is not a difficult task, but a modern graphics system must be able to provide more high-level control than a simple list provides. We therefore explore what is necessary to implement a PHIGS-like hierarchical graphics database in sort-first. Such a database allows definitions of "structures" (containing groups of primitives and various rendering state elements) and provides hierarchical manipulation by allowing one structure to "instantiate" or "call" other structures [FOLE90].

We examine the choices for how to distribute the database hierarchy and the primitives contained therein. When keeping track of primitive migration, we find that we must distinguish between the declared primitives versus the instantiated ones, and this leads us to introduce two distinct bookkeeping methods, referred to as the "min-set" and "max-set" methods. To support these methods, we find we should perform bookkeeping with a quantum consisting of groups of primitives rather than individual primitives, and we address the resulting issues of group creation and the group-overlap factor. For each bookkeeping method, we lay out the data structures and algorithms that are necessary to support sort-first operations on a distributed hierarchical graphics database. Finally, we examine the performance implications of both solution methods, and we outline some possible paths for improvement.

1.3.3 Additional Contributions

Though load balancing and database management are the key issues that are specific to sort-first implementation, we also provide a guide to cover certain hardware-related design issues. For the communications network, we provide an example to illustrate sort-first bandwidth needs, and discuss design choices that would favor sort-first. We also discuss rasterizer design and how the choices are affected by sort-first.

In order to prove our thesis statement, we perform a detailed comparison between sort-first and sort-middle (having already ruled out sort-last as a potential candidate). We detail the differences in communications requirements, and we offer analytic models to describe the computational overhead of each architecture. The comparison reveals that while sort-first has a somewhat higher computational overhead (on the order of 20-50% extra transformations), its savings in communications bandwidth (generally sending only one fourth the number of primitives or less) more than make up for this and therefore make it the better candidate for the targeted application area.

1.4 Organization of this Dissertation

In chapter 2 we explore more fully the field of parallel rendering. We consider the application demands on interactive graphics systems, and we present a detailed description of the sort-first, sort-last, and sort-middle architectures. We look at each architecture in light of the application demands and compare their communications bandwidth requirements. For high-resolution rendering, this puts sort-last out of the picture, leaving only sort-first and sort-middle. Sort-first's advantage lies in its ability to make use of frame-to-frame coherence, and we show the results of a simple experiment to demonstrate this advantage.

With chapter 3 we examine the issue of load balancing with sort-first. As with any parallel algorithm, good efficiency demands careful attention to this issue since poor load balancing can completely negate the advantages of parallel processing. We examine existing static and adaptive load-balancing strategies and present the new MAHD algorithm. This is followed by an explanation of certain primitive-handling issues and more details concerning the implementation of the MAHD algorithm. From there we present results of various load-balancing studies, showing us the limitations of static load-balancing and the promise of MAHD. Next we develop a parallel, software, sort-first implementation based upon MAHD, and we examine its performance characteristics with some additional data sets. Finally in this chapter we examine other system issues and how they relate to load balancing. These include the handling of complex primitives, the display of multiple viewpoints, and the design of the rasterization hardware.

In chapter 4 we study the topic of primitive database management in sort-first. First we describe the various types of graphics databases: primitive lists, hierarchical graphics databases, and immediate mode. We focus on hierarchical databases and the problems of bringing such a database into a parallel system. With sort-middle or sort-last, these issues are mostly just ones involving representation and distribution. Sort-first adds various primitive redistribution issues such as bookkeeping of distribution data and efficiently sending and receiving primitives. In addition, one must also keep other database operations efficient as well. We also examine the issue of immediate-mode, where the database is generated "on the fly."

In chapter 5 we examine various practical issues of sort-first implementation. We start out with a performance goal and then examine some of the system characteristics that are necessary to meet this goal. Of chief concern are the communications subsystem design and the bandwidth characteristics. Also in chapter 5 we examine how sort-first can be implemented over an existing hardware platform. The platform in question is PixelFlow, a graphics machine developed by UNC, Division, and Hewlett Packard. We describe the machine and its conventional operation, and then we examine the challenges involved in implementing sort-first with it.

Chapter 6 is a return to the thesis statement. We once again examine how the various fully-parallel architectures perform for large-database applications rendering to very high-resolution displays. Having eliminated sort-last as a candidate in chapter 2, and now having a more detailed understanding of sort-first, we present in this chapter an analytic comparison between sort-first and sort-middle, with the goal of determining which is best-suited to the desired target application area.

Finally in chapter 7 we offer some conclusions and lay out some directions for future research.

2. Parallel Rendering (Why Sort-First?)

Let us step back for a moment and consider the larger picture surrounding interactive computer graphics systems. Such systems serve as the backbone for interactive or real-time graphics applications. These applications include vehicle simulation, architectural walk-through, computer-aided design, and many different kinds of scientific visualization.

Within a given application, the job of the graphics system is to draw an image corresponding to a view of all the objects in a given environment. In order to be interactive, this process must be repeated (with updated parameters) many times per second. The task is further complicated by the fact that there may be hundreds of thousands of primitives and millions of pixels.

Such is the case with the set of applications mentioned. Demands for greater detail and realism mean that display databases with hundreds of thousands (or even millions) of primitives are becoming common. While sometimes a large fraction of the primitives can be culled from the view, this is not always the case. Sometimes an engineer or researcher desires to see a multi-million primitive model on screen, and no amount of culling or simplification can reduce the polygon count enough for interactive performance without sacrificing the representation accuracy [ALIA98]. On the output side of the rendering task, the numbers are growing as well. A “standard” high-resolution display device can display about one million pixels, while new display devices are becoming available that can display tens of millions of pixels (see section 2.1.3). For interactive applications requiring the display of such large numbers of primitives and pixels, we find few graphics systems that are up to the challenge.

Only “fully-parallel” graphics systems are capable of dealing with such challenges. In this chapter, we shall explore the taxonomy of such systems and examine the suitability of each toward complex rendering tasks. First, however, we shall go over the application requirements a little more closely.

2.1 Interactive Graphics Applications

There are many types of interactive graphics applications, each with varying demands from the graphics system. For now, we will only consider some basic demands: the graphics system performance, the size of the graphics database, and the nature of the output device(s).

2.1.1 Performance Requirements

The challenge that we are concerned about is not just rendering images, but doing so in an interactive or real-time manner. “Interactive” means that the system responds quickly enough such that the user can adjust the controlling inputs in rapid response to the output. “Real-time” is an even more stringent requirement, dictating that the system must always respond with updated outputs within a certain small fixed amount of time. To simplify matters, we will use the term “interactive” to cover both cases as long as a distinction is not necessary.

The exact response time needed for interactive graphics varies depending upon the application and the user interface. One can place certain bounds on the throughput requirements, however. For many applications, thirty frames per second (fps) are required to maintain the illusion of fluid motion [PADM92]. Fewer frames per second may be adequate in certain visualization applications, but anything under one frame per second is challenging the definition of interactive. The minimum necessary update rate is dependent upon how fast objects in the simulation are moving relative to the display. Large ship simulations may find 10 fps to be a reasonable update rate, while tactical helicopter simulations can demand 60 fps or more [LATH94]. The display system involved is an important factor as well. Head-mounted displays (HMDs) and other moving-display systems can cause a fair amount of relative motion that must be considered in the equation.

For any interactive application, what is of equal importance to update rate is latency. This refers to the time from when the graphics system receives all the necessary information to begin producing an image until the user sees the finished image. Too much latency in interactive applications can lead to a variety of problems, from difficulty of control to “simulator sickness” [HELM93]. Again, actual latency requirements vary by application. Simple visualization applications might tolerate latency up to one second. HMD applications ideally should have zero latency, especially if the display has see-through capability. This is necessary in order to register the real world with the computer-generated one. One attempts to achieve zero latency by using prediction of the user's head position, but this requires low, predictable latency from the system to start with. For HMD applications, it is estimated that prediction is only useful up to about 80 ms into the future, a time that must include head tracking and viewpoint computation time as well as rendering time [AZUM94].

2.1.2 Database Sizes

The "size" of the display database is a term with many meanings. It can refer to the number of primitives the application has in total, or it can refer to the number of primitives that the graphics system must actively consider for any given frame. We will attempt to distinguish these as necessary.

The total sizes of databases can easily reach millions of primitives or more. Data from the Defense Mapping Agency can be used to generate large-area databases for tactical training simulators [WALE83]. Other examples of large databases include:

- large-scale architectural models
- architectural models tessellated for radiosity calculations
- detailed CAD models of large-scale constructs (factories, vehicles, etc.)
- models of physical objects created by high-resolution digital scanning
- finite element models from flow simulations (air flow, heat flow, etc.)
- databases from volume data applications (e.g., polygonalized CAT scan data)
- particle simulation models used for astronomical research
- particle simulation models for meteorological research
- circuit models in VLSI design

In all of these areas, the desire to show more complex, more detailed models has been pushing graphics systems up to and beyond their limits. Current graphics systems are reaching a peak performance of approximately ten million polygons per second [MONT97], meaning that such systems cannot display, in interactive fashion, databases larger than about one million polygons.

For the applications involved with these databases, the percentage of the database likely to be on-screen varies from a small fraction to unity. Thus all stages of a graphics architecture must be able to scale to the appropriate size. For many such large-database applications, the real challenge has become the reduction of the database into a small enough size that can be handled by the graphics system. While progress is being made on this front (using both sophisticated culling and simplification techniques), the desire for the graphics system to handle larger and larger databases is still very pressing.

2.1.3 Display Devices

The desire for displaying more pixels is pressing as well. The “Holy Grail” of display devices is a device that can replicate real images to the same extent as the human eye can sense them. There is almost no current display technology that can provide a practical “true” 3D display (with objects displayed at different depths of focus), and thus we consider only images presented at a single depth of focus. Imagine for a moment a display device that surrounds a user with pixels that are too small to be discerned. Given that human visual acuity (near the fovea) is estimated at one arc minute [DOEN85], a few calculations tell us that we are talking about nearly 200 million pixels.

One current approach on the way towards this limit is the CAVE [CRUZ93]. This system uses projection CRTs to cover 3 walls and the floor of a room with computer-generated stereo imagery. With current graphics systems, this amounts to five million pixels, while projector technology allows for at least 16 million [LACR92]. A similar approach is being employed in “the office of the future,” [RASK98], where multiple projectors are used to make every surface a part of the display.

A more compact approach to immersive display technology is the head-mounted display, or HMD. Current technology has already achieved the million-pixel-per-eye mark [LACR94] and is expected to go much further. Kaiser Electro-Optics has proposed (for a DARPA-sponsored project) to create an immersive HMD system with 4.6 million pixels per eye [DEFO96].

Even for applications that do not demand immersion, the number of pixels can be large. For many years, the state of the art in CRT display technology has been 1.25 million full-color pixels on a twenty-inch diagonal display. Proposed HDTV standards aim at about two million pixels. What is really desirable on such a display is the same resolution one can get on paper: approximately eight million pixels for a start (300 dots per inch on an 8.5x11 inch page). New technologies are slowly making this a reality [WERN93].

2.2 Basic Rendering

Before we examine parallel rendering, we start by taking a look at traditional serial rendering. We note that while there are many different approaches that can be used to render images, there is typically only one choice that is applicable for real-time image generation. This is usually referred to as the standard or traditional graphics pipeline, which has the following major steps:

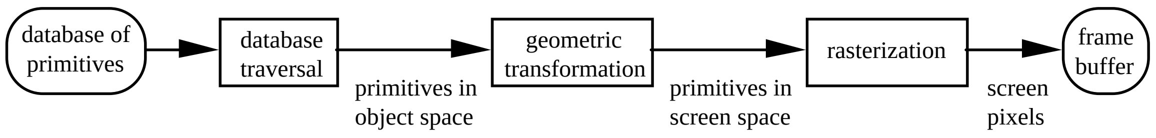


Figure 2.1 Basic Graphics Pipeline

At the start of the pipeline, we have a model consisting of hundreds of thousands of geometric primitives represented in a model coordinate system. The model may include a set of texture images to apply to the primitives' surfaces. In addition, we must have information about the choice of viewing location and direction as well as other information that affects the generated image (such as lighting information). At the end of the pipeline, we have the millions of pixels that make up the desired image. For a detailed discussion on the various steps, please refer to [FOLE90]. For the purposes of this dissertation, we generalize the pipeline into only the following steps: traversal, geometric transformation, and rasterization.

Traversal is typically considered the first stage in the pipeline. It begins with the process of examining the entire graphics database and deciding which parts must be passed on to the rest of the pipeline. Thus, some high-level culling can be done here. Traversal may also involve converting primitives from a specific model-format to a format that is desired by the rest of the pipeline.

The traversal process may either be embedded within the graphics application, or it may be considered a part of the graphics system itself. In the former case, the application keeps track of the model database and feeds a stream of primitives to the graphics system. This is referred to as "immediate-mode" operation. In the latter case, the graphics system itself stores the model database, allowing the application to make changes or "edits" upon it. This is referred to as "retained-mode" operation, which is the mode of operation that this research focuses upon. Immediate mode will receive attention in section 4.7.

Geometric transformation involves a set of floating-point calculations that, among other things, transforms the model-space coordinates of the vertices of the primitives into a set of viewing-space coordinates. In addition, several other calculations may be performed in order to prepare the primitives for rasterization. These may include computations such as normal-vector transformation, texture-coordinate transformation, or lighting-vector calculation. These are all floating-point computations that are suited to general-purpose microprocessors.

Rasterization is the process of taking a primitive defined by a set of viewing-space coordinates and raw surface characteristics and generating a set of pixels in image space to represent that primitive. The source data may also include a texture image that will be mapped over the surface of the primitive and used to modulate the color or other surface properties of the primitive. The amount of work required to rasterize a given primitive depends upon the on-screen size of the primitive. A given primitive may be smaller than a single pixel or it may occupy the entire screen. The average size tends to vary with respect to the overall model complexity (larger models have “smaller” primitives). Sizes from ten to fifty pixels are fairly common.

The amount of rasterization work is also affected by the amount of anti-aliasing desired. Anti-aliasing refers to the correct filtering of a rendered image by taking into consideration the color contribution of each primitive at the sub-pixel level. Anti-aliasing is often done by super-sampling, where one computes multiple color samples at different places within each pixel and then correctly averages these samples together to produce the final pixel color. If four samples are computed per pixel, then four times as much rasterization work is required, eight samples require eight times the work, etc.

Parallel rasterization is a term with various meanings. Typically the term is used to refer to systems that compute, in parallel, color values for multiple pixels (or multiple samples) for a given polygon. This is a level of parallelism that has been in use for a long time. For truly high performance, a system must rasterize multiple *polygons* in parallel, using parallel processing for each polygon. This is the meaning that we will use when talking about parallel rasterization.

In order to reach the necessary level of performance for the applications mentioned in the previous section, a graphics system must have very high performance at all these major stages of the pipeline. Given the processing demands of the applications and the level of performance of current processors, this implies that parallel processing must be used for both the transformation and rasterization pipeline stages. Thus in the next section we examine the ways to implement this kind of parallelism.

2.3 Fully-Parallel Rendering Architectures

Consideration of the traditional rendering algorithm will reveal that it is essentially a sorting algorithm [SUTH74]. Consider each output pixel as a storage bin. Each input primitive must be sorted by placing its contribution into the right set of bins. The sorting really happens along three dimensions: one must choose the correct row and column to select the correct bin (pixel), and then one must choose the correct depth within the pixel in order that the contribution is added correctly.

Given this insight and an understanding of the graphics pipeline, one can form a taxonomy of parallel rendering architectures based upon it. The taxonomy is based upon the fact that the required sorting can be done at different places along the graphics pipeline. The sorting can in fact happen either before or after the transformation or rendering stages, leading to the possibilities of sort-first, sort-middle, and sort-last [MOLN94]. In each case, the items being sorted vary: sort-first sorts object-space primitives, sort-middle sorts screen-space primitives, and sort-last sorts pixels or samples.

We now examine each of sort-first, sort-middle, and sort-last, including their major limitations and how they respond to the application challenges mentioned in the introduction. In the following descriptions, we consider a framework of an application host computer working with a graphics computer subsystem. The latter consists of many parallel processors working to produce the desired images in real time. We assume that the display database is initially partitioned and distributed among all the graphics processors.

2.3.1 Sort-First

In sort-first, each graphics processor is assigned a portion of the screen (consisting of one or more screen “regions”) to render. First, the primitives are classified according to the current viewing parameters. This is an initial transformation step to decide to which processors the primitives actually belong. The decision is typically based upon which screen regions a primitive's bounding box overlaps. During classification, primitives are redistributed to make sure that all processors have the complete set of primitives that fall in their respective regions.

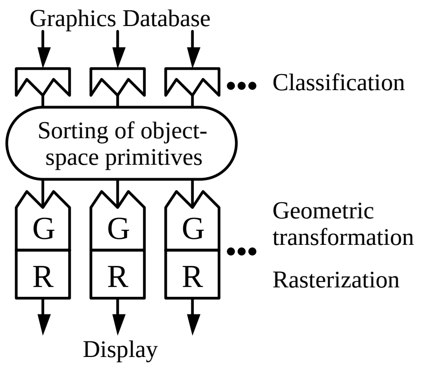


Figure 2.2 Sort-First Graphics Pipeline

Following classification, each processor performs the rest of the transformation and rasterization steps for all of its resulting primitives. This point deserves some emphasis: after classification, each processor can work independently to completely render its portion of the screen. Since each processor has tightly-coupled transformation and rasterization sub-systems, this allows much flexibility in terms of the rendering algorithms that may be used. After rasterization, the finished pixels are sent to one or more frame buffers to be displayed.

Since sort-first depends entirely upon screen-space subdivision to partition the workload, it requires appropriate load-balancing to deal with uneven distributions of primitives across the screen. Since the load-balance is so much at the mercy of the screen-space primitive distribution, load-balancing is more problematic for sort-first than for either of the alternative architectures (though this issue affects sort-middle almost as much). Using screen-space subdivision to partition the work also places a cap upon the number of processors that can be used in the system. This point is reached when adding another processor would increase the overhead and redundancy due to the partitioning more than it would increase the performance due to adding more processing power.

Using sort-first in combination with retained mode allows an important optimization to be made. The optimization is one that takes advantage of frame-to-frame coherence, an inherent quality of interactive graphics applications. In such applications, the viewpoint usually changes very little from frame to frame, and thus the on-screen distribution of primitives does not change appreciably either. Using retained mode with sort-first means that the resulting database distribution from one frame can form the initial distribution for the next frame. Thus the primitives will migrate from processor to processor, and they will only need to be communicated as they cross over boundaries between different processors' screen regions.

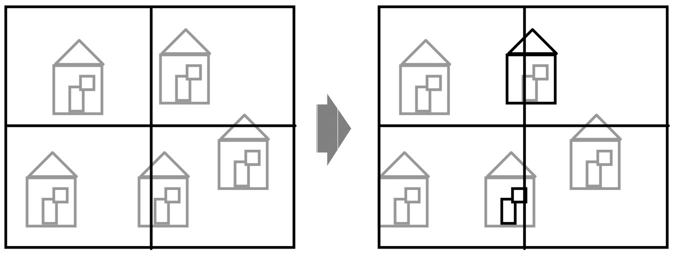


Figure 2.3 Frame-to-frame Coherence
After the left frame, only the highlighted primitives must be sent to draw the right frame.

Thus far, there is no known implementation of sort-first that uses this optimization. The only systems that vaguely fit into the sort-first model are certain multiple screen systems where there is a complete hardware rendering pipeline associated with each screen [SCHU80, LATH85, CRUZ93, SCHW98]. Such systems will either use an immediate-mode interface to feed each pipeline, or else each pipeline will have its own copy of the database to traverse. Given that such systems have limited traversal performance and offer limited or no load-balancing capability among the different pipelines, we offer them no further consideration.

The only other documented sort-first system in evidence is [COX95]. Cox created a sort-first implementation of the Renderman REYES graphics pipeline on the Connection Machine CM5. Because Renderman is aimed at realistic rendering, this was not an interactive system. It was based upon an immediate-mode input stream, and thus did not make use of frame-to-frame coherence as mentioned above. This implementation also used only static (interleaved) load balancing, which (as we shall see later) limited its performance.

2.3.2 Sort-Middle

In sort-middle, there is a set of transformation processors and a set of rasterization processors. The two jobs might actually be done on the same physical processor, or there may be a separate type of processor for each job, but in either case there are still two logically separate sets of processors. Each transformation processor completely transforms its portion of the primitives. The resulting primitive information is classified by screen region and sent to the correct set of rasterization processors. The transformation processors will typically use the screen-space bounding box of a primitive to decide which rasterization processors must receive it. After a rasterizer finishes rasterizing the primitives in its region, it sends the final pixels to the frame buffer(s).

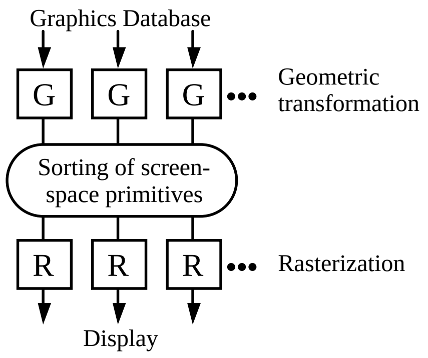


Figure 2.4 Sort-Middle Graphics Pipeline

In contrast to sort-first, the original distribution of primitives is always maintained on the transformation processors. For each frame, all of the on-screen transformed primitives must be routed to the correct set of rasterization processors. This communication issue presents some limits to sort-middle's scalability, as we shall discuss later.

As with sort-first, the screen-space partitioning of the rasterization work requires careful load-balancing and presents additional scalability issues. With sort-middle, however, the load-balancing concerns of the geometric transformation stage can be considered independently of those of the rasterization stage. In this vein, sort-middle enjoys another advantage over sort-first and sort-last: it offers the possibility of easily balancing the transformation performance with respect to the rasterization performance by varying the number of processors at each stage.

Because the geometric transformation stage is not as closely tied to the rasterization stage as in the other architectures, sort-middle may impose certain limitations on what rendering algorithms may be used. The loose coupling in the middle of the pipeline can limit feedback from the rasterization stage back to the transformation stage. This can make certain visibility culling algorithms either less efficient or infeasible with sort-middle.

Because of the way it builds upon traditional graphics pipelines, sort-middle is a fairly natural architecture that has resulted in many implementations. Some examples are [AKEL93], [CROC93], [DEER93], [ELLS94], [FUCH89], and [WHIT94]. Some of these implementations are software algorithms that run on general purpose MIMD systems. Others consist of custom-built hardware designed specifically for graphics rendering. For a complete study of sort-middle systems, please refer to [ELLS96].

2.3.3 Sort-Last

For sort-last, each processor has a (nearly) complete rendering pipeline and produces an incomplete full-area image by transforming and rasterizing its fraction of the primitives. These partial images are composited together, typically by depth sorting each pixel, in order to yield a complete image for the frame buffer(s). The composition step requires that pixel information (at least color and depth values) from each processor be sent across a network and sorted along the way to the frame buffer.

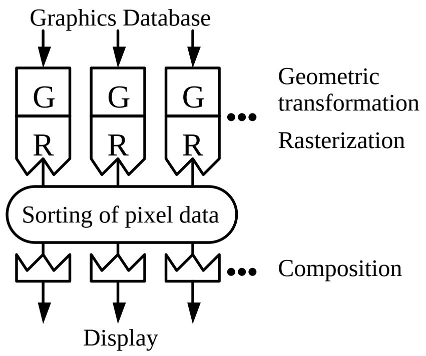


Figure 2.5 Sort-Last Graphics Pipeline

Load-balancing on sort-last is usually much less dependent upon the screen-space distribution of the primitives than the other architectures. Assuming that each processing node has a full-screen image buffer (sort-last full), then the load-balancing is affected only by the number and type (and perhaps size) of onscreen primitives on each processor, independent of where they fall. Some sort-last designs utilize a tiled rendering technique (sort-last sparse), requiring tile-by-tile synchronization across the machine during the rendering phase. With such designs, the distribution of the primitives across the different tiles can create load-balancing problems unless proper consideration is paid to this issue.

The sort-last composition process can be carried out very efficiently with the proper communications hardware. An efficient design would stream the pixel information out from each processor in a coordinated fashion. When a given pixel value arrives at a processor, the communications hardware on that processor can compare the incoming pixel with the corresponding pixel produced locally and forward the correct one on to the next processor. With this setup, the time to composite all the pixels in the system is nearly the same as the time to send all the pixel information from one processor to another.

Note that for sort-last to produce a high-quality, anti-aliased image, the process is complicated due to the late sorting process. Producing a correctly anti-aliased image with sort-last requires that every separate pixel sample be sent and composited during the sorting stage. Thus the amount of communication required is directly affected by the degree of anti-aliasing desired.

Finally, because visibility is not decided until after the composition stage, sort-last places limitations on the kinds of rendering algorithms that may be used. The choice of algorithms available for rendering transparent polygons becomes limited, for example, and visibility culling algorithms are less useful on sort-last.

The sort-last architecture is covered well in [MOLN91] as well as [MOLN92], [EYLE97], and [NISH96]. PixelFlow is a high-performance hardware implementation of sort-last; more details about this system may be found in chapter 5.

2.4 Application Suitability Analysis

Each architecture has its own set of advantages and disadvantages. We have outlined some of these above; for additional comparisons, one may refer to [MOLN94]. In this section, we shall consider each architecture with regard to how well it can handle the growing application demands for large primitive databases with high-resolution output.

2.4.1 Sort-Last

Sort-last is a very promising architecture as it offers excellent scalability in terms of the number of primitives it can render interactively. One can easily add more processors to a sort-last system in order to handle greater numbers of primitives. With appropriate communications hardware, such scaling has no significant effect upon the composition process or its bandwidth requirements. There are of course limits to such scaling, these having to do with database distribution issues and the coordination of all the processors; these factors are only likely to be a problem for very large numbers of processors.

The output side of the pipeline is another story. Sort-last does not scale very well with respect to the number of pixels to be rendered. Every pixel rendered on each board must be sent over the composition network, and for each pixel there is a significant amount of information to send. Minimally, each pixel must include both depth and color information; this is approximately double the amount of data versus sending the color alone. However, as pointed out earlier, anti-aliasing adds an additional multiplicative factor to the amount of data that must be sent. Thus eight-sample anti-aliasing increases the amount of pixel data to be sent by a factor of eight. There may exist ways to reduce this penalty, but the alternatives examined in [MOLN91] added other complications or tradeoffs in visual quality.

To consider the limitations imposed by sort-last pixel communications, we present the table below. For this table, we assume that 32 bits of data are used for color information and 32 bits are used for depth (Z) values, resulting in a total of eight bytes per pixel composited. ("AA" is the anti-aliasing factor.)

Resolution	AA	Data to composite	Bandwidth @ 30 Hz	@ 60 Hz
1024x1024	16	128 MB	3.8 GB/s	7.7 GB/s
2048x2048	16	512 MB	15.3 GB/s	30.7 GB/s
4096x4096	16	2048 MB	61.4 GB/s	122.9 GB/s
8192x8192	16	8192 MB	245.8 GB/s	491.5 GB/s

Table 2.1 Sort-Last bandwidth figures

For comparison, the state-of-the-art UNC/Hewlett Packard PixelFlow system offers a point-to-point composition network bandwidth of 6.4 GB/s [EYLE97]. We can see that this is completely inadequate for very high resolution output, and that an order of magnitude increase in bandwidth is necessary just to get started into the territory of very high-resolution output. For this reason, we will dismiss sort-last as a candidate for these applications. Aside from bandwidth issues, the late-visibility issues mentioned in section 2.3.3 may also get in the way of using sort-last.

2.4.2 Sort-Middle

In contrast to sort-last, sort-middle only needs to send finished pixels from the rasterizers to the framebuffers. Thus, for the cases in the table above, only 1/32th of the bandwidth shown is necessary for sort-middle pixel traffic. This suggests that pixel communications issues do not stand in the way of using sort-middle for very high-resolution output applications.

We then turn our attention to the issue of primitive communications. With sort-middle, every on-screen primitive must be sent from some transformation processor to some rasterization processor. This presents a many-to-many communications pattern that puts a limit on the number of processors one can add to a sort-middle machine. This limit is only reached when the primitive communications network is saturated, however. We thus consider the possible amount of bandwidth that this may consume.

To analyze sort-middle, we need to make some assumptions about the data being communicated. We shall assume that all primitives are triangles. We compute the number of bytes per primitive as shown in the table below. The command field indicates that a triangle follows, and we assume that rendering flags are necessary to indicate how the triangle should be drawn.

Data description	Bytes	Total
command field	4	4
rendering flags	4	4
triangle vertices	3 * (4 + 4 + 4)	36
triangle normals	3 * (4 + 4 + 4)	36
vertex colors	3 * (4)	12
texture coordinates	3 * (4 + 4)	24
Total bytes		116

Table 2.2 Triangle components

Given this value, we can now determine the amount of bandwidth that is necessary to redistribute a given number of on-screen primitives at interactive rates. This is shown in the next table.

No. of triangles	Amt. of data	Bandwidth @ 30 Hz	@ 60 Hz
50 K	5.8 M	0.17 GB/s	0.35 GB/s
100 K	11.6 M	0.35 GB/s	0.70 GB/s
500 K	58 M	1.74 GB/s	3.48 GB/s
1 M	116 M	3.48 GB/s	6.96 GB/s
5 M	580 M	17.40 GB/s	34.80 GB/s
10 M	1160 M	34.80 GB/s	69.60 GB/s

Table 2.3 Sort-Middle bandwidths

We can see that as we approach one million triangles per frame, the bandwidth figures start to become a large concern. One can use various techniques to reduce the amount of data being sent, such as making use of triangle strips, display-list caching, and geometry compression techniques [FOLE90, MONT97, DEER95]. However, the use of such techniques will only extend so far.

A better way to reduce the amount of primitive communication would be to take advantage of frame-to-frame coherence. If one can avoid resending primitives that are going to the same processors each frame, one may potentially reduce the communication to a small fraction, allowing for larger databases to be rendered at interactive rates. However, changing the architecture in this manner results in a new architecture: sort-first.

2.4.3 Sort-First

Depending upon the implementation details, one may suggest that sort-first is very similar to sort-middle. Both require some amount of transformation followed by redistribution of the input primitives. Both output regions of completed pixels to the framebuffer. Technically, one may distinguish the two by asking whether object-space primitives or screen-space primitives are redistributed. However, the most interesting difference between the two architectures appears when one considers retained-mode databases. With such databases, sort-first can keep the redistributed database from one frame to the next, making use of frame-to-frame coherence.

Interactive applications generally exhibit a good deal of frame-to-frame coherence. This benefits sort-first since it means that fewer primitives will need to be communicated during each frame. Only primitives that move from one processor's region to another's will need to be sent. The number of primitives that will need to be sent will be inversely proportional to the amount of coherence that exists from one frame to the next. This leads to an interesting point, since the amount of coherence in an application typically increases as the update rate increases. Thus a faster update rate potentially allows for better performance with a sort-first system.

With regard to application suitability, sort-first looks very good. Like sort-middle, only finished pixels will need to be sent to the framebuffers. With regard to primitive communications, it appears that sort-first can do much better than sort-middle, due to its use of coherence. We shall study this aspect more in the next section.

However, sort-first is not without its share of problems. Load-balancing is perhaps one of the biggest concerns: because the on-screen distribution of primitives may be highly variable, much thought must go into how the processors are assigned screen regions. Also, management of a set of migrating primitives is a complex task. Finally, even sort-first places restrictions on the type of rendering algorithms to be used. In the rest of this dissertation, we intend to lay out and solve these major difficulties of sort-first.

2.5 Coherence Study

Because the main advantage of sort-first lies in its ability to take advantage of the coherence of on-screen primitive movement, it is important to analyze this characteristic of applications and determine what kind of savings might actually be achieved. To this end, we performed several experiments and describe the results here.

2.5.1 Experiment Description

The goal of the experiment was to determine approximately what fraction of primitives (vs. the number of primitives on-screen) might have to be sent from one processor to another in a sort-first implementation running an actual application. We will call this value the communications coherence factor. Getting an estimate of this factor will help us to determine the value of sort-first as a rendering architecture. Because sort-first has many implementation issues that are unresolved, the testing was done using a simulation with several simplifying assumptions.

The testing involved two phases. The first was making recordings from actual applications running on UNC's Pixel-Planes 5 graphics system [FUCH89]. The resulting recordings contain a series of viewpoint specifications for each frame rendered while the application was run. The second phase was to take this information and the graphics database files and feed them to the simulation program.

Two different applications were used for the different test cases. “PLB” is a simple program that spins a given database around the screen's vertical axis (named after a graphics performance benchmark from [NCGA92]). “Xfront” is a joystick-controlled visualization program that allows one to manipulate an arbitrary display database. Two three-axis joysticks allow one to easily control rotation and translation.

The simulation program is based upon a framework written by David Ellsworth for his study of sort-middle systems [ELLS94]. The framework provides support for reading in the database file and processing the recording file (see figure 2.6). For each frame it generates a list of primitives transformed to screen space and calls upon a partitioning algorithm to distribute the primitives among the processors. It then calls an evaluator to analyze the resulting distribution. Code was added to implement a sort-first partitioning and to calculate the resulting primitive traffic.

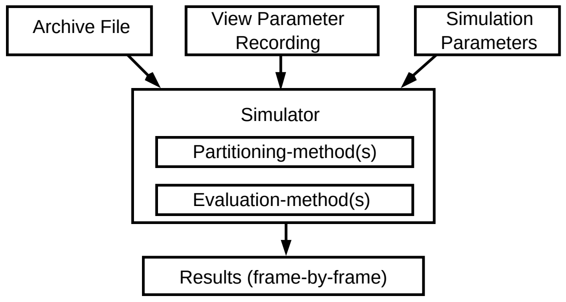


Figure 2.6. The simulation test-bed

The setup for this test was as follows:

- The database is simply a list of polygons (no structure).
- The aspect ratio of the screen is square.
- The screen is subdivided into equal-size square regions with one region assigned to each processor.
- The primitives are initially randomly distributed (the first frame's data are ignored for this reason).
- Primitives are redistributed according to the regions overlapped by their bounding boxes.
- If a primitive falls into multiple regions, the processor at the upper-left region is deemed to be “in charge” of it.

In these particular tests, the screen resolution is irrelevant. The only concern is how many regions (and thus processors) there are. Several configurations of regions were tested:

Image 2.1
PLB: PLB head

Image 2.2
Xfront: Sierra

4x4, 8x8, and 16x16. The experiments produced a series of values, one per frame, representing the communications coherence factor: the total number of primitives being sent divided by the number of primitives on-screen. For the latter value, we use the number of original primitives (i.e., if a primitive appears on two regions, it only counts once). From these figures, we calculate the arithmetic mean, the high value, the standard deviation, and the 95th percentile value.

2.5.2 Experiment Results

For PLB, the database is a scanned model of a human head (see image 2.1 above). The model is placed in the center of the screen and spun at 4.5 degrees per iteration around a vertical axis through its center (as in [NCGA92]).

Dataset: PLB head: 59,592 polygons, 80 frames

# of regions:	4x4	8x8	16x16
mean	4.06 %	8.80 %	18.07 %
high	5.19	10.30	20.80
std-dev	0.54	0.70	1.05
95-percentile	5.07	9.92	20.06

Table 2.4 PLB/Head Results

For Xfront, the model is a terrain database of a section of the Sierra Nevada mountains (image 2.2). The model undergoes a series of rotations and translations; after each manipulation, the viewpoint is abruptly reset in order to recenter the model.

Dataset: Sierra: 162,690 polygons, 234 frames

# of regions:	4x4	8x8	16x16
mean	3.17 %	6.08 %	11.51 %
high	98.07	102.26	107.38
std-dev	7.68	9.53	11.76
95-percentile	5.04	10.36	20.53

Table 2.5 Xfront/Sierra Results

Increasing the number of regions increases the percentage of primitives that have to be communicated. This is fairly obvious, since increasing the number of region borders will increase the chance of a primitive crossing them. The averages reflect this very well.

The high values for the Xfront/Sierra results exceed 100% for two of the cases. These large values resulted from the abrupt transitions (there were three transitions in this sequence). The values exceeded 100% for two of the cases since primitives that fall into multiple regions may need to be sent more than once.

The percentiles perhaps are of greatest interest, since they leave out the discrepancies resulting from the abrupt transitions. They show that for these applications, 95% of the frames to be rendered require communication of only about 20% of the primitives or less. This experiment shows that temporal coherence can provide sort-first with a dramatic savings in the amount of communication that is necessary. The amount of savings is related directly to the nature of the application and the number of regions the screen is divided into. As the number of regions increases, the amount of savings decreases in proportion, but remains quite substantial even for fairly large numbers of regions.

2.6 Conclusion

We have taken a look at the space of application areas and seen an existing desire for an interactive graphics system that can deal with very large numbers of input polygons and very large numbers of output pixels. We have examined the space of graphics architectures that can handle these needs. The space includes the fully-parallelized graphics architectures with the categories of sort-first, sort-middle, and sort-last. A look at sort-last shows that it cannot satisfy the demand to generate very high resolution output images. A look at sort-middle shows that while it might be able to meet the desired application demands, primitive communication appears to be a limiting factor. Finally, a look at sort-first reveals it to be the most likely candidate, since it does not have the pixel communications bottleneck of sort-last, and its use of frame-to-frame coherence means it requires only a fraction of the primitive bandwidth that sort-middle does.

3. Load Balancing Sort-First

Having presented an overview of sort-first issues and challenges, we now begin to look into finding solutions. The first issue, load-balancing, centers around efficiency. In order for sort-first to be an efficient parallel architecture, it must be designed such that the various processors each receive an approximately equal amount of work. Primarily, for sort-first, this means that the processors should receive a similar number of primitives to render (assuming for the moment that the primitives are similarly-sized). Since this distribution is determined by the screen partitioning, we will first focus on that aspect of the load-balancing problem. However, there are other factors that we must consider as well, since the overall work involved in rendering an image involves more than simply transforming and rasterizing primitives.

In this chapter we begin by considering the various tradeoffs involved with load balancing. We then examine several load-balancing algorithms and present a new algorithm called mesh-based adaptive decomposition ("MAHD"). We explore the various facets of these load-balancing algorithms, and we then examine their performance, first through simulation and then via software sort-first implementation. Finally, we examine other practical issues of a sort-first rendering system and how they are impacted by load-balancing considerations.

3.1 Basic Load-Balancing Issues

In order to evaluate different load-balancing methodologies, we need to understand the tradeoffs. These include the degree of load-balancing achieved, the amount of *direct overhead* from the load-balancing algorithm itself, and the amount of *indirect overhead* resulting from the work-load partitioning given by the load-balancing solution.

The better the overall load-balance, the less time that processors will spend idling while they wait for other processors to finish. Less wasted time means better efficiency and therefore better performance. We measure load-balancing efficiency by calculating the utilization, which is the average amount of work done by all the processors divided by the maximum amount of work done on any given processor. This can also be calculated as the total amount of work done divided by the product of the maximum work-load times the number of processors. For sort-first, the work load of a given processor is typically expressed as the number of primitives each processor must render. However, for various reasons, some primitives may take longer to render than others. These differences must be considered in the overall load-balancing equation.

The direct overhead of a load-balancing algorithm is entirely dependent upon the actual load-balancing algorithm chosen. Some algorithms may have no direct overhead; with these the screen partitioning and assignment is chosen statically. Other algorithms may work to gather primitive distribution statistics and use this to decide upon the partitioning dynamically. In such cases, all of this extra work contributes to the direct overhead. Direct overhead may also result from synchronization delays due to any synchronization points added by the load-balancing algorithm.

The indirect overhead from load-balancing is the extra work that results due to the screen subdivision. This work comes from primitives that overlap multiple regions and from primitive communication. This overhead is in fact intrinsic to sort-first; however the amount of the overhead is greatly affected by the subdivision chosen for each frame.

Primitives that overlap multiple regions will need to be processed by all of the processors that deal with those regions. Each copy of the primitive must be fully transformed, clipped, and rasterized. This results in decreasing efficiency as more and more primitives overlap more and more regions. We characterize this decrease in efficiency by examining the overlap factor, O. This variable represents the average number of regions overlapped by each primitive. Thus an overlap factor of 1.5 means that, on average, each primitive overlaps one and a half regions, or that there is a 50% increase in the number of primitives that must be rendered (due to overlap). There is also a formula that can be used to calculate O for certain situations (derived in [MOLN91] and [MOLN94]). Its assumptions are that primitives have equally-sized bounding boxes, and that primitives may fall anywhere on the screen with equal probability. The formula is:

$$O = ((W+w)/W)((H+h)/H) \text{ where}$$

W, H = width and height of a screen region
w, h = width and height of a primitive's bounding box

This formula gives us some useful information about what types of partitioning strategies are likely to have lower overhead than others. We will examine some of the implications of this formula in the next section.

Primitive communication is another indirect overhead factor that is affected by the load-balancing strategy. For example, the more the screen is partitioned, the greater the chance of a primitive crossing between regions and thus needing to be communicated. In addition, if the load-balancing strategy moves the region boundaries, this may also lead to greater amounts of primitive communication. This type of overhead must be carefully evaluated when formulating a screen partitioning strategy.

3.2 Screen-Region Assignment

The choice of strategy for mapping screen regions to processors has a critical impact on the performance of a sort-first system. A simple strategy, such as dividing the screen into as many equal rectangles as there are processors and assigning them one-to-one, can result in severe load-balancing problems. If the greatest concentration of primitives happens to fall into a single region, the parallel advantage of the system will be lost as the non-busy processors wait for the overloaded one to do its job. There are various approaches one can take to partitioning the screen such that the processors have nearly equal loads. In this section we examine some of the choices and study what their implications are.

We broadly categorize the screen partitioning methods as either static or adaptive. Static methods choose a fixed screen partitioning and processor assignment, while adaptive methods vary these parameters during run time. We will examine each category in turn.

3.2.1 Static Methods

Since static assignment does not provide any active means of load-balancing, a passive strategy must be used. The general approach is to divide the screen into more regions than there are processors and assign the regions to the processors in an interlaced fashion (figure 3.1). The idea is that if the screen is divided finely enough, each processor will have similar portions of both the populated and the sparse areas of the screen, and thus they should have nearly equal loads.

1	2	1	2
3	4	3	4
1	2	1	2
3	4	3	4

Figure 3.1 Static Region Assignment
The screen is subdivided and regions are assigned to 4 processors.

One problem with this approach is that there is still the possibility that a high concentration of primitives will fall into one region, no matter how small the regions are. However, in practice, this should not happen very often. A well-designed application will use level-of-detail control and various types of culling algorithms to prevent this possibility [CLAR76, COSM81, ZHAN97].

Another obvious drawback is that increasing the number of regions per processor increases the amount of overhead. Still, the simplicity of this approach makes it worth studying. With that in mind, several concerns come up: how should the regions be shaped, how many of them should there be, and how should they be assigned?

The obvious choices for region shapes are horizontal strips, vertical strips, or rectangles. For several reasons, rectangles are the preferred choice. Consider a simulation where objects disappear into the horizon. Making strips parallel to the horizon would make load balancing difficult since the majority of objects may fall into the strip containing the horizon. Making vertical divisions would ease the load-balancing problem, but this solution would create more primitive traffic as the viewer turns. It would also face the same problem as horizontal strips should the viewpoint roll.

Also, both the overlap factor and primitive traffic are proportional to the total linear length of the region boundaries. This is intuitively easy to see, since having more boundaries will provide more opportunity for primitives to cross them. By examining the overlap-factor formula (see above), we can see that reducing O means making both the width and height of regions as large as possible (shown analytically in [ELLS96]). All of the preceding arguments then call for making the regions as square as possible.

The number of regions per processor is commonly called the granularity ratio. For overhead considerations we would like to keep this number as small as possible, whereas having a larger granularity ratio would generally improve the load-balancing performance. The ratio should be high enough to separate dense clumps of primitives into at least as many regions as there are processors. If such a clump of primitives occupies 1/16th of the screen, then a granularity ratio of at least 16 is needed to partition this clump. Since this clumping behavior is very scene dependent, determining the ideal granularity ratio to use is not easy, as many researchers have discovered [WHIT92, ELLS93, COX95]. We examine the effects of the granularity ratio in the experiments described below.

The question of region assignment can perhaps be addressed more simply. The goal is to scatter each processor's regions uniformly over the screen. This can be achieved readily enough using regular rectangular interlacing. For this method, one first establishes a rectangular "footprint" of processor regions, one region assigned to each processor. One then repeats this pattern over the screen, again in a rectangular fashion, to achieve a number of footprints equal to the granularity ratio. This technique and its considerations are discussed in [ELLS96].

3.2.2 Existing Adaptive Methods

Adaptive region assignment allows a variety of solutions. One approach is to fix the number of regions equal to the number of processors, but vary the shape of each. Another is to partition the screen into regions statically, but dynamically allocate them to processors. Various combinations of the strategies are possible. Adaptive solutions offer the possible benefit of keeping the number of divisions to a minimum, but at the cost of increased overhead and complexity. The goals for developing an algorithm are as follows:

- optimal primitive load-balance across processors
- minimum algorithm computation and communication costs
- minimum number of regions (one per processor is ideal)
- minimum overlap factor
- minimum primitive traffic between frames

One factor that affects several of the goals is the shape of the screen regions. The subdivision algorithm could be simplified greatly by making only horizontal or vertical cuts across the entire screen. However, this would tend to adversely affect the overlap factor and primitive traffic. As mentioned in the discussion of static methods, the ideal shape is closer to square, meaning that both horizontal and vertical divisions are necessary. Forcing the divisions to go all the way across the screen is not necessary, however. One can have more flexibility in placing the divisions, though the cost of this flexibility may be more difficulty in determining which primitives overlap which regions.

We turn our attention toward several proposed algorithms and consider them in light of the above goals. From there, we develop an algorithm that appears to combine many of the beneficial ideas from these proposed ones.

3.2.2.1 Roble's Method

Roble describes an algorithm [ROBL88] that starts with a standard rectangular decomposition (i.e., the screen is divided into as many equal rectangular regions as there are processors). According to the number of primitives in each region, lightly loaded regions are combined and highly loaded regions are split in half and assigned to the processors freed by the combining. The main problem with this algorithm is that there is no information on exactly how to divide the highly loaded regions, and thus the resulting splits may add little benefit if most of the region's primitives fall on one side of the split.

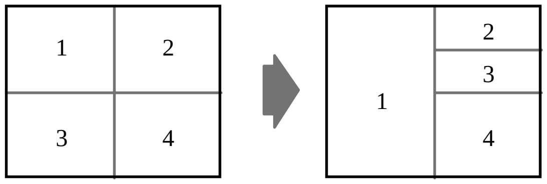


Figure 3.2 Roble's Method
Regions 1 and 3 are combined; processor 3 helps with original region 2.

3.2.2.2 Whelan's Method

Whelan proposes an algorithm known as median-cut [WHEL85]. Median-cut splits the screen into subregions based upon the distribution of the centroids of each polygon. The cuts recursively divide the screen (along the longer dimension of the given region) until the number of regions equals the number of processors. This approach requires the calculation of primitive centroids and the sorting of these centroids along both horizontal and vertical dimensions. The primitives must then be partitioned with each cut made. As a result, this algorithm requires a fair amount of calculation overhead. Also, since the algorithm is based only upon the centroids of primitives, it is not sensitive to the on-screen area of the primitives.

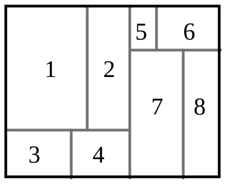


Figure 3.3 Whelan's Method
According to primitive centroids, the screen is recursively subdivided.

3.2.2.3 Whitman's Method

Another strategy is Whitman's top-down decomposition [WHIT92], which starts by tallying up primitives based upon how their bounding boxes overlap a fine mesh. A unit is added to each mesh cell that the bounding box overlaps. After all of the tallying, adjacent mesh cells are combined and summed hierarchically to form a tree structure (i.e., adjacent cells are combined, then adjacent pairs of cells are combined, etc.). The tree is then traversed top-down by splitting the region with the most primitives in half each time. Because this method does not directly try to make regions with equal numbers of primitives, the resulting regions may have largely varying numbers of primitives. To compensate for this, Whitman subdivides beyond the number of processors, creating ten times as many regions as there are processors. Dynamic task assignment is used to even out the processor load balance. However, the resulting finer granularity of the regions means a higher overlap factor, resulting in more overhead for this method.

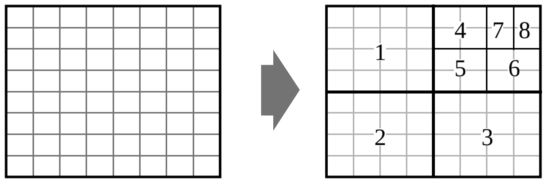


Figure 3.4 Whitman's Method
Mesh cells are combined hierarchically; the screen is split by traversing the hierarchy.

3.2.3 New Adaptive Method: MAHD

Combining some of the above ideas leads to the algorithm presented here. This adaptive algorithm also uses a fine mesh to tally the primitives. For each cell covered by a given primitive's bounding box, we tally an amount proportional to the rendering costs for that primitive (see section 3.4.2 for more about this). Once all the primitives have been counted, the cells are added up to form a summed area table [CROW84]. Finally, the screen is divided along cell boundaries using a hierarchical approach similar to that of median-cut. The summed-area table allows a quick binary search operation to determine the location of each cut. Also, the algorithm allows for using a number of processors that is not a power of two by choosing appropriate split ratios rather than always dividing regions equally. Hereafter, we refer to this algorithm as the mesh-based adaptive hierarchical decomposition algorithm, or MAHD for short. In section 3.4, we shall examine the MAHD algorithm in detail.

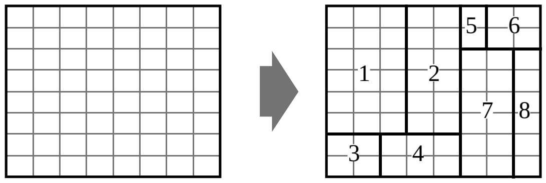


Figure 3.5 The MAHD Method
Mesh cells are used to accelerate Whelan-like subdivision.

3.3 Database Issues vs. Load-Balancing

Thus far we have mainly examined load-balancing as it concerns partitioning the work of rendering the on-screen primitives. However, we have thus far not considered the work associated with finding out which primitives go to which processors and making sure that they get there. For various reasons, the work involved here is not as straightforward as it may seem. This work is tied in to the database handling issue, which will be discussed in detail in the next chapter. For the purposes of this chapter we shall consider only some of the basic database issues as they related to load-balancing.

3.3.1 Mastership

We begin with the issue of making sure that primitives are sent to the correct processors. If only one processor has a copy of a given primitive, then it is obviously up to that processor to make sure that the primitive gets distributed to the correct set of processors needing it (according to where the primitive falls on the screen). When a primitive overlaps multiple screen regions, then several processors may need a copy of the primitive. In the case where multiple processors have a copy of the same primitive, there must be a consensus on which processor will make sure that the primitive gets distributed appropriately.

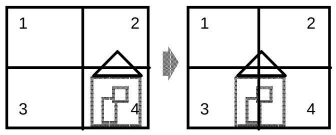


Figure 3.6 Primitive mastership
Which processor is responsible for transmitting the highlighted primitive?

We solve this problem by designating a “master” processor for each primitive. The master processor will make sure that the primitive gets distributed to the proper set of processors. The other processors that have a copy of the primitive just for rendering purposes will be referred to as “slave” processors for that primitive. Similarly, a processor will have some master primitives and some slave primitives. Slave primitives are kept on a processor as long as they remain in that processor’s region; once they move away they can be discarded.

The next issue that arises is what should happen when a primitive moves away from the screen region for which its master is responsible for rendering. One possible answer is that the mastership of that primitive should be transferred to a new processor, a processor that is rendering a region that the primitive does overlap. We will call this strategy “dynamic mastership.” The alternative is to keep mastership responsibilities fixed once assigned. This strategy will be called “static mastership.” We will review them each in turn.

3.3.2 Dynamic Mastership

If a primitive leaves the region of its master processor and falls upon several new processors’ regions, the current master must decide which of those processors will become the new master. There are several alternatives for deciding which processor should be the new master. Some choices include:

- the lowest (or highest) numbered processor in the group
- the processor responsible for the region where the upper-left corner (or other corner) of the primitive's bounding-box lies
- the processor where the primitive’s midpoint falls
- the processor responsible for the region that the primitive's first (or *n*-th) vertex falls in
- a random processor from the group

For the most part, the choice of alternative seems to be simply a matter of convenience: whichever convention is easiest to implement. Still, some caution is necessary; some of the alternatives will tend to favor certain processors over others, thus unevenly distributing the mastership duties. The first two alternatives may have this problem, while the others do not. Assuming that primitives are small with respect to the regions, this effect may be fairly negligible.

With dynamic mastership also comes the issue of what to do with off-screen primitives. When a primitive leaves its master processor’s region and no longer falls in any region what should happen to it? One option is for the primitive to remain on its old master processor. However, this introduces two problems: load-balancing and overflow. In general, off-screen primitives still must be transformed in order to determine when they come back into view. Since primitives generally go out of view by leaving the edge of the screen, the processors responsible for handling these regions would soon get loaded down more heavily with off-screen primitives than the processors responsible for the interior regions of the screen. In fact, if all the primitives were to move off the same edge of the screen, certain processors could end up with the majority of the primitives in the database, possibly overflowing their memory capacity. Since the primitives should not (or perhaps cannot) all go to a few processors, it is apparent that they have to be sent away somewhere eventually once they have gone off-screen. Where should they be sent?

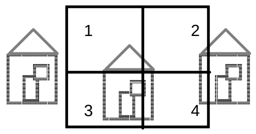


Figure 3.7 Off-screen primitives
Which processor is responsible for transmitting them?

One could send off-screen primitives to “neighboring” processors, since this might offer a communications advantage; however these processors might then become overloaded themselves, unless they also send the primitives away. However, sending off-screen primitives more than once appears counterproductive.

A better solution would be to send them to a processor that is under-loaded. The immediate problem with this approach is that it requires processors to distribute information about their primitive loads. There is the further problem that several processors may all decide to send their off-screen primitives to the same under-loaded processor, overloading it in the process. Additional logic would be required to prevent this.

The best approach is probably to have each processor attempt to distribute its freshly off-screen polygons evenly among the entire set of processors. This can be achieved by either round-robin or stochastic means. Assuming that the processors are fairly evenly loaded to begin with, this approach should maintain an even load balance. Some consideration to minimize the total number of messages sent by each processor may help reduce communications overhead. This redistribution approach appears the most efficient since off-screen primitives only need to be sent once and very little work is required to decide where they must be sent.

We can also examine how off-screen primitives might interact with different load-balancing methods. With static load-balancing, the flexibility in dealing with off-screen primitives can be used to compensate for the lack of flexibility in dealing with on-screen primitives. Processors that have few on-screen primitives may be assigned additional off-screen primitives to deal with. Again, this would require processors to distribute their load-balance information. One must be cautious with this approach, however, since sending primitives unnecessarily, just for the sake of load-balancing, can be counter-productive.

With adaptive load-balancing methods, we can take another tack. Such methods can handle uneven balances of off-screen primitives by modifying the balance of the on-screen primitives to compensate. Thus if all the processors send a count of their off-screen primitives at the same time that they send information about the on-screen primitive distribution to the gathering processor, then both types of information can be used in determining the screen-partitioning. Doing this appropriately requires a measure of how much time it takes to classify a primitive versus the time required to fully render one. This coefficient can then be used to adjust the target number of primitives for each processor to have as a result of screen subdivision.

Dynamic mastership also introduces another problem concerning database editing. If the application program should need to edit primitives, the application processor must be able to quickly locate the desired target primitives among the graphics processors. If the primitives are constantly moving around, then quickly locating them becomes more difficult.

In order to locate the primitives at all, we consider various possible solutions. One solution is for each processor to remember where it sends away its master primitives. With this method, the host can send edit messages to the initially-assigned master processor for a primitive, and the message can be propagated, if necessary, by following a chain of pointers to where the primitive was sent. Depending upon the application run-time properties, however, this can result in several messages being sent for each edit operation.

Another solution is for each processor to notify the original master of a primitive (or just notify the host) each time it sends the primitive away. This can result in excessive communication congestion at the host. A third solution is to have the host simply broadcast all primitive editing commands to all the graphics processors. If broadcast communication is cheap, then this is perhaps the best solution. Regardless of which solution is chosen, the host only needs to send messages indicating the *identity* of primitives that are being changed. The master processors of such primitives just mark those primitives for deletion. The actual primitives that will replace the old ones only need to be sent to one new master processor each.

3.3.3 Static Mastership

Dynamic mastership adds the challenges of dealing with off-screen primitives and primitive editing. We address these problems by considering an alternate strategy for handling mastership: static mastership. If we follow this approach naively, we find that we have approximately doubled the amount of primitive transformation occurring within the system. This is due to the fact that, most of the time, a processor’s master primitives will actually fall in other processors' screen regions, and thus there will be nearly as many slave primitives as there are master primitives in the system.

To reduce this extra transformation burden, we introduce primitive grouping. With this strategy, we put multiple primitives together into a group and keep track of a simplified bounding volume around each group. Rather than applying the concepts of master and slave to each individual primitive, we apply them to each group. The benefit is that the work of mastership is reduced to only a fraction of the number of primitives.

A master processor for a group will examine which screen regions are overlapped by the bounding volume of the group and send that group of primitives to the appropriate slave processors. The slave processors transform the primitives within the group and render them as necessary. The group is kept on the slave processor until the entire group moves away from that processor's region.

Primitive grouping introduces new problems of its own.

- how do we choose which primitives to group together?
- how many primitives do we put in each group?
- what kind of bounding volumes and bounding tests do we use?
- what overhead results from using groups?

To answer the last question first, the overhead from using groups comes from the group overlap factor. The higher the group-overlap factor, the more regions will be overlapped by each group. If the bounding box of a group overlaps a particular region, it may be the case that few or even none of the primitives within the group actually overlap that region, yet the associated processor must transform and cull each of the primitives within the group to find this out. Also, as groups overlap more regions, they will need to be sent to more processors, increasing the communications overhead as well. Thus as the group overlap factor increases, we expect an increase in both the processing and communications overhead of the system. In answering the questions regarding group construction, size, and bounding volumes, we must therefore pay attention to the group overlap factor, since each of these has an effect on this overhead. We will examine answers to these questions in section 4.3.2.

Aside from making static mastership feasible, the idea of grouping primitives together offers benefits to other areas of primitive handling. Primitive sorting can be sped up as well, since rather than transforming and classifying every individual primitive, only the groups need to be processed. The amount of necessary bookkeeping data is reduced as well, since these data are only kept per group rather than per primitive. In addition, grouping can facilitate memory management by increasing the granularity of the managed units, in turn decreasing the number of units that must be managed. Both computation and storage costs for managing the groups are amortized over the primitives in the group, this occurring at the expense of the overhead from the group overlap factor. Choosing an appropriate group size may result in a net savings of work (see section 4.3.2). Because of this, one may wish to use primitive grouping in situations other than where static mastership demands it.

We now consider one more way to take advantage of primitive groups. We suggest that the bounding volume for a primitive group approximates the primitives within the group. Given this, we can use the bounding volumes instead of the primitives themselves to approximate the on-screen primitive distribution. This idea lets one reduce the amount of work needed to compute the distribution information since one no longer needs to transform the actual primitives to do this. The tradeoff is that this estimation of the distribution will not be as accurate as using information about the individual primitives themselves.

3.4 Focus on the MAHD Algorithm

In the following sections we describe in more detail the implementation of the MAHD algorithm.

3.4.1 Mesh Size

As mentioned, the MAHD algorithm uses a “fine mesh” in order to calculate the primitive rendering load across the screen and also as a quantum basis for making screen subdivisions. The question then arises of how fine this mesh should be: smaller cells allow more precise measurement and partitioning flexibility, but increasing the number of cells increases the algorithm overhead. These are similar to the tradeoffs we faced when considering the granularity ratio for static load-balancing methods. In fact, we will use the same term, granularity ratio, to talk about the number of mesh cells with regard to the number of processors. Thus a granularity ratio of 16 implies that there are 16 times as many cells as there are processors in the system.

As with the static load-balancing partitioning, we need a mesh that is fine enough to separate dense clumps of primitives into enough different cells such that the rendering load can be evenly partitioned. The same guideline applies: if we want to partition a clump of primitives occupying 1/16th of the screen, the granularity ratio should be at least 16. As before, the ideal granularity ratio is dependent upon the scene being rendered by the given application.

In the case of static load-balancing, increasing the granularity ratio increases the indirect overhead of the system. No more load-balancing work is required, but there is more primitive overlap and communication as a result. Increasing the granularity ratio of the MAHD algorithm on the other hand affects the direct overhead. The load-balancing operations require more work, but primitive overlap and communication are not much affected as a result.

The load-balancing operations that are affected by the mesh size are: tallying, communication of the tally results, and the screen-splitting computation. The overhead of the tallying is the cost of computing the tally amount (see below) plus the cost of an addition for each mesh-cell overlapped for each primitive being tallied. Thus as the granularity ratio increases, so does the mesh-cell-overlap factor. However, we expect that, on average, each primitive covers very few cells (large databases generally have small primitives), and since the time factor is just the cost of an addition per cell, the tallying overhead is likely to be fairly negligible. Also, it may not be necessary to tally every single primitive (see section 3.3.3), and thus this cost is reduced even further.

Gathering the tally results requires that each processor send its meshes to the gathering processor. At most, this amount of communication is the number of mesh cells times the number of processors (minus one). If the number of processors is large, this communication can be done in multiple stages, performing a reduction at each stage. However, we do not expect the number of mesh cells to be very large, and thus the direct communications method is probably reasonable for most systems. One might be able to reduce the required amount of communications by sending only the non-zero mesh cells. If the primitives being tallied by each processor are local to its screen region, then this optimization will reduce the communication to an amount only a little larger than the total number of cells in the mesh.

3.4.2 Tallying

The value that we tally into each cell should be representative of the cost (in time) to render that primitive. Before tallying, we divide this value by the number of cells that the primitive (or its bounding box) overlaps. This division is necessary because when we add together the values from groups of cells, we want the sum to approximate the overall time that will be necessary to render the primitives contained in that cell grouping. Inaccuracies will result from primitives that aren't totally contained within the cell grouping, however.

One could calculate the value to tally to each cell as follows:

$$\text{value}_1 = (\text{Tprim} + \text{Tpixel} * A) / C \quad \text{where}$$

Tprim = per-primitive time cost to render one primitive
Tpixel = per-pixel time cost to render one pixel of one primitive
A = area of the primitive in pixels
C = number of mesh cells covered by the primitive's bounding box

If the values are used only to calculate the screen partitioning, then it does not matter if they are all scaled uniformly. In this case, one could simplify the above formula as follows:

$$\text{value}_2 = (1 + R * A) / C \quad \text{where } R = \text{Tpixel} / \text{Tprim}$$

To further simplify the formula, one could use the assumption that the area of the primitive is approximately proportional to the number of mesh cells it covers:

$\text{value}_3 = 1 / C + R * Ka * Acell$ where

Ka = typical (primitive area / its bounding box area)
Acell = pixel area of one mesh cell

The Ka coefficient is an approximation for the relation between a typical primitive's area and its bounding box area. For triangles, the actual value varies between 0 and 1/2 depending upon the triangle's shape and orientation.

One final assumption can be made in the case where we have a highly pixel-parallel rasterizer, and thus the only primitive rendering costs are per-primitive, with negligible per-pixel costs. In this case, the second term of value_3 drops out, and the value is just:

$\text{value}_4 = 1 / C$

The above simplifications only apply if all the primitives are of the same type (all triangles, for instance). Since different kinds of primitives have different processing costs, we may find this formula to be more useful:

$\text{value}_5 = (Tprim / C) + Ka * Tpixel$

Note that while Ka will vary according to primitive type, the second term is still just a per-type constant, and thus the computation is still very simple. Of course, determining good values to use for Tprim and Tpixel will require some research for the given hardware and rasterizing algorithms used in each specific case.

3.4.3 Splitting

We use a hierarchical splitting algorithm to partition the screen into the same number of regions as there are processors. Hierarchical means that the initial split cuts all the way across the screen, giving two subregions, while the next two splits cut each subregion into two sub-subregions, and so forth. This is also known as a kd-tree algorithm [SAME90]. With MAHD, the splits are always performed along cell boundaries.

For each split, there is the choice of using a horizontal cut or a vertical cut. Our initial approach is to always choose the same direction for the first level cut and then to alternate directions for each successive level. An alternate approach is to determine the direction of each cut based upon the longer dimension of the region being cut. The tradeoff is that if different cut directions are chosen in successive frames, a loss of coherence will result, requiring more primitive redistribution.

In order to partition the screen into an arbitrary number of regions, we must determine an appropriate split ratio for each split operation. For numbers of processors that are powers of two, the split ratio is always 1:1. That is, we try to place each split such that there is an equal amount of rendering work on either side. For other numbers of processors, we can compute the split ratios as follows. First, we construct a minimum height, left-aligned binary tree with as many leaves as there are processors. In each leaf node, we place the value 1. In higher nodes we place the sum of the values of the immediate children. Each non-leaf node in this tree then corresponds to a split in the partitioning algorithm, with the root node corresponding to the initial split, the next nodes down corresponding to the next two splits, and so forth. The split ratio for any given split is given by the ratio of the values of the children of the corresponding node. For example, in the figure below we have the tree corresponding to a five node system.

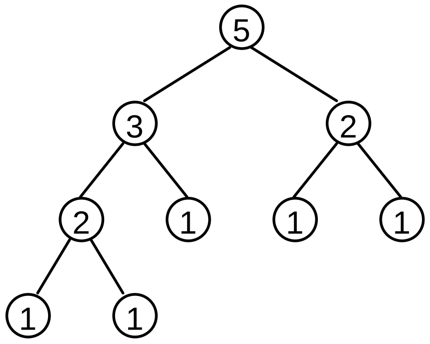


Figure 3.8

We thus make the initial split with the ratio of 3:2. The next level splits use ratios of 2:1 and 1:1, and the final split uses a ratio of 1:1 also. In the end, each final region (each leaf node) has approximately the same amount of work as the rest.

Once we know the split ratio, we must then determine the split location that will let us get closest this ratio. To determine this we use the values gathered in the cell mesh. First we convert the cell mesh into a summed-area table by storing sums accumulated across the rows and columns [CROW84]. This table lets us determine the sum of the original cells in any rectangular area by only considering the values at the corners of the area in the summed-area table. By using a binary search with the table, we can quickly partition a given region into two subregions with the correct split ratio. We actually construct a "partition tree" which details all the splitting operations:

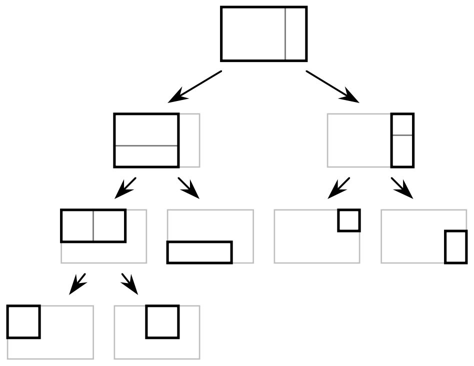


Figure 3.9

There are two potential problems that we have thus far not considered. One is the possibility of creating zero-area regions, and the other is the possibility of needing to avoid creating excessively large area regions. Creating a zero-area region gives a processor zero work to do. One may need to avoid large area regions due to hardware limitations (such as the amount of local frame-buffer memory per node). Avoiding either of these problems is easy: one simply limits the range of any given split operation. However, this solution may have some negative consequences on the overall utilization. This is a topic that will be left for future investigation.

3.4.4 Classification

Given that the resulting regions do not form a regular grid of rows and columns, we need to find a fast way of classifying primitives. That is, for a given primitive, we need a fast way to calculate which regions it covers. We will assume that we are actually interested in finding out which regions are covered by the primitive's bounding box, since this can be computed more easily.

One could use a quad-tree algorithm for this task [SAME90]. The recursive formulation of this algorithm is as follows:

```
bounding_box : the bounding box of the primitive in question
region_list : the set of regions overlapped by the bounding box; initially empty
partition_tree : a partition tree as described above, describing the screen partitions
node : the node in the tree that we are considering at each step; initially the root
```

```
classify(bounding_box, region_list, partition_tree, node)
{
    if (bounding_box does not overlap this node of partition_tree) return
    if (node is a leaf node) add the corresponding region to the region_list
    classify(region_list, bounding_box, partition_tree, node's left child)
    classify(region_list, bounding_box, partition_tree, node's right child)
}
```

The run time of the algorithm is approximately:

regions covered * $\log_2$ (total number of regions)

We can do better. A faster algorithm is to create another mesh of cells, and in each one store two values. One value is a bit mask indicating all the regions that could be overlapped by a bounding box having its upper left corner in this cell. The other value is a bit mask indicating all the regions that could be overlapped by a bounding box having its lower right corner in this cell. To find the regions covered by a given bounding box, we look up the bit masks corresponding to the region's upper left and lower right corners and logically AND them together to arrive at the proper set of regions. This algorithm requires only constant time. Creating the bit masks is straightforward, however the mesh data structure can be rather large.

In fact one does not really need a two-dimensional mesh in order to solve this problem. We can make do with four one-dimensional arrays. One array would store bit masks indicating possibly-overlapped regions for a bounding box with a given left edge. Others would supply similar bit masks for the right, bottom, and top edges. The proper four masks would be ANDed together to achieve the final set of regions. This algorithm requires less setup time and less storage space than the previous one.

3.4.5 Parallel and Scheduling Issues

There are additional implementation issues that still need to be addressed. We have not described how the load-balancing algorithm maps to a parallel machine, nor have we discussed the scheduling of the algorithm steps in order to fit within an interactive rendering framework. The latter issue is especially critical to the overall system performance.

Regarding the parallel implementation of the algorithm, we offer a relatively simple solution. Each processor will tally the primitives for which it is responsible into its own copy of the mesh. At some point, the multiple copies of the mesh must be combined into a single copy. We simply have all the processors send their values to a single processor. This processor can combine the values and perform the rest of the partitioning algorithm and finally broadcast the results back to the other processors. This approach is reasonable given that the partitioning algorithm is a relatively light computational task, the amount of data to be sent is not large, and the number of processors will be limited.

Each processor will have to send a copy of its local mesh to the gathering processor. If the primitives for which a given processor is responsible for tallying are mostly located within its own region, then the amount of data that each processor needs to send is likely to be proportional to the size of its region (i.e., empty cell counts do not need to be sent). On the other hand, if each processor's primitives are scattered about the screen, we will not be able to take advantage of this minor optimization.

We now turn our attention to the scheduling of the steps of the MAHD algorithm. If we choose the most obvious layout of the algorithm, the sequence of steps on a single processor would resemble this:

1. Receive the current frame parameters.
2. Transform and tally all the primitives.
3. Send off the tally results (if necessary).
4. Wait for the calculation results of the screen partitioning algorithm. (Or calculate and distribute the results if this is the gathering processor.)
5. Examine the primitives again and route them to the correct processors(s) as necessary.
6. Proceed with the remaining graphics pipeline steps.

There are two major problems with this approach. One is that it requires two full passes over all the primitives, and another is that processors must synchronize with each other between these two passes (at step 4). These problems mean increased latency and decreased efficiency.

One approach to increasing the efficiency is to reduce the cost of the first pass by sampling the database rather than processing every primitive. In doing so, we also attempt to reduce the possibility of delay resulting from the first synchronization point. To achieve this, we arrange the algorithm steps as follows:

1. Receive the current frame parameters.
2. Transform and tally a representative sample of the primitives.
3. Send off the tally results.
4. Start to transform the remainder of the primitives.
5. Receive the results of the screen partitioning computation.

(Or calculate and distribute the results if this is the gathering processor.)
Ideally, this happens before step 4 has finished.

6. Start again processing the entire set of primitives:
   Transform (as necessary) and route the primitives.
7. Proceed with the remaining graphics pipeline steps.

This algorithm reduces the possibility of processors idling while waiting to receive the partitioning data. We shall refer to this algorithm as the *stochastic MAHD* method, and it introduces some new issues. First, we must investigate the assumption that a sample of the primitives approximates the overall on-screen distribution well enough for our purposes. This depends upon the nature of the sample that is used, of course. In addition, we must also consider the overhead added by this rearrangement of the algorithm steps. These issues will be discussed in the next section.

Another solution to the efficiency problem is to eliminate the extra pass by performing both operations at the same time. This solution requires using the primitive distribution from frame *n* to compute the subdivision for frame *n*+1. The steps are:

1. Receive the current frame parameters.
2. Transform, tally, and route all the primitives.
3. Send off tally results.
4. Proceed with the remaining graphics pipeline steps.
5. Wait for results of screen partitioning computation.
   (Or calculate and distribute the results if this is the gathering processor.)

This is basically a pipelined implementation of the MAHD algorithm. With only a single pass over the database, the additional synchronization step introduced above is removed. The disadvantage of this solution is that the current frame's subdivision is based upon “old” data. However, because of the expected temporal coherence of frames, we expect the old data to be representative of what the new frame will be like. We investigate this algorithm further in section 3.4.7.

Before we take a more detailed look at the various algorithms, we consider how the scheduling affects communications issues. With either algorithm, there will be a burst of communication as the processors begin sorting primitives. This will result in congestion at this point in time, but since this occurs early in the frame, it should not be a problem, assuming that processors do not block as a result of the congestion. The congestion should also clear up well before the end of the frame. In addition, starvation should not be an issue since all processors will perform sorting first, followed by transformation of local primitives, before dealing with received primitives.

3.4.6 Stochastic MAHD

With the stochastic MAHD algorithm we must choose a method for sampling the primitive database to obtain the approximate on-screen primitive distribution. One method is to simply choose every “one in n” primitives while traversing the database. This method can sometimes run into problems when the primitives are laid out in patterns that are based upon multiples of the chosen n. Another approach can be used if the primitives are organized into groups with bounding volumes (see section 3.3.3). The primitive distribution could be estimated by examining where the primitive groups fall, rather than examining all of the individual primitives. The advantage of this method over the first is that all of the primitives are taken into account. The disadvantage is that the empty space within the group bounding volumes is counted towards the rendering work.

Assuming one does the sampling based upon the primitive bounding volumes, then one must consider what work is reasonable for the processors to perform while waiting for the partitioning results. One possibility is that they can transform their slave primitive groups, since the sampling itself only requires that each processor examine its master primitive groups. If static mastership is used, each processor will have a good number of slave primitive groups to transform. However, with dynamic mastership, there will be relatively few slave groups per processor, and hence this work may not be enough to overcome any unevenness in the number of master groups being examined. In this case, there appears to be no reasonable way to avoid this particular inefficiency.

The accuracy of the sampling process can be skewed due to other factors. On-screen visibility culling algorithms, for instance, attempt to avoid transforming and rendering entire objects if it can be determined that they would be occluded by nearer scenery [COOR97, ZHAN97]. This is just one of a variety of algorithms that attempts to reduce the graphic workload (others include level-of-detail and texture-substitution techniques). When such algorithms are used, one can no longer assume that the set of primitives within the viewing frustum is the same as what will be rendered. Thus unless the culling information can be worked into the sampling process, then the results of the sampling can be very misrepresentative of the actual rendering work distribution. In such cases, one may be better off using the pipelined MAHD algorithm.

We now examine the overheads associated with the stochastic MAHD algorithm. First, we note that the tallying cost is reduced since we are not tallying every primitive. However, extra overhead is introduced by making additional passes over the database.

The cost of the additional passes depends mainly upon how much data must be touched during each one. Appropriate thought put into the design of the database data structures can help reduce this factor (see chapter 4). There is also the possibility of a synchronization delay while processors are waiting for the partitioning results; however providing the processors with enough work to perform until the results come back reduces the possibility of any delay.

3.4.7 Pipelined MAHD

The pipelined MAHD algorithm depends upon good frame-to-frame coherence in order to produce a good estimate of the on-screen primitive distribution. However, for real applications, there will always be some differences between successive frames, and thus the distribution information for a given frame will usually be suboptimal for use with the next frame. If the viewer is rotating, for instance, then objects will always be shifted slightly “behind” in the distribution mesh compared to where they actually appear. This will result in the partitioning algorithm always choosing slightly unbalanced work distributions.

One can attempt to compensate for this error. One possibility is to resample the distribution mesh from the last frame based upon an offset computed by examining the differences between the previous and current viewing transformation matrices. Without extra information, this method can only be used to compensate for changes in the rotation component of the viewing matrix. If one retains some Z information for each mesh cell, then one can also attempt to compensate for translation in the viewing matrix. This method still does not help when the object motion results from changes in object matrices, however.

One can also attack the coherence problem by attempting to increase the amount of frame-to-frame coherence. One can do this by increasing the frame rate. In most cases, however, the graphics hardware is running at the maximum possible frame rate already. Beyond adding more processors, increasing the frame rate can be achieved by reducing LOD accuracy requirements, reducing the screen resolution, or by adjusting various culling metrics.

Aside from the coherence issue, the pipelined MAHD algorithm is very nicely structured. It requires no intra-frame synchronization points, since the primitive distribution information is not required until the start of the next frame. One can incorporate ideas from the stochastic algorithm: the tallying of the distribution information can be done on a per-group basis, rather than per-primitive. This reduces the amount of work required for this operation.

3.5 The Load-Balance Experiments

We are now prepared to evaluate the various load-balancing methods. In this first round of experiments, our overall goal is to compare the static load-balancing method against the two variations of the MAHD adaptive method.

3.5.1 Goals

The purpose of this experiment is to answer the following questions:

Static Load Balancing:

1. How many regions per processor are necessary for good load-balancing?
2. How does the number of regions per processor affect the overhead?

Adaptive Load Balancing (MAHD):

3. How good is the load-balancing of the MAHD algorithm?
4. How does the mesh-cell-size parameter affect the results?
5. How does using the previous frame's data affect the load-balancing?

Both:

6. How do the methods scale with respect to the number of processors?
7. How do the static and adaptive methods compare?

A discussion of load-balancing brings up the question of what is a good load-balance? This is a difficult question to answer precisely, since load-balance is only one of many interdependent factors involved in determining the final performance of a system. For instance, there is a tradeoff between load-balance and communications costs, especially with static load-balancing. Given this factor, load-balance goals are dependent upon the ratio of processor speed over communications bandwidth. Decreasing the ratio means a more even load-balance is desirable, while increasing it allows for more variability in load, assuming this allows for a decrease in communications costs. For this first set of tests, we will make a somewhat arbitrary choice and declare a load-balance adequate if the utilization exceeds two thirds. We will come back to this question in section 3.6.

3.5.2 Procedure

To answer the questions posed above, the simulator system used in section 2.5 to evaluate coherence was suitably modified. Both static and adaptive load-balancing schemes were added to the system. The adaptive scheme has the option of using the current frame's primitive distribution or that of the previous frame in order to determine the screen subdivision. These options are labeled “CF” and “PF” (respectively) in the results section below. The tallying formula used was $value_4$ from section 3.4.2. Aside from these changes, the setup as presented in section 2.5.1 remains the same for the new tests.

For these tests, we collect data concerning the primitive distribution across the system. For each recorded frame, we compute the minimum, maximum, and average number of primitive fragments per processor (we say fragments since here the primitives are counted according to how many regions their bounding boxes overlap; a single primitive may lie in one processor's multiple regions.) We also compute the standard deviation of this figure. We normalize these values by dividing them by the ideal value (the database size divided by the number of processors) and express them as a percentage. The utilization (expressed as a percentage below) is calculated by dividing the average value by the maximum. In order to consider a given method's communications overhead, we record the total number of primitives that need to be sent each frame. This value is also normalized by dividing it by the database size, and it is expressed as a percentage as well.

The following tests were performed for each application:

Static Algorithm:

Processors	Region Configuration
16	4x4 - 40x40 (1*1-10*10 regions per processor)
64	8x8 - 64x64 (1*1-8*8 regions per processor)

Adaptive Algorithm:

Processors	Mesh Configuration
16	16x16, 32x32, 64x64
64	64x64, 128x128, 256x256

Table 3.1 Experimental Setup I

Each run of the adaptive algorithm was performed twice, once using the previous frame's distribution data and once using the current frame's.

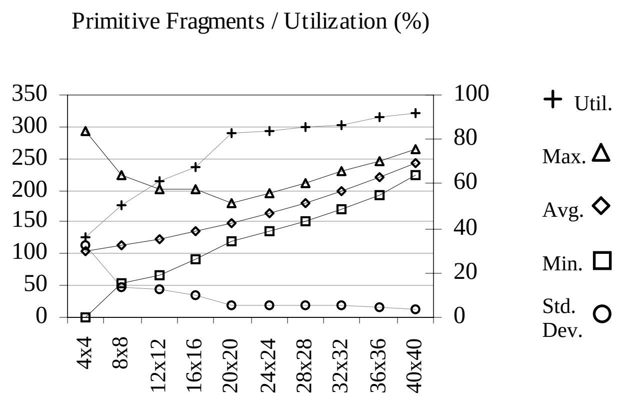
3.5.3 Results

In this section we present graphs showing the summary results from the experiments. The summary results were obtained by averaging the various statistics over the frames of the particular test run. For the communications figures, we also show various statistics taken over the frames of each run. For the PLB Head case, these include the maximum, minimum, and standard deviation. For Sierra, the maximum values were all over 100% (see section 2.5.2), so we show the 95th percentile values instead. The minimum values were all near zero, and thus were left out.

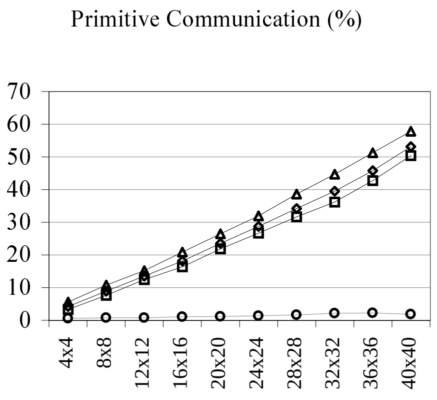
The utilization figures are plotted with respect to the right-side scale on the graphs below.

Dataset: PLB Head

Static Averages: 16 processors, 1-100 regions per processor

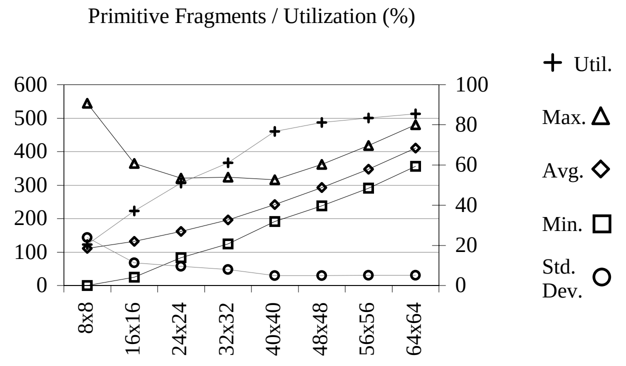


Graph 3.1a

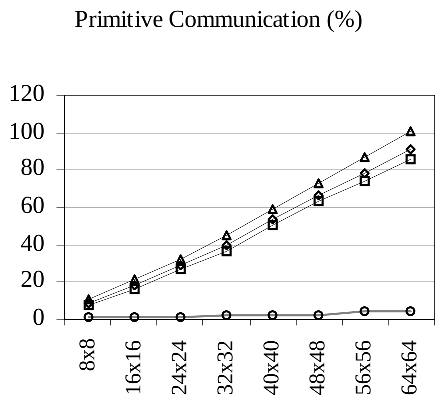


Graph 3.1b

Static Averages: 64 processors, 1-64 regions per processor

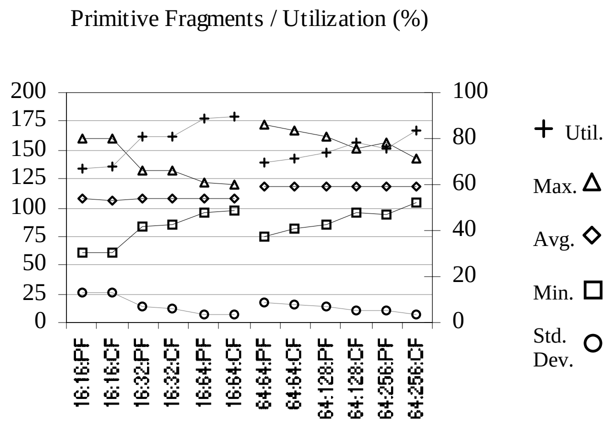


Graph 3.2a

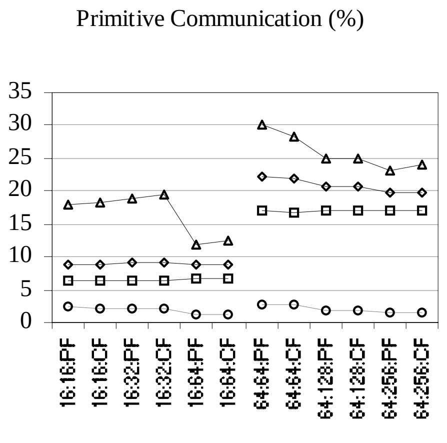


Graph 3.2b

Adaptive Averages: 16 or 64 processors, 16x16-256x256 mesh, PF or CF algorithm



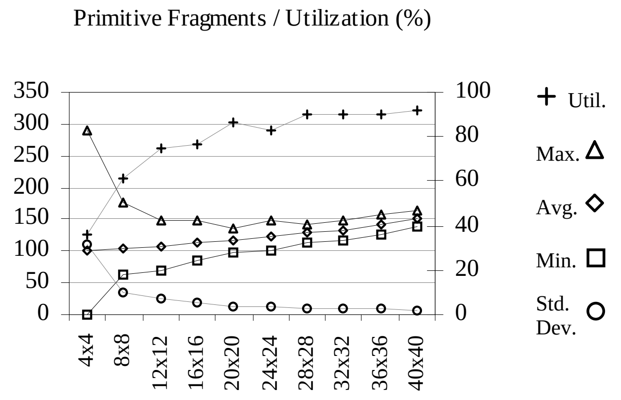
Graph 3.3a



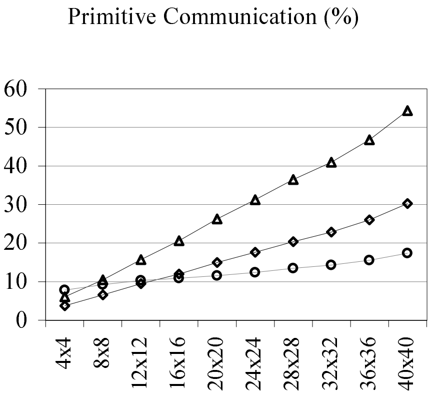
Graph 3.3b

Dataset: Sierra

Static Averages: 16 processors, 1-100 regions per processor

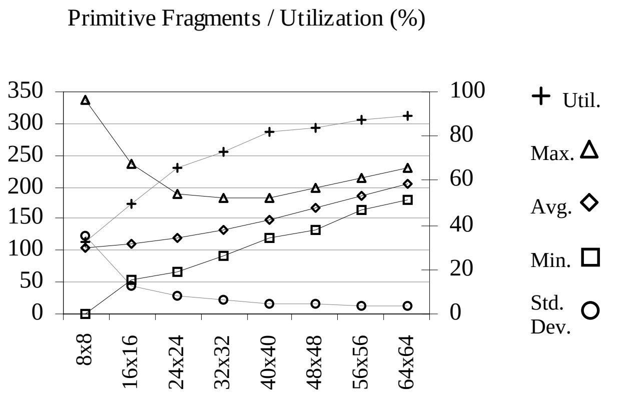


Graph 3.4a

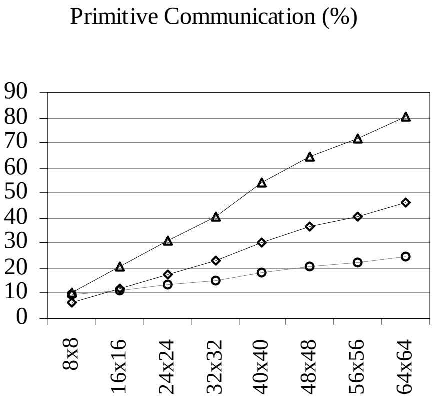


Graph 3.4b

Static Averages: 64 processors, 1-64 regions per processor

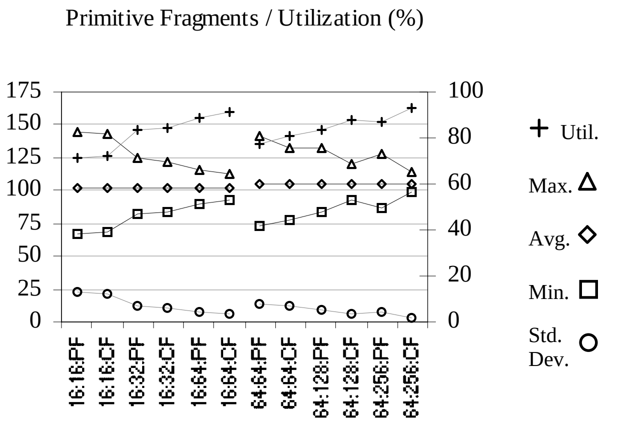


Graph 3.5a

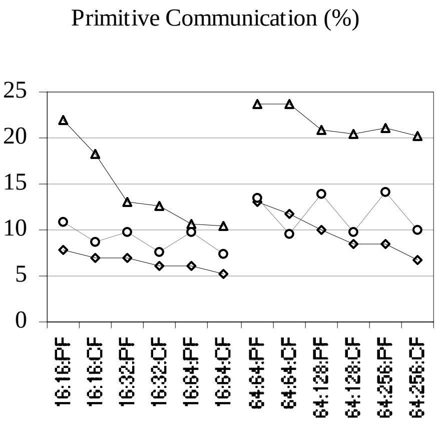


Graph 3.5b

Adaptive Averages: 16 or 64 processors, 16x16-256x256 mesh, PF or CF algorithm



Graph 3.6a



Graph 3.6b

In the appendix we show some of the detailed graphs which plot the frame-by-frame results for certain test runs. For the adaptive PF cases, we found that the first frame default partitioning (equal-sized regions) combined with the use of the previous frame's distribution data led to high communication for the first two frames of the run. Thus the first two frames were omitted from the statistics for all the test cases.

3.5.4 Discussion

We now consider the questions posed in light of the data.

Static Methods:
1. How many regions per processor are necessary for good load-balancing?

Having chosen a utilization of two-thirds or more to be good, we can look at the utilization graphs (3.1a, 3.2a, 3.4a, 3.5a) and see that between 9 and 25 regions per processor must be used before we achieve a reasonable load-balance. This value varies according to both the dataset and the number of processors being used. PLB Head, being the smaller dataset, required more regions per processor than Sierra.

2. How does the number of regions per processor affect the overhead?

Because the number of regions per processor has a direct bearing on the size of each region and thus on the overlap factor, we naturally expect that increasing the number increases the overhead. Graphs 3.1, 3.2, 3.4, and 3.5 bear this out: both the amount of communication required as well as the number of primitive fragments that must be processed increase in direct relation to the number of regions into which the screen is divided. The increase is directly proportional to the total length of cuts made across the screen. Doubling the cut length (by increasing the number of regions per processor, say, from 4 to 16) approximately doubles the amount of communication as well as the number of *additional* primitive fragments in the system. Thus if communication is costly, then it is vital to keep the number of screen regions to a minimum.

Adaptive Methods:
3. How good is the load-balancing of the MAHD algorithm?

With the proper choice of cell size (see below), the algorithm is able to achieve respectable load-balancing without problem. This is shown in graphs 3.3a and 3.6a. Limiting the cell-size too much results in poor algorithm performance, as we will discuss next.

4. How does the mesh-cell-size parameter affect the results?

The choice of mesh-cell size is a very important factor in the performance of this algorithm, as it determines how precisely primitive distribution is measured and how finely the screen can be subdivided. Starting from a very large cell size, reducing the cell size results in a dramatic benefit to the resulting load-balance, as graphs 3.3a and 3.6a show. Each halving of the cells' dimensions results in nearly a halving of the standard deviation of primitive distribution.

Also, in contrast to the static cases above, the change in cell size results in almost no change in the total number of primitive fragments. This number is mainly dependent upon the number of screen regions, which cell-size does not affect. The average amount of communication required does not change much either, though peak communication needs are decreased as the cell-size decreases. This likely happens because the finer mesh allows the screen subdivisions to follow the primitive motion somewhat, reducing primitive border crossings.

5. How does using the previous frame's data affect the load-balancing?

For the applications tested here, we can see that using the primitive distribution data from the previous frame does not result in any significant performance degradation. The data for the sixteen-processor cases shown in graphs 3.3 and 3.6 are nearly identical for the PF and CF cases. For the 64-processor cases, one can see that the CF cases perform slightly better (around 5% increased utilization). For Sierra we also see a very small savings in the number of primitives to be communicated using the current frame's distribution data.

One must remember that the applications tested here were fairly simple in nature and did not have a great deal of motion in them. Further testing with different types of applications would be valuable. However, before this can be done, the simulation must be further refined to handle problems such as off-screen primitives.

<u>Both:</u>
6. How do the methods scale with respect to the number of processors?

To answer this question, we consider the amount of overhead added as the number of processors varies while still maintaining a reasonable load-balance. Both methods suffer increased performance overhead with an increasing number of processors, again due to the problem of dividing the screen into more and more regions.

We compare graphs 3.1, 3.2, 3.4, and 3.5 to see the results. Graph 3.1 shows that PLB does fine with 16 processors, while 64 is too many; the average number of primitive fragments doubles by the time a reasonable load-balance is reached. On the other hand, Sierra, a database nearly three times larger, does fine with up to 64 processors. Sierra does have smaller primitives (reducing its overlap-factor overhead), but this is typical of larger databases.

As for the adaptive method, graphs 3.3 and 3.6 show both cases are fine up to 64 processors. Since overhead is due largely to the overall number of regions, we can look at the Sierra static test cases and suggest that the adaptive method can handle much larger numbers of processors, perhaps over a thousand, before overhead becomes too much of a problem.

As with many parallel systems, there are lots of application issues that determine the limits of how well the architecture scales. Any screen-subdivision architecture will suffer from increased overhead as the number of subdivisions increases. The amount of extra overhead is determined partly by the overlap factor and, in the case of sort-first, partly by the dataset dynamics.

7. How do static and adaptive region assignment compare?

The static method requires 9 to 25 regions per processor to achieve the same load balance that the adaptive method can achieve with one region per processor. This means that the static method needs 3 to 5 times the communications bandwidth of the adaptive method to take care of the additional primitive shuffling overheard. The static method will also require an increase in processing capability to account for the increased number of primitive fragments.

The tradeoff for this is that the static method has no overhead for any primitive distribution measurements or screen subdivision procedures. Additionally, the classification algorithm is simpler and the rasterization stage does not have to deal with varying-size regions. Finally, the processor-to-frame-buffer mapping is fixed for the static algorithm; this fact could be used to an advantage for designing a parallel framebuffer system.

The question is then whether these simplifying factors make up for the additional overhead of having multiple regions per processor. While the simplicity of the static method may appeal to designers of lower-cost systems, it seems that designers of high-performance system would opt for the adaptive method.

3.6 Load-Balance Experiments II

Having found MAHD to be a well-performing load-balancing algorithm, we have developed a more detailed implementation of sort-first in software in order to study the architecture further. In the next section we describe this implementation, and in the following sections we describe additional experiments which were performed using this implementation.

3.6.1 Sort-First Software Implementation

We have written a parallel sort-first software-based implementation. PVM is used to provide multiprocessing capability [GEIS94]. A host program spawns processes for each graphics processor and the framebuffer. The system implements the MAHD load-balancing algorithm in both the stochastic and previous-frame forms. It also implements two systems for dealing with hierarchical graphics databases; these will be discussed in the next chapter. As discussed in section 3.3.3, the system places primitives into groups, and sorting and communication of primitives happens at the group-level. Mastership of primitive groups is static, and bounding boxes around each group are used to accelerate primitive culling. One can vary the maximum number of primitives per group in addition to several other parameters.

The placement of primitives into groups is done in a sorted method as described in section 4.6.2. The bounding boxes placed around the primitive groups are initially axis-aligned; no attempt is made to orient them for tightest fit. During database traversal, the bounding boxes are transformed into screen space, and their bounding rectangles are computed. This information is used to perform both view-frustum culling and bucket-sorting of the primitive groups. These choices are described in chapter 4.

3.6.2 Goals II

We would like to answer the following questions:

1. How do the stochastic and previous-frame algorithms compare?
2. What are realistic values for the communications coherence factor?
3. How does varying the number of processors and the group size affect processor utilization and communication?

3.6.3 Procedure II

For this set of experiments, we chose to use more complex datasets. The datasets were taken from the GPC's OpenGL Performance Characterization (OPC) project's Viewperf benchmark set [SPEC00]. Three datasets were taken from the Data Explorer, Design Review, and Lightscape benchmarks. They have the following characteristics:

Benchmark	Dataset	Primitives	Type
Design Review	*gyda2*	63,608	triangles
Data Explorer	*dx*	91,581	triangles
Lightscape	*parlment*	450,029	quads

Table 3.2 Dataset Descriptions

These datasets were chosen to cover an interesting range of database sizes and types. The *gyda2* and *parlment* datasets include recorded flythrough paths, whereas the *dx* dataset stays onscreen while rotating on its axes. A front-end program was written which loads the models and controls the animation sequences. This is essentially a simplified version of Viewperf written to use the software sort-first implementation; the latter outputs frame-by-frame, per-processor statistics to various data files during each run.

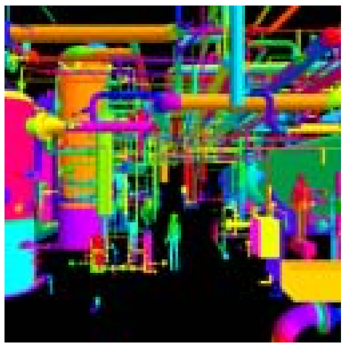


Image 3.1
Gyda2

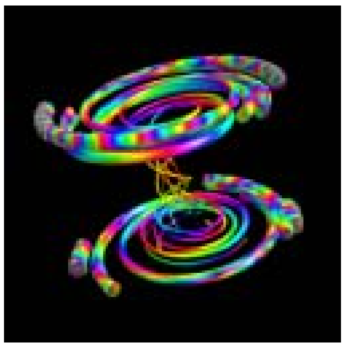


Image 3.2
DX

Image 3.3
Parlment

The following additional information about the datasets will help explain some of the experimental results:

Dataset	Onscreen primitives	Animation characteristics
gyda2	50K-63K (59K avg.)	no rotation, only slow zoom, translation
dx	all	object rotates ~5 deg. per time step
parlment	58K-425K (205K avg.)	view rotates up to 7.2 deg. per time step

Table 3.3 Dataset Animation Properties

Runs were done using the following setups:

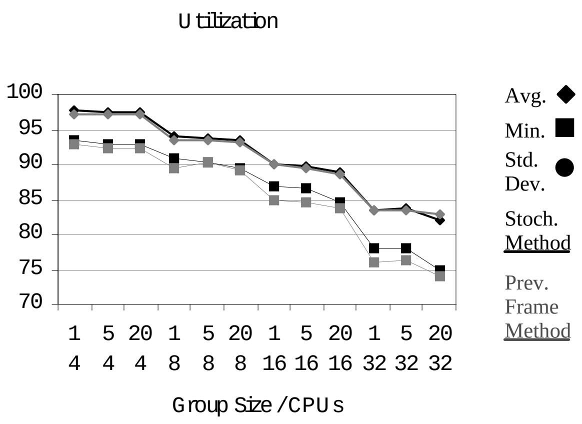
Number of Processors: 4, 8, 16, 32
Group sizes: 1, 5, and 20 primitives per group
Load-Balancing Methods: previous frame and stochastic
Framebuffer sizes (from dataset): 700x700 (*gyda2*, *dx*), 1248x960 (*parlment*)
Mesh sizes (chosen experimentally): 140x140 (*gyda2*, *dx*), 156x120 (*parlment*)
Back-facing primitives are not culled, thus are counted for utilization.
Primitives are sorted into groups using the method described in section 4.3.2.

3.6.4 Results II

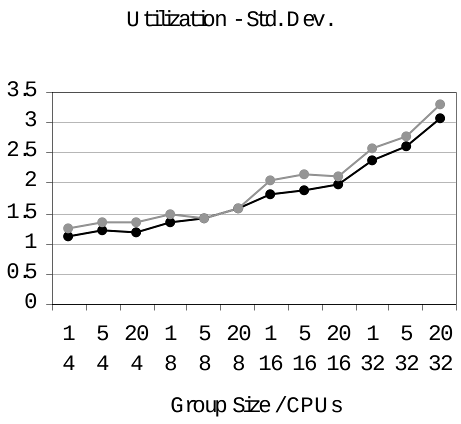
As before, we present graphs containing the summary results of all the tests, while the appendix contains frame-by-frame results for a select number of test runs. The summary results are produced by computing the averages (and standard deviations) over the various measured values from the frames of each test run.

All the data below are expressed in percentages as defined in section 3.5.2.

Dataset: *Gyda2* / Primitive Utilization & Standard Deviation

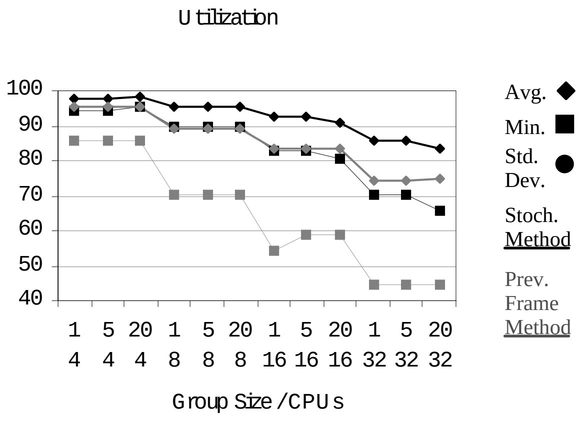


Graph 3.7a

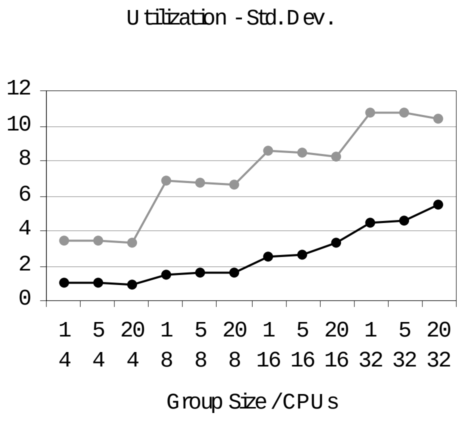


Graph 3.7b

Dataset: *DX* / Primitive Utilization & Standard Deviation

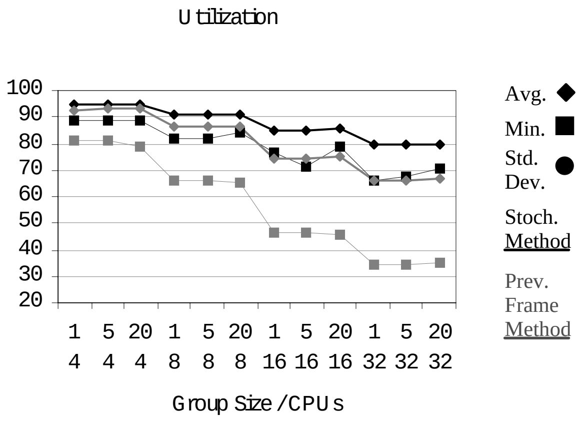


Graph 3.8a

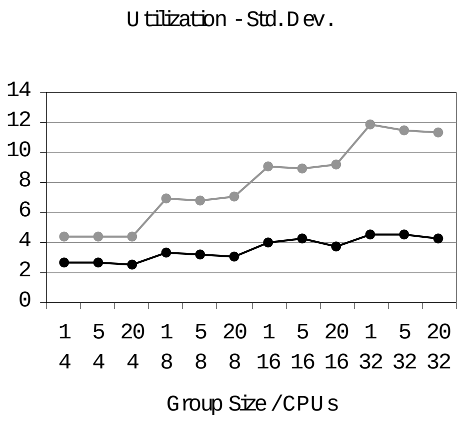


Graph 3.8b

Dataset: *Parlment* / Primitive Utilization & Standard Deviation

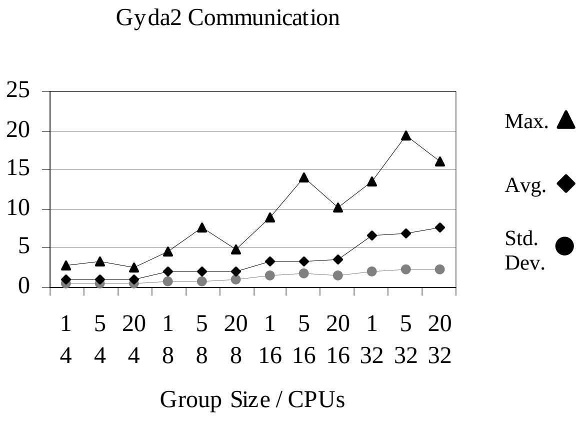


Graph 3.9a

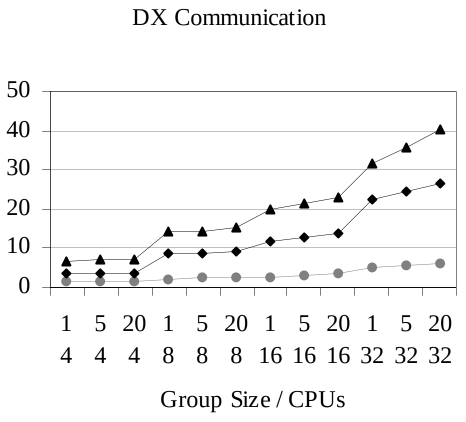


Graph 3.9b

Communications: Maximum, Average, and Standard Deviation

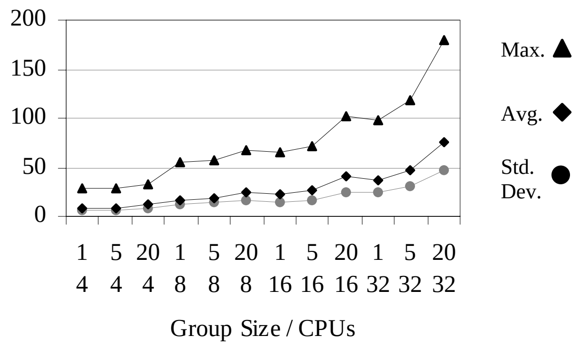


Graph 3.10a



Graph 3.10b

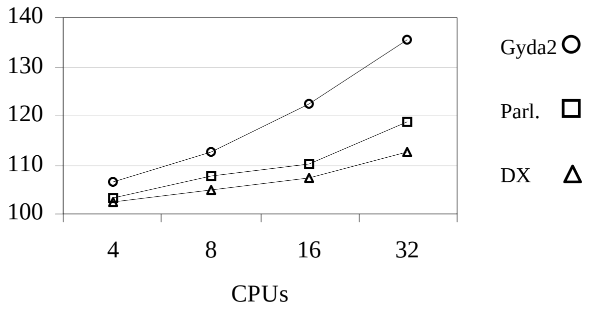
Parlment Communication



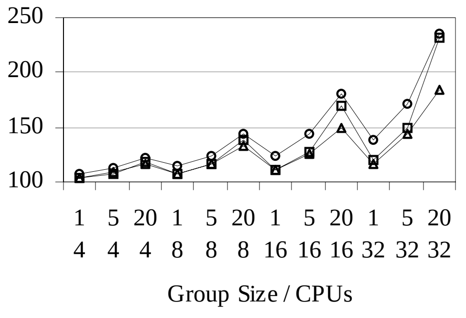
Graph 3.10c

Primitive and Group Overlap Factor

Primitive Overlap Factor



Group Overlap Factor



Graph 3.11a **Graph 3.11b**

3.6.5 Discussion II

1. How do the stochastic and previous-frame algorithms compare?

Looking at the utilization graphs, we can see that for the *gyda2* dataset the methods perform nearly identically, whereas for the other two datasets the performance diverges somewhat. For the *gyda2* dataset, the figures are all within a percentage point of each other. The difference grows to as much as 11% for *dx*, and up to about 14% for *parlment*. These increasing differences are mainly due to the nature of the animation of the various datasets; greater differences reflect more active animations (with correspondingly less frame-to-frame coherence, upon which the previous-frame method depends). As one would expect, the difference in performance between the two methods is smaller when there are fewer screen regions, and it grows as the screen is divided into more numerous, smaller regions. This is because the inefficiency of the previous-frame method is relative to how much the primitives move out of their screen regions from one frame to the next, and this value grows as the screen regions become smaller and more numerous (while the amount of motion stays the same).

More differences between the two methods can be seen in the standard deviations and worst-case utilization values. For the stochastic method, the standard deviations are typically a few percent smaller and the worst-case values are somewhat better (9-28% for *dx*, 8-35% for *parlment*). This is not surprising, since we expect that the coherence between frames (which affects the efficiency of pipelined MAHD) will vary more often than the difference between the estimated and actual frame's primitive distribution (which affects the efficiency of stochastic MAHD).

All in all, we find that while the average difference in performance between the two methods is not very large, the worst-case performance differences can be fairly substantial. This suggests that load-balancing using the previous frame's distribution data is a reasonable approach for applications where one can count on a good amount of frame-to-frame coherence. For situations where the coherence is less guaranteed, one should choose the stochastic method.

2. What are realistic values for the communications coherence factor?

The graphs 3.10a-c plot the communications coherence factor for the various test cases. Again, these values represent the number of primitives being communicated vs. the number of primitives on-screen, averaged across the frames of each test run. The figures for the previous-frame method are virtually identical to the figures for the stochastic method, and thus only one set of figures is shown. As the graph shows, the coherence factor is directly proportional to both the number of screen regions used and the number of on-screen polygons. This is to be expected, since both factors directly affect the likelihood of a primitive crossing a screen region boundary. There is some discrepancy in the data as the group size varies, and this is discussed in the next section below.

Looking at the standard deviations and maximum values, we can see that there can be a great amount of variance in the communications coherence factor. For the smaller datasets, the values are not large enough to be of much concern. However, for *parlment*, the largest dataset, the maximum factor runs over 100% for certain cases with larger numbers of processors. Partly, the size of this figure is due to the group overlap factor (see below). Mostly, however, the large factor is due to the nature of the animation itself. In the *parlment* dataset, the camera performs a rapid rotation of 180 degrees in just 25 steps. With the dataset's 60-degree field of view, this means the primitives are moving close to 1/8th of the screen in each time step. With 32 processors, the screen is divided horizontally into 8 regions, and thus it is difficult under these circumstances to escape having 100% or more of the primitives moving between frames. We speculate that such rapid rotations are relatively uncommon. We note that the eye cannot focus easily during such a rapid rotation, which suggests that one may attempt to trade off accuracy in rendering for bandwidth savings to deal with this situation. In addition, an increase in the system's frame rate can also greatly decrease the amount of communication (by increasing coherence). Finally, we note that it is conceivable that one may have a load-balancing scheme that includes relative primitive movement as a factor in determining screen partitioning. Such a scheme could sample relative primitive movement and use this information to help reduce the amount of communication required by defining the regions accordingly.

The communications coherence factor results show us that it is difficult to pin down any specific values for the factor without first defining several variables. However, we feel confident that for systems of up to sixteen processors, the number of primitives being sent between frames can in most cases be kept well under fifty percent of the number of on-screen primitives, provided that a reasonable amount of coherence exists.

3. How does varying the number of processors and the group size affect processor utilization and communication?

Looking again to the utilization graphs, we observe that the utilization appears to decrease a small amount for each doubling in the number of processors. The amount of decrease varies depending upon the dataset and the load-balancing method. For the stochastic method, the decrease ranged from 3-7%; the decrease is larger for the previous-frame method (in the range of 6-12%) for the reasons mentioned above. Various factors account for the decrease in utilization. First of all, the primitive distribution information is based upon primitive groups. Even with a group-size of one, this information is only estimated, and errors in the estimation grow as the average screen-region size decreases. Second, screen splits can only be done along cell boundaries. Even though the cell sizes used in the experiments appear small, there were still many instances where regions were only a few cells wide or tall, indicating that going to a finer division may be helpful.

Looking at the communications coherence factor vs. the number of processors, we see exactly the expected results. Each increase in the number of processors adds to the total length of the inter-region boundaries. The coherence factor tells us how often primitives move across these boundaries, and therefore it increases proportionally to the total boundary length. The figures illustrate the boundary lengths for various numbers of processors.

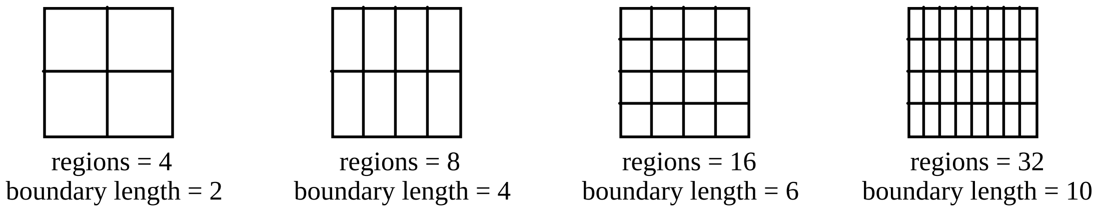


Figure 3.10

The same relationship is shown by the primitive overlap factor (see graph 3.11a). Whereas the communications coherence factor tells us about primitives moving across the boundaries, the primitive overlap factor reveals how often primitives simply lie across region boundaries. Comparing the numbers for the different databases tells us about the relative on-screen primitive sizes: larger primitives are likely to cross more regions. As expected, *gyda2* appears to have relatively large primitives, while *parlment* and *dx* have much smaller ones. Even though *parlment* is a larger dataset, *dx* has smaller on-screen primitives. This is not too surprising given that *dx* is viewed completely on-screen, whereas *parlment* is a walk-through.

If we look at the effect of the group size upon the communications coherence factor, we see the expected result of increasing communication as the group size increases. The amount of the increase appears to be mostly a reflection of the speed of the animation of the dataset. With the datasets where the animation is more limited, the increasing group overlap factor does not significantly change the communications coherence factor. When the amount of primitive motion increases greatly compared to the screen region size, then the effect of the group overlap factor increases significantly.

3.7 Primitives other than Polygons

Thus far we have made the assumption that the only type of graphics primitive in the system is a polygon. For many applications, this assumption is valid. However, for various reasons, users often desire a greater variety of primitive types. For instance, the triangle strip provides greater efficiency by reducing the number of points that must be specified for a given dataset. The same can be said for the height field. In addition, more powerful primitives, including surface patches such as NURBS [PIEG95], allow complex surfaces to be modeled with many fewer primitives than if triangles were used. We cover some of the issues involved in using these kinds of primitives with a sort-first system.

Triangle strips pose a problem since they can contain an arbitrary number of triangles. This is a problem since a large database can consist of a few triangle strips, thus making it difficult to distribute and load-balance this work on a per-primitive basis. The usual solution to this problem is to subdivide large triangle strips into smaller pieces, then distribute these pieces. This solution works with sort-first as well. The per-primitive rendering cost for a triangle-strip section is thus a function of the number of vertices, and the per-pixel cost is typically related to the bounding box. While it is possible to construct triangle-strips with high per-pixel rendering costs that lie within a small volume (by overlapping the triangles), such constructs are probably not very useful. Note that each time a triangle-strip is split, two vertices must be duplicated in order to provide continuity for the new strip.

Height fields have the same problem as triangle strips: they can contain an arbitrary number of triangles. Fortunately, the solution is the same as well. Height fields can be easily partitioned in order to allow more even distribution of the data across the rendering nodes. Just as with triangle-strips, one must keep in mind that each partitioning increases the total number of vertices that must be processed. Thus while a 51x51 field has 2601 vertices, splitting it into quarters yields four 26x26 fields, or 2704 vertices.

Surface patches allow the specification of complex curved surfaces by specifying only a relatively small number of control points. Unfortunately, graphics rendering hardware cannot render surface patches directly, and instead the patches must be tessellated into a number of polygons. The tessellation may be either static or dynamic, done using a variety of techniques [KUMA96]. This tessellation process is a complicating factor when surface patches are brought into a parallel graphics system. For instance, patches that are adjacent in model space require the same amount of tessellation along their borders in order to avoid cracks or seams. When adjacent patches are located on separate processors, the tessellation algorithm must guarantee that the borders will be subdivided equally. With sort-first, we have two ways to handle the tessellation and redistribution. A single processor can tessellate and redistribute the polygons, or else the patches themselves can be redistributed with each processor performing the tessellation separately. Each method has tradeoffs.

With a single processor tessellating a patch, the overall amount of computation is reduced at the expense of additional communication. A problem that results is that any time the patch must be re-tessellated, the polygons that have been distributed to other processors must be deleted and the new polygons must be distributed. This makes the rendering algorithm very complicated. Distributing the patch itself avoids these problems. In this case, each processor must do its own tessellation of the patch, increasing the total amount of computation required. Each processor must be sure to perform the same tessellation algorithm, since otherwise patches that cross screen boundaries will have cracks or seams.

Load-balancing of patches can be done in various ways. One might do the tessellation first, then perform the necessary tallying based upon the generated polygons. Since the polygon-distribution information is not available until after tessellation, this method is perhaps better paired with a load-balancing scheme that uses the distribution information from the previous frame. Alternately, one may tally based only upon the patch and its bounding box, making some assumptions about the rendering complexity contained therein. This method is a better choice when computing the load-balancing for the current frame since it allows the tallying to be completed as quickly as possible. However, the assumptions made may not necessarily lead to an accurate estimate of the amount of rendering work to be done, perhaps leading to load-imbalance.

3.8 Multiple Viewports

We now consider how sort-first can continue to operate efficiently when rendering to multiple viewports (also known as channels). In this section the viewports we are talking about all provide a view of the same database, but each one may have different viewpoint information.

Given multiple viewpoints, there are two approaches that a graphics system can take to render them. It can consider each viewpoint in turn, using all the machine resources to render each sequentially, or it can consider all the viewpoints as part of a single composite frame, distributing the machine resources among the portions of the overall frame. Depending upon how the viewpoints are related, one approach might offer lower overhead or latency compared to the other.

If the views are presented independently, then the sequential approach can offer lower latency. This approach is also easier to implement, since the machine must only deal with a single viewpoint at a time. On the other hand, treating the multiple viewpoints as a single frame allows the overall frame to be divided into fewer regions (thus lowering the overlap factor) and also potentially allows individual processors to each deal with only a single viewpoint (and thus a single traversal over the database), decreasing overhead. However, this method is more complex to implement.

Let us consider the sequential approach for a moment. We establish the notion of a viewing context; within a given viewing context there is frame-to-frame coherence. One may establish multiple contexts and then switch between them. Thus by keeping track of individual contexts the graphics system can keep track of frame-to-frame coherence even when the successive frames rendered by the system are from unrelated viewpoints.

Given multiple viewing contexts, we must rethink off-screen primitive handling. Off-screen primitives now fall into two categories: off-screen for a given context, or off-screen for all contexts. A primitive can only be discarded if it is in the latter category.

In the next few sections, we examine some of the common arrangements of multiple viewports. For each, we consider whether it is more advantageous to treat the viewports sequentially or simultaneously. We also examine other ways to optimize the rendering.

3.8.1 Stereo Channels

With stereo display, two viewpoints are created that are slightly offset from one another and facing in nearly the same direction. The scenes from the viewpoints will be fairly similar depending upon the offset and discrepancy in gaze direction. Due to this similarity, it is likely that the primitives present on a given processor for each context will be similar, and that the primitives that need to be communicated for each context will be similar as well. Since the views are dependent and must be updated simultaneously, there is no latency advantage to finishing one view before starting the next (a “swimming” effect occurs if the views are not updated simultaneously). On the other hand, since the viewpoints differ, it is not possible to easily treat the two images as a single composite frame. For the best efficiency, one would make special code paths to deal with the optimizations one could get out of stereo by working on the two images simultaneously. However, for ease of implementation, one would more likely treat each view as a different context and render each one sequentially.

3.8.2 Spatially-adjacent Channels

Spatially-adjacent channels are often used together with large field-of-view displays. Such displays may consist of multiple projection CRTs typically laid out side-by-side, often with each viewscreen angled to face the viewer. Sometimes the edges of the images from adjacent CRTs are overlapped to eliminate any possibility of a “seam” appearing. Eliminating seams also requires that the images sent to each CRT be updated simultaneously. What is unique about this display configuration is that the viewpoints share a common eyepoint. This makes it easy to treat the multiple viewpoints as a single composite frame. This also allows certain optimizations to be made.

Given the common eyepoint, if the field of view of the entire display is less than 180 degrees, one can accelerate certain aspects of primitive handling. Once a primitive has been projected onto the view plane for one viewpoint, one can determine how the primitive projects onto the view planes for the other viewpoints. One can do this by projecting the other (secondary) viewports onto the first (primary) view plane. The relative location of a primitive in the primary viewing plane with respect to a secondary reprojected viewport will be the same as the relative location of the primitive when it is projected onto the secondary viewplane. Thus one can quickly perform any culling or binification operations based upon only a single projection. A slight difficulty arises if the viewing planes are set at different angles. In this case, the reprojected viewports will be distorted by the reprojection. Thus if regions in the viewports are axis-aligned rectangles, they will become trapezoids when reprojected.

3.8.3 Independent Channels

If multiple viewpoints are used to drive completely independent display channels, such as for multiple users, then there is a latency advantage to having the entire system process each channel sequentially. This arrangement reduces the amount of time taken to produce the individual images for each channel. Since the views are completely independent, there is not much to be gained from attempting to treat them as a single composite image.

3.9 Hardware Constraints

We now investigate how various hardware factors and implementation issues affect the overall sort-first algorithm and load-balancing strategies. We consider issues relating to the design of the communications, rasterizer, and framebuffer hardware.

3.9.1 Communications issues

For the most part, we can ignore many communications issues when designing a sort-first system, provided that in the end we have enough bandwidth to support the desired applications. However, there are a few algorithmic design decisions that affect the communications patterns and bandwidth requirements. Among these, perhaps the main one is the decision between static and dynamic mastership (see section 3.3).

With static mastership, the initial distribution of the primitives determines which processor is responsible for making sure that a given primitive (or group) is sent to all the processors that need a copy. It is likely that for a given frame, each processor will need to send some primitives to most of the other processors. Therefore, with regard to communications patterns, static mastership implies that there will be much many-to-many communication.

Using dynamic mastership means that mastership responsibilities for a given primitive (or group) get passed along to a processor whose region contains that primitive. The effect upon communications is that messages will become more localized, with processors tending to communicate mostly with other processors that have screen regions near to their own. Assuming a point-to-point communications network, this fact can be used to decrease the overall amount of communications.

One final issue concerning communications and load-balancing is the transfer of pixels from the rasterizers to the framebuffer. As mentioned in section 3.4.3, we do not limit the size of any given region during the screen subdivision process. As a result, if a large portion of the screen has few primitives, this region may be assigned to a single processor. This processor will then have to send a large region of pixels to the framebuffer. Thus it must be the case that either such large transfers do not get in the way of the rest of the system (due to bandwidth or blocking restrictions), or else the subdivision process should limit the maximum size of any region to a figure that poses no such problems.

3.9.2 Rasterizer Design

Before we consider how sort-first and rasterizer design interrelate, we examine more closely how rasterizers work. While there are many possible ways to construct a polygon rasterizer, two common approaches stand out. One is the conventional framebuffer-based architecture, and the other is the tile-based (or region-based) architecture. [FOLE90, TORB96, COX97]

With a conventional rasterizer architecture, a full-screen framebuffer is used to store all of the necessary pixel information. Minimally, this includes at least two color buffers and a Z-buffer. When a particular primitive is being rendered, the rasterizer computes which pixels will be affected, reads any necessary information from the framebuffer, and writes out the new pixels to the framebuffer as appropriate. This is repeated for all the primitives in the scene. With this approach, primitives may be processed as soon as they are available, and the image in the framebuffer is complete as soon as the last primitive has been processed.

Extending the conventional rasterizer architecture into a fully-parallel graphics system presents problems. The straightforward approach would place a full-screen framebuffer on every rendering node, each of these feeding one or more framebuffers that actually display the completed images (or see 3.9.4 for a slightly different alternative). This approach would be wasteful of memory for a screen-partitioning graphics architecture since the majority of the pixels on any given node will not be used (each node is only working on a fraction of the screen). A solution to this problem is to only place a limited amount of framebuffer memory on each node. This will either limit the largest fraction of the screen that any one node can work upon, or else it will dictate a region-based approach, as we discuss next.

With a region-based architecture, the pixel information for a fraction of the screen is kept locally within the rasterizer itself (sometimes even completely on-chip). The fraction is a rectangular tile or region of the screen, typically between 32x32 and 128x128 pixels in size. One region of the screen is processed completely before working on the next. This requires that the primitives be sorted according to the regions they occupy, and this further implies that all of the primitives be transformed and sorted before rasterization can really get underway. Once a region is finished, the color information is sent to a separate full-screen framebuffer (the pixels' Z and other intermediate values do not have to be sent).

Note that a hybrid architecture is possible. Such an architecture would use the memory within the rasterizer as a cache for the full-screen framebuffer. Sorting the primitives by region would increase the locality of access, though it would not be necessary to have the primitives be fully sorted, thus eliminating the latency involved in waiting for the sorting to finish.

With either architecture, parallel pixel processing can be employed. As the amount of pixel parallelism increases, the time taken to process individual primitives decreases. Once the number of pixel processors begins to exceed the number of pixels in a primitive, per-primitive rasterization time approaches a constant value.

3.9.3 Load-balancing vs. Rasterizer Design

In any graphics system it is important to match the performance of the rasterization hardware to that of the other components in the graphics pipeline, especially the geometry transformation and processing components. With sort-middle, this can often be done easily enough by simply varying the number of each type of processor. With sort-first or sort-last, however, the transformation processors and rasterizers are paired together, and they should be closely matched in performance.

3.9.3.1 Load-Balancing Rasterization

The cost of rasterization is a factor that should not be ignored when performing load-balancing calculations. If one had a system where all primitives had equal rasterization costs, then load-balancing the transformation would load-balance the rasterization as well. This ideal situation is only achieved if all the primitives have similar on-screen area (not a realistic assumption for most applications) or if each rasterizer generally has more pixel processors than each primitive has pixels (possibly a realistic assumption for some systems and applications). More commonly, a primitive's rasterization cost will be proportional to its area.

For the typical less-than-ideal situation, one should include rasterization costs into the calculations for computing screen partitions for load-balancing. This has already been discussed in section 3.4.2. As mentioned in that section, there is a tradeoff involved: computing accurate rasterization costs for each primitive increases the direct load-balancing overhead. This is why bounding-box estimates are typically used.

3.9.3.2 Rasterizer Constraints on Screen Partitioning

As mentioned above, if there is a limited amount of framebuffer memory (less than full-screen size) on each rendering node, then this will either limit the largest region size that any one node may work upon, or else it will dictate using a region-based rendering approach. Limiting the region size too much can lead to poor load-balancing solutions in certain circumstances (such as when the primitives are crowded into one corner of the screen).

A region-based rendering architecture brings up some questions of its own. Should the load-balancing algorithm be concerned with the region-size of the rasterizer? Generally speaking, regardless of how the screen is partitioned (whether along rasterizer-region boundaries or not), there will be approximately the same number of primitives to rasterize. However, partitioning along rasterizer-region boundaries does offer some advantages. For one, reducing the total region boundary length (by aligning rendering regions with screen partitions) will reduce the overhead from the primitive overlap factor. Such alignment can also reduce the total number of regions that must be processed (by avoiding fractional regions), thus lowering the per-frame overhead.

3.9.3.3 Rasterizer Design vs. System Pipelining

As mentioned earlier, a region-based architecture requires that all primitives be sorted according to screen region before the rasterizer can finish working on the first region. This is due to the fact that we must be sure there are no more primitives to be rendered in this region before working on the next one, and we cannot be sure of this until sorting is complete. For sort-first, this has some possible consequences that we should examine. For instance, we wish to avoid having a global-synchronization point introduced into the algorithm in order to avoid extra delays. We also wish to avoid possible network congestion if all the nodes finish rasterizing their first region and try to send the pixels to the framebuffer at the same time.

So far, the algorithm has been developed as follows:

1. Receive the current frame parameters.
2. Sort the primitives across the nodes in the system.
3. Proceed with the remaining graphics pipeline steps.

We will now expand step 3:

3a. Send a “done sorting” message to all other rendering nodes.
3b. Prepare all local on-screen primitives for rasterization.
    This includes sorting by screen region and any necessary rasterizer setup.
3c. Process incoming on-screen primitives as above.
3d. Continue until “done sorting” message is received from all other nodes.
3e. Begin rasterization.

Because step 3e cannot begin until all other nodes have reached step 3a, there is the possibility of a delay here. This possibility is mitigated by two factors. First, since each node must process local and received primitives (steps 3b and 3c) before waiting on the other nodes (step 3d), this gives a chance for the slowest node to finish step 2 to catch up with all the others and perform step 3a before the fastest node has to do any waiting. Also, if static mastership is used, it is possible to load-balance the sorting work very evenly (independently of the rendering work), thus making it likely that nodes will reach step 3 at about the same time.

3.9.4 Framebuffer issues

We have already covered some ways in which the framebuffer design affects load-balancing. Yet for the most part, we have been considering systems wherein the mapping of rasterizers to framebuffers does not really affect the load-balancing algorithm. However, there are various framebuffer arrangements where this is not the case.

One possible arrangement involves placing a framebuffer on every rasterizer, and then synchronizing and combining the output of multiple framebuffers using video blending or dynamic video switching. This arrangement would allow “communication” of the rendered pixels without actually having to send them anywhere. If every framebuffer is connected to just a single output, then the load-balancing process is limited only by the video-switching restrictions. For example, if the video switching can only be done on a per-scan-line basis, then screen regions will be limited to horizontal strips (assuming no other pixel communication is done). Another scenario is if the framebuffer outputs are grouped together with multiple groups feeding multiple displays. In this case, the image space is partitioned by the grouping arrangement, and screen regions can only be shifted around within each partition.

A similar yet slightly different problem arises when we consider the use of multiple projection displays. If all of the displays are projecting overlaid images upon the same screen, then the situation is much as above. However, with such displays, we are not limited to having each display exactly overlay the other. We could imagine multiple displays lined up horizontally (to provide a wide field of view), yet with a considerable amount of overlap between the displays in order to allow for load-balancing without pixel communication. This limits the load-balancing algorithm to only moving the screen partitions around within the overlapped areas.

3.10 Load-Balancing Wrap-up

In this chapter we have addressed many aspects of load-balancing with regard to the sort-first architecture. We examined static and adaptive approaches to load-balancing, and we found in our experiments that adaptive load-balancing is the more desirable approach due to the fact that it requires partitioning the screen into fewer regions, greatly decreasing the overhead due to the overlap factor. We introduced the MAHD adaptive load-balancing algorithm, which uses many advanced techniques to keep the direct overhead low. We also examined the idea of load-balancing and sorting based upon groups of primitives instead of doing these operations on a per-primitive basis. This idea offers benefits by reducing overheads, but it also introduces new overheads of its own. We will continue to examine these grouping-related issues in the next chapter.

Our investigations into load-balancing also looked at several practical issues. We looked at various ways of dealing with complex primitives, and we also established the notion of rendering contexts as a solution of maintaining frame-to-frame coherence when rendering to multiple viewports. Finally in this chapter we examined various hardware issues and either how they may affect the load-balancing algorithm or how they may be affected by it. Among these issues were processor-to-processor communication, rasterizer design, and framebuffer design.

4. Graphics Database Management

In any parallel system, the management of the application data structures is a matter of no small concern. Having a database distributed among several processors creates problems when processors need portions of the database that they do not have. One must carefully decide what data should be divided among the processors and what data should be replicated across all the processors. Then one must adapt the various algorithms to make sure that each processor gets the data it needs and that this happens efficiently. In solving these problems, one must first consider the structure of the data itself and the set of operations that must be performed on it. Therefore, we first review some basic facts about graphics databases.

4.1 Types of Graphics Databases

The graphics database consists of a description of all the objects that the application desires to render. Combined with a set of viewing parameters, this is all the information needed by the graphics system to render a particular scene. The database may be stored on the graphics system, in which case it is called a "retained-mode" database, or it may be kept on the host and used as an "immediate-mode" database. Retained-mode databases come in various forms; we will discuss primitive lists and hierarchical graphics databases.

Primitive lists are straightforward: they consist of a list of graphics primitives, each containing its own descriptive properties. While easy to implement, primitive lists do not support any high-level operations: you can only change the database by adding or deleting individual primitives. Hierarchical graphics databases combine multiple primitive lists with state and control primitives to add support for high-level operations. Immediate-mode systems take a different approach: they depend upon the host system to feed the graphics system a stream of primitives for every frame rendered. In the next sections, we will discuss each of these database options and how they relate to sort-first.

4.1.1 Primitive Lists

The primitive list is the simplest form of graphics database. Each primitive in the list contains all the information necessary to render it, with the exception of the viewing parameters that are common to all of the primitives. The only operations on the database are to add or delete primitives. A primitive list provides the minimal functionality necessary to implement a graphics system, and this kind of interface is adequate for simple graphics applications. However, the lack of any high-level operations makes this interface unsuitable for more complex applications. Operations on entire objects (groups of primitives) would require many manipulations of individual primitives. Such operations could easily consume a burdensome (and unnecessary) amount of computation and bandwidth resources.

We will not specifically address the issues of primitive-list support. However, we note that the primitive list is a subset of the hierarchical graphics database, which we will discuss in detail. From these discussions one can infer what is necessary to support primitive lists alone.

4.1.2 Hierarchical Graphics Databases (HGD's)

In order to overcome the shortcomings of the basic primitive list, most systems use some form of hierarchical graphics database [FOLE90]. We can describe the HGD by starting with the primitive list concept and extending it in several ways. First, we allow certain primitive attributes to be specified separately from the primitives; once an attribute is specified, it is put into the global state and it binds to all subsequent primitives. Thus if many primitives have a common attribute, such as color or texture, the attribute need only be specified once for the entire primitive group.

Rather than using a single transformation matrix for all of the primitives, we add the capability of specifying additional transform matrices within the list. With each matrix we can specify how it is to be concatenated with previous matrices. As with other attributes, each transform matrix affects subsequent primitives within the list.

Next, we go beyond a single primitive list by providing multiple named lists, called structures. Thus primitives common to a single object can be put into their own structure. Finally, we add the concept of a structure call (execute). A structure can contain one or more calls to other structures. Each call creates a new instantiation of the called structure.

The caller's global state is passed on to the called structure, but not vice-versa. This is achieved by the use of a state stack.

If we prohibit recursive structure calls, then the resulting call graph forms a hierarchy. The use of structure calls makes it simple to create multiple instances of objects. One can also easily switch an object on or off or switch among multiple representations of the same object by simply changing a structure call. By combining structure calls and matrix transforms appropriately, one can easily animate an entire scene by changing only a few pieces of information within the hierarchy.

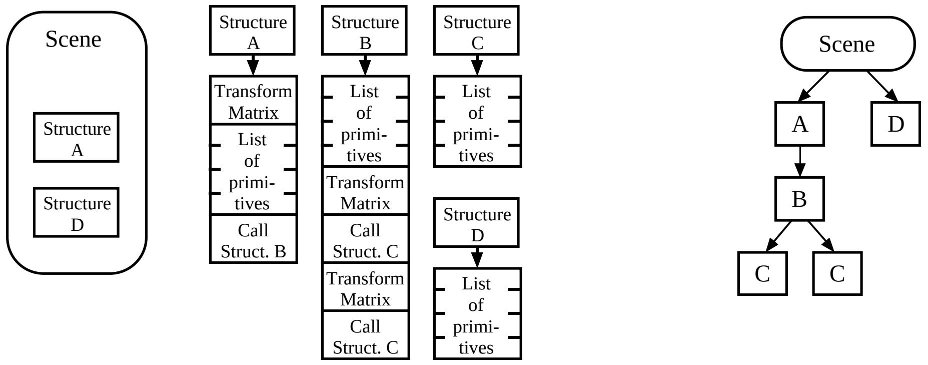


Figure 4.1 Hierarchical Graphics Database Example
The example scene contains structures A & D. Structure A calls B which calls C twice.
A diagram of the scene is shown to the right.

4.1.2.1 HGD Terminology

We define some terminology that relates to hierarchical graphics databases. We have already described the contents of the database by describing structures, which contain primitives, attributes, transforms, and structure calls. The following are the basic operations on structures:

- create a new structure, add elements to it, close it
- delete a structure
- post a structure (add it to the scene)
- unpost a structure (remove it from the scene)
- change an element in a structure to a similar element
  (i.e., change a matrix, color, execute, etc.)

With an HGD, the scene consists of:

- a viewing matrix (plus other viewing parameters)
- default primitive attributes
- the list of posted structures

The scene can in fact be viewed as a main, root structure to which you can only add or remove execute calls. We will use the term “instance” to refer to a particular instantiation of a structure that is being executed, whether due to the structure being posted or due to an execute call by some “parent” structure.

Drawing the scene starts with the traversal process, wherein the system starts with the first posted structure and traces a path through the database based upon the primitives and commands contained within the structures. Traversal of a hierarchical graphics database is complicated by the notion of state inheritance. State information is passed from a parent structure to its children (usually not from sibling to sibling). A state stack can be used during regular traversal to easily manage proper inheritance. However, traversing the hierarchy in any non-standard order requires saving all of this state information. This is an issue for sort-first since the primitives arriving at a given node are not guaranteed to be in any particular order.

4.1.2.2 HGD Design Issues

Sometimes it makes sense to bind attributes directly to primitives rather than indirectly through the global state. An example is a model where every primitive is a different color. Such attribute binding can be supported by either providing multiple types of primitives or else providing a flag with each primitive indicating which attributes are bound.

Another design decision is the flexibility with which the various structure elements may be intermixed within a structure. The different types of elements can be either segregated or interspersed, and certain restrictions (such as having at most one transformation matrix per structure) also do not limit the generality. Databases with different semantic restrictions can typically be converted from one type to the other, and therefore such choices can be determined by the resulting efficiency of their implementation.

There are some additional database issues that will receive little or no discussion. One of these is image-based texture mapping. With fully-parallel rendering architectures, image-based texture mapping typically requires that each rendering node have a copy of all textures that might appear in the scene being rendered. This solution works in the case where there are few textures that are shared by many primitives. If one has an application wherein nearly every primitive has a unique texture, then one may be better off by treating textures as primitives. This issue will not be covered since the texture-mapping problem is common to all highly-parallel architectures, and the solutions will likely be topics of dissertations themselves.

Another issue that we will not address is database paging. Database paging is a method for dealing with databases that are larger than the rendering system can store in its memory. There are two scenarios here: one is where a very large "world" is being defined, yet only part of it will appear on-screen at once; the other is where a very large number of primitives will appear on-screen. The former scenario can be handled easily enough by having the application delete structures that are known to be not visible. The latter scenario requires either virtual memory on the graphics processors or else API extensions to deal with streaming (immediate-mode) primitive data. Immediate-mode is discussed next.

4.1.3 Immediate Mode

Rather than retaining the graphics database within the graphics subsystem, some computer systems keep this functionality solely within the application program. Thus the application program must keep all information about the database and present this information to the graphics subsystem anew each frame. This is referred to as "immediate-mode operation."

The advantage of immediate mode is that it gives the application more control over the database, both in terms of storage and manipulation. The main disadvantage is that it typically requires much more communication between the processor(s) running the application program and the processors in the graphics subsystem. This is the case for applications in which the basic object appearance remains mostly the same from frame to frame. Only when the majority of the primitives are changing from frame to frame do the communications requirements tend to even out or perhaps even favor immediate mode. Certain applications designed to display the results of scientific simulations fall into this category. A discussion of the relative merits of immediate mode vs. retained mode may be found in [AKEL89].

Since immediate mode is a common application-program interface (due to the popularity of SGI's OpenGL API [NEID93]), we will consider how one can implement such an API with a sort-first architecture. This is done in section 4.7. However, since retained mode is more applicable to sort-first and very suitable for a large number of interactive graphics applications, this is the API that the thesis will focus upon.

4.1.3.1 Types of Immediate Mode

There are two basic varieties of immediate mode command streams: stateless and stateful. In the former, all the necessary information to render a primitive (with the exception of certain viewpoint-related information) is bundled together with that primitive. In the latter, primitive attributes are separated (to various extents, depending upon the API) and specified independently. The property specification commands affect the “current state”, and the properties of a given primitive are determined by the current state at the point in the command stream where the primitive is defined. Thus the information provided at a given time affects the definition of primitives declared later.

Stateless immediate mode potentially involves a greater amount of communication, but distributing a stateless command stream to a group of processors is relatively easy. Stateful immediate mode removes redundant state specification information from the command stream but makes the resulting stream more difficult to distribute.

4.1.3.2 Immediate Mode for Parallel Systems

When considering the immediate-mode interface in the context of parallel machines, one must address some questions about the application computer system and its connection to the graphics subsystem. For instance, is the application parallel or serial? If the application is parallel, is it running on the same hardware that the graphics system is running on? How does the application communicate with the graphics system? Does each application node talk to one or to many graphics nodes? While immediate mode is well defined for a single application processor talking to a single graphics processor, other configurations are not as well defined. We will have to address these issues before discussing how to implement immediate mode on sort-first.

When considering immediate-mode with sort-first, several more issues arise. Does the application attempt to perform any sorting? Does it attempt to keep track of the primitives' frame-to-frame coherence? And if they must, how do the graphics nodes redistribute immediate-mode primitives? We examine all of these issues in section 4.7.3.

4.2 Parallel HGD Issues

We now examine what is involved in using a hierarchical graphics database in a parallel system. We first examine the issue by looking at what the issues and solutions are for sort-middle and sort-last. We then examine these issues with respect to sort-first. Due to primitive migration, implementing HGD’s on sort-first is more complicated than for the other parallel architectures. We examine the nature of the problem, and then chart out two different solution branches.

4.2.1 HGD's in Sort-Middle and Sort-Last

The basic strategies for dealing with HGD's are similar with either sort-middle or sort-last [ELLS90, MOLN91]. We describe these first since the ideas apply to all distributed database systems. In the following sections we will see how these ideas are extended to work with sort-first.

The main issue is how to distribute the database, and the common solution is as follows: replicate most of the hierarchy except for the primitives, which are distributed across the processors. This “distribute-by-primitive” method is shown in figure 4.2. The goal of the distribution scheme is to give each processor a similar work load. To help achieve this in sort-middle and sort-last, one scatters the primitives across the processors such that each will likely have a similar number of on-screen primitives to render.

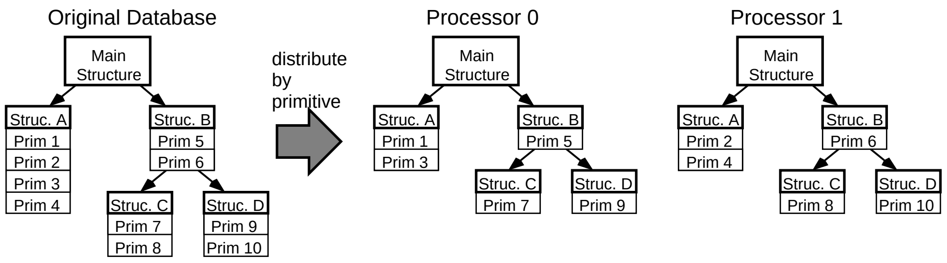


Figure 4.2 Distribute-by-primitive method

All of the operations are relatively straightforward. During rendering, each processor simply traverses the hierarchy and processes the subset of the primitives that it has. Most editing operations are performed by broadcasting the appropriate command to all processors, which change their copies of the database appropriately. The only exception is when the command is to create or change a given primitive. In this case, the command should go to only the relevant processor.

Note that with this kind of strategy, only the work related to the primitives is parallelized, while all other kinds of operations are replicated. Some optimizations are possible to reduce the extra work. A common one involves batching primitives together, then removing redundant state changes and any structures that become completely empty on any given processor [ELLS90]. This can be very helpful for applications that either have frequent state changes or that use a large number of small structures. On the negative side, it makes keeping track of the database somewhat more complicated, since all processors will no longer "see" the same hierarchy.

4.2.2 HGD's in Sort-First

Now we examine the issues surrounding the implementation of HGD's in sort-first. As with the other architectures, we must consider the question of how to distribute the actual database among the processors. We must also deal with primitive migration, meaning that each processor must receive the right set of primitives to render each frame. This issue will be discussed in section 4.2.3, and various solutions will be discussed in the following sections.

We begin with the issue of database distribution. With sort-first, we have the choice of using either the same "distribute-by-primitive" method (figure 4.2) as with the other architectures, or the alternative "distribute-by-structure" method (figure 4.3). In this method, one tries to subdivide and distribute the database by structure, thus giving each processor only a portion of the overall hierarchy. Such a distribution scheme would appear to better parallelize the traversal load.

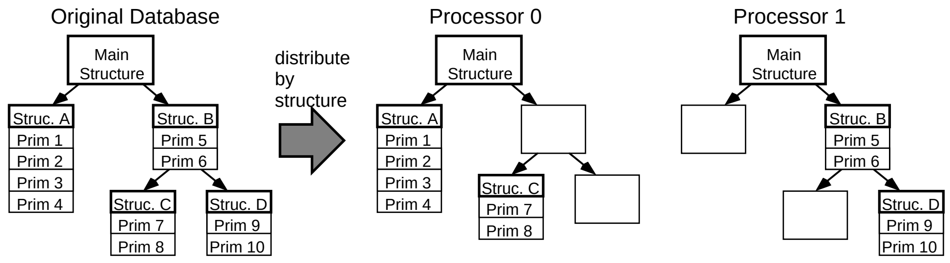


Figure 4.3 Distribute-by-structure method

However, this scheme makes traversal very complicated as a result of instancing (structure executes). This is because the inherited state (including the transformation matrix) needed for a given structure depends upon all of its parent structures, but those structures may not be present where they are needed (as in Fig. 4.3, structure C). To eliminate this problem, we can replicate all such state information, but this makes the method very similar to the distribute-by-primitive approach.

Given that the hierarchy and state information are replicated, we are left with the choice of how to actually distribute the primitives themselves. Scattering the primitives finely usually guarantees that they will be evenly distributed across the processors, regardless of which primitives wind up on screen. Distributing the primitives in large consecutive clusters or groups provides no guarantee of an even distribution unless one bases the group size on a-priori knowledge about the structure sizes (something most API's do not provide for). Even if one provides each processor with a group of equal size, there is no guarantee that the on-screen groups will be equally sized. Choosing groups of intermediate size is another possibility.

For sort-last, one should choose a relatively fine primitive scattering since this is a main factor in determining the overall processor load-balance. For sort-first or sort-middle, the initial primitive distribution determines the processor load-balance for the sorting pass over the database, but not for the overall rendering. Thus a fine scattering is good for this purpose, but a coarser scattering has the desirable characteristic that it may leave some structures empty. With empty structures, one has the opportunity to skip over them during database traversal, improving the parallel efficiency of the system (see section 4.5).

Also with sort-first, the actual primitive data will tend to cluster together due to the very nature of the architecture. Whether or not this provides a window for greater efficiency is determined by various bookkeeping decisions, which are discussed in sections below.

4.2.3 Primitive Migration Concerns

With sort-first, unlike sort-middle or sort-last, we must deal with primitive migration. This means that each processor must end up having the right set of primitives to render each frame. This set consists of every primitive that has an instance appearing in a screen region for which this processor is responsible. Thus aside from making sure that each processor has a copy of the necessary primitive definitions, we must also make sure that each processor knows which instances of those definitions it must render. Our database solution must allow for efficient representation and migration of this information, as well as solutions for dealing with these data during subsequent frames. There are many efficiency criteria: a given processor should not have to store or transform many more primitives than those that fall in its region; the cost of the basic editing and traversal operations should be kept minimal (they should not take much longer than sort-middle or sort-first require); finally, of course, the amount of bookkeeping information and its management should not weigh too heavily.

Given that only primitives (with their intrinsic state) will be migrating, we can see that it is necessary to keep primitives distinct from the other types of database contents. However, depending upon the database semantics, some primitive attributes may be tied into the primitive's position in the database, and thus we must be able to efficiently maintain these ties as primitives migrate. These attributes may include items such as the current color or the latest transformation matrix. Manipulating the database semantics to limit such ties is one possible solution (for instance, only allow a single current color and matrix per structure), but another possibility is to maintain efficient lookup tables for such information and assign appropriate indices to each primitive. We will provide more details about this method in the relevant sections below.

4.2.4 Database Representation Issues

In order to illustrate various issues associated with database representation and migration, we set up an example database scenario:

Structure A	Structure B	C (cont'd.)
xform 1	xform	primitive e
execute B	color	primitive f
xform 2	execute C	primitive g
execute B		primitive h
xform 3		primitive i
execute B	Structure C	primitive j
xform 4	primitive a	primitive k
execute B	primitive b	primitive l
	primitive c	primitive m
	primitive d	primitive n

Figure 4.4 Example database contents

The database consists of three structures: A, B, and C. Structure A is posted. The hierarchical structure of the scene can be viewed two ways:

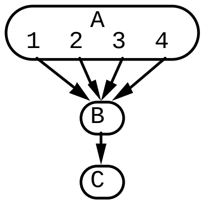


Figure 4.5a Minimal view

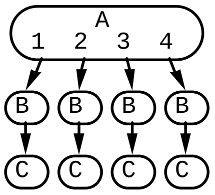


Figure 4.5b Maximal view

Figure 4.5a shows a *minimal* view of the database by presenting it as a directed-acyclic graph. This contains all the information we need to traverse the database. Figure 4.5b shows a *maximal* view of the database by presenting it as a tree. This time each execute points to a separate instance of each structure. This view gives us a clearer picture of how much work this database requires to actually render.

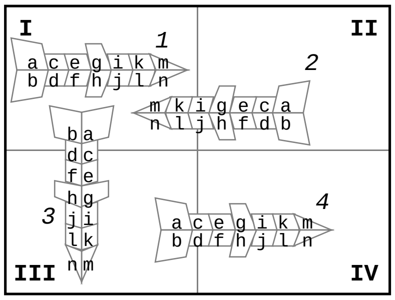


Figure 4.6 Example database scene

Now let us suppose an actual rendering of this scene appears as shown in figure 4.6. The screen regions are the quadrants in roman numerals, while the given instances of C's primitives are numbered in Arabic.

For this scenario, we notice that the processors for regions I and III must receive the entire set of primitives, while those for regions II and IV only need to deal with a subset. Given that they both have a copy of the entire database, how do the processors for regions I and III know which portion of it they must render? We need a way of determining which portions of figure 4.5b must be rendered by each processor.

We attack this problem by considering two main solution methods. The methods are based upon which view of the hierarchy is stored and managed:

1. The minimal set (figure 4.5a) containing only defined instances of primitives.
2. The maximal set (figure 4.5b) containing all of the instances of all primitives.

Each method has an interesting set of advantages and disadvantages.

4.3 HGD's: The Minimal Set Method

Keeping track of only the minimal set of database information has a number of advantages. Since this is the same procedure used by sort-middle and sort-last, it means that both the storage space required and the cost of basic operations will be similar among sort-first, sort-middle, or sort-last. However, this method has a significant drawback: for a given traversed instance of a structure, a processor does not know which primitives (if any) it must render.

Let us examine the traversal problem by considering figure 4.6. We see that instance 1 of structure C falls completely in region I. Instances 2 and 3 fall partially in this region, while instance 4 falls completely outside of this region.

The processor for region I must have all of the primitives in C because it will need them to draw instance 1. The processor also has the entire hierarchy information and thus it knows about all the other instances. How will this processor know that it only needs to render instance 1, parts of 2 and 3, and not worry at all about instance 4?

4.3.1 Min-Set Implementation

In order to efficiently decide which instances of a structure each processor must render, we compute bounding volumes for each structure [CLAR76, ARVO89, GOTT96]. This allows a given processor to quickly test whether or not an instance of a structure falls in the processor's region, thereby making the min-set method practical. The processor will first check whether or not a structure's bounding volume intersects the region and only act on the enclosed primitives if it does.

This will help deal with most cases. For the example case used above, the processor for region I can use this information to quickly decide that it need not worry about instance 4 of structure C, since it falls completely outside of its region. It still must deal with instances 2 and 3, however. Since structures can have arbitrarily large numbers of primitives, it is necessary to be able to place bounding volumes around portions of structures (groups of primitives). Issues concerning the type of bounding volumes to use and the forming of primitive groups will be discussed in the next section.

Given that one has bounding volumes around each structure, one could consider building a hierarchy of bounding volumes and using various techniques to further speed up the rendering by reducing the amount of traversal [CLAR76]. However, this can only be readily applied to the max-set (where transformations that link hierarchy elements are defined), not to the min-set.

Aside from bounding-volume creation, one must also worry about the effects of editing on bounding volumes. If primitives within a structure are changed, the bounding volume may need to grow or shrink. Growing bounding volumes is typically easy; shrinking them may require reevaluation of all the contained primitives or primitive groups. Fortunately, editing of primitives within existing structures is not expected to be a frequent operation.

4.3.2 Bounding Volumes & Grouping Concerns

We must choose a bounding volume to use around each structure or primitive group. There are many different choices available; some examples are spheres, axis-aligned boxes, oriented boxes, and multiple-slab intersection [ARVO89, GOTT96]. These are listed in order of increasing complexity and tightness of fit, which together represent the major tradeoff associated with the choice of which to use. Tighter fitting volumes help to decrease the overhead resulting from the group overlap factor. This affects both the number of primitives that each processor must deal with as well as the amount of primitive communication. Since bounding-volume tests must be performed for every primitive group in every instance of every structure for each frame, it is also important for the tests to be as simple and fast as possible. Various techniques have already been developed to very efficiently cull-test bounding boxes [HOFF96]; for this reason, we have chosen for our experiments to use axis-aligned bounding boxes in model space, which become oriented bounding boxes in screen space.

We must also decide how the primitives will be placed together into groups. The goal of this operation is for each group to be as compact in space as possible (again, to reduce the group overlap factor), while not taking a lot of time to do this. The grouping of primitives into groups can be done in a number of different ways. One can group the primitives as they are received just by putting every N primitives in a new group. For applications that generate primitives in a spatially-coherent order, this technique works quite well. For other applications, one can buffer the primitives as they are received and then proceed to sort them into spatially-coherent groups. The techniques for doing this are similar to the adaptive techniques used for the load-balancing schemes mentioned in the previous chapter, except that they operate in three dimensions. For our experiments, we use a sorting technique very analogous to Whelan’s median-cut method (see section 3.2.2). In section 4.6, we take a closer look at various group-forming techniques.

If the primitives are only loaded during initialization, then one can afford to spend some time sorting them into compact groups. If primitives are being continually loaded during run time, then forming groups very quickly becomes very important, and sorting may require too much time. For this reason, a system may want to allow a choice of methods for forming the groups.

Finally, one must decide how many primitives should be placed in each group. Putting more primitives into each group would result in fewer overall groups to deal with, and having fewer groups would speed up the traversal process. However, putting more primitives into a group would also likely increase the bounding volume and therefore the group overlap factor. The experiments in chapter 3 looked into this problem somewhat; further explorations in this topic are presented in chapter 6.

4.3.2 Primitive Migration / Communication

We now examine the problem of determining how primitives will be handed off from processor to processor. We will continue to use the idea of placing primitives in groups; thus rather than keeping track of and sending primitives around individually, a processor would only manage and send whole groups.

There are two solutions for the problem of handing off primitives:

1. Push method: each group from the database is assigned to a particular processor, and each processor is responsible for making certain that its groups are sent to the right places.
2. Pull method: a processor will, upon seeing that it needs a given group, request that it be sent from some known location.

4.3.2.1 Min-Set Push Communication

For the push method, we must decide which processors will be responsible for which groups. A processor must make sure that the groups for which it is responsible are located on the correct set of processors. This bucket-sorting process is done by checking the bounding volumes for each group against the screen subdivisions (in our experiments we use the 2D bounding box of the 3D bounding volume). In order to know which processors must be sent a given group, the sending processor must store for each group a bit vector indicating which processors already have a copy.

As we did in previous chapters, we will use the terminology of calling a processor a "master" of a given group if it is responsible for the distribution of that group, while other processors that have a copy of the group solely for rendering purposes will be called "slaves".

Assuming that the assignment of groups to masters is static, there will typically be two sets of copies of the primitives present at all times: the static master set (where the groups were assigned) and the dynamic slave set (where the groups actually appear). If the assignment of mastership is allowed to be dynamic, this can be reduced to a single copy. However, without active load-balancing, the distribution of mastership may vary widely from processor to processor. Due to structure instancing, it is not clear that load-balancing the rendered primitives (the max-set) does anything to balance the master primitives (the min-set). Static mastership solves this problem since the master distribution can be assigned evenly while the database is being created. Having a good master load balance means that the initial sort-first pass (or two) over the database will be well balanced among the processors.

There are other issues between static and dynamic mastership. The latter increases the possibility that, on a given processor, structures will become empty, providing the opportunity to skip over such structures during traversal (see section 4.5). But dynamic mastership exhibits the off-screen primitive problem (section 3.3.2), where master primitive groups accumulate at processors rendering the screen edges and thus require redistribution. Static mastership avoids this problem.

During traversal, a processor will treat “master groups” and “slave groups” differently. For master groups, the processor must execute the bucket-sorting algorithm: it determines the set of regions in which the group appears based upon bounding-volume overlap. The master must then compare this set with the existing set of slaves and finally send the group to any new slaves.

For the slave groups, the processor needs only to perform a simple cull test to see whether or not the group appears in its region and proceeds accordingly. However, slave groups introduce a new challenge: one must decide when to delete them. One can only delete them when no instance of the group appears in this processor’s region. Since one cannot know this until after a complete traversal, one must either perform an additional “cleanup” pass or else pipeline the process and perform the cleanup during the next frame.

4.3.2.2 Min-Set Pull Communication

With the pull method, the job of making sure that a processor receives the right set of primitives is left to the receivers (as opposed to the senders). A processor will transform a bounding volume and see if it must render a given group. If it does not have the group, then it will request it from a known location (the group’s “master” processor). To establish known locations, however, requires again that there be two distributions of primitives: the static, known (master) distribution, and the dynamic (slave) distribution.

Unlike push communication, the pull method does not require master processors to store and manipulate the bit-vector indicating which slave processors have a copy of a given group. A larger advantage of the pull method is that the processors need only perform the simple cull test and not the full bucket-sorting algorithm for each group.

The disadvantages of the pull method are two-fold. One is the need to send request messages. This adds a significant number of messages to the rendering process compared to the push method. Using buffering to reduce the number of messages sent will likely increase rendering latency. The request messages also require processors to interrupt their normal work in order to respond. Finally, the pull method also requires that all processors transform and test all groups' bounding volumes for visibility (as opposed to only testing those groups that are already locally present). No portion of the hierarchy can be skipped over, since all groups are always potentially present until tested otherwise.

Load-balancing considerations are similar to those of the push method, except that the only additional work required of masters is the processing of request messages and the sending of groups as a result.

Considering the two methods, it appears that the push method is the better choice. Since it does not involve any primitive request messages, it has lower communication bandwidth and latency. The push method has much lower computational requirements and it also offers more potential for optimizations.

4.3.2.3 Bookkeeping for Arriving Primitives

With either communications method, there are some additional problems that must be dealt with, both concerning arriving primitives. When a processor receives a group of primitives, it must be able to store the primitives efficiently in the correct place in the hierarchy, and it must efficiently transform and render them, using the correct state information (e.g., transformation matrix).

We solve these problems by first observing that sort-first always performs at least two passes over the database: a sorting pass followed by a rendering pass. The first pass must be a complete hierarchical traversal. During this pass, we can save various information to make the second pass easier, as well as to solve the problems of dealing with arriving primitives. This information includes:

• A linear index table of all structures. Since all processors have a copy of the hierarchy skeleton, the structure numbers can be consistent across processors. One can use the structure numbers to specify where arriving primitive groups should be inserted into the hierarchy, and the index table allows the insertion to be done in small constant time.

• Combined transformation matrices. These would normally be discarded during single-pass traversal. By saving them using appropriate lookup tables, one can easily find the appropriate set of matrices to use to transform arriving primitive groups.

• Information about which primitives groups to render. We can efficiently deal with the set of already-present primitive groups that overlap this processor's region. During the hierarchical traversal, we create a local max-set view of the database, or a "flattened" view. In constructing this list, we can store pointers to the primitive groups, along with indices to the state information needed to render the appropriate instance of each group. By constructing this list, we can avoid a second hierarchical traversal of the database during the rendering pass; instead, we can perform a much simpler linear traversal of this list.

There is still a small problem to be worked out concerning arriving primitives. This is the fact that the receiving processor will not necessarily know which max-set instances should be rendered for the new min-set groups. Processors will need to be able to identify all the max-set transformation states that correspond to a given min-set structure. Such lists can be constructed by linking together corresponding max-set information structures. When a primitive group is sent, it can include information about which was the first instance where the group appeared in the destination processor's region. That processor can start from this point in the list and then perform cull tests for this group as it is transformed according to the rest of the instances in the list.

4.4 The Max-Set Method

With the minimal-set method, a processor must always transform and cull every instance of every group it has. With the maximal-set method, we reduce this work by choosing instead to keep track of each primitive-group instance separately. This information can be distributed among the processors rather than requiring each to compute it for every frame. With the max-set method, bounding volumes are not a necessary component. However, they can be retained with primitive groups to provide the bookkeeping advantages mentioned earlier (in section 3.3.3).

The max-set method has its disadvantages, however. It must generate and store bookkeeping data for all the primitive groups within the max-set. The problem is that the max-set can change significantly as the hierarchy is modified, and thus the basic database operations become more complex.

Let us consider the max-set method with respect to the example shown earlier. We have reproduced the necessary figures below. Each processor now keeps track of a max-set structure similar to figure 4.7b (perhaps in addition to the min-set structure in figure 4.7a). However, on a given processor, what is present in the structure instances marked "C" will vary from instance to instance. Consider the processor for region I (figure 4.8). For instance 1, it will have data for all of the primitives in the original structure; for instances 2 and 3, it will have data for only portions of the primitives; and for instance 4, it will contain no primitives at all.

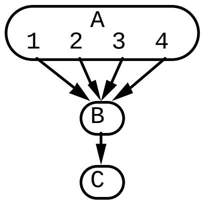


Figure 4.7a Minimal view

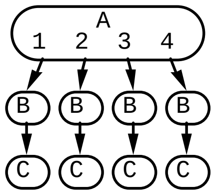


Figure 4.7b Maximal view

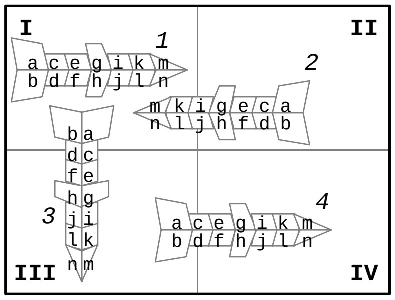


Figure 4.8 Example Database Scene

Now consider modifying the original database. If one wants to change a primitive in structure C, there are four instances that are affected. If one wants to modify the execute in B (perhaps to execute some other structure not shown), then all the information about the primitives within the four instances in C becomes invalid. If we imagine that structure A is a substructure itself and if it is executed an additional time, then the entire A-B-C subtree must be replicated to maintain the maximal set.

4.4.1 Max-Set Implementation

To keep track of all the primitive group instances, we could consider a simple replication strategy whereby we generate the max-set by simply making copies of the original data for every structure instance that comes into existence. However, this method has major drawbacks. Should any part of a given structure change, one must find and change every instanced copy. Thus each edit becomes a linear-time (or worse) operation rather than constant. Also, whenever a group instance is sent from one processor to another, the entire group must always be sent, since there is no notion of the primitive data vs. an instance pointer.

A much better way of building the max-set would be to first start with the min-set, then make a tree of pointers that corresponds to the max-set. Each max-set instance would contain only pointers to the original data within the min-set. Given the much greater feasibility of this pointer method over the replication method, the pointer method is the only one we will explore further.

Thus with the pointer method, we maintain two appearances of the database. In the min-set copy we store the exact same information as for the min-set method. For a given processor, this information includes (at least) primitive groups for which any instance appears in that processor's region. The max-set copy of the database contains an entire skeleton of the actual max-set. Within each max-set structure, there will be pointers to only the groups that appear in this processor's region for that instance. In addition, each max-set structure will point to its original min-set version. This will allow one to quickly find the non-primitive information for a given structure. Finally, the max-set can also be used to store certain information generated during traversal (namely, combined transformation matrices). One difference between the min-set and max-set skeletons in that the min-set uses "soft-links" for its execute calls (the calls are by structure ID), whereas the max-set would use "hard-links" (memory pointers).

4.4.2 Max-Set Operations

All editing operations are initially performed upon the min-set copy, and then the system must eventually update the max-set copy. As before, we retain the idea of master and slave processors for primitive groups that overlap multiple regions. The master processor is responsible for making sure that the group goes to all the necessary slave processors. To extend this idea to the max-set method, we make the master processors responsible for properly distributing all the group instances. Thus the master must consider not only which processors have a copy of a given min-set primitive group, but also which processors have a link to that group for a given max-set instance.

Now let us consider how to perform the basic operations on the max-set database. The creation of new structures is straightforward; it only affects the min-set. The deletion of a structure is similar: you should only delete non-instanced structures. Posting a structure requires the construction of a max-set tree, while unposting a structure requires deletion of the same. The expense of posting structures can be minimized by deferring the real work until scene traversal time. Similarly, unposting can be done by putting the tree in a garbage-collection list, to be cleaned up after traversal and rendering. Editing parameters and matrices within the hierarchy is straightforward. Editing primitives can also be straightforward assuming one uses the “static” master system mentioned earlier (meaning the primitive will be easy to find). Finally, changing an execute is potentially complex, but again we minimize the work by delaying it.

4.4.3 Max-Set Traversal

All of the complex work is delayed until scene traversal time. During the first pass, a processor must simultaneously traverse both the min-set and the max-set, making sure that the max-set appropriately parallels the min-set. When there are pieces of the min-set that do not exist in the max-set (due to a new post or execute), the processor creates them on the spot. When the processor comes across pieces in the max-set that have become obsolete due to database edits, these are added to the garbage-collection list.

When a new instance of a structure is to be created in the max-set, a processor places within it pointers to all of its master groups for that structure. This guarantees a complete yet disjoint initial distribution for the max-set structure instance.

During traversal, a processor considers each group pointer in the max-set structure. If it points to a master group, then this processor must see that this group instance (and its corresponding min-set data) is properly distributed to all necessary slaves. If the group pointer is directed at a slave group, then the processor only decides if it still needs to keep the pointer. If it does not need it anymore (due to the associated instance no longer falling in this processor’s region), then it can delete it and possibly also the associated slave group data. To know about this possibility, reference counts can be used to keep track of how many max-set instances are pointing to the min-set data. One issue here is that the slave group data cannot be removed until after the frame has been rendered. This is because pointers to the data may be received from other processors (see below). A solution to this problem is to implement a “delete candidates” linked list. The slave group data is added to this list whenever its reference counts go to zero. Any incoming pointers to this data can take it off the list. After the frame has been rendered, the remaining delete candidates can be deleted.

The bookkeeping just discussed is greatly simplified if static masters are used. If dynamic mastership is used, then one must be careful to migrate all the information associated with a given primitive group: which processors have copies of the actual group, and which processors have a copy of each instance. Not transferring this information together leads to greater complications.

4.4.4 Max-Set Primitive Migration

We now devote some additional attention to the problem of transmitting primitives, since new complications arise due to the fact that we have to worry about both min-set primitive groups and their max-set instances.

When a processor sees that it must send a group instance, it will go ahead and place a message regarding that instance into the appropriate outgoing buffers. The processor must also check to see if the receiving processor has a copy of the original group data. If not, this is also queued to be sent.

The difficulty in this process is finding an efficient way for the master processor to pass along the instance pointer information. The pointer must refer to the original group by an ID (since memory pointers are not valid across processors). The receiver must have a quick way of looking up the ID to see where in its memory the group is found. One solution is to place the group IDs in a hash table. Keeping a group index table for each min-set structure may also be a practical solution.

Thus when a processor receives a group of primitive data, it just places it in the appropriate structure in the min-set. When a processor receives a group pointer, it places the pointer in the appropriate max-set structure and links it to its min-set data. It then (or eventually) renders that max-set group.

4.5 Improving Efficiency

For various reasons, in a distributed graphics database one may find structures that contain no primitives. One reason is when the number of primitives in that structure is smaller than the number of processors times the number of primitives per group. Applications sometimes have large numbers of small structures (in addition to smaller numbers of larger structures), and thus locally empty structures are likely. In addition, with dynamic mastership, sort-first presents the possibility that primitives will migrate out from structures and leave them locally empty on some processors.

Given the presence of many empty structures, it pays to optimize traversal by not traversing into empty portions of the hierarchy. Bounding volume hierarchies (see section

4.3.1), applicable to the max-set method, provide this possibility for locally off-screen portions of the hierarchy. It does not help with empty on-screen structures.

A more general method is to simply keep a flag for each structure to indicate the presence of any primitives. For the min-set, this is about all one can do. During traversal, one can skip empty structures, though backtracking will be required if non-empty structures are found further down the tree.

For the max-set, the flags can be made hierarchical, indicating the presence of primitives within entire branches of the structure hierarchy. One difficulty with this method is that any primitive groups that arrive in previously "empty" portions of the hierarchy require backtracking to compute the relevant transformations to use. Another small problem is the fact that, if all the processors are not performing the same traversal, it is not as easy to generate synchronized structure indices. One can overcome this problem either by always doing light traversal (to compute structure numbers), or by keeping track of the number of sub-structures under any given structure.

4.6 Sort-First Implementation

As mentioned in chapter 3, we have constructed a sort-first software implementation that provides the load-balancing mechanisms discussed in that chapter as well as both of the database-handling algorithms discussed in this chapter. It was through the implementation of these algorithms that we discovered many of the facts that have been presented thus far. In this section, we present some additional information that was gleaned during the implementation and testing of this system.

4.6.1 Description of Setup

Section 3.6.1 gives an overview of the system. As mentioned there, this parallel system uses PVM in order to provide separate processes for the host program, each graphics processor, and the framebuffer. Four different main rendering loops were written to provide for the combination of two load-balancing methods and two database-management methods.

In order to test the hierarchical features of the database, we obtained a hierarchical model of a Ford Bronco. The Bronco consists of 466 structures and 74,647 triangles. We constructed a test case wherein the Bronco was instanced three times, with the instances distributed around a circle. In the animation, the camera revolves around the circle while viewing approximately one half of the circle's area. The images below show representative frames from the animation.

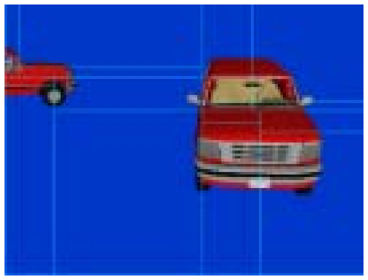


Image 4.1a

Image 4.1b

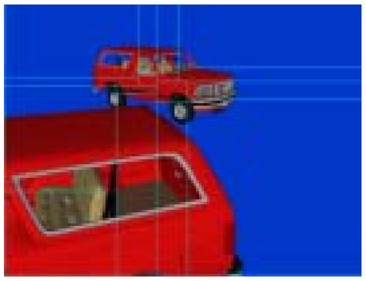


Image 4.1c

4.6.2 Primitive Group Formation

Our initial approach to placing primitives into groups was the simplest. Referred to as "unsorted", this strategy simply takes every N primitives and places them into a group, where N is the selected group size. However, we had concerns that this method would result in overly large groups (with bad consequences as a result of the group overlap factor), and thus we implemented two other strategies as well. One strategy ("partially sorted") involved first inserting the primitives from the input file into an octree data structure, then recursively traversing the octree cells (in a spatially-coherent order) and putting every N primitives into a new group. The next strategy ("sorted") was to read in the primitives in each structure and form the groups by performing a 3D spatial-subdivision. The algorithm uses a median-cut method to subdivide the bounding box of the primitives repeatedly until the proper number of groups is achieved (the splitting techniques from section 3.4.3 are incorporated to handle arbitrary numbers of groups). Chart 4.1 details the results of these different strategies for the Bronco database:

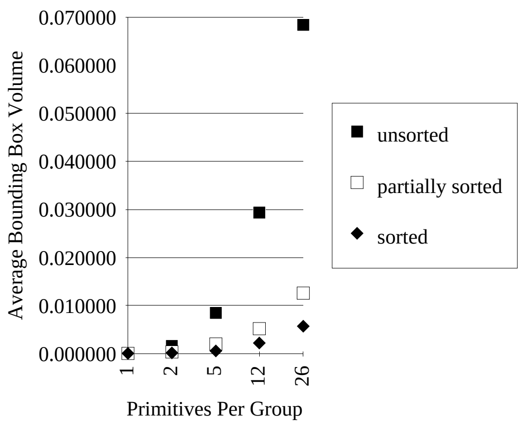


Chart 4.1 Grouping strategy results

As one can see, the sorted method resulted in the smallest average bounding box volume. Because of this, the sorted method was chosen for use with the experiments done here and in chapter 3. As the sorting is only done at database load time, this is a reasonable choice for a model with static structures.

4.6.3 Min-set vs. Max-set

In writing the code for the implementation, we discovered that the differences between the min-set method and the max-set method were not so great as initially believed. As mentioned in section 4.3.2.3, one constructs a max-set-like list of primitives to be render even in the min-set case. Our implementation took advantage of this by using the same max-set data structures for either case. Thus the biggest difference is that the min-set method creates and destroys the max-set structure for each frame, whereas the max-set method retains it and only modifies it as necessary.

This is not the only difference, of course. Another difference has to do with the communication. The min-set method always sends primitive groups, whereas the max-set method sends both primitive pointers and primitive groups. In practice, however, nearly every primitive pointer will be accompanied by a primitive group, since (for many applications), it is rare that the same processor has multiple instances of a given primitive group appearing within its screen region. An obvious small optimization would be to combine the group with a pointer, since a pointer must accompany every group (but not every pointer must be accompanied by a group).

The other main difference between min-set and max-set has to do with the number of transformations performed by each method. In this case, the differences become very application dependent. If an application creates only one instance of each structure, then the number of transformations performed in either case is identical. However, as more and more instances are created, then the max-set method gains an advantage. While both methods will perform the same number of transformations for the master primitive groups, the number of transformations done for the slave primitive groups differs. The min-set method must transform every instance of each slave group, whereas the max-set method only transforms those instances that last appeared within its region.

To help us quantify this transformation difference between the min-set and max-set methods, we gathered statistics on several test runs of the Bronco animation. Both the min-set and max-set bookkeeping methods were tested using varying group sizes and numbers of processors. The number of primitives per group included: 5, 12, and 26. The number of processors included: 4, 8, 12, 16, and 20. For each frame drawn, we counted the total number of transformations that occur in each. This includes transformations of the master groups, the slave groups (including newly received ones), and of all the primitives in the groups that overlapped any processor's region. We count one transformation for each group or primitive. For a given set of test conditions, the statistics across the processors and across the frames in the animation are averaged together. This provides us with a broad view of each test case. The results are shown in chart 4.2.

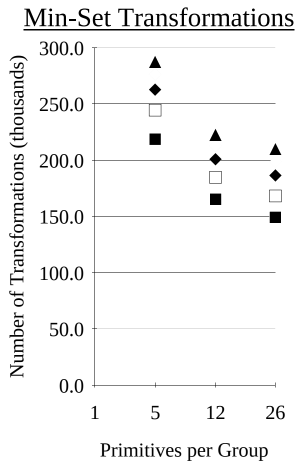


Chart 4.2a

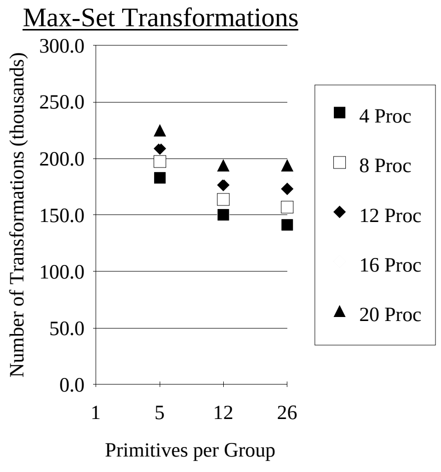


Chart 4.2b

The total number of transformations has an interesting relationship to group size. Larger groups mean that there are fewer of them to transform. However, as mentioned before, larger groups increase the overlap inefficiency. Comparing the results on charts 4.2a and 4.2b show us that the max-set method has much less transformation work to do than the min-set method when the group size is small, yet this difference diminishes rapidly as the group size is increased. The source of this difference comes from the number of slave group bounding boxes that are transformed. This figure is three times higher for the min-set method, which is what we expect, given that there are three instances of the Bronco being executed.

At the group size of five primitives, the difference in the number of transformations is large enough to suggest that the max-set method has a good advantage. At the group size of twelve primitives or larger, that advantage appears greatly diminished. Given that the implementation differences between the two methods are not very large, it seems reasonable to choose the max-set method in any case. The effects of the group size upon communications and load-balancing were discussed in chapter 3, and further studies into the effects of group size will also be discussed in chapter 6.

4.7 Immediate Mode

There are certain applications for which immediate mode operation is the only practical choice. An example is a scientific visualization application that is generating its graphics data base in real time as a simulation is progressing. It is therefore worth some consideration to investigate how immediate mode and sort-first would work together.

4.7.1 Immediate Mode Variations

As mentioned in section 4.1.3, immediate mode comes in at least two different varieties. These include "stateless" and "stateful," differentiated by whether or not complete rendering state information is always bundled together with every primitive or whether it can be specified separately for groups of primitives. Given that popular immediate-mode APIs are based upon the "stateful" model, we will focus discussion on this method. Many of the problems that will be discussed only apply to this method; some problems will concern both.

In a serial immediate-mode application, a single stream of graphics commands is passed from the host to the graphics system. The commands include both primitives and state-changing operations such as matrix transformations, surface parameter changes, or rendering mode changes. Because it is a single stream, the association of state changes to primitives is simple: each state change affects any primitives that follow. Also, the drawing order of the primitives is given by their order in the command stream.

Parallel immediate mode has several meanings:

- the graphics system may be parallel
- the application host may be parallel
- both systems may be parallel

Since we are interested in immediate mode only as it relates to sort-first, we consider only the first and last possibilities.

4.7.2 Parallel Immediate Mode Issues

When moving from a serial immediate mode context to a parallel context, many issues arise. One must find a way to distribute the stream and still maintain the semantics of the command stream. One must address both the state semantics as well as primitive ordering semantics. If there are several command streams (coming from a parallel host), one must decide on a set of semantics that makes sense, and then choose a distribution scheme that affords them as well as provides for other parallel issues such as load-balancing. One must also decide issues involving communications parameters such as buffering. Finally, when we combine sort-first with immediate mode, we cannot address the issue of distribution without considering sorting. Should the application host processor perform the sorting, or should the graphics processors do this? We start by looking at distribution strategies.

4.7.2.1 Distribution Strategies

If we consider a single command stream feeding a parallel graphics system, it is obvious that the stream must be subdivided and distributed among the graphics processors for efficient processing. For sort-first, there are three methods one may consider for command stream distribution. One is for the host to simply distribute the command stream to the graphics processors with the only concern being to provide the various graphics processors with approximately equal portions of the command stream. In this case, it is up to the graphics processors to perform the region-based sorting of the primitives after they have been received.

Another possible distribution method is for the host to take on some or all of the sorting work. In this case, the host is provided with a mapping of screen regions to graphics processors, and it distributes the primitives to the processors based upon screen-region overlap. This method moves the initial work of the sort-first graphics pipeline up to the host, since it requires the host to determine how the primitives map to the screen. For some applications, this may be an excessive amount of work with which to burden the host.

The final distribution method relieves the host of the sorting work by providing an additional processor between the host and the graphics processors. In this setup, the host will send its undivided stream of commands to the intermediate processor. This processor will then perform the sorting and distribution of the command stream. This sorting processor can (or rather, should) be closely coupled to the host processor such that the host's communication with it does not consume bandwidth in the network connecting the sorting processor and the graphics processors.

If we now assume that the application is running on a parallel host producing multiple streams of graphics commands, we must not only consider how the streams will be distributed but also what kind of semantics should apply. A reasonable semantic structure is for each stream to be considered a somewhat unique context, with only certain global state information (viewport mapping, background color, etc.) being shared among all streams. [IGEH98, MACI98]

There are various ways that a parallel host can communicate with a parallel graphics system. If the ratio of application processors to graphics processors is one-to-one and each application processor is producing similarly balanced streams, then it makes sense for each application processor to talk to a single graphics processor. In any other context, even distribution of the command load suggests that each application processor should distribute its command stream to all of the graphics processors. The strategies for this are the same as those just mentioned for the single application stream. However, each graphics processor must now maintain a context for each command stream that it is receiving.

4.7.2.2 Dividing the command stream

The choices for dividing the stream are scattering and clustering. Scattering sends consecutive primitives to different processors, while clustering sends consecutive groups of primitives to different processors. In fact, these choices actually just represent different points along a continuum of how many consecutive primitives to send to each processor before switching to the next processor. Clustering is the better choice for sort-first. Consecutive primitives are often likely to be spatially coherent, and keeping them together is desirable for sort-first. Also, the more one subdivides the stream, the more work one has to do to preserve the stream semantics (discussed below).

Having chosen clustering, we must decide how large the clusters should be. Since preserving semantics requires that state information must be added to the start of each cluster (see below), having larger clusters would help to amortize this overhead. The larger clusters would also have the potential for less communication overhead, since there would be fewer clusters to send. However, making clusters too large can also increase the system latency, since no work begins until the first cluster is sent, and the frame cannot finish until the last cluster arrives. Also, the number of clusters relative to the number of processors should be large enough that each processor will have close to an equal cluster load. Thus cluster size is a very application and system dependent issue.

4.7.2.3 Maintaining Stream Semantics

Preserving the stream semantics with a distributed command stream can be done in a straightforward (if inefficient) manner. With this method, the application host must maintain an out-going buffer for each graphics processor. All state-changing commands are “broadcast” by putting a copy in each buffer. Primitives are placed only in the single buffer for the destination processor.

With this approach, there is the likelihood that a given buffer will contain redundant state-changing commands. To avoid these requires additional bookkeeping. Assuming that the various states that can be changed are relatively orthogonal, one can keep a table of state that has changed for each out-going buffer. Each time a given state is changed, the Boolean entries in all tables are updated. When switching to a new buffer, the application host must examine the table and place in the buffer appropriate state-changing commands with the most recent values for any changed state. At switch time, the table for the old buffer is also reinitialized to “no changes.”

If one must sort primitives on the graphics processors, preserving semantics becomes more difficult. This is because a given processor will not have any information about the state context on other processors. To solve this problem, one must either create a common reference or else send complete state information with each batch of redistributed primitives. The latter strategy is fairly self-explanatory, though costly. For the former, one must always broadcast every state-change command and assign to each set a sequence number. When primitives are redistributed, the appropriate sequence numbers are attached to provide the rendering state context. The graphics processors will have to retain a lengthy queue of the state information to refer to given the sequence numbers. In addition, building and extracting the proper information from the queue can involve a costly overhead. With either method, one could reduce the overhead by limiting the amount of state information in the command set.

4.7.2.4 Maintaining Ordering Semantics

The issue of primitive-ordering semantics complicates parallel graphics system. Forcing primitive ordering can be required for several reasons. One is for non-Z-buffered rendering, where priority-based rendering is used to solve the occlusion problem. Another reason may be to support certain anti-aliasing algorithms that require ordered rendering. Ordering is also useful with Z-buffered rendering to make certain that nearly co-planar surfaces are rendered correctly. Finally, primitive ordering is necessary for transparent objects to be rendered correctly.

If sorting is done at distribution time, then no special work is needed. If sorting occurs later, the original primitive order can be lost, and one must work around this. In some cases, only a few primitives must be kept in order with respect to each other. For this situation, one solution is to keep all order-related primitives in the same cluster. If the cluster size is exceeded by such a group of primitives, then additional clusters may be used, but they must be sent to the same processor. Databases with large order-dependent sections would require a different solution. If database order is necessary only to solve coplanar-polygon issues in a Z-buffered system, then a solution is to append the primitive sequence number to the low-order side of the value used for Z-buffering. If true ordering is required, a synchronization problem occurs: a given processor cannot proceed past a given point in the command stream until it knows that it will receive no more primitives from other processors which might be inserted before that point. This case is best solved by doing sorting during distribution.

4.7.3 Sorting (Classification)

As mentioned, for immediate-mode command streams on sort-first, one can choose to sort the primitives either during or after distribution time. We saw that there are various problems that occur when performing the sort later. In this first section, we consider how one can efficiently implement the sort process during distribution time.

4.7.3.1 Sorting During Distribution

The strategies discussed in this section apply equally to sorting performed on the host or sorting performed on a sorting processor. To simplify things, we will discuss the issues from the point of view of the host performing the sorting.

There are two possible strategies for the host to sort the primitives: sorting in screen space and sorting in object space. Sorting in screen space involves performing some of the same steps that would otherwise be performed on the graphics processors. The host must transform the vertices into screen space and determine which screen regions are overlapped. As mentioned earlier in the dissertation, one can perform classifications on bounding boxes in small constant time. For the host to transform the primitives requires it to keep track of the transformation matrices the same way that the transformation processors do. One can avoid transforming every primitive's vertices if one computes a bounding volume for all the primitives in a given cluster (group). In this case, one must only transform the bounding volume and sort based on its overlap.

If one wishes to avoid transformation of primitives on the host, one may perform the sort in object space. Doing so requires that one transform the viewing frustum and its partitions into object space. Since the usual transformations specified in the command stream transform object space into screen space, one would need to compute the inverse of these transformations to transform the viewing frustum from screen space into object space. For the common simple transformation, finding the inverse is a simple computation. Of course, if the user is concatenating a given transform matrix with the top of the matrix stack (as is frequently done), then the inverse transforms must also be managed and concatenated in the same way.

Transforming the individual screen partitions into object space can be simplified by specifying the partitions parametrically (i.e., using normalized viewport coordinates). Then one only needs to transform the four corners of the viewport into object space and finally compute the vertices of the partitions using simple linear interpolation.

Once one has determined the positions of the screen partitions in object space, classifying the primitives (or groups of primitives) becomes a matter of plane tests. Unfortunately, this is the big disadvantage of classification in object space: the author knows of no simple constant-time algorithm for performing this kind of classification. Using a recursive algorithm, classification in object space requires between O(log N) time for small primitives to O(N) time for screen-size primitives, where N is the number of screen regions.

With this problem in mind, it appears that classification in screen space is the simpler and faster method. If one performs classification based upon the bounding boxes of groups of primitives, then this appears to be a reasonable amount of work to transfer to the host system. Again, by performing the sorting on the host, maintaining the semantics of the command stream across the parallel system becomes a simple matter. Assuming that multicast communication is efficient, host-side sorting also reduces the total communications cost needed by the sorting, since primitives need only be sent once (rather than sent and then redistributed).

4.7.3.2 Sorting After Distribution

Sorting on the host, of course, takes away computing resources from the host system. If one wishes to avoid this without the expense of adding sorting processors, then one must turn to sorting primitives on the graphics processors. As most of the problems of sorting after distribution have already been discussed, we summarize the issues here.

When sorting on the graphics processors (GPs), we have the host simply distribute batches of primitives to the GPs. The distribution only attempts to send equal numbers of primitives to the GPs and to maintain the command stream semantics. Once a batch of primitives is received on a given GP, that GP must transform them into screen space and sort the primitives to where they need to go. Thus it will likely have to send away about (N-1)/N percent of the primitives (where N is the number of graphics processors), since there is only a one in N chance of a primitive being sent to the correct processor initially.

The main difficulties with GP-side sorting are the additional overhead needed to maintain command-stream semantics, the complications caused by ordered primitives, and the additional communications bandwidth required. The main advantage of GP-side sorting is that the host is freed of all sorting computations. It still must worry about some distribution and semantic issues, but these involve relatively little computation.

4.7.3.3 Sorting and Communications

With a retained mode database, sort-first is able to take advantage of frame-to-frame coherence to reduce the communications bandwidth needed for sorting primitives. With immediate mode, one loses this advantage since primitives are always generated anew and must be completely resorted for each frame. Thus an immediate-mode system will require more processor-to-processor communications bandwidth than a retained-mode system.

Immediate mode already assumes a high-bandwidth connection between the application host and the graphics system. This is especially important for a system involving a single host processor, since the bottleneck in the graphics pipeline will likely be the communications of primitives from the host memory to the graphics system. Some immediate mode interfaces (such as OpenGL) attempt to overcome this bottleneck by adding retained structures (display lists) to the immediate-mode command set. If an application can greatly benefit from such structures, then it will likely be better off using a retained-mode interface to begin with.

4.7.4 Load Balancing

We have discussed the issue of seeing that graphics processors get similar portions of the command stream, but now we consider how the screen-subdivision and region assignment techniques discussed in chapter 3 apply to immediate mode. In brief, most of the load-balancing methods suggested for retained mode apply equally well to immediate mode. Practically speaking, adaptive load-balancing algorithms would have to use the primitive distribution information from the previous frame to determine the screen subdivision for the current frame. Using the current frame's information would require buffering all the current frame’s primitives prior to sorting and rendering them. This would increase latency significantly and require large amounts of buffering space.

4.7.5 Conclusion on Immediate Mode

With immediate mode, sort-first becomes very similar to sort-middle. As discussed in [MOLN94], there is still a notable difference, though. Sort-first is redistributing raw application primitives, while sort-middle is redistributing transformed primitive data. The difference can be significant if the application primitives are high-level primitives. A high-level primitive generally requires less data than a display primitive does, especially if the high-level primitive tessellates into many display primitives (like a polynomial surface patch). Thus in such cases there may be a communications advantage in sort-first over sort-middle. In other cases, the advantage is less clear, and the distinction between the architectures becomes blurred.

5. Sort-First Implementation

In this chapter, we look at various issues concerning sort-first hardware implementation. In section 5.1, we examine various communication and design issues that one must consider when designing sort-first hardware. In section 5.2, we look at the possibility of implementing sort-first over an existing hardware system, namely the UNC-CH PixelFlow system.

5.1 Sort-First Hardware Implementation Design Guide

In this section, we will discuss the design of a sort-first hardware implementation. We will focus on the questions of which design decisions are important for a sort-first architecture, rather than attempting to discuss every possible detail. Before we begin, we will first set some performance goals that we wish to attain with the system design. These goals will help guide some of the design decisions by providing a context in which we can discuss specific performance figures. After this, we consider the overall system architecture, and then we discuss issues involving the communications network, the rasterizer design, and the frame-buffer.

5.1.1 Performance Goals

We wish to design a sort-first system capable of the following:

- 30 frames per second
- one-million triangle (on-screen) scene
- 16-million pixel screen, using four 4-million pixel framebuffers
- 16-sample antialiasing
- average 50-pixel triangles
- rendering from a retained-mode hierarchical database
- 5% of the database changes between each frame

The framebuffer setup was chosen based upon the fact that a four-million pixel display is approximately the largest one can expect from a single display. Our goal is for a very high-resolution system, and thus the target system uses four such displays for producing a larger, composite output screen.

We will assume that the triangles are to be textured, lit, Gouraud-shaded, Z-buffered, and fogged. We shall also assume that colors will be specified with 24-bit precision plus an 8-bit alpha value, and that all coordinate values will be specified with 32-bit precision (including surface normal values and texture coordinate values).

We expect that to achieve this goal, a system of approximately sixteen rendering nodes will be required. This is based upon the fact that four million triangles per second is a maximum rendering rate that one can expect from each node [3DLABS, NEC99]. This allows us to assume an average rendering rate of two million triangles per second and allow for two-to-one load imbalances. Given that the screen will be subdivided into sixteen regions, and based upon the primitive statistics, we can also assume that the primitive overlap factor will be fairly negligible.

5.1.2 Overall Architecture

The overall system architecture will follow the same architecture we have assumed throughout the dissertation. This consists of a host computer system attached to a parallel graphics subsystem. The graphics subsystem consists of many processing nodes attached to a common communications network. One may view the host system as a special node (or set of nodes, for a parallel host) attached to the same network.

We will refer to the processing nodes of the graphics subsystem as rendering nodes. Each such node will contain a CPU and additional hardware for rasterizing polygons. The design of the rasterizer will be discussed in section 5.1.4. Some of the graphics processing nodes will serve as framebuffer nodes. Whether these nodes are also rendering nodes is an issue that will also be discussed in section 5.1.4.

The system operation is also as already detailed. The host system sends commands to create a retained hierarchical database on the graphics subsystem. When the command is given to render a frame, the rendering nodes each render the primitives in their portion of the frame and send the resulting pixels to the appropriate framebuffers.

5.1.3 Communications Network

While a sort-first system does not place many specific demands on the communications network, there are some capabilities that are desirable. The ability to do broadcast communication is desirable when transmitting commands and data from the host to the graphics nodes. This is also useful when the rendering node in charge of load-balancing computations needs to send the screen partitioning information to the other rendering nodes.

A communications network that provides multiple point-to-point communication paths is desirable over one that provides only a single, blocking communications medium. During rendering, while sorting the primitives, each rendering node will need to send batches of primitives to several other nodes. This many-to-many communication pattern may congest a bus or other single-channel communications setup.

In order to create an easily and economically scalable system, it is desirable that the communications medium be easily embedded in a backplane. Each rendering node would be contained on a circuit board that would plug into this backplane. The communications networks that allow easy embedding into a backplane are point-to-point networks, arranged in either a linear or ring fashion. Aside from easy scalability, these arrangements also offer short, uniform-length signaling paths. These properties allow the possibility of reducing complexity and expense when designing high-speed communications links.

At this point in time, let us examine the communications bandwidth requirements for our desired sort-first system. First, let us examine system-wide bandwidth:

System Item	Derivation	Size
editing messages	model size * % changing * prim. size	5.8 MB
primitives being sorted	C * on-screen database size * prim. size	58.0 MB
load-balancing info.	mesh cells (16K) * cell size (4)	< 0.1 MB
pixels sent to FB	resolution * pixel size (4)	64.0 MB
Total data per frame		~128 MB/frame
Bandwidth	(at 30 frames/sec)	3840 MB/sec

To compute the bandwidth from the editing messages, we assume that five percent of the model is changing during each frame. The primitive size is 116 bytes. This is the same number as taken from Chapter 2. (This figure was chosen for simplicity; more realistically, one would use triangle strips to reduce the bandwidth and computation requirements.) For the primitives being sorted, we assume a redistribution coefficient (C) of 50%, representing a worst case figure for which the frame rate must be maintained. The mesh size chosen was 128x128.

Now we consider the bandwidth with regard to single nodes. First, we examine the traffic for a rasterization node. We start with the received data:

Rasterizer Item	Derivation	Size
editing data received	model size * % changing * prim. size	5.8 MB
primitives received	58 MB * 1/N * imbalance ratio (2)	~7.3 MB
other data received		< 0.1 MB
Total received / frame		~14 MB/frame
Receive Bandwidth	(at 30 frames/sec.)	420 MB/sec.

For the primitives being sorted, we divide the total figure by the number of processors, and then consider how imbalanced this figure may become. Based upon the experiments from Chapter 3, we use a maximum/average ratio of 2.0 as the worst case for the primitives being received, and a ratio of 1.1 for the primitives being sent (below). We do a similar calculation for the number of pixels being sent, although the imbalance ratio in this case is only estimated. The entry for "other" data is in reference to any load-balancing data and frame-drawing commands that must be sent or received. Here we summarize the data being transmitted:

Rasterizer Item	Derivation	Size
primitives sent	58 MB * 1/N * imbalance ratio (1.1)	~4.0 MB
pixels sent	64 MB * 1/N * imbalance ratio (4)	16.0 MB
other data sent		< 0.1 MB
Total sent / frame		~21 MB/frame
Send Bandwidth	(at 30 frames/sec.)	630 MB/sec.

Next we consider a framebuffer node. In this case, data are only being received:

Framebuffer Item	Derivation	Size
pixels received	resolution * pixel size (4)	16.0 MB
other data received		< 0.1 MB
Total data per frame		~17 MB/frame
Bandwidth	(at 30 frames/sec)	510 MB/sec

If one were to combine a framebuffer node with a rasterizing node, the total data received per frame would be approximately 31 MB in the worst case. This would require a bandwidth of 930 MB/sec. at 30 frames per second.

A network utilizing 160-bit bi-directional data paths operating at 200 MHz would be capable of providing the 4 GB/sec bandwidth necessary for the operation of this sort-first system. The back-end network of PixelFlow provides a 256-bit bi-directional data path operating at 200 MHz; a modified version would be suitable for use with sort-first.

From the tables above, we see that there is a significant amount of traffic due to both primitives and pixels. One might consider implementing a separate communications network for the pixel traffic versus the other traffic (primitives and commands). This question comes down to whether having two slower networks is better than having one faster network. Usually, the versatility of having a single faster network is preferred, since this does not place a firm boundary on the amount of bandwidth available to each kind of traffic. Another design decision to consider is whether there should be separate network ports on a node, with one attached to the transformation engine (the node CPU) and another attached to the rasterizer and/or framebuffer. This looks like a good choice, given that the primitive traffic alone will be a fairly heavy burden on the CPU's memory bus. Putting the pixel traffic on its own port prevents adding further load to this bus.

We would like to have incoming messages arrive and be buffered such that they do not block the reception of other messages. Also, we wish to avoid performing unnecessary memory-to-memory copies of incoming data. To best handle these needs, the system should have intelligent processing of incoming messages. It should be able to take different kinds of incoming messages from the receive FIFO and store them directly in main memory according to the data structures appropriate for the message contents. Depending upon the processor characteristics and requirements, this task can be handled by either a main processor interrupt or else by a separate communications processor.

Sending messages should be handled in an intelligent fashion as well. We desire to be able to send data without first copying it to an in-memory message buffer, and without stalling the main processor while the transmit hardware is busy. A simple DMA command processor would be able to satisfy these requirements.

5.1.4 Rasterizer and Frame Buffer Setup

With regard to sort-first, we have two main considerations regarding the rasterizer and framebuffer design. One is that the local framebuffer will vary greatly in size, from only a few pixels up to perhaps 1/4 of the total viewport. Second, we require a fast connection between the local framebuffer memory and the communications network.

As mentioned in section 3.9.2, there are two common approaches to rasterizer architecture: the conventional framebuffer-based approach, and the tile or region-based approach. Given that in sort-first, the local framebuffer size can vary so much, a region-based approach appears to be more practical. With the conventional rasterizer architecture, each sort-first rendering node would need a large local framebuffer (close to full-screen size) capable of storing all the necessary pixel information. These large local framebuffers would be unnecessary with a region-based rendering approach. We note that requiring a region-based approach does not necessarily require a region-based architecture. A conventional rasterizer having only a limited amount of local framebuffer memory could perform region-based rendering as well.

Thus a region-based rasterizer in a sort-first system would render its portion of the final image one region at a time, sending the resulting pixels to the appropriate framebuffer or framebuffers. In order to eliminate rendering delays that may be caused by communications bottlenecks, the rasterizer should be able to buffer one or more regions' worth of pixel color values after they have been computed. This way the rasterizer can continue to work on new regions while the buffered regions are waiting to be sent.

The disadvantage of using region-based rendering comes from the fact that an entire region must be finished before starting on the next one. This requires sorting of the input primitives on a per-region basis. Thus all of the input primitives must be buffered. Since the sorting cannot be finished until all of the input primitives have arrived at a given node, the rasterizing cannot really proceed until this point has been reached. The need to wait for all the primitives to arrive increases the rendering latency. [ELLS89]

Eliminating the additional latency requires additional pixel memory. One solution is just to switch to the conventional architecture and have local framebuffers large enough to render any local screen partition. A more cost-effective solution may be to combine a region-based rasterizer with several regions' worth of pixel storage memory. Thus the rasterizer could begin to work on one region before all of the primitives have arrived. When it runs out of primitives for that region, it would store aside the unfinished pixels and then start working on another region. This process would repeat until the rasterizer either runs out of storage space or regions left to process. During this time, the rest of the primitives will hopefully have arrived and been sorted, and the rasterizer can then reload each unfinished region's pixels and finish rasterizing them with the remaining primitives. Compared to the conventional architecture, the advantage of this approach is that the additional buffer memory need not have extremely fast random-access characteristics (as required by the conventional architecture).

5.2 Sort-First Using PixelFlow

In the previous sections, we discussed some of the issues relevant to putting together a sort-first graphics system. In the following sections, we examine an existing graphics system and see how it can be adapted to sort-first operation. The graphics system we examine is PixelFlow, a system conceived at UNC-CH [MOLN91, MOLN92] and brought into existence with efforts from Division and HP [EYLE97]. First we describe the system and its conventional operation. Next, we examine how the sort-first algorithm needs to be adapted to fit this system.

5.2.1 PixelFlow Description

5.2.1.1 Overall Architecture

PixelFlow is designed to be a scalable sort-last graphics system. As with other high-end graphics systems, the system design includes a host computer running an applications program communicating over a high-speed network with several graphics processing nodes. What makes it a sort-last system is an additional fast and wide network designed explicitly for pixel communication between the GP nodes. A system diagram is shown below, and each part will be discussed briefly in turn.

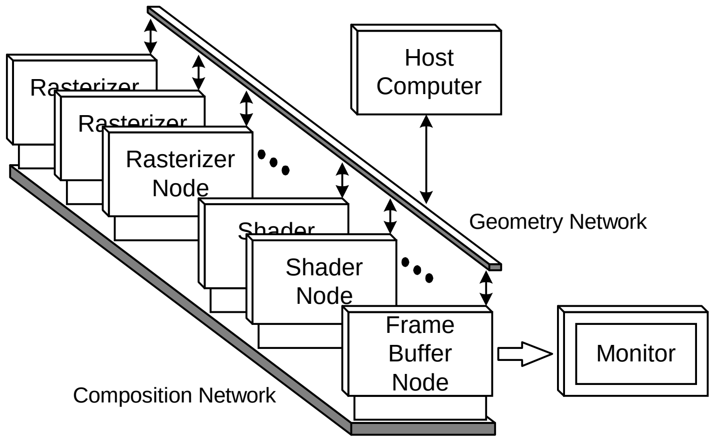


Figure 5.1 PixelFlow System Diagram

5.2.1.2 PixelFlow Node

A PixelFlow node consists of a “GP” (geometry processor) board and a rasterizer board located on opposite sides of a midplane. These two boards are coupled closely together by connections going through the midplane. A node may also have an adapter board attached to it, as described below. Figure 5.2 below illustrates the major parts of a node board.

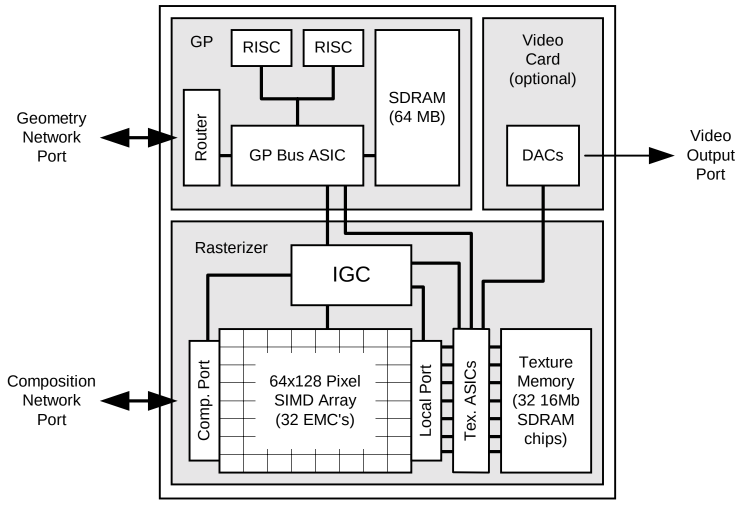


Figure 5.2 PixelFlow Node

A GP board contains two Hewlett Packard PA-8000 RISC processing units attached to the system processor bus. A custom ASIC ("RHInO") connects the processor bus to the main memory and to the other system components. A GP board may be populated with 64 or 256 MB of memory. The RHInO chip contains a semi-intelligent DMA engine that sends data from main memory to the rasterizer unit. Another port on the RHInO is attached to an ASIC ("GeNIe") that implements the front-end communications network, described below.

The rasterizer board houses a custom-designed 128x64 SIMD array of pixel processors. This array is realized through 32 enhanced-memory chips (EMC's), each of which contains 256 pixel processors. Each pixel processor contains an 8-bit ALU, 256 bytes of main memory and 128 bytes of buffer memory. The SIMD array is controlled by an "IGC" unit (image-generation controller) that accepts commands from the DMA engine and converts these into microcoded instructions for the pixel processors. Linked very tightly with the SIMD array is a texturing unit consisting of several control ASICs and an array of texture memory SDRAM chips. The EMC's are also connected to the image composition network discussed below.

The rasterizer unit has the capability of rendering a region of 128x64 pixels with one sample per pixel, or it can compute up to 8 samples per pixel in parallel, reducing the effective region size to 32x32 pixels. There is a limited amount of inter-pixel-processor communication designed to allow bringing together multiple sample values into a single processor for the purpose of averaging the values together.

5.2.1.3 Front-end Communications Network

Connecting the GP units together is a front-end network referred to as the *geometry network*. This network consists of point-to-point communication links that are 32 bits wide operating bi-directionally at 200 MHz. This provides 800 MB/sec. of full-duplex bandwidth between each pair of GP nodes. The GeNIe chip which implements the network provides three ports on each board: one to the GP memory (limited to 240 MB/sec.), one to the rasterizer texture unit (800 MB/sec.), and one to an optional adapter board such as the host interface (240 MB/sec.). The network uses packet-based communication and allows multicast messaging.

5.2.1.4 Composition Network

Connecting the rasterizer units together is the composition network. This network consists of point-to-point communications links that are 256 bits wide operating bi-directionally at 200 MHz. This provides 6.4 GB/sec. of full-duplex bandwidth between each rasterizer unit. This network incorporates special handshaking logic to control the synchronous flow of regions of pixels through the network. As pixels flow into a given rasterizer unit, they can passed straight along to the next unit, they can be modified using composition logic, or this rasterizer can send out its own pixels instead. Each board can choose to save or discard the incoming pixels.

5.2.1.5 Adapter Boards

An optional adapter board may be attached to each node. One required adapter is the host interface. This connects to a port on the GeNIe chip as mentioned above. Another possible adapter is a framebuffer board. The framebuffer memory store is actually located within the texture memory of the rasterizer unit. Scan-out is controlled by the texture ASICs. Thus a framebuffer adapter is mainly just a RAMDAC unit. The interface to the texture memory is bi-directional, and thus a video input adapter could be implemented as well.

5.2.1.6 Conventional Operation

The GP boards are divided into three functional groups: rasterizers, shaders, and framebuffers. PixelFlow uses a region-based rendering architecture. Each of the rasterizer boards receives a portion of the primitives in the scene. To render a frame, first all the RISC processors on the rasterizer boards transform the primitives and sort them according to screen region. A system-wide "rendering recipe" governs the order in which the screen regions will be processed. Next, the SIMD processors on the rasterizer boards rasterize the first region's primitives and interpolate their intrinsic properties. Once all the primitives in this region are done, these values are stored in an output buffer and various semaphores are set to enable a transfer to begin. Meanwhile, the SIMD processors start working on the next region.

When all of the rasterizers have finished with the first region, the semaphores enable a composition cycle to begin. The pixel values for the first region are transferred to a free shader board. During the composition cycle, for a given pixel position, only the values representing the surface nearest to the viewer are passed on to the next board. Thus the pixels from all the boards are composited together as they are sent to the shader. Once the transfer has finished, the shader board can begin computing the final pixel color values based upon the intrinsic pixel values as well as relevant image texture data contained in the texture memory.

Having more than one shader board allows more time to be spent shading each region compared to rasterizing. After a shader board finishes computing the final color for each pixel, and when the rendering recipe allows, the pixel color values are transferred to a framebuffer board. Once an entire frame's worth of regions have arrived, the framebuffer can display the new completed image.

5.2.2 General Sort-First-On-PixelFlow strategy

Implementation of sort-first on PixelFlow is generally straightforward. The biggest issue is transfer of pixels to the framebuffer over the composition network. Since composition network use must be coordinated across all the rendering nodes, we must find a way to efficiently schedule the use of the composition network and make this information available to all of the nodes at an appropriate time. Thus we must look closely at the sort-first algorithm on a region-based architecture and see how this can be implemented (section 5.2.3). In section 5.2.4 we examine the issue of database communications and its implementation on PixelFlow.

5.2.3 Rasterizer and Pixel Communication Scheduling

In an ideal sort-first system, each rendering node would just send each region of pixels to the framebuffer and let the communications hardware handle the details. With PixelFlow, the use of the composition network must be scheduled, and this schedule must be known to all the nodes in advance of any transfers. We briefly consider dynamic scheduling first, and then we look at static scheduling.

The idea behind dynamic scheduling is that as a node finishes rendering a region, it would broadcast a message in order to schedule that region to be transferred next. There are several problems with this approach. First of all, having a broadcast message for each rasterizer region is not very practical (for a 4096x4096 display with 8-sample antialiasing, 16384 broadcast messages would be required per frame). Also, rasterization of one frame typically happens in parallel pipelined fashion with sorting and transformation work on the next frame. Dynamic scheduling would involve interrupting that work frequently to handle the transfers. Finally, if other nodes are busy rasterizing, then the transfer commands will not be issued until those regions are finished as well. This introduces additional latency to the transfers.

Because of these problems, we turn to static scheduling, wherein we decide at one point during the frame the order for all the transfers to take place. The questions are at what point this should be and how the schedule is made. One possibility is to create the new schedule after all the primitives have been sorted and transformed. At this point in time, each node knows exactly how many primitives are to be rendered in each region, and one may accurately estimate how long it takes to render each region. This information could be gathered together on one node, which would then compute and distribute the transfer schedule. The main problem with this approach is that the system must wait for all the nodes to finish transformation before the first transfer can take place. Effectively, this holds up rasterization until all the nodes have finished transformation. This introduces a global synchronization point into the system that can result in delays as well as congestion.

Instead, we look for a point earlier in the graphics pipeline when the scheduling can be done. The best candidate appears to be when sorting is finished. At this point, all of the primitive groups will have been processed and tallied for the load-balancing algorithm. If there is a correspondence between the cell sizes used for the load-balancing algorithm and the rasterizer region size, then we will have information we can use to estimate the rendering time for each region. Since these tallies are based upon primitive groups, they will not be as accurate as the information available after transformation, but they should be adequate for these estimations. In addition, this information must also be used to estimate the transformation time required on each node, so one will estimate not only how long each region takes to rasterize, but also when each node will begin rasterization.

Thus after sorting, each node will send its tally results to one master node. This node can then estimate when each region will be finished and produce the composition network schedule. This information is then broadcast to all the nodes. As soon as each node receives this message, it would have to send semaphore commands to its rasterizer to enable the proper number of other-board transfers to take place. This allows the node that finishes rasterizing the first scheduled region to go ahead with its transfer without needing to wait for the other nodes to all finish transformation.

We note an additional complication in the scheduling process. When in multi-sample antialiasing mode, efficient use of the hardware suggests that one should process as many regions as there are samples before communicating the results. For instance, in eight-sample mode, all 8192 pixel processors work to compute eight samples of a 32x32 region. This is repeated seven more times, storing the results at a different address each time. After eight regions have been computed, rasterizer processor-to-processor communication is used to effect an 8x8 transpose of this sample data. Thus before the transpose each processor has one sample value for eight regions (with different samples in different processors), and afterwards each processor has eight sample values for one region (with different regions in different processors). After the transpose operation, each processor performs a weighted averaging of the samples to produce a single color value, finally resulting in a full set of 8192 pixel values (containing eight regions of 32x32 pixels) ready to be sent to the framebuffer. The scheduling algorithm therefore must take this processing into account when scheduling the transfers on the composition network.

Finally, we note that determining the exact order in which the different rasterizers will be ready to communicate is unnecessary since buffering is available to help prevent rasterizers from stalling while they wait to use the composition network. Since only color values need to be buffered, potentially more buffering space is available than with the sort-last algorithm (which must buffer at least the color and Z information). On the other hand, since final shading is computed on each board, we need to leave enough free pixel memory to perform these computations. The exact amount of room needed varies depending upon the shading computations performed (from zero extra bytes for Gouraud shading, 24 extra bytes for Phong shading, up to all available bytes for a complex procedural shading function).

5.2.4 Geometry Communications

Communications of primitive data will be a limiting factor on PixelFlow. Although the front-end network provides 800 MB/sec. of bandwidth between rendering nodes, design restrictions limit the effective GP-to-GP bandwidth to 240 MB/sec. In order to achieve the rendering goals for the example application mentioned earlier in the chapter, an effective bandwidth of closer to 420 MB/sec. is desired. This figure is in consideration of 30 frames/sec. rendering, a primitive redistribution coefficient of 50%, and a given node needing twice as many primitives vs. the average number being redistributed. The example also assumes a per-node rendering rate that is higher than PixelFlow's capability (approximately one million triangles per second per node [WEST97]). Adding more PixelFlow rendering nodes would address the polygon performance issue, but it would not likely help the communications issue. Adding more nodes to the system has the potential to decrease the per-node bandwidth, but this is offset by the fact that adding more nodes tends to increase the communications requirements. In order to handle the example application, PixelFlow would need either a smaller redistribution coefficient (e.g. under 25%) or a primitive-redistribution imbalance ratio that is closer to one. One could also reduce the amount of data to be sent by reducing the per-primitive data (e.g. removing vertex colors, or using fewer bits of precision for the various values).

5.3 Conclusion

In this chapter we have discussed the important factors in the design and implementation of a sort-first graphics system. A communications network that can keep up with the demands of sort-first is a key factor in such a design. The design of the rasterizer has obvious impact upon the cost and the latency of the overall system, and it is factor that demands careful consideration as well. Beyond these factors, building a sort-first system is not much different than building any other highly-parallel graphics system.

We also discussed the implementation of sort-first on PixelFlow. There were two major challenges: the scheduling of pixel transfers over the composition network, and the limited amount of bandwidth for geometry communication. We believe that the static scheduling as described provides the best solution for the first challenge. On the other hand, the geometry bandwidth issue appears to be a performance-limiting stumbling block. As it stands, for scenarios such as those presented in this chapter, PixelFlow will only be able to keep up under ideal conditions (i.e., when the primitive-redistribution coefficient is small).

6. Sort-First vs. Sort-Middle

6.1 Introduction

In section 2.4, we compared sort-first, sort-middle, and sort-last in order to see which would be the most suitable for applications demanding high-resolution rendering of highly complex scenery. We were able to easily eliminate sort-last due to the excessive bandwidths resulting from the pixel traffic when running at high resolutions. However, in that section, we only suggested that sort-first would be a better candidate than sort-middle, and we were not able to go further with this statement until we explored the issues involved with sort-first rendering. Now that we have covered these issues, we return to the question of which is the best architecture for the aforementioned applications.

6.2 Similarities and Differences

Sort-first and sort-middle have many similarities. Both use screen partitioning in order to distribute the rasterization load. Both redistribute primitives in order to make sure that the node rasterizing a given screen region has all the primitives that fall in that region. In addition, if one uses the database-handling methods suggested in chapter 4, then both use a scattered distribution of the primitive database in order to distribute some of the transformation work.

Now we consider the differences between the two architectures. In sort-first, untransformed database primitives are redistributed between nodes, whereas in sort-middle transformed display primitives are sent. This can matter a little or a lot depending upon the kind of primitives the system handles. A more significant difference between the architectures is the caching of primitives done on sort-first nodes. This affects many aspects of the architecture, including the load-balancing and all aspects of primitive database management.

In order to evaluate the main impact of these differences, we focus on the amount of primitive communication that takes place in either architecture, followed by the amount of extra processing that must be done to support an efficient sort-first or sort-middle system.

6.3 Primitive Communication Comparison

There are a variety of factors that can affect the amount of communication that takes place in either sort-first or sort-middle. The overlap factor, which is a function of the number of processors and of primitive size, is one factor. Another set of issues revolves around the database organization and method of distribution. Finally, with regard to sort-first specifically, there is the amount of coherence, which is affected by the motion in the scene and also the number of processors. In the following sections, we look at each of these factors and finally in section 6.3.6 we examine what they mean for typical cases of sort-first communication vs. sort-middle.

6.3.1 Database Primitives vs. Display Primitives

Sort-first and sort-middle redistribute different kinds of primitives. In sort-first, we redistribute database primitives, whereas in sort-middle we redistribute display primitives. Database primitives are untransformed primitives as they are found in the hierarchy, whereas display primitives have already been transformed to screen space and are ready to be rasterized. Depending upon the kinds of primitives a system supports, there may be little or there may be a lot of difference between database and display primitives.

For polygons, the difference between the two forms is not necessarily very great. A database primitive will contain the original primitive definition along with the information necessary to indicate its position within the database, whereas a display primitive will contain all the information necessary to rasterize it. Typically, the attributes have already been bound to a display primitive, and thus it may be slightly larger. The extent of the difference is really determined by the format of the input required by the rasterizer. Some systems may need only a command token followed by the raw data [examples?]. In systems such as Pixel-Planes 5 and PixelFlow, a display primitive is in the form of an SIMD command stream containing the instructions and data necessary to rasterize it. In this case, the display primitive becomes larger. As the tables below show, the display primitive data can be more than twice as large.

Database triangle		Bytes
command field	4	4
rendering flags	4	4
triangle vertices	3 * (4 + 4 + 4)	36
triangle normals	3 * (4 + 4 + 4)	36
vertex colors	3 * (4)	12
texture coordinates	3 * (4 + 4)	24
Total: 29 words = 116 bytes		

Table 6.1 Database Triangle Contents

Pixel-Planes 5 triangle rasterization instructions	Word Length
// first, second, and third triangle edge equations	
IGC_FEDGE_L3(cp, A, B, C);	4
IGC_TREEgeZERO_L3(cp, A, B, C);	4
IGC_TREEgeZERO_L3(cp, A, B, C);	4
// Z compare and load	
IGC_MEMleTREE_L3(cp, ZLsb, ZLen, GA, GB, GC);	4
IGC_LOAD_L0(cp, ZLsb, ZLen);	1
// shading information	
IGC_CLEAR(cp, NoShadeBit, 1);	1
// difuse color, alpha, and specular	
igcLOADCLAMP_L3(cp, DifRedLsb, DifLen, GA, GB, GC);	5
igcLOADCLAMP_L3(cp, DifGreenLsb, DifLen, GA, GB, GC);	5
igcLOADCLAMP_L3(cp, DifBlueLsb, DifLen, GA, GB, GC);	5
igcLOADCLAMP_L3(cp, AlphaLsb, AlphaLen, GA, GB, GC);	5
IGC_DLOAD_S1(cp, SpecLsb, SpecLen, <spec color>);	2
// normals	
IGC_LOAD_L3(cp, XRefLsb, RefLen, GA, GB, GC);	4
IGC_LOAD_L3(cp, YRefLsb, RefLen, GA, GB, GC);	4
IGC_LOAD_L3(cp, ZRefLsb, RefLen, GA, GB, GC);	4
// texture ID, exponent, U, V, and scale factor	
IGC_DLOAD_S1(cp, TindexLsb, TindexLen, <texture ID>);	2
IGC_LOAD_C1 (cp, TShiftLsb, TShiftLen, <exponent bits>);	2
IGC_LOAD_L3 (cp, UTexLsb, TextureLen, GA, GB, GC);	4
IGC_LOAD_L3 (cp, VTexLsb, TextureLen, GA, GB, GC);	4
IGC_DLOAD_S1(cp, ScaleLsb, ScaleLen, <scale factor>);	2
Total: 66 words = 264 bytes	

Table 6.2 Pixel-Planes 5 Triangle Contents

For primitives such as triangle strips or height fields, the difference may be more significant, again depending upon what the rasterizer is able to accept as input. If the rasterizer can accept such primitives as input, then the situation is not much different than for polygons. On the other hand, if the rasterizer requires such primitives to be separated into individual polygons, then the resulting display primitives will require much more communications bandwidth in order to be redistributed. This will also be the case for surface patches.

6.3.2 Primitive Overlap Factor

The overlap factor has already been discussed in chapter 3 of this dissertation. The overlap factor becomes a large concern when primitives are sizeable compared to screen regions. When screen regions are very large compared to the primitive size, then the overlap factor becomes fairly negligible. For sort-first, we have each processor working on only one screen region, thus tending to maximize region size. Since complex databases tend towards smaller primitives, the primitive overlap factor is not much of an issue for sort-first. With sort-middle, the impact of the overlap factor depends upon the method chosen for load-balancing the rasterization work. Many common sort-middle load-balancing methods suggest using a granularity ratio higher than one [ELLS96]. This decreases the average screen region size and can increase the primitive overlap factor substantially. Chart 6.1 below shows the primitive overlap factor (expressed in percentage) for the OPC datasets from the experiments in section 3.6; note that the number of processors is the same as the number of regions.

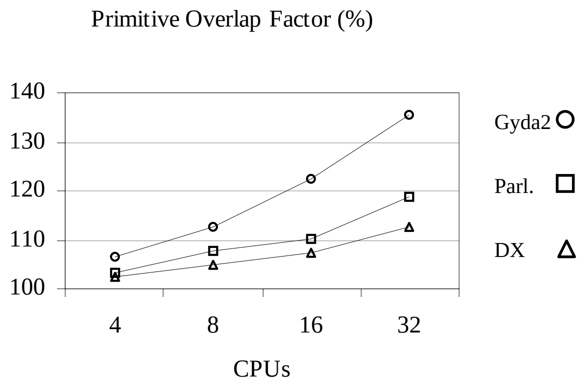


Chart 6.1 Primitive Overlap Factor

The effect of the overlap factor upon communication is different for sort-first than it is for sort-middle. With sort-middle, the overlap factor figures directly in to the amount of primitive communication that is necessary per frame. Due to primitive caching, the situation for sort-first is different. A change in the overlap factor caused by increasing primitive size will not affect sort-first as drastically as it will affect sort-middle. However, a change caused by increasing the number of processors will affect sort-first in the same way as sort-middle.

6.3.3 Group Overlap Factor

In sort-first, we reduce the amount of computation required for sorting by grouping primitives together and redistributing the groups. The disadvantage of this is that primitive groups have a larger overlap factor than the individual primitives, and this results in more communication as well as transformation overhead. The size of the group overlap factor is highly dependent upon the grouping algorithm as well as on the ratio of the average group size in screen space to the average region size. In chart 6.2 (below) we see the group overlap factor (in percentage) for the OPC datasets.

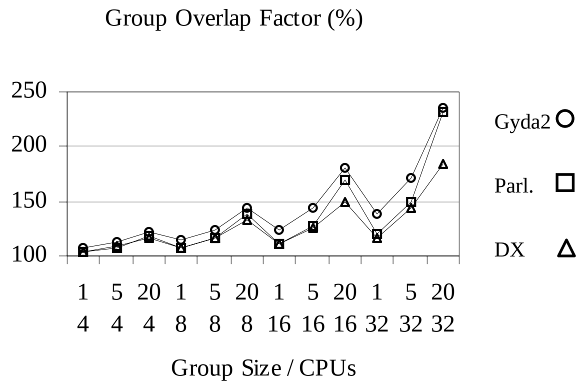


Chart 6.2 Group Overlap Factor

The group overlap factor increases communication by requiring primitives within the group to be sent even when they might not overlap the destination processor's region. Some of these primitives may eventually overlap the destination processor's region later on, in which case their being sent too early is not necessarily a wasted overhead. However, some will never appear in the destination processor's region before the group is discarded, and this is where the communication overhead figures in.

6.3.4 Backface Culling

In sort-middle, each primitive is transformed to screen space before being sorted. This makes it possible to cull backfacing primitives prior to sorting and thus avoid redistributing them. This optimization is not possible in sort-first as it has been described, since sorting is based upon primitive group overlap. Backfacing primitives generally account for one half of the onscreen primitives. As a result, for applications that allow culling of backfacing primitives, this is a clear benefit for sort-middle.

6.3.5 Primitive Caching

With sort-middle, every onscreen, forward-facing polygon must always be redistributed every frame. With sort-first, primitives are effectively cached once they have been sent to a given processor, and they remain as long as they appear in that processor's screen region. A primitive only needs to be redistributed when its group crosses into a new processor's region. The communications savings that result from this caching behavior can be large, but it varies greatly depending upon several factors. The main factors include the amount of primitive motion, the number of screen regions, and the load-balancing algorithm. The latter is a factor since it affects how the screen is partitioned. In addition, primitive group size also affects the amount of primitive communication.

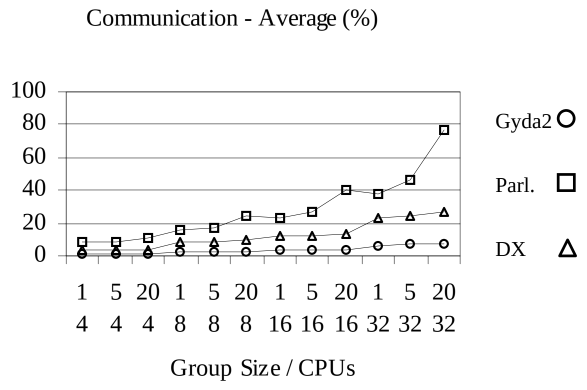


Chart 6.3 Communication

Chart 6.3 above shows for the OPC datasets the average number of primitives that must be communicated as a percentage of the on-screen primitives. For two of the datasets, this value always remains under 30%, while it climbs close to 80% for the third, with this peak value highly dependent upon the group size.

6.3.6 Putting it all together

We now look at the whole picture for primitive communications in sort-first and sort-middle by examining actual cases. From the simulations done in section 3.6, we can count the number of on-screen slave primitives. This value will correspond exactly to the number of primitives that must be communicated if sort-middle were used in place of sort-first. In the chart below we show how this value compares to the amount of sort-first communication by plotting the ratio.

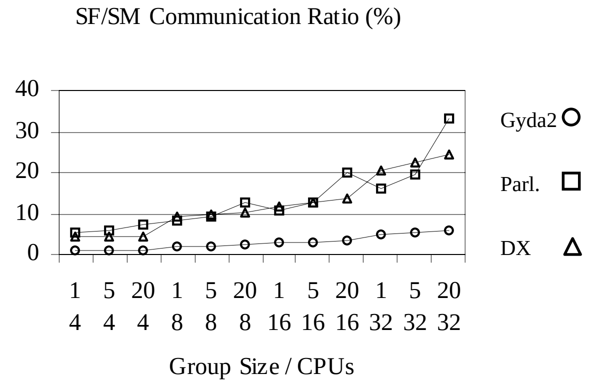


Chart 6.4 SF/SM Communication

There are two things to note. One is that the data is the ratio of the number of primitives, not of the data size. Thus the differences in size between database and display primitives are not accounted for. Also, these experiments did not cull backfacing primitives, and this factor is not accounted for either. These two factors work in opposite directions to similar extents, however, and thus we speculate that the ratios shown above should be close to the values as if both factors were considered.

Overall we can see a clear advantage for sort-first over sort-middle in terms of the amount of primitive communications required. In the worst case, sort-first required only about one third the amount of primitive communications compared to sort-middle. In most cases, this figure was under 25%. Having claimed the communications advantage, we turn our attention to differences in processing overhead.

6.4 Bookkeeping/Transformation Comparison

We now examine the differences between sort-first and sort-middle in terms of the amount of work that happens prior to rasterization. This includes the database traversal, culling, and sorting; we exclude the communication costs since these were discussed in the previous section. We will use analytic techniques to perform this comparison.

In order to compare sort-first and sort-middle on a level playing field, we will assume that both systems use grouping of primitive clusters in order to perform high-level culling. In sort-first, the processing nodes will perform bounding-box tests on each group of primitives not only to perform off-screen culling but also to route them to the correct processors. In sort-middle, the bounding-box tests are only used for off-screen culling, while the primitives are sorted individually.

We define the following variables:

C = coherence: fraction of onscreen primitives requiring redistribution
Gp = number of primitives in a group
Ng = number of primitive groups
Nn = number of processing nodes
Og = overlap factor for primitive groups
Op = overlap factor for individual primitives
Os = fraction of groups that are onscreen
Sd = time to distribute one item (place it in a communications buffer)
Sg = time to sort a group
Sp = time to sort a primitive
Tg = time to transform/cull-test a group bounding box
Tp = time to transform/cull-test a primitive
Xp = time to finish transforming a primitive

We start with sort-first. The first step is to transform and cull-test all master primitive groups. This step involves a bounding-box transformation, finding the box's extremities, and comparing against the screen extent:

group culling: Tg * Ng

The next step is to sort all of the groups that remain on screen. This involves the classification algorithm discussed in section 3.4.4. Once we compute which processors must receive the group, the group is placed in the appropriate outgoing buffers:

group sorting: (Sg + C * Sd * Og) * Os * Ng

Next, all of the slave primitive groups must be transformed and cull-tested. The number of slave groups is relative to the number of processors, assuming static mastership is used.

slave group culling: Tg * (Nn - 1) / Nn * Og * Os * Ng

At this point all of the primitives from the on-screen groups must be transformed and cull-tested:

primitive culling: Tp * Gp * Og * Os * Ng

Finally, we finish the transformation of all of the primitives that are on-screen. From each group, we expect that only one half of the primitives are forward-facing:

primitive transformation: Xp * 1/2 * Op * Gp * Os * Ng

Aside from these computations, there is some additional processing that must happen with sort-first, but we have omitted them from the analysis since they are fairly minimal. One of these is the memory management for receiving and discarding slave primitive groups; this only requires some flag tests and pointer manipulations. Another factor is the slightly more complicated traversal process that is needed for sort-first. Most of this only happens on a per-structure basis, and the number of structures in a database is typically small compared to the number of primitives. The sort-first traversal may also require some per-group pointer manipulation, but this is only a small constant-time operation.

We now turn to sort-middle. Again, the first step is to transform and cull-test the original primitive groups:

group culling: Tg * Ng

Once we know which groups are on-screen, the primitives must be transformed and cull-tested:

primitive culling: Tp * Gp * Os * Ng

Finally, the forward-facing primitives undergo final transformation, binitization, and sorting:

primitive transformation and sorting: (Xp + Sp + Sd * Op) * 1/2 * Gp * Os * Ng

If we assume that the unit primitive-sorting cost is approximately equal to the unit group-sorting cost, then we can compute a ratio of the amount of sort-first work to sort-middle work as follows:

$$\frac{Tg + Os*[(Sg+C*Sd*Og) + Tg*(Nn-1)/Nn*Og + Tp*Gp*Og + Xp*1/2*Op*Gp]}{Tg + Gp*Os*[Tp + (Xp+Sg+Sd*Op)*1/2]}$$

If we can assign some reasonable relative values to each of these variables, we can get an assessment for the value of this ratio. We will use the following set of values:

sort-first coherence factor:	C = 30%
primitives per group:	Gp = 5 or 20
number of processors:	Nn = 16
group overlap factor:	Og = 1.1, 1.2, ... 2.0
primitive overlap factor:	Op = 1.15
on-screen primitive fraction:	Os = 0.1, 1.0
group distribute time:	Sd = 5
group sort time:	Sg = 10

group transform/cull-test time:	Tg = 100
primitive transform/cull-test time:	Tp = 100
primitive final transformation time:	Xp = 100

We chose the values for C and Op based upon the typical results from the OPC test cases. The value for Nn makes little difference in the formulation, and sixteen was deemed reasonable. With regard to the time values, the exact numbers chosen are irrelevant; only the relative scale of the numbers matter. The time to transform and cull a group was set at 100. Given this, the primitive transform time is about equal in scale, as is the final transformation time. The distribute and sort operations, on the other hand, are much simpler, and are scaled appropriately.

Given these values, we produce chart 6.5 below. This chart shows us the ratio for the amount of group and primitive transformation work for sort-first over sort-middle.

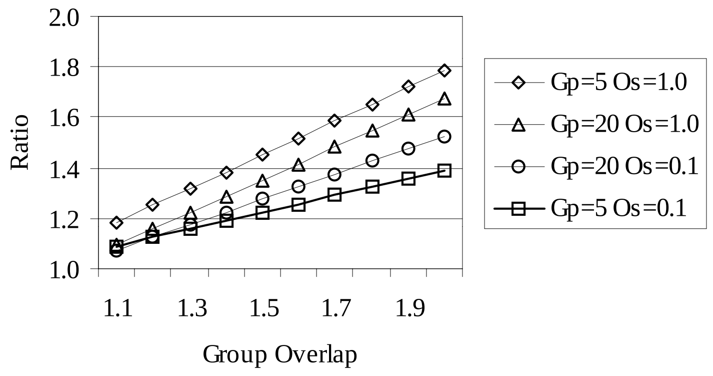


Chart 6.5 SF-SM Ratio

We observe that the group overlap factor is a critical quantity in determining the overhead of sort-first compared to that of sort-middle. This can be seen in the formula above as Og appears in several of the terms in the numerator. We also note an interesting inversion with respect to the group size and the on-screen fraction of the primitives. When all the primitives are on-screen, reducing the group-size helps sort-middle more than it helps sort-first (increasing the ratio). When the fraction of on-screen primitives decreases beyond a certain point, the opposite situation occurs. For these values, the cross-over happens when the on-screen fraction drops to around 25%.

From the graph we can see that the processing overhead for sort-first processing is not exactly negligible. With a small group overlap factor (1.1-1.2), the overhead is perhaps small enough to disregard (7%-25% more work for sort-first); however we have seen that typical values for the group overlap factor can easily approach 2.0, and with this value the processing overhead of 40%-80% is rather significant. For a “typical” group-overlap value of 1.5, the processing overhead is in the range of 22%-45%. We will consider the implications of these values in the next section.

When considering this overhead, we note that the amount of work represented by this formula covers only a certain fraction of the graphics pipeline. Most significantly, the work of rasterization has been omitted. In most cases the rasterization work would be equivalent for either architecture. It would also likely be performed by a separate processor in parallel with the transformation work, and thus it would factor out. In addition, both architectures admit certain unique optimizations that must be factored in for a complete comparison. For instance, sort-first’s architecture allows for geometry culling algorithms based upon rasterizer feedback, whereas such feedback designs are less practical for sort-middle.

From the formulations above, we can examine the effects of one sort-first optimization. If we modify the slave group culling term to account for dynamic mastership (instead of static mastership), the term changes into:

slave group culling: Tg * (Og - 1) * Os * Ng

This is based upon the idea that dynamic mastership reduces the number of slave primitive groups to just those added by the group overlap factor. With this change, we see a 4-9% improvement for a group size of 5, and only a 1-5% improvement for a group size of 20.

6.5 Conclusion

Sort-first offers a clear communications advantage over sort-middle. However, this advantage comes at the cost of additional processing overhead. The communications advantage is a factor of two to ten, whereas the additional processing overhead (for group and primitive transformation) is a factor of seven to eighty percent. In order to understand how these differences affect the overall performance of a sort-first or sort-middle system, one has to understand where the bottleneck is in such systems. In this situation, we only need to worry about two kinds of bottlenecks: inadequate communications network bandwidth, and inadequate transformation processing power.

In order to evaluate communications network bandwidth, we look at memory bus bandwidth, since this would be an upper limit on the bandwidth for any one node. On a typical high-performance microprocessor system, this bus operates at 100 MHz using 64-bits, providing a bandwidth of 800 MB/sec [INTEL]. From the analysis of chapter 5, we can see that we need a bandwidth of approximately 540 MB/sec. to support the sending and receiving of primitive data for the application detailed in section 5.1.1. This number is derived from certain worst-case expectations. For sort-middle, we expect to require about the same amount of bandwidth, except that this is not a worst-case condition, but rather the expected case. The argument that the figure is similar is based upon the fact that we assumed the sort-first primitive redistribution coefficient to be 50%; sort-middle will send 100% of the primitives, but only the half which are front-facing, thus reaching the same 50% figure. As explained in section 6.3, there are other differences which may make sort-middle's bandwidth requirements even higher (such as the primitive data size). In any case, what we see is that for either a sort-first worst case or a sort-middle typical case is that primitive communication is likely to be a system bottleneck. In addition, given the rate of change of communications speed, we do not expect this situation to change in the near future [POUL99].

Now we examine processing. A typical high-performance microprocessor can transform polygons at the rate of about two million triangles per second, while specialized processors can do six million per second or more [DIEF99]. However, already on the horizon is a *game console* (the Sony PSX2) which is expected to be able to transform triangles at the rate of 36 million per second [DIEF99]. Given this rate of advance in CPU technology, we do not expect transformation power to be a bottleneck for a parallel graphics system.

Thus the fact that sort-first requires more processing is made up for by the fact that systems have processing power in abundance compared to communications bandwidth. From these observations, we believe that sort-first offers a clear performance advantage over sort-middle, given that sort-first has lower communications requirements.

7. Conclusions/Future Work

In this dissertation we have examined the field of highly highly-parallel computer graphics systems, with focus on the sort-first architecture. Our goal was to find an architecture that is well-suited for handling both very high-resolution output and very large datasets. Our thesis was that, among the choices of sort-first, sort-middle, and sort-last, we would find sort-first to be the best choice. We have demonstrated this thesis by offering a description of how one would efficiently implement sort-first and also by providing a comparison between sort-first and the competing architectures.

In our comparison we briefly looked at sort-last. This architecture is desirable for its excellent scalability properties for dealing with large datasets. However, it suffers from bandwidth constraints when one desires to produce very high-resolution output. We then examined sort-middle. With this architecture, pixel communication bandwidth is not a large issue, but primitive communication bandwidth could be a potential problem when rendering very large datasets. This left the door open for sort-first, an architecture that offers lower primitive communications requirements than sort-middle through its use of frame-to-frame coherence. However, before we could provide a detailed comparison between sort-first and sort-middle, we first had to address the issues surrounding sort-first implementation, principally efficient load-balancing and bookkeeping for a migrating hierarchical graphics database.

In our search to find an efficient load-balancing algorithm for sort-first, we examined both static and adaptive load-balancing methods. We found that the static methods could easily result in high overheads as the screen must be partitioned into many times more regions than there are processors. This resulted in big increases in the overlap factor as well as the amount of communication. Adaptive methods allow for the ideal situation of having one region per processor. After looking at existing adaptive algorithms, we introduced MAHD, a new adaptive load-balancing algorithm that provides a good work distribution with very low overhead.

Our look into managing a migrating hierarchical graphics database revealed two alternative approaches to the problem, which we termed “min-set” and “max-set.” With the min-set method, one keeps track of only the defined structures in the database, while with the max-set method, one keeps track of these and all the instantiated structures as well. The min-set method requires the use of bounding volumes around database geometry; the max-set method does not require this, although it can benefit from it. We suggested that primitives be placed in groups, and that the group be the quantum unit for primitive management. With this, we had in place a solution for database management.

Our comparison of sort-first and sort-middle showed us two things. First, sort-first clearly offers the advantage in terms of lower communication bandwidth requirements. It’s ability to take advantage of frame-to-frame coherence means that sort-first only has to send about one quarter of the primitives compared to sort-middle. However, sort-first requires more computational overhead to sort and transform the primitive database compared to sort-middle. This extra overhead varies from 7% to 80%, and it is due to the choices made for sort-first’s database management algorithm.

With regard to this apparent split decision, we reiterate our conclusion from chapter 7. If we look at the current state and pace of computer technology, we can see that communication is much more likely to be a bottleneck than computation. With VLSI technology still following Moore’s Law, we’re seeing computing capacity double every eighteen months, whereas off-chip communication speed only improves about 21% in the same time period [POUL99]. What this means is that extra transformation overhead is not an issue compared to extra primitive communication overhead. Given this, we suggest that sort-first is the best choice for applications demanding high-resolution rendering of complex datasets.

7.1 Future Work

There are many opportunities for extending this work and making sort-first more efficient. It is likely that if this research were carried forward, one could greatly reduce the extra computational overhead of sort-first and thus remove this advantage of sort-middle. We will first discuss future work concerning load-balancing issues, followed by work concerning database management issues.

7.1.1 Load-Balancing Improvements

With the MAHD algorithm, the mesh cell-size is currently set manually. For the experiments presented, we simply kept decreasing the cell-size until we reached a desired level of load-balancing performance. It would be better if the system were able to determine an appropriate cell-size automatically. This could be done adaptively, based upon data from the last-frame's load-balancing performance.

Our experiments also only focused upon one method of tallying, even though several methods were presented. We conjectured that the choice of method would not make a significant load-balancing performance difference. However, this conjecture should be investigated. We also did not investigate some of the different choices available for the screen-splitting algorithm. It may also be worthwhile to investigate whether making the splitting algorithm more flexible will improve load-balancing performance without exacting too much of a toll on communication requirements.

Since we found various situations in which the pipelined version of the MAHD algorithm is more appropriate than the stochastic version, we think it would be useful to investigate ways of making the pipelined version more efficient. Most importantly, we would like to compensate for the "lag" that results from using the last-frame's distribution data to determine the work-load partitioning for the current frame. We mentioned some suggestions in section 3.4.7. Other possibilities involve keeping track of primitive distribution data on an object-by-object basis, then attempting to update and combine this data to determine the overall primitive distribution. Some ideas from Microsoft's Talisman architecture, which groups objects into depthwise layers, may prove helpful here [TORB96]. Finally, there is more work to be done to make sure that complex primitives will be computed and load-balanced in an efficient manner. Our thoughts in section 3.7 only begin to examine this issue.

7.1.2 Database Management Improvements

Our implementation of sort-first database management was chosen only as a starting point due to its relative simplicity. However, there are several design choices that make this implementation less than optimal. The two major ones are the use of primitive groups and the choice between static and dynamic mastership.

As we can see from chart 6.5, the group-overlap factor is the largest factor that makes sort-first less efficient than sort-middle. When a group overlaps two or more regions, all of the primitives within the group are transformed and cull-tested in all of the overlapped regions. This source of inefficiency can be eliminated by simply doing away with groups and making the primitive the quantum unit for database management. However, this choice greatly increases the amount of bookkeeping work. Remember that with static mastership, there is a "double" distribution of the primitives: the master distribution and the slave distribution. With primitives in groups, the added overhead from this is reduced from “double the primitives” to “double the groups,” a much smaller factor.

Thus we observe that eliminating primitive groups also requires the choice of dynamic mastership in order to avoid the double-distribution issue. However, as mentioned in section 3.3.2, dynamic mastership brings up its own set of issues. These include dealing with off-screen polygons, tracking primitives for editing purposes, and the fact that the master primitive distribution can become unbalanced. None of these problems appears to be a show-stopper. By developing a sort-first system that implements dynamic mastership on individual primitives, one can eliminate most of the extra transformation overhead that sort-first currently requires over sort-middle.

7.2 Final Thoughts

The author hopes that the insights presented here prove useful for future researchers who wish to implement sort-first rendering in a hardware system. With the rise of the performance of PC graphics to levels heretofore only achieved by the highest-end parallel graphics systems, one might question the need for fully-parallelized graphics systems. This is only a short-sighted view, however, since the desire for higher rendering performance is ever present. It may simply be that future parallel graphics systems will come on one card rather than in an entire card cage. With a new set of design limitations, the question of sort-first, sort-middle, or sort-last may once again be open.

References

[3DLABS]	3Dlabs, Inc., Oxygen GVX210 product specifications, http://www.3dlabs.com/products/oxygengvx210.html.
[AKEL89]	K. Akeley, "The Silicon Graphics 4D/240GTX Superworkstation," IEEE *Computer Graphics & Applications,* July 1989, pp. 71-83.
[AKEL93]	K. Akeley, "RealityEngine Graphics," *Computer Graphics* (Proc. of Siggraph), Aug. 1993, pp. 109-116.
[ALIA98]	D. Aliaga, J. Cohen, A. Wilson, H. Zhang, C. Erikson, K. Hoff, T. Hudson, W. Stuerzlinger, E. Baker, R. Bastos, M. Whitton, F. Brooks, D. Manocha, "A Framework for the Real-time Walkthrough of Massive Models," UNC TR #98-013, March, 1998.
[ARVO89]	J. Arvo and D. Kirk, "A Survey of Ray Tracing Acceleration Techniques," Chapter 6 in *An Introduction to Ray Tracing*, edited by Andrew S. Glassner, Academic Press, New York, 1989.
[AZUM94]	R. Azuma and G. Bishop, "Improving Static and Dynamic Registration in an Optical See-through HMD," *Computer Graphics* (Proc. of Siggraph), July 1994, pp. 197-204.
[BODE95]	N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W.-K. Su, "Myrinet - A Gigabit-per-Second Local Area Network," *IEEE Micro*, February 1995, pp. 29-36.
[CLAR76]	J. H. Clark, "Hierarchical Geometric Models for Visible Surface Algorithms," *Communications of the ACM*, October 1976, Vol. 19, No. 10, pp. 547-554.
[COOK84]	R. L. Cook, "Shade Trees," *Computer Graphics* (Proc. of Siggraph), July 1984, Vol. 18, No. 3, pp. 223-231.
[COOR97]	S. Coorg and S. Teller, "Real-Time Occlusion Culling for Models with Large Occluders," *Proceedings of the 1997 Symposium on Interactive 3D Graphics*, Providence, RI, April 27-30, 1997, pp. 83-90.

[COX95] M. Cox, *Algorithms for Parallel Rendering*, doctoral dissertation, Princeton University, May 1995.

[COX97] M. Cox and N. Bhandari, “Architectural Implications of Hardware-Accelerated Bucket Rendering on the PC,” *1997 Siggraph/Eurographics Workshop on Graphics Hardware*, Los Angeles, CA, August 1997.

[COSM81] M. Cosman and R. Schumacker, “System Strategies to Optimize CIG Image Content,” *Proceedings of the Image II Conference,* Scottsdale, Arizona, 10-12 June, 1981.

[CROC93] T. W. Crockett and T. Orloff, "A Parallel Rendering Algorithm for MIMD Architectures," *Proc. Parallel Rendering Symposium*, ACM Press, New York, 1993, pp. 35-42.

[CROW84] F. C. Crow, "Summed-Area Tables for Texture Mapping," *Computer Graphics* (Proc. of Siggraph), July 1984, Vol. 18, No. 3, pp. 207-212.

[CRUZ93] C. Cruz-Neira, D. J. Sandin, and T. A. DeFanti, "Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE," *Computer Graphics* (Proc. of Siggraph), Aug. 1993, pp. 135-142.

[DEER93] M. Deering and S. R. Nelson, "Leo: A System for Cost Effect 3D Shaded Graphics," *Computer Graphics* (Proc. of Siggraph), Aug. 1993, pp. 101-108.

[DEER95] M. Deering, “Geometry Compression,” *Computer Graphics* (Proc. of Siggraph), August 1995, pp. 13-20.

[DEFO96] DeFoe, Douglas, Kaiser Electro-Optics, Inc., “Full Immersion Head Mounted Display System,” WWW URL http: //www.darpa.mil/ ETO/Displays/HMD/Factsheets/Immersion.html.

[DIEF99] K. Diefendorff, “Sony’s Emotionally Charged Chip,” *Microprocessor Report*, Volume 13, No. 5, April 19, 1999.

[DOEN85] P. K. Doenges, "High Performance Image Generation Systems: Overview of Computer Image Generation in Visual Simulation," Siggraph 1985 Course Notes, July 1985.

[ELLS89] D. Ellsworth, “Pixel-Planes 5 Rendering Control,” University of North Carolina Department of Computer Science Technical Report, TR89-003.

[ELLS90] D. Ellsworth, H. Good, and B. Tebbs, “Distributing Display Lists on a Multcomputer,” *Computer Graphics*, Vol. 24, No. 2, March 1990, (*Proceedings 1990 Symposium on Interactive 3D Graphics*, Snowbird, Utah, March 25-28, 1990), pp. 147-155.

[ELLS94] D. Ellsworth, "A New Algorithm for Interactive Graphics on Multicomputers," *IEEE Computer Graphics & Applications*, July 1994, Vol. 14, No. 4, pp. 33-40.

[ELLS96] D. Ellsworth, *Polygon Rendering for Interactive Visualization on Multicomputers*, doctoral dissertation, TR 99-008, University of North Carolina at Chapel Hill, 1996.

[EYLE97] J. Eyles, S. Molnar, J. Poulton, T. Greer, A. Lastra, and N. England, "PixelFlow: The Realization", *Proceedings of the Siggraph/Eurographics Workshop on Graphics Hardware*, Los Angeles, CA, August 3-4, 1997, pp. 57-68.

[EVAN] *ESIG-3000 Technical Description*, Evans and Sutherland, Salt Lake City, Utah.

[FOLE90] J. D. Foley, A. Van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practice*, 2nd Ed., Addison-Wesley Publishing Co., Inc., Reading, Mass., 1990.

[FUCH89] H. Fuchs et al., "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories," *Computer Graphics* (Proc. of Siggraph), July 1989, Vol. 23, No. 3, pp. 79-88.

[GEIS94] A. Geist, A. Beguelin, J. Dongarra,, R. Mancheck, W. Jiang, and V. Sunderam, *PVM: A Users' Guide and Tutorial for Networked Parallel Computing*, MIT Press, Cambridge, Mass., 1994 (also http://www.netlib.org/pvm3/book/pvm-book.html).

[GOTT96] S. Gottschalk, M. Lin, and D. Manocha, "OBBTree: A Hierarchical Structure for Rapid Interference Detection, " *Computer Graphics* (Proc. of Siggraph), August 1996, pp. 171-180.

[GREE93] N. Greene, M. Kass, and G. Miller, "Hierarchical Z-Buffer Visibility," *Computer Graphics* (Proc. of Siggraph), August 1993, pp. 231-238.

[HELM93] J. L. Helman, "Designing Virutal Reality Systems to Meet Physiological and Psychological Requirements," Siggraph '93 Applied VR Course Notes, Aug. 1993.

[HOFF96] K. Hoff, "A 'Fast' Method for Culling of Oriented-Bouding Boxes (OBBs) Against a Perspective Viewing Frustum in Large 'Walkthrough'

Models," http://www.cs.unc.edu/~hoff/research/vfculler/viewcull.html, 1996.

[IGEH98] H. Igehy, G. Stoll, and P. Hanrahan, "The Design of a Parallel Graphics Inteface," *Computer Graphics* (Proc. of Siggraph), July 1998, pp. 141-150.

[INTEL] Intel Corporation, Pentium II processor specifications, http://www.intel.com.

[KUMA96] S. Kumar, *Interactive Rendering of Parametric Spline Surfaces*, doctoral dissertation, TR96-039, University of North Carolina at Chapel Hill, 1996.

[LACR92] M. Lacroix, "A HDTV Projector for Wide Field of View Flight Simulators," *Proceedings of the 1992 Image VI Conference*, July 1992, pp. 492-500.

[LACR94] M. Lacroix, and J. Melzer, "Helmet-Mounted Displays for Flight Simulators," *Proceedings of the 1994 Image VII Conference*, June 1994, pp. 34-40.

[LATH85] R. Latham, "A VLSI-Based Digital Image Generator," Proceedings of the Interservice/Industry Training Systems and Education Conference, 1985.

[LATH94] R. Latham, "Advanced Image Generator Architectures," Image VII conference tutorial, June 1994.

[LI88] K. Li, "IVY: A Shared Virtual Memory System for Parallel Computing," *Proceedings of the 1988 International Conference on Parallel Processing*, August 1988, pp. 94-101.

[MACI98] B. MacIntyre and S. Feiner, "A Distributed 3D Graphics Library," *Computer Graphics* (Proc. of Siggraph), July 1998, pp. 361-370.

[MOLN91] S. Molnar, *Image-Composition Architectures for Real-Time Image Generation*, doctoral dissertation, TR 91-046, University of North Carolina at Chapel Hill, Oct. 1991.

[MOLN92] S. Molnar, J. Eyles, J. Poulton, "PixelFlow: High-Speed Rendering Using Image Composition," *Computer Graphics* (Proc. of Siggraph), July 1992, pp. 231-240.

[MOLN94] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs, "A Sorting Classification of Parallel Rendering," *IEEE Computer Ggraphics & Applications*, July 1994, Vol. 14, No. 4, pp. 23-32.

[MUEL95] C. Mueller, "The Sort-First Architecture for High-Performance Graphics," Proceedings of the 1995 Symposium on Interactive 3D Graphics (Monterey, California, April 9-12, 1995), special issue of Computer Graphics, ACM SIGGRAPH, New York, 1995, pp. 75-84.

[MUEL97] C. Mueller, "Hierarchical Graphics Databases in Sort-First," Proceedings of the 1997 Parallel Rendering Symposium (Phoenix, Arizona, October 25-26, 1997), special issue of *Computer Graphics*, ACM SIGGRAPH, New York, 1997, pp. 49-57.

[MONT97] J. S. Montrym, D. R. Baum, D. L. Dignam, and C. J. Migal, "InfiniteReality: A Real-Time Graphics System," *Computer Graphics* (Proc. of Siggraph), August 1997, pp. 293-302.

[NCGA92] National Computer Graphics Association Picture-Level Benchmark, *GPC Quarterly Report*, Vol. 2, No. 4, 1992.

[NEC99] NEC, Inc., "The TE4 Professional 3D Accelerator," Hot 3d Systems presentation, *SIGGRAPH/ Eurographics Workshop on Graphics Hardware*, August 1999 (available on the WWW at http://www.merl.com/hwws99/presentations/te4.pdf).

[NEID93] J. Neider, T. Davis, and M. Woo, *OpenGL Programming Guide*, Addison-Wesley, 1993.

[NIGU94] S. G. Nigus, "New System Advances Visual Simulation Core Technologies," *Proceedings of the 1994 Image VII Conference*, June 1994, pp. 86-95.

[NISH96] Satoshi Nishimura, Tosiyasu Kunii, "VC-1: A Scalable Graphics Computer with Virtual Local Frame Buffers," *Computer Graphics* (Proc. of Siggraph), August 1996, pp. 365-372.

[PADM92] P. Padmos and M. Milders, "Checklist for outside-world images of simulators," International Training Equipment Conference and Exhibition (ITEC), Luxembourg, April 1992, pp. 2-14.

[PCI] PCI Bus Specifications, PCI Special Interest Group, Hillsboro, Oregon.

[PIEG95] L. Piegl and W. Tiller, *The NURBS Book*, first edition, Springer-Verlag Berlin Heidelberg, 1995.

[POUL99] J. Poulton, informal presentation concerning electrical signaling technologies, University of North Carolina at Chapel Hill, March 1999.

[RASK98] 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,” *Computer Graphics* (Proc. of Siggraph), July 1998, pp. 179-188.

[ROBL88] D. R. Roble, “A Load Balanced Parallel Scanline Z-Buffer Algorithm for the iPSC Hypercube,” *Proceedings of Pixim ’88*, Paris, France, October 1988.

[ROHL94] J. Rohlf and J. Helman, “IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics,” *Computer Graphic*s (Proc. of SIGGRAPH), July 1994, pp. 381-394.

[SAMA99] R. Samanta, J. Zheng, T. Funkhouser, K. Li, and J. Singh, “Load Balancing for Multi-Projector Rendering Systems,” *SIGGRAPH/ Eurographics Workshop on Graphics Hardware*, August 1999.

[SAME90] H. Samet, *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*, Reading, Mass., Addison-Wesley, 1990.

[SCHU80] R. A. Schumacker, “A New Visual System Architecture,” Proceedings of the Second Interservice/Industry Training Equipment Conference, Salt Lake City, Utah, November 1980.

[SCHW98] M. Schwenden and B. Miles, *Onyx2(TM) Rackmount Owner’s Guide*, Silicon Graphics Inc. document number 007-3457-004, 1998.

[SPEC00] Standard Performance Evaluation Corporation, “SPECopc Group Develops Benchmarks for Systems Based on OpenGL API,” http://www.specbench.org/gpc/opc.static/overview.htm, 2000.

[STRA92] P. Strauss and R. Carey, “An Object-Oriented 3D Graphics Toolkit,” *Computer Graphics* (Proc. of SIGGRAPH), July 1992, pp. 341-350.

[SUTH74] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker, "A Characterization of Ten Hidden Surface Algorithms," *ACM Computing Surveys*, Mar. 1974, Vol. 6, No. 1, pp. 1-55.

[TORB96] J. Torborg and J. Kajiya, “Talisman: Commodity Realtime 3D Graphics for the PC,” *Computer Graphics* (Proc. of SIGGRAPH), August 1996, pp. 353-363.

[WALE83] C. E. Wales, and M. A. Cosman, "DMA and CIG: A Shotgun Wedding," *Proceedings of the 5th Interservice/Industry Training Equipment Conference*, Nov. 1983, pp. 97-104.

[WERN93] K. I. Werner, "Xerox's 6.3-million-pixel LCD," *IEEE Spectrum*, Nov. 1993, Vol. 30, No. 11, pg. 21.

[WEST97] L. Westover, personal communication, August 1997.

[WHEL85] D. S. Whelan, *Animac: A Multiprocessor Architecture for Real-Time Computer Animation*, Ph.D. dissertation, California Institute of Technology, 1985.

[WHIT92] S. Whitman, *Multiprocessor Methods for Computer Graphics Rendering*, AK Peters, Ltd., Wellesley, Massachusetts, 1992.

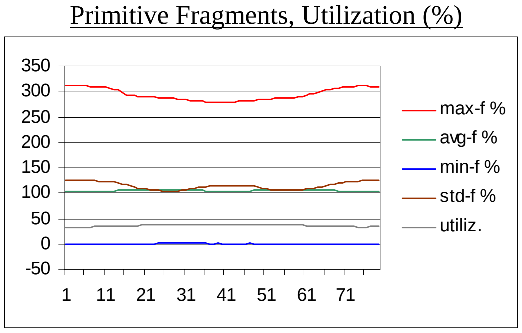
[WHIT94] S. Whitman, "A Task Adaptive Parallel Graphics Renderer," *IEEE Computer Graphics & Applications*, July 1994, Vol. 14, No. 4, pp. 41-48.

[ZHAN97] H. Zhang, D. Manocha, T. Hudson, and K. E. Hoff, “Visibility Culling usinng Hierarchical Occlusion Maps,” *Computer Graphics* (Proc. of Siggraph), August 1997, pp. 77-88.

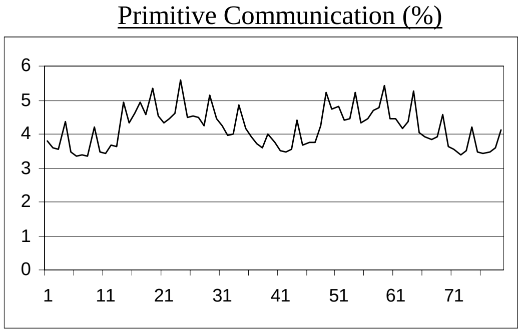
<u>Appendix A</u>

Graphs for Load-Balance Experiments I (section 3.5)

PLB Head, Static, 16 processors, 4x4 regions:

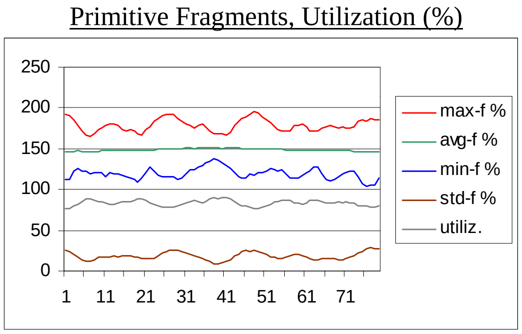


Graph A.1

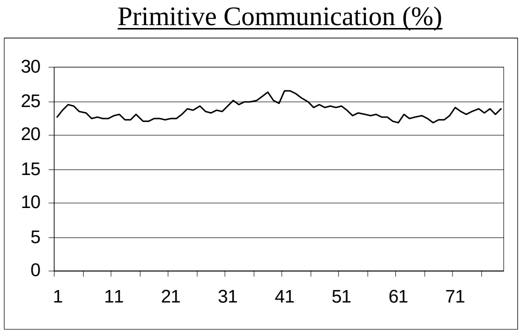


Graph A.2

PLB Head, Static, 16 processors, 20x20 regions:

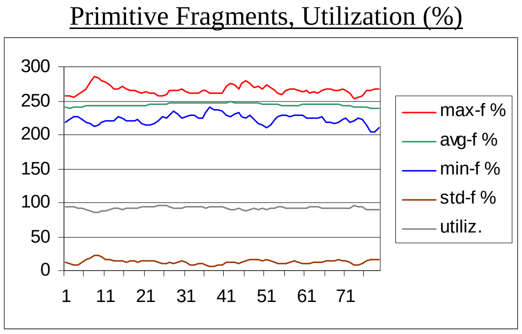


Graph A.3

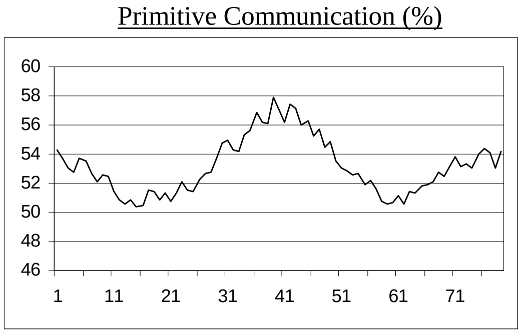


Graph A.4

PLB Head, Static, 16 processors, 40x40 regions:

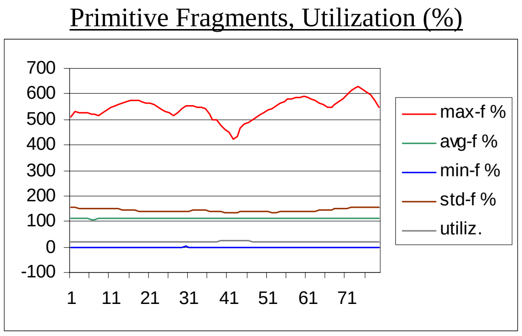


Graph A.5

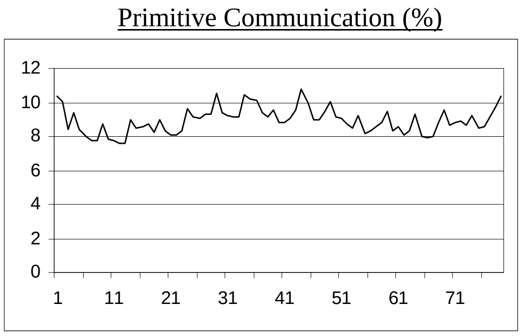


Graph A.6

PLB Head, Static, 64 processors, 8x8 regions:

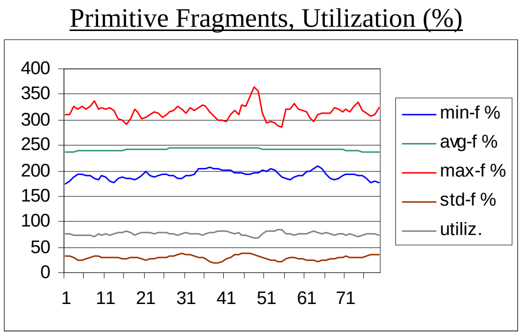


Graph A.7

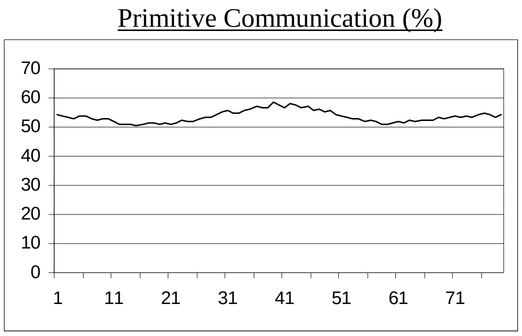


Graph A.8

PLB Head, Static, 64 processors, 40x40 regions:

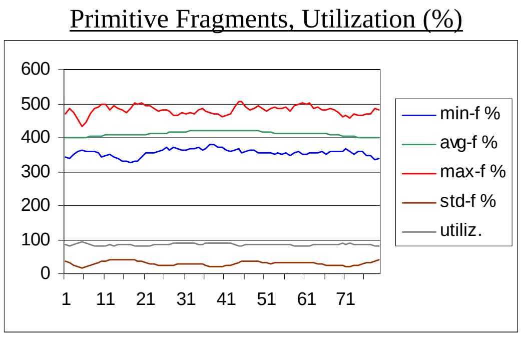


Graph A.9

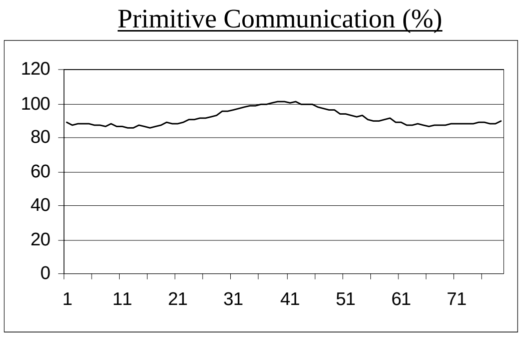


Graph A.10

PLB Head, Static, 64 processors, 64x64 regions:



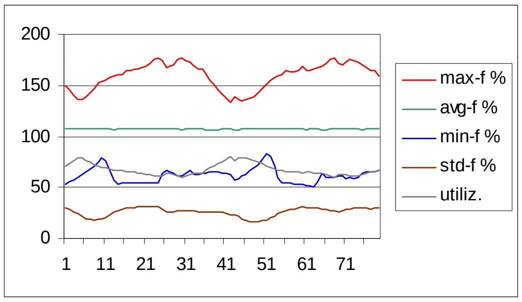
Graph A.11



Graph A.12

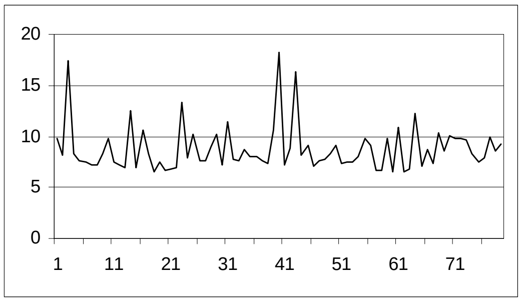
PLB Head, MAHD, 16 processors, 16x16 cells, current-frame:

Primitive Fragments, Utilization (%)



Graph A.13

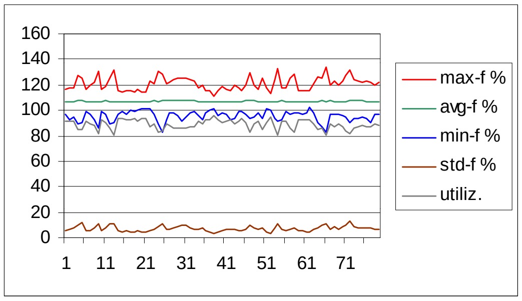
Primitive Communication (%)



Graph A.14

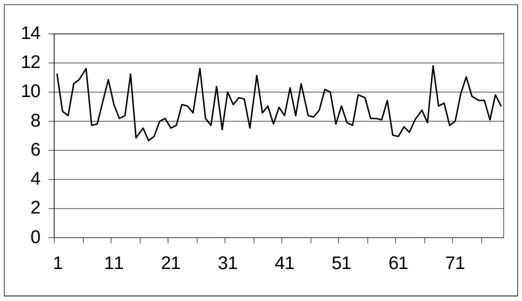
PLB Head, MAHD, 16 processors, 64x64 cells, previous-frame:

Primitive Fragments, Utilization (%)



Graph A.15

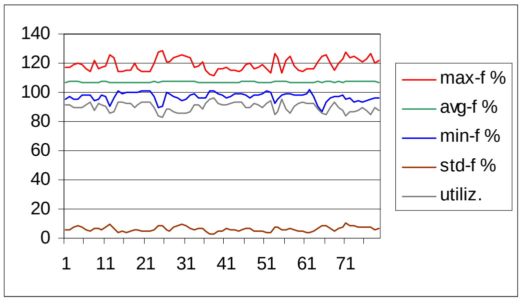
Primitive Communication (%)



Graph A.16

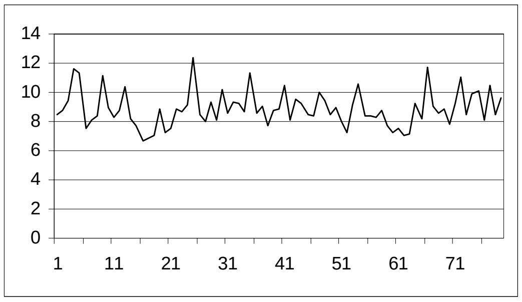
PLB Head, MAHD, 16 processors, 64x64 cells, current-frame:

Primitive Fragments, Utilization (%)



Graph A.17

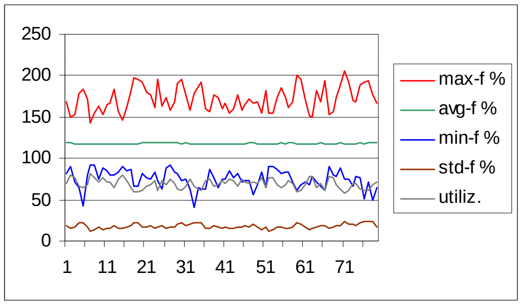
Primitive Communication (%)



Graph A.18

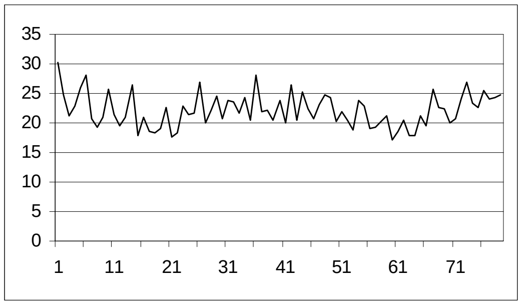
PLB Head, MAHD, 64 processors, 64x64 cells, previous-frame:

Primitive Fragments, Utilization (%)



Graph A.19

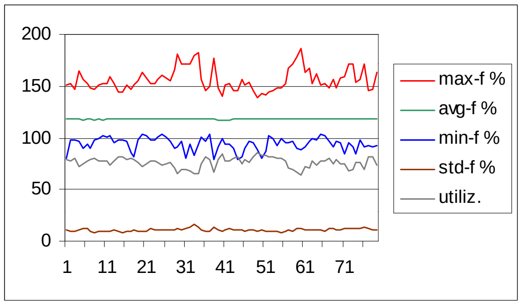
Primitive Communication (%)



Graph A.20

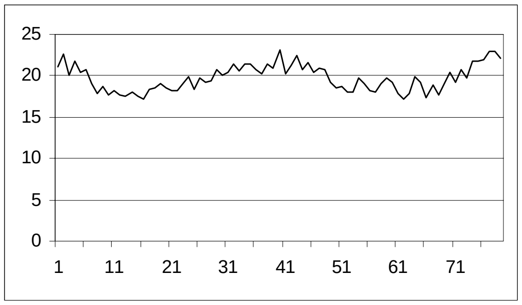
PLB Head, MAHD, 64 processors, 256x256 cells, previous-frame:

Primitive Fragments, Utilization (%)



Graph A.21

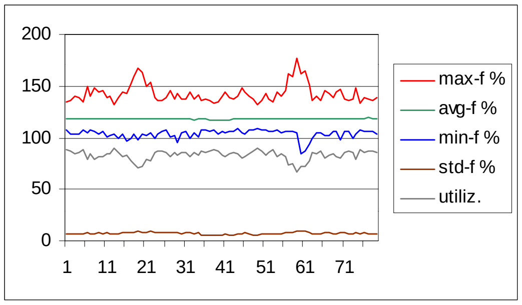
Primitive Communication (%)



Graph A.22

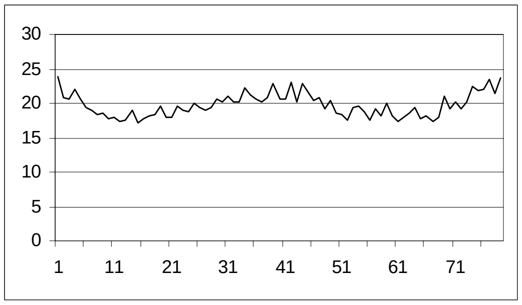
PLB Head, MAHD, 64 processors, 256x256 cells, current-frame:

Primitive Fragments, Utilization (%)



Graph A.23

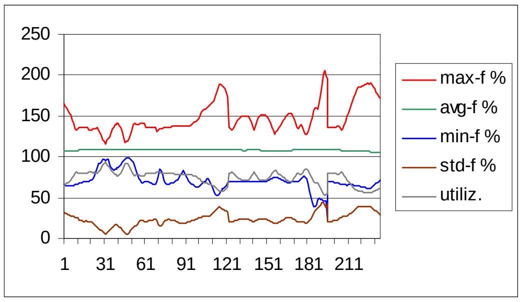
Primitive Communication (%)



Graph A.24

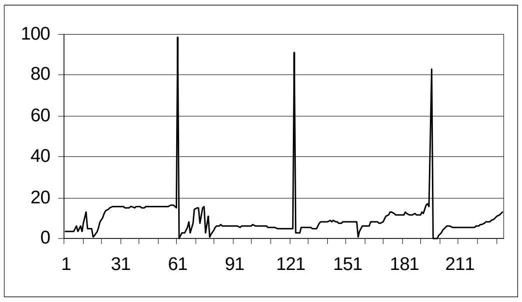
Sierra, static, 16 processors, 12x12 regions:

Primitive Fragments, Utilization (%)



Graph A.25

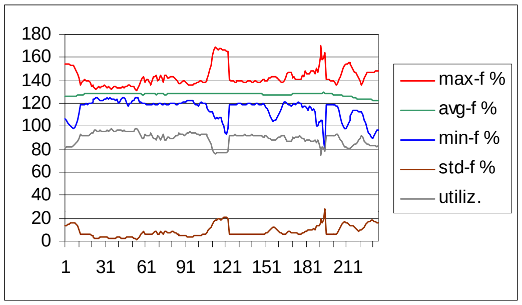
Primitive Communication (%)



Graph A.26

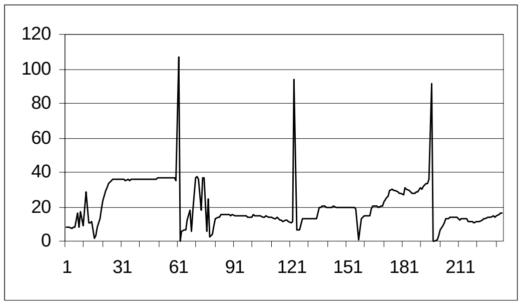
Sierra, static, 16 processors, 28x28 regions:

Primitive Fragments, Utilization (%)



Graph A.27

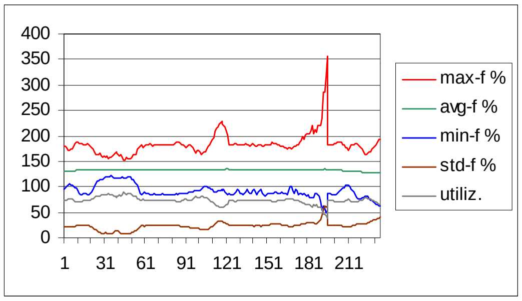
Primitive Communication (%)



Graph A.28

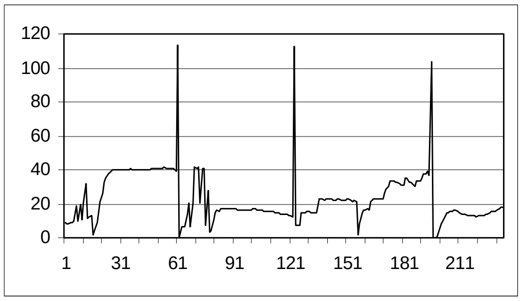
Sierra, static, 64 processors, 32x32 regions:

Primitive Fragments, Utilization (%)



Graph A.29

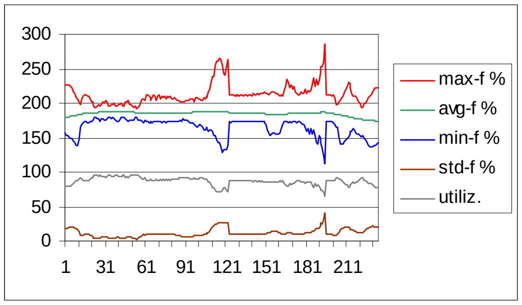
Primitive Communication (%)



Graph A.30

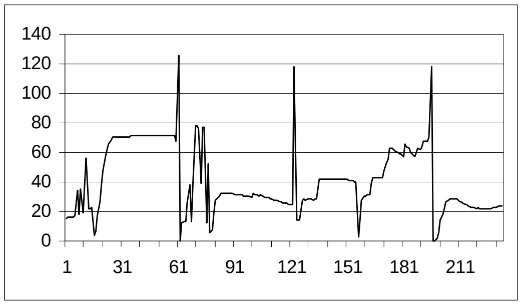
Sierra, static, 64 processors, 56x56 regions:

Primitive Fragments, Utilization (%)



Graph A.31

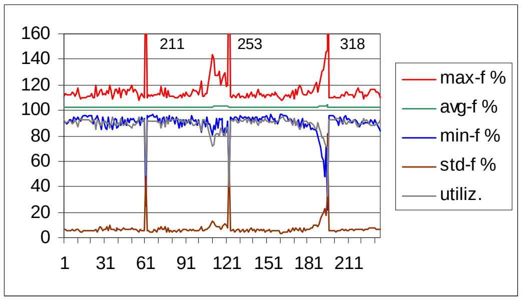
Primitive Communication (%)



Graph A.32

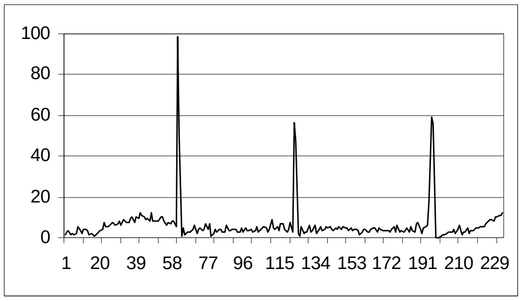
Sierra, MAHD, 16 processors, 64x64 cells, previous-frame:

Primitive Fragments, Utilization (%)



Graph A.33

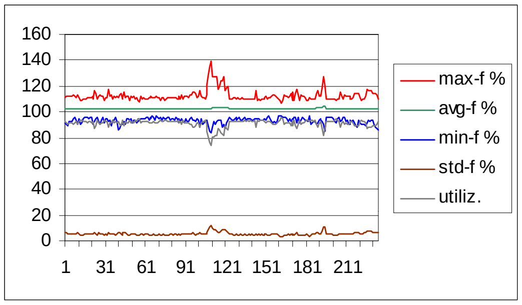
Primitive Communication (%)



Graph A.34

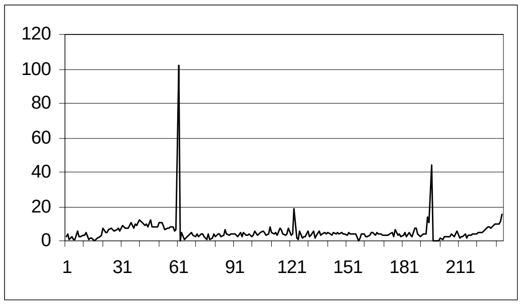
Sierra, MAHD, 16 processors, 64x64 cells, current-frame:

Primitive Fragments, Utilization (%)



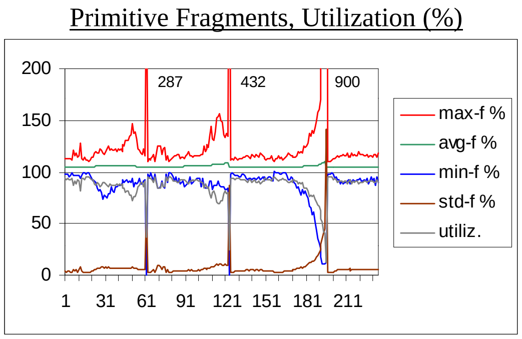
Graph A.35

Primitive Communication (%)

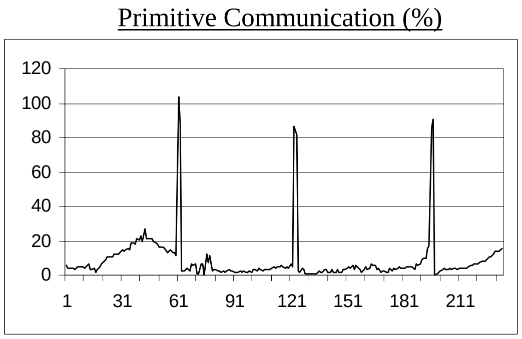


Graph A.36

Sierra, MAHD, 64 processors, 256x256 cells, previous-frame:

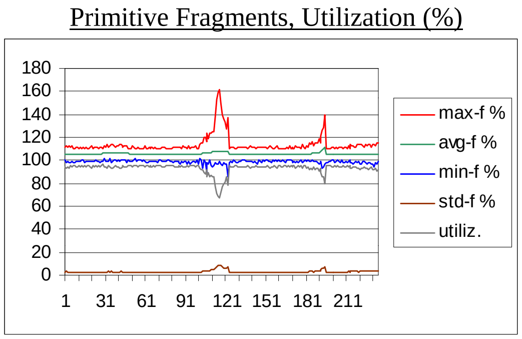


Graph A.37

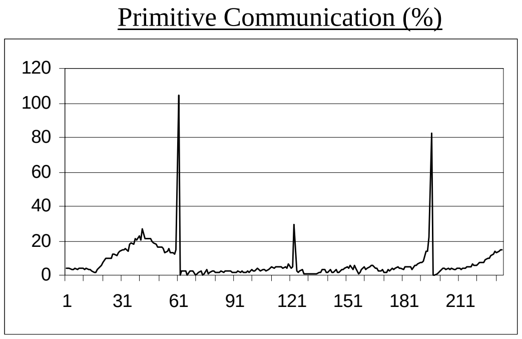


Graph A.38

Sierra, MAHD, 64 processors, 256x256 cells, current-frame:



Graph A.39

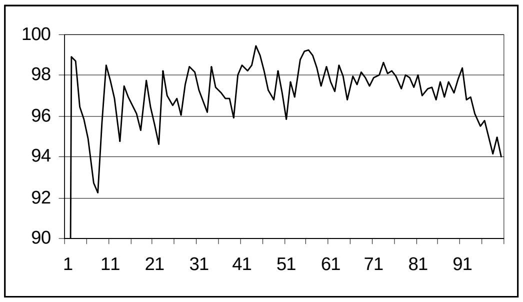


Graph A.40

Graphs for Load-Balance Experiments II (section 3.6)

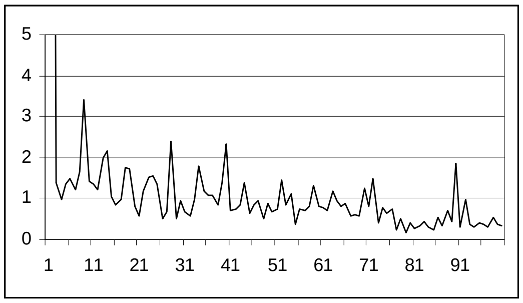
Gyda2, 4 processors, 5 prims/group, previous-frame:

Primitive Utilization (%)



Graph A.41

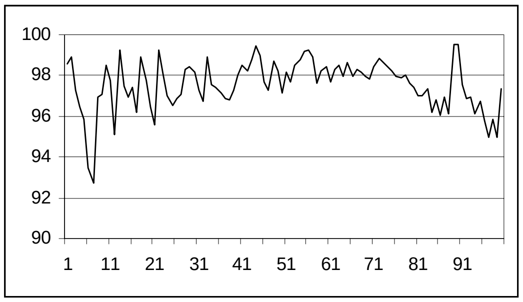
Primitive Communication (%)



Graph A.42

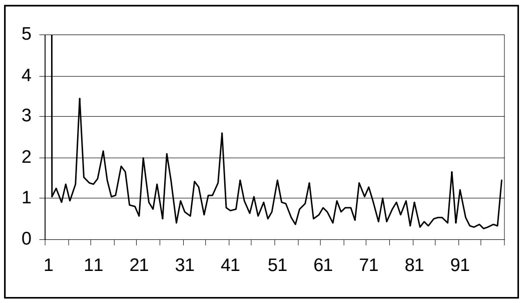
Gyda2, 4 processors, 5 prims/group, stochastic:

Primitive Utilization (%)



Graph A.43

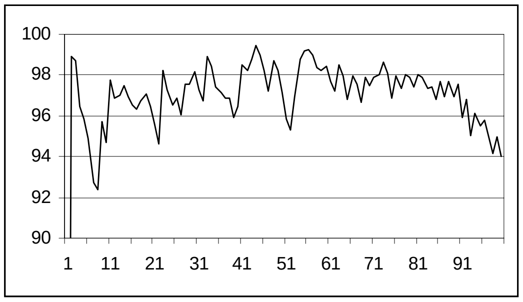
Primitive Communication (%)



Graph A.44

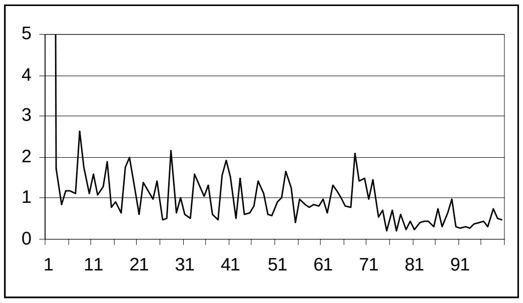
Gyda2, 4 processors, 20 prims/group, previous-frame:

Primitive Utilization (%)



Graph A.45

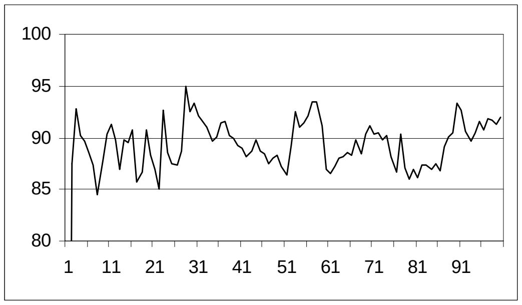
Primitive Communication (%)



Graph A.46

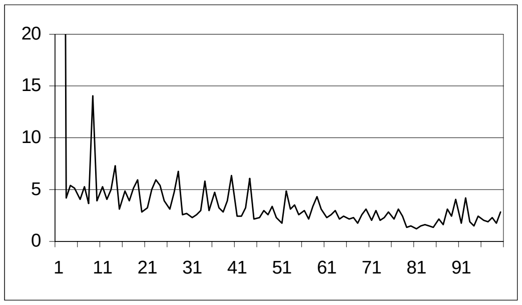
Gyda2, 16 processors, 5 prims/group, previous-frame:

Primitive Utilization (%)



Graph A.47

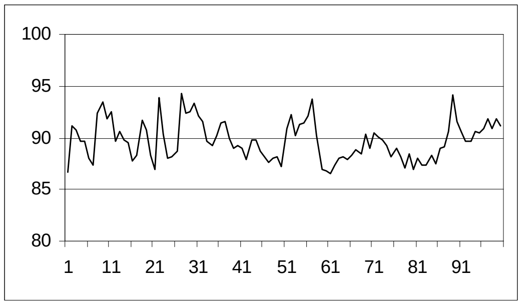
Primitive Communication (%)



Graph A.48

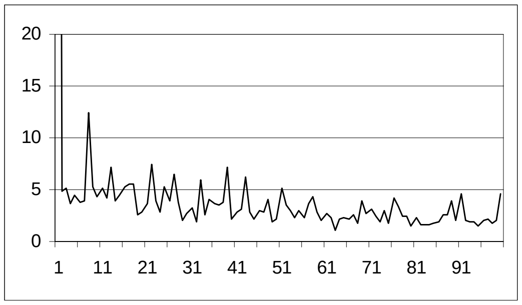
Gyda2, 16 processors, 5 prims/group, stochastic:

Primitive Utilization (%)



Graph A.49

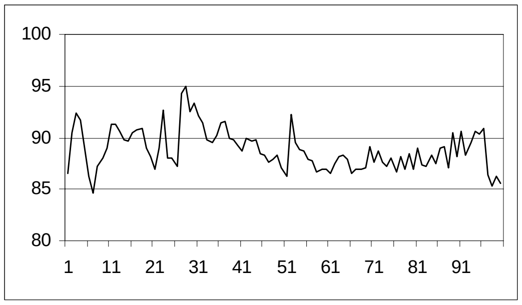
Primitive Communication (%)



Graph A.50

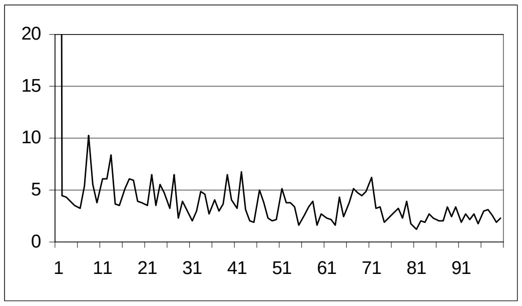
Gyda2, 16 processors, 20 prims/group, stochastic:

Primitive Utilization (%)



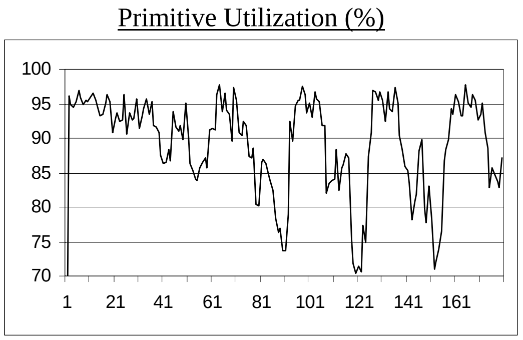
Graph A.51

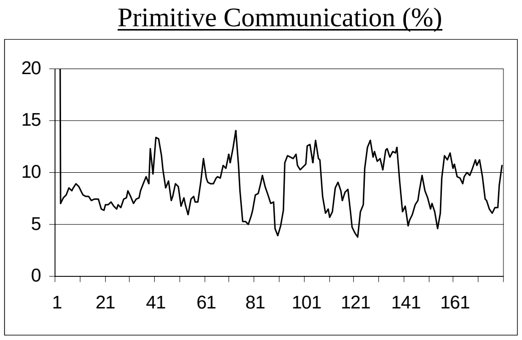
Primitive Communication (%)



Graph A.52

DX, 8 processors, 1 prim/group, previous-frame:

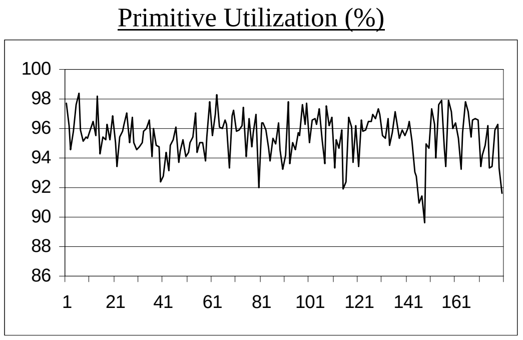


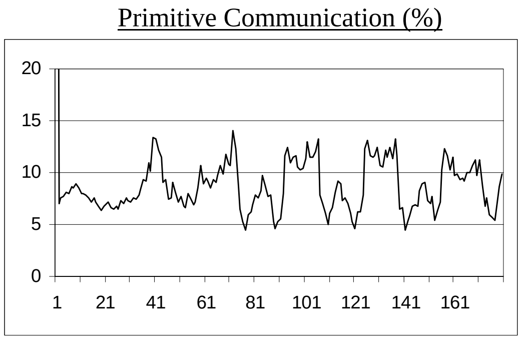


Graph A.53

Graph A.54

DX, 8 processors, 1 prim/group, stochastic:

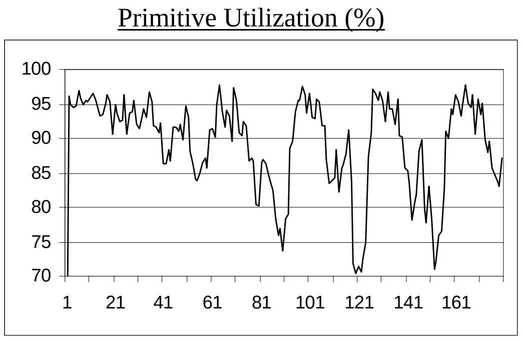


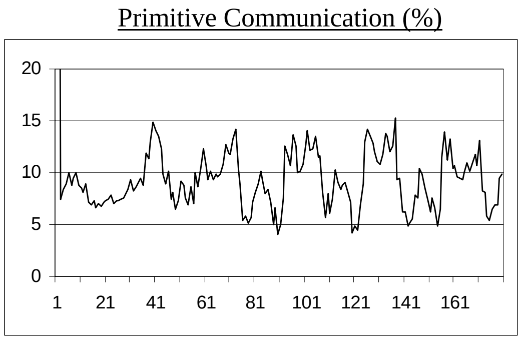


Graph A.55

Graph A.56

DX, 8 processors, 20 prims/group, previous-frame:



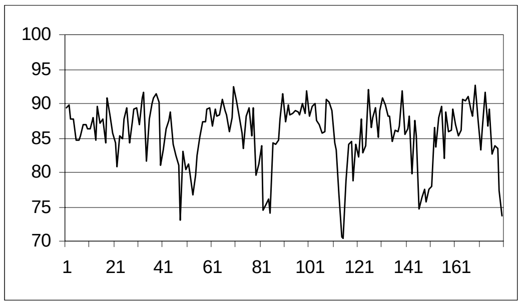


Graph A.57

Graph A.58

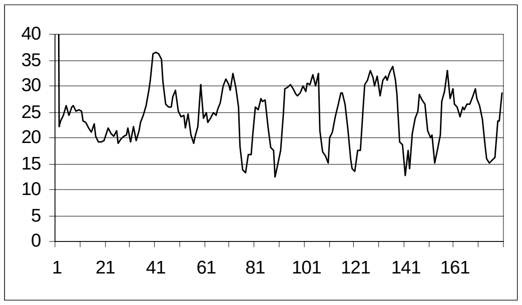
DX, 32 processors, 5 prims/group, stochastic:

Primitive Utilization (%)



Graph A.59

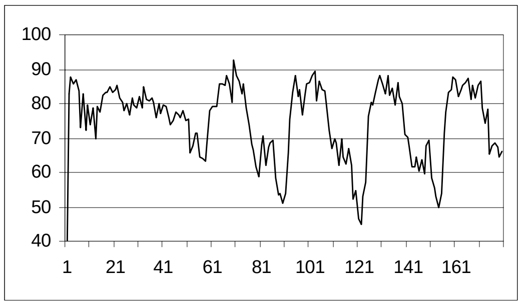
Primitive Communication (%)



Graph A.60

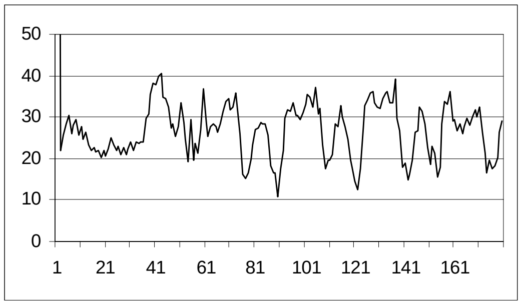
DX, 32 processors, 20 prims/group, previous-frame:

Primitive Utilization (%)



Graph A.61

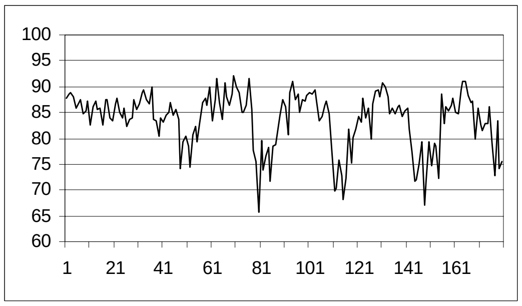
Primitive Communication (%)



Graph A.62

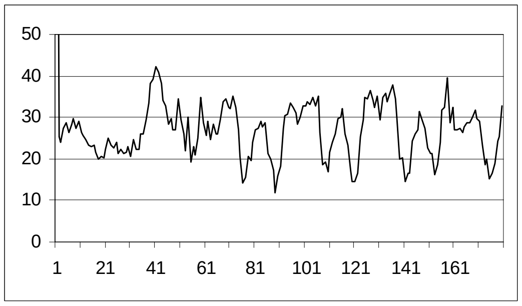
DX, 32 processors, 20 prims/group, stochastic:

Primitive Utilization (%)



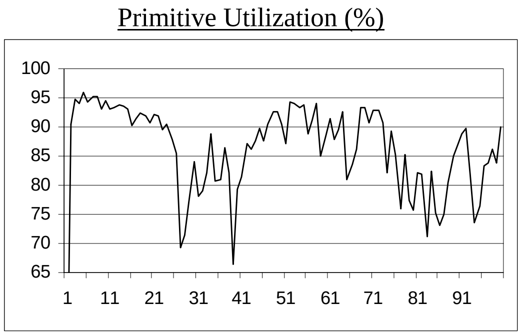
Graph A.63

Primitive Communication (%)

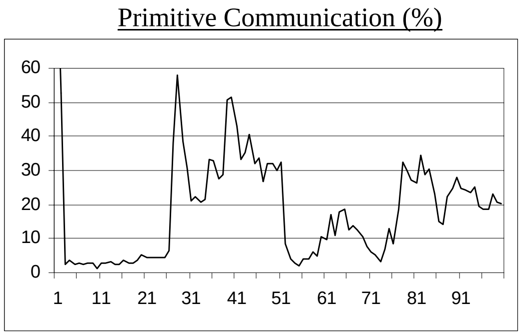


Graph A.64

Parlment, 8 processors, 5 prims/group, previous-frame:

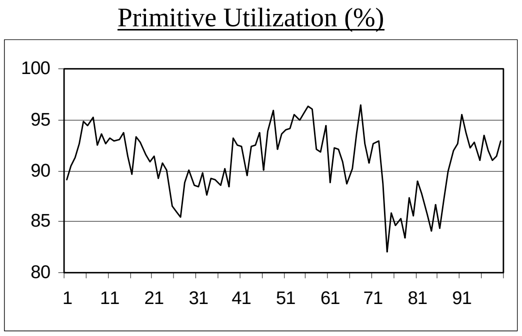


Graph A.65

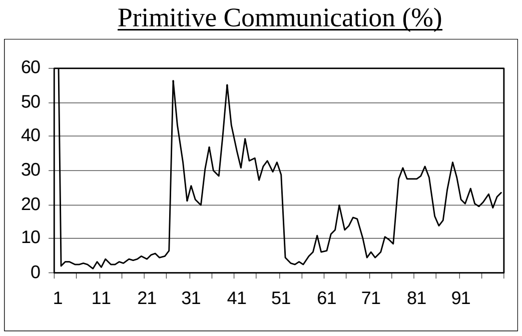


Graph A.66

Parlment, 8 processors, 5 prims/group, stochastic:

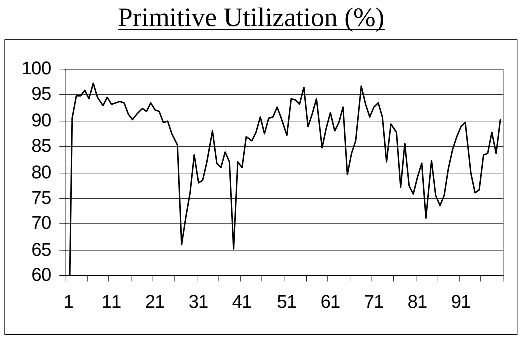


Graph A.67

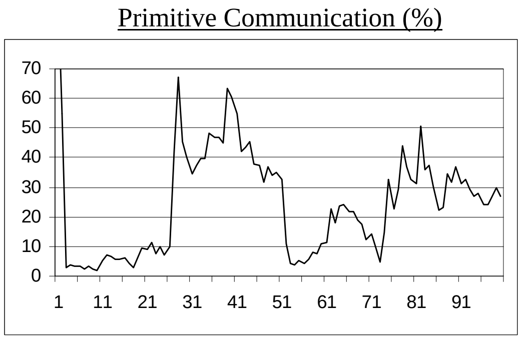


Graph A.68

Parlment, 8 processors, 20 prims/group, previous-frame:



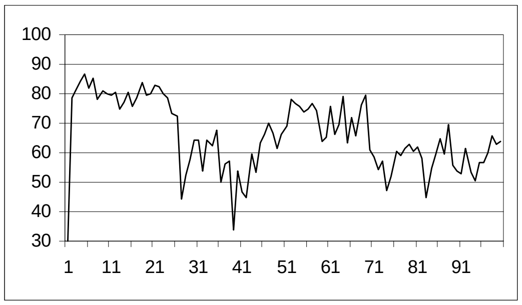
Graph A.69



Graph A.70

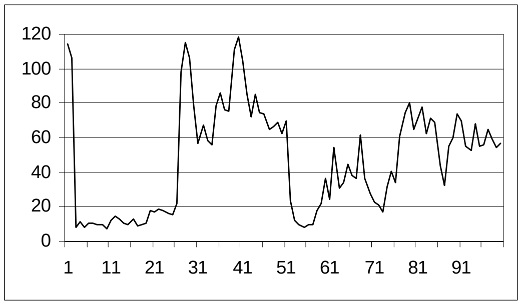
Parlment, 32 processors, 5 prims/group, previous-frame:

Primitive Utilization (%)



Graph A.71

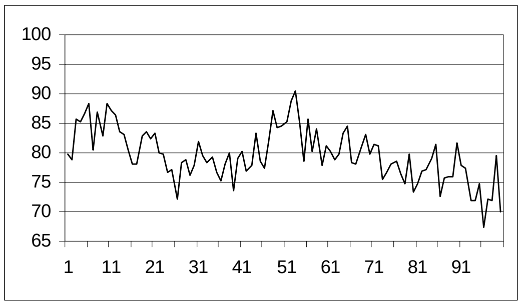
Primitive Communication (%)



Graph A.72

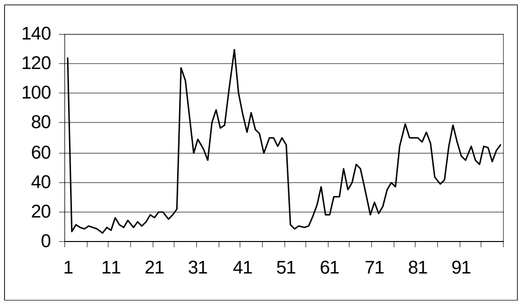
Parlment, 32 processors, 5 prims/group, stochastic:

Primitive Utilization (%)



Graph A.73

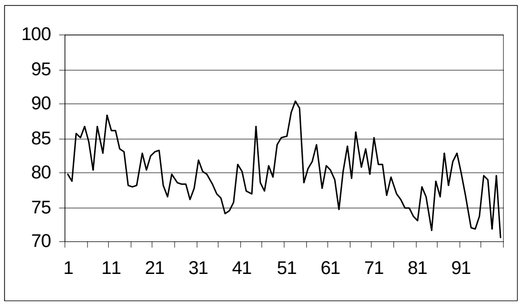
Primitive Communication (%)



Graph A.74

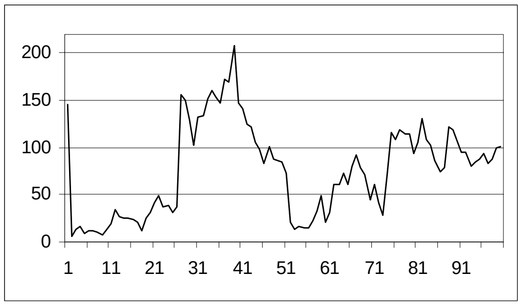
Parlment, 32 processors, 20 prims/group, stochastic:

Primitive Utilization (%)



Graph A.75

Primitive Communication (%)



Graph A.76

Appendix B

Recent Developments

A group of researchers from Princeton University implemented a type of sort-first system based upon multiple commodity-PC workstations [SAMA99]. Eight PCs with 3D graphics-accelerator cards were networked using a Myrinet 100 MB/s network [BODE95]. A ninth PC served as a master controller. The PCs were attached to eight LCD video projectors arranged in a four by two fashion to provide a large, seamless image.

With this system, the researchers implemented and tested three different algorithms for load-balancing. These will be discussed shortly. In order to simplify database handling and keep network bandwidth down, a static database scheme is used. All nine PCs read the same 3D database and store it entirely in memory. In order to draw a frame, the master PC first culls the database using various algorithms to produce a potentially visible set of leaf nodes from the database hierarchy. It then sends rendering commands to the slave PCs indicating which leaf nodes must be rendered for which regions of the screen. Since only a single PC is performing the traversal and culling step, this portion of the graphics pipeline is both a potential bottleneck as well as a source of latency. Whether or not this is a real problem depends upon the granularity of the database leaf nodes. Larger leaf nodes reduce the amount of traversal and culling work. However, this also results in more overhead just in the same way that using a larger group size does for the solutions proposed in this thesis.

The load-balancing solutions examined in this work were called “grid-bucket assignment” (GRID for short), “grid-bucket union” (UNION), and “KD-split.” The GRID algorithm starts with a regular rectangular subdivision (a fixed granularity ratio is chosen for any given run). The master processor considers the 2D axis-aligned bounding box of each leaf node and forms an estimate of the time to render each tile of the rectangular subdivision. Tiles are initially assigned to the PC whose projection region contains the tile. Each tile is considered an independent work unit. The master then uses a greedy algorithm to reassign tiles based upon a cost-benefit computation that compares the local rendering benefit vs. the remote rendering cost of rendering the tile elsewhere. The algorithm terminates when the most loaded PC cannot reassign any tiles in a beneficial manner.

The UNION algorithm is similar to the GRID algorithm. However, it aims to reduce the work that results from the primitive overlap factor. Rather than consider each tile independently, adjacent tiles are merged together, and all primitives overlapping any tile in the set are rendered only once. This process affects the cost-benefit computation, making it more beneficial to reassign regions that are adjacent rather than scattered.

The KD-split algorithm is similar to Whelan's median-cut method. However, rather than considering primitive centroids, the splitting algorithm considers the 2D bounding boxes of database leaf nodes. It still requires sorting the vertices of the bounding boxes in both X and Y dimensions, but by taking into account the 2D extents, it attempts to choose splits that avoid overlaps.

The researchers results were not very surprising. The GRID algorithm required a high granularity ratio in order to produce acceptable load-balancing, yet using such a high ratio resulted in excessive overhead due to the overlap factor. This matches our findings from the static interleaved algorithm. The UNION algorithm was able to reduce this overhead, but it suffered from high computation time (for high granularities) and a load-balancing difficulty when database leaf nodes are large compared to screen tiles. A positive aspect of both GRID and UNION is that pixel redistribution time is kept very low. The KD-split algorithm, on the other hand, traded off pixel-redistribution time for good load-balancing and very low overlap overhead. The tradeoff was a favorable one, usually resulting in better overall performance than the other algorithms. However, the computation time for the KD-split algorithm is also relatively high, typically on par with the expected ideal rendering time.

The load-balancing performance of the KD-split algorithm may be marginally better than that of the MAHD algorithm. This statement is a conjecture, since no direct comparison is available. Average efficiency (computed as avg/max rendering time) from the test cases shown appears to be around 90-95%. However, the computational overhead for KD-split is significantly higher due to the need to sort and iterate over the coordinates of all the database leaf nodes. This seems a high cost to pay for the small potential gain.