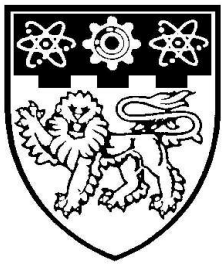
Nanyang Technological University



A Framework for a Realtime Distributed Rendering Environment

Submitted as the Dissertation in fulfillment of the requirements for the degree of Doctor of Philosophy

by

Zhu Huabing

Supervisor: Dr. Tony Kai-Yun Chan

Division of Computing Systems
School of Computer Engineering
Nanyang Technological University
Singapore 639798

August 2004

Abstract

Graphics hardware capability is being pushed to its limits by the demands of interactive visualization applications. Not only is the complexity of the models increasing, but their images are also displayed at higher resolution. As a consequence, many researchers have looked into the speeding up the computation by distributing the rendering tasks over a cluster of computational units. However, to the best of our knowledge, little research has been done for parallel rendering on distributed environments with multiple clusters. UNC researchers have devised a taxonomy of parallel rendering architectures, naming the classes "sort-first", "sort-middle" and "sort-last". From their analysis, none of the homogeneous architectures is a clear winner under all conditions; rather, each is potentially useful for some set of applications and implementation constraints.

This dissertation describes a specific heterogeneous architecture, named "sort-both", which combines both sort-first and sort-last strategies, for distributed rendering environments where there is disparity between the fast connections inside a cluster and slow connections between clusters. However, to achieve good performance in these environments, it is important to show how sort-first and sort-last can be implemented efficiently. A new dynamic load-balancing sort-first algorithm is presented which allows for efficient workload distribution. A novel sort-last method is also developed which requires smaller communication bandwidth than the traditional one.

This system architecture has some important advantages. At the inter-cluster level, it uses the sort-first algorithm to achieve low communication cost and reduced redundant rendering. At the cluster level, the sort-last algorithm achieves good load balancing and scalability with small pixel redistribution cost. Sort-both is a viable and very com-

petitive architecture. The issues are explored thoroughly in this dissertation. Based on these advantages, our thesis proposes that sort-both is the best choice for interactive rendering involving large retain-mode database, high-resolution displays in distributed computing environments. While this dissertation clears the way for the sort-both implementation, there are still many opportunities for further exploration in this area.

Acknowledgement

I would like to express my deepest gratitude and respect to my supervisor, Dr. Chan Kai Yun Tony for his remarkable guidance, heuristic advice and encouragement. My research work benefits a lot from his enormous supportive efforts, even in writing up of this dissertation. Without his help, I could not done so far at all. I also would like to thank Mr. Wang Lizhe, for his creative discussion and valuable advice that inspire me greatly.

Many thanks to Mr. Zeng Yi, Mr. Tang Ming, Dr. Cai Wentong, Mrs.Irene Goh, Dr. Chow Kin Keung, Dr. Kang Zhijian, Dr. Simon See for their selfless help and worthy cooperation. There are many people, both in and out of NTU, whose presence has helped me in innumerable ways during my stay here. I thank you all for opening your hearts and minds to me.

Finally, I would like to give special thanks to my wife and my parents for providing constant support and encouragement during my graduate career.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This is a dissertation on high-performance interactive computer graphics architectures for distributed computing environments. Our research focuses on parallel rendering on distributed-memory message-passing (DMMP) architectures as found in most clusters and Grid [26] computing environments.

1.1 Basic Rendering

The term "rendering" refers to the computational process of generating an image from an abstract description of a scene [15]. Computationally, the rendering process is a simulation problem in which light's interactions with the supplied scene description are computed [47]. Rendering research mainly deals with computationally efficient algorithms and models for image synthesis. Rendering systems can be classified according to two axes: the representation used to describe the environment and the algorithm used to create the images [39].

In general, many algorithms can be used for a given scene representation, and many representations may be used in conjunction with an algorithm. The various representa-

tions of 3D environments can be classified into surface-based representation, solid-based representation, and image-based rendering. Surface models (e.g., NURBS, subdivision surfaces, and polygon meshes) describe 3D objects as 2D manifolds in a world. They are the most popular representation formats. Solid models include constructive solid geometry (CSG) and volume models. In CSG, simple primitives are combined by means of a regularized Boolean set of operators that are included directly in the representation [23]. A CSG object is organized by a tree with Boolean operators at the internal nodes and simple primitives at the leaves. Volume data represent characteristics of the 3D environment (e.g., opacity) as a function of a 3D position in space. Often, this function is parameterized by uniformly spaced samples in space called voxels. Currently, the major application area in this field is medical imaging, where volume data is available from X-ray Computer Tomography (CT) scanners or Magnetic Resonance Imaging (MRI) scanners. They produce three dimensional stacks of parallel plane images that each consists of an array of X-ray absorption coefficients. 3D volume data sets give the immediate advantage that the information can be viewed from any view point rather than as individual planes. Image-based representations synthesize images directly from other images without a three dimensional geometric representation. McMillan [47] covered this topic well in his Ph.D dissertation.

Given the plethora of representation formats, the right choice for an application depends on the ease of modelling the environment, the phenomena used in the rendering, and the time constraints of rendering. While the algorithms used for rendering are quite varied, often being directly designed for a particular representation. They may be classified as projection-based or ray-based. With projection-based rendering algorithms,

the primitives are projected onto the image plane by a set of transformation. Because of regularity in computation and access patterns, these types of algorithms tend to be fast. Ray-based algorithms trace the path of specularly reflected and transmitted (or refracted) rays through an environment. A ray is traced, for each pixel, from an eye or view point through the pixel and into the scene. The ray-based algorithms can generate "super"-real images that are never experienced in every day life [73]. The higher computational costs lead them to be much slower. Again, the right choice for a rendering algorithm depends on the phenomena and time constraints. Ray tracing algorithms are not suitable for interactive systems because of their high computational demands. Compared with ray-based algorithms, projection-based algorithms tend to be more computationally efficient and most interactive rendering systems use these algorithms because of the performance level required by human interaction. Therefore, the following discussion is focused on project-based rendering of polygon model.

The "standard graphics pipeline" for these algorithms has two major steps. Starting with primitives (polygons) in object space, a geometry processing step transforms the primitives into screen space. This is followed by a rasterization step to convert the primitives into a set of screen pixels. They finish with a set of appropriately colored pixels in the frame buffer. Each step includes several computationally intensive procedures. Figure 1.1 shows the simplified graphics pipeline.

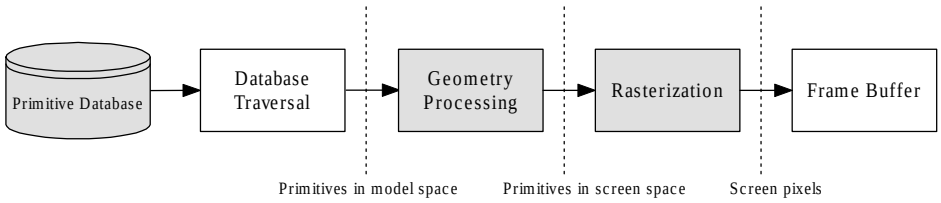


Figure 1.1: Standard Graphics Pipline

1.2 Interactive Rendering

"Interactive" means that the system responds quickly enough such that the user can adjust the controlling inputs in rapid response to the output [58]. "Real-time" means that the system must always keep updating outputs within a certain small fixed amount of time [58]. Some real-time applications, such as key frame animation, do not directly interact with humans and human interaction is confined to a relatively constrained period [23]. In particular, if the latency exceeds beyond approximately 1 second, the human will feel that computer's responses is too slow to interact seamlessly [39]. Researchers reports that minimal latency detectable by a human has been shown to be approximately ten milliseconds [64]. Thus, we define interactive rendering as the process of creating images from 3D environments at rates between approximately 1 and 100 frames per second. Therefore, increasing the frame rate from 100 frames per second to 1000 frames per second is not useful due to the perceptual characteristics of the human brain [39].

However, various interactive applications require different response time. A frame rate, 20 frames per second, is required to fool the human eye into perceiving motion in discrete frames as a continuous motion. It is known as motion fusion rate. The minimum necessary update rate is dependent upon how fast objects in the simulation are moving relative to the display. Padmos and Milders report that thirty frames per second are required to maintain the illusion of fluid motion [63]. Large ship simulations may find 10 frame per second to be a reasonable update rate, while tactical helicopter simulations can demand 60 frames per second or more [44]. The display system involved is an important factor as well. For example, head-mounted displays (HMDs) ideally should

have zero latency, especially if the display has see-through capability.

Because interactive rendering must maintain high frame rates, when massive data sets are involved in some applications, they can require hundreds of MFLOPS of floating-point performance and gigabytes per second of memory bandwidth, far beyond the capabilities of a single processor [53]. As a consequence many researchers have looked into speeding up the computation by parallelizing it over distributed computational resources.

1.3 Parallel Rendering

In computer graphics, when the scene is complex, or when high-quality images or high frame rates are required, the rendering process becomes computationally intensive. To provide the necessary levels of performance, parallel computing technologies have been applied in this field. In the early days of the field, its initial use was primarily in specialized applications. Today, parallel hardware is routinely used in graphics workstations, and numerous software-based rendering systems have been developed for general-purpose parallel architectures. Parallel rendering refers to the exploitation of parallelism in performing the rendering computations [15]. Parallelism of various types may be employed at many levels: for example, functional parallelism (pipelining) can speed critical calculations and data parallelism can be used to compute multiple results at once. Common data-parallel approaches are by object and by pixel or portion of the screen. Temporal parallelism can be applied by off line application. The essence of the rendering task is to calculate the effect of each primitive on each pixel. Due to the arbitrary nature of the modeling and viewing transformations, a primitive can fall

anywhere on (or off) the screen. Thus rendering can be viewed as a problem of sorting primitives to the screen. Molnar et al. [53] describe a classification scheme based on where the sort from object coordinates to screen coordinates occurs, which is believed to be fundamental whenever both geometry processing and rasterization are performed in parallel. This classification scheme allows computation and communication cost to be analyzed and encompasses the bulk of current and proposed highly parallel renderers. To summarize their classification scheme, parallel rendering algorithms fall into one of three classes: sort-first, sort-middle, or sort-last.

1.4 Geographical Distribution

Parallel rendering usually requires a large amount of computing capability. However, for most users, it becomes more difficult to address all requirements on a single computing platform or for that matter in a single location. At the same time, high-speed networks and the advent of multidisciplinary science mean that the use of remote resources becomes both feasible and necessary [25]. In a distributed computing environment, shown as Figure 1.2, various resources are available, e.g. large volume data storage, supercomputer, and video equipments and so on.

Geographically distributed resources can be harnessed as parallel rendering components. However the implementation of a distributed parallel rendering system encounters various challenges. One problem is that the developer have to deal with resources with heterogeneous capabilities. For example, networks ranges from multi-gigabit/s Myrinet to dialup lines. Even apparently identical resources may be configured in different ways at different sites or provide different interfaces and services. As the capabilities

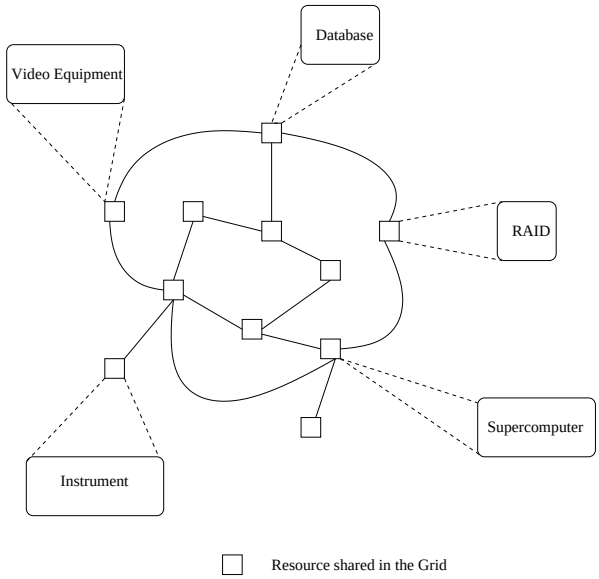


Figure 1.2: Geographical Distribution of Resources

of remote resources are typically not known to the application developer, applications need to be able to discover and then act upon configuration information [25]. Another problem is different policies. Distribution also leads to variation in the policies that govern who can use resources, how resources are paid for, and what resources can be used. Because the resources in a distributed environment are shared by many users, resource behaviors are often dynamic and - from the point of view of the user at least - unpredictable [25]. Therefore, some special mechanisms, such as resource reservations and mirroring, are required for reducing the impact of such unpredictability; in addition, applications must be able to deal with changes in resource characteristics.

1.5 Thesis Statement and Contributions

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

" Among the classes of fully-parallel graphics architectures, a hybrid of sort-first and sort-last architecture named sort-both is the best choice for the interactive rendering involving large retained-mode database, high-resolution displays in distributed computing environments. "

The exact values for " large " and " 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 " high-resolution " suggests rendering over eight million pixels [58]. In this thesis, the term, distributed computing environment, suggests the networked computing resources are geographically distributed, especially in a multi-cluster environment. This dissertation concentrates on message-passing systems because they are the predominant class of scalable parallel system in use today. It will be seen, however, that the relative cost of data transmission is an important parameter which can influence a number of design decisions.

This dissertation is not the first attempt to apply parallel rendering technology in distributed computing environments. Parallel rendering in single cluster has been defined, explored, and implemented in many different contexts (see Chapter 2). Most of them use homogeneous parallel rendering strategy.

This dissertation differs from previous work in that it uses a heterogenous strategy to achieve high rendering performance in multi-cluster environments. It addresses the problem of low bandwidth connections between clusters.

In order to demonstrate the above thesis, two tasks should be accomplished. First, it should be shown that sort-both is a viable architecture in distributed computing environments. This requires us to address the major difficulties in creating a sort-both

implementation. Namely, these are the need to find an efficient load balancing algorithm for sort-first and the need to find an efficient low overhead algorithm for sort-last. Both of these are complex and interrelated issues which will be discussed in the following sections.

Second, it should be shown that among the fully-parallel architectures, sort-both is best suited to the application domain suggested in the thesis statement. This task includes three comparisons. Compared with three homogenous architectures, sort-both shows its advantages and this establishes our thesis.

1.5.1 Sort-both Architecture

In a distributed environment, the system distributes rendering tasks (geometry data) among the rendering nodes, collects results (image data) from them and sends them to a remote display equipment. How to achieve maximum computational potential to serve the rendering task is the key problem. This problem involves load-balancing, minimizing the computational overhead and minimizing the bandwidth overhead.

A typical characteristic of distributed computing environments is the large gap between the fast connection inside a cluster and slow connection between clusters. An application designed for uniform speed interconnects can lead to performance degradation. Sort-both architecture incorporates both sort-first and sort-last [53] algorithms to achieve high performance when rendering complex data sets with high-resolution displays and distributed computing resources. It takes the advantage of the hierarchical nature of the distributed computing environment. It decomposes the image space into tiles with sort-first algorithm and distributes them over multi-clusters and employs a

peer-to-peer sort-last scheme to compose sub-images inside a cluster. In this way, low communication cost is achieved by the sort-first strategy between clusters. As with most sort-first algorithms, the number of partitions will impact the scalability. In this way, because the partition number is small, the redundant rendering introduced by sort-first is relatively small. Inside the cluster level, the sort-last algorithm achieves good load balancing and scalability. Sort-last algorithms have natural load balancing without redundant rendering. As each image tile is a low-resolution sub-image, it cuts down the pixel redistribution cost.

Many implementation details are examined regarding the sort-both architecture. We examine how the architecture fits into distributed computing environments. This involves various load-balancing, communication, and scalability issues.

1.5.2 Efficient Load-balancing Algorithm for Sort-first

In a sort-first algorithm, screen-space is partitioned into non-overlapping 2D tiles (i.e., regions or partitions), each of which is rendered independently by a tightly-coupled pair of geometry and rasterization processors, and the sub-images for all 2D tiles are composed (without depth comparisons) to form the final image. The main advantage of sort-first algorithms is that the communication requirements are relatively small, and they scale well with increasing numbers of processors. However, sort-first systems are susceptible to load imbalance [53]. Load balancing is critical for distributing the rendering workload evenly among the processors and preventing a single overloaded processor from becoming a major bottleneck. Static load-balancing methods are typically not well-suited to sort-first architectures due to the extra overhead created when subdivid-

ing the screen into more regions than there are processors (which is necessary in static load-balancing to even out the processor load) [58].

A new load-balancing sort-first algorithm, known as "DPBP" (Dynamic Pixel Bucket Partition), 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 DPBP only assigns one region per processor, it does not suffer from the overhead that upsets the static load-balancing method. DPBP subdivides the screen using a binary tree, and it works for an arbitrary number of processors (it is not limited to powers of two).

1.5.3 Sort-last Parallel Rendering without Depth Comparison

In a typical sort-last algorithm, each rendering processor is assigned arbitrary subsets of the primitives. Each computes pixel values for its subset, no matter where they fall in the screen. Rendering processors then transmit these pixels over an interconnect network to composing processors which resolve the visibility of pixels from each rendering processor with depth comparison. The interconnect network, however, must handle all of the pixel data generated on all of the rendering processors. The high communication cost for pixel redistribution make it inapplicable for interactive or real-time applications that render high-quality images.

A novel sort-last method without depth comparison is presented in this thesis. It is named Image Layer Composition with Alpha (ILCA). It sorts the primitives along the Z axis and partitions the primitives into different image layer volumes. Each rendering processor renders a subset of the primitives in one volume and generates a sub-image.

The sub-images are composed with the alpha value in each pixel. In this way, the bandwidth requirement is greatly cut down. On the other hand, this method can adopt the parallel pipeline composition [45] which is useful to take full advantage of the computational capability to speed up image composition.

1.5.4 Additional Contributions

In order to analyze the performance model of the sort-both architecture, a 3D overlap factor equation is derived. From this equation, it can be found that compared with homogeneous sort-first architecture, sort-both introduces less redundant rendering.

In order to prove our thesis statement, we perform a detailed comparison between sort-both, sort-first and sort-last (having already ruled out sort-middle as a potential candidate). We detail the differences in communications requirements, and we offer an analysis to describe the computational overhead of each architecture. The comparison reveals that sort-both saves in communications bandwidth and introduces less computational overhead. Therefore it makes sort-both the better candidate for the targeted application area.

1.6 Organization of the Dissertation

In Chapter 2, the field of parallel interactive rendering is more fully introduced. A detailed description of the sort-first, sort-last, and sort-middle architectures are presented. The architectures and algorithms of some successful systems are reviewed.

In Chapter 3, we look at each homogeneous architecture in the light of application demands and compare their communications bandwidth requirements. From the dis-

cussion, it is found that none of homogeneous architecture is superior to the other in distributed computing environments. In order to solve the problem, a hybrid architecture named sort-both is presented. It involves both sort-first and sort-last strategies at different levels.

In Chapter 4, the issue of load balancing with sort-first is examined. 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. DPBP is an adaptive load-balancing sort-first algorithm for partitioning the screen into multiple sub-regions. At the same time, it also partitions the geometry data into groups according to its overlapping of the sub-regions.

In Chapter 5, the novel sort-last strategy, ILCA, is presented. With this method, the geometry data is decomposed into multiple image layer volumes alone the Z axis. Each rendering processor generates an image layer with the assigned primitives. To compose the final image, this strategy simply layers these image layers on top of one another according to the visibility order of the layers. In this way, the bandwidth requirement is cut down. In this chapter, a 3D overlap factor is proposed to examine the redundant rendering of sort-both architecture.

In Chapter 6, The sort-both, a hierarchical architecture, is introduced in detail. A possible software model implemented on Computational Grids is discussed. Compared with homogeneous sort-first and sort-last architectures respectively, sort-both architecture demonstrates its good scalability.

Finally in chapter 7, we offer some conclusions and include some discussion which is out of our focus in this dissertation.

Chapter 2

Fully Parallel Rendering

This chapter contains some background information for the reader. It first introduces some terminology by describing the graphics pipeline. Additional information on these topics can be found, respectively, in standard graphics texts [23]. It also introduces the classification of the parallel rendering architectures. The properties of each architecture are presented. The remaining sections describe previous work in parallel graphics. Much of the work focuses on polygon rendering. Some algorithms that render primitives other than polygons or perform ray tracing are included since many of their load-balancing algorithms are applicable to polygon rendering. Some successful systems are also introduced. The reader can find other surveys of parallel algorithms and architectures in [8], [75], and Chapter 18 of [23].

2.1 Graphics Pipeline

The rendering pipeline is a logical mode for the computations needed in a raster-display system, but is not necessarily a physical model since the stages of the pipeline can be implemented in either software or hardware [23]. The stages are:

- Geometry processing, which performs per-vertex operations such as coordinate transformations, lighting, texture coordinate generation, and clipping (may be hardware-accelerated).
- Rasterization, which performs per-pixel operations such as the simple operation of writing color values into the frame buffer, or more complex operations like depth buffering, alpha blending, and texture mapping (may be hardware accelerated).

However, the detail stage of each type of shading is different. Figure 2.1 shows the graphics pipeline of Gouraud [32] or Phong [7] shading algorithm.

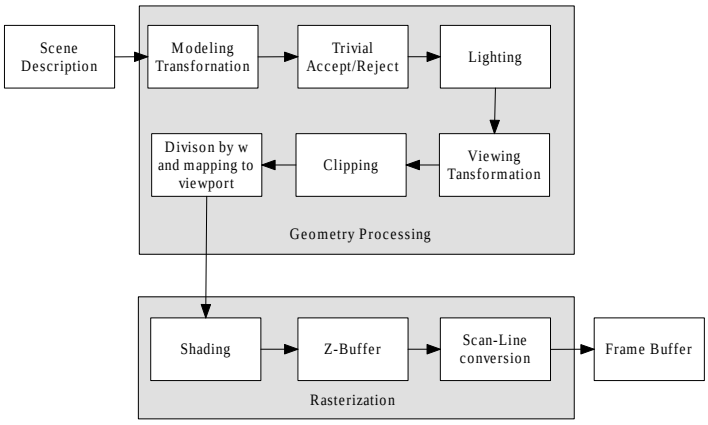


Figure 2.1: Graphics Pipline of Gouraud or Phong Shading

In the geometry processing stage, the modelling transformation procedure transforms the graphics primitives from the object-coordinate system to world-coordinate system. In addition, one or more surface-normal vectors may need to be transformed depending on the shading method to be applied. In the trivial accept/reject classification procedure, each vertex is tested against the six bounding planes of the viewing frustum. The primitives completely out side the viewing frustum are rejected. Depending on the shading algorithm to be applied, an illumination model must be evaluated

at various locations in lighting procedure. When viewing transformation, primitives in world coordinates are transformed to normalized projection coordinates. In some cases, modelling and viewing transformation matrices can be combined into a single matrix. Then, only one transformation procedure is required in the pipeline. In the clipping procedure, lit primitives that were not trivially accepted or rejected are clipped to the view volume. If back-face culling is enabled, the primitives with their back facing to the eye are removed. The procedure of division by w and mapping to view port perform the perspective division, and map the primitives to the appropriate window coordinates.

In the rasterization stage, scan-line conversion: determine the interior of the polygon. This can be done by first calculating the first and last scanlines that the primitive crosses, and then, for each scan line, calculating the first and last pixels that are inside the polygon. Z-buffer: determine each pixel visibility by comparing the pixel depth with the stored Z value. Shading: according to the illumination model applied, the pixel color values are calculated. When performing Phong shading and/or texture mapping, the last step is often modified to interpolate the normal vector and/or texture coordinates, and the shading is then calculated from the interpolated values.

Although both pipelining and parallelism can be used to build high-performance rendering system, parallelism offers serval advantages, including reconfigurability for different algorithms [23]. The performance of a pipeline system is limited by the throughput of its slowest stage, pipelines do not scale up as readily as do parallel systems. Parallel systems, on the other hand require more complicated synchronization and load balancing and cannot use specialized processors as well as can pipelined systems.

2.2 Rendering Parallelism

Several different types of parallelism can be applied in the rendering process. These include functional parallelism, data parallelism, and temporal parallelism [16]. Each type is suitable for a set of applications or specific rendering methods.

- Functional parallelism. Different processors perform different functions in the rendering pipeline. It is used in pipelines of processors, where the operations to be performed are distributed among a set of different processors. As a processor completes work on one data item, it forwards it to the next processor, and receives a new item from its upstream neighbor. The degree of parallelism achieved is proportional to the number of functional units in the pipeline. Therefore, the available parallelism is limited to the number of stages in the rendering pipeline.

- Data parallelism. The data are split up among the processors, and each processor performs operations on a portion of the data. The parallelism achievable with this approach is not limited by the number of stages in the rendering pipeline. However the performance is challenged by the network bandwidth and the number of processors which can be incorporated into a single system. Data parallelism can be classified into two categories: object parallelism and image parallelism [16, 21]. Because the data-parallel approach can take advantage of larger numbers of processors, it has been adopted in one form or another by most of the software renderers which have been developed for general purpose "massively parallel" systems. Data parallelism also lends itself to scalable implementations, allowing the number of processing elements to be varied depending on factors such as scene

complexity, image resolution, or desired performance levels.

- Temporal parallelism. It is also mentioned as frame parallelism. In animation applications, the tasks can be decomposed in the time domain. Different processors work on successive frames in an animation sequence or interactive session. The main advantage of frame parallelism is its linear speedup, as the usual performance is proportional to the number of processors. However, the performance increase is not accompanied by a reduction in latency.

Obviously, data parallelism is most suitable for interactive rendering with massive data sets. To achieve the highest performance, each step of a parallel rendering algorithm must have the associated data divided among multiple processors. Such algorithms are called *fully parallel* [21, 53] algorithms. In a fully parallel architecture, both geometry processing and rasterization must be performed in parallel.

2.3 Class of Parallel Rendering Architectures

2.3.1 Rendering as a Sorting Problem

The essence of a rendering task is to calculate the contribution of each primitive on each pixel. However, a primitive can fall anywhere on (or off) the screen. Thus rendering can be viewed as a problem of sorting primitives to the screen [70]. If each pixel is regarded as a storage bin, each primitive must be sorted by placing its contribution into the set of pixel bins overlapped by this primitive in screen space. That means each primitive must choose the correct pixel bins, and then one must choose the correct depth within the pixel bins in order that the contribution is added correctly.

For fully parallel rendering, this sort involves a redistribution of data between processors. The location of this sort largely determines the structure of the resulting parallel rendering system. Molnar et al. [53] have classified parallel rendering strategies into sort-first, sort-middle and sort-last. The classification is based on where the sort takes place in the translation from object coordinates to screen coordinates.

2.3.2 Sort-first

Figure 2.2: Sort-first. Redistributes raw primitives during geometry processing. [53]

In sort-first architectures, the image space is partitioned into non-overlapping 2D regions (i.e., tiles) and each processor does just enough geometry processing to determine the region of the raster image that a primitive will belong. The primitive is then sent to the appropriate processor to perform both the geometry processing and rasterization for that region of the raster image. The main advantage of sort-first is low communication

requirements. Another advantage is that the processors implement the entire rendering pipeline for a portion of the screen. These strong points benefit the system performance in distributed environments such as clusters and computational Grids. However, it is susceptible to load imbalance because of the random distribution of the primitives in the image space. Another significant disadvantage is the extra rendering work, which can be characterized by the overlap factor, the ratio of the total rendering work performed over the ideal rendering work required without redundancy. Since the overlap factors grow with increasing the number of processors, the scalability of sort-first systems is limited.

2.3.3 Sort-middle

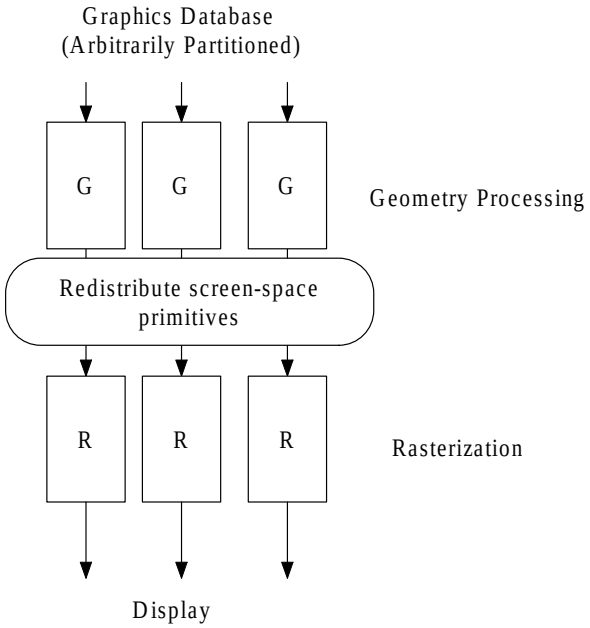


Figure 2.3: Sort-middle. Redistributes screen-space primitives between geometry processing and rasterization. [53]

Sort-middle architectures are those that perform sorting and data redistribution at

the obvious place - between geometry processing and rasterization. In a sort-middle architecture, each geometry processor computes screen coordinates for each primitive that has been assigned to it. It then determines the rasterizing processor to which it will send a primitive's scan line information, based on the raster image's partition among processors. One problem with sort-middle algorithms is that they are susceptible to load imbalance of rasterizing processors due to non-uniformly distributed primitives. Another problem is high communication cost if the tessellation ratio is high. Sort-middle is best suited for tightly-coupled systems that use a fast, global interconnection to send primitives between geometry and rasterization processors.

2.3.4 Sort-last

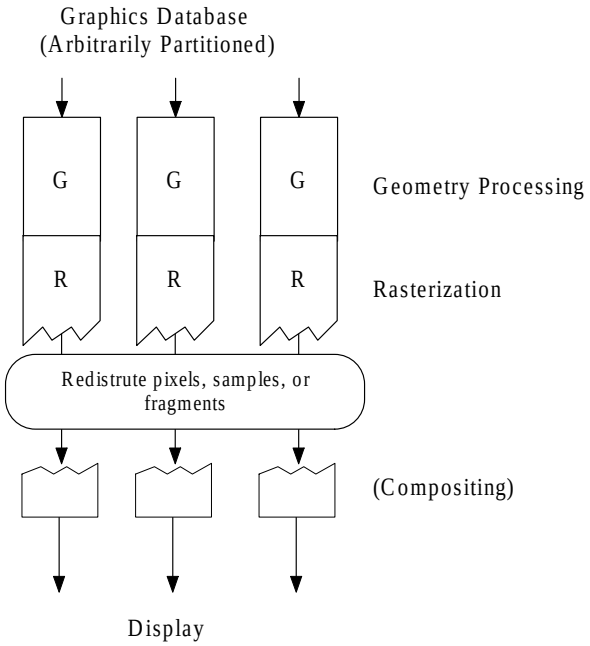


Figure 2.4: Sort-last. Redistributes pixels, samples, or pixel fragments during rasterization. [53]

The sort-last architectures defer sorting until the last stage. Primitives are arbitrar-

ily distributed to available processors. Each processor then performs both geometry processing and rasterization of its assigned primitives, regardless of where they fall within the raster image. The final stage is to compose (sort and merge) the raster images from all rendering processors to form the final image with depth information. The advantage of the sort-last algorithms is that, it is less prone to load imbalance and is scalable. The main disadvantage is that it usually requires an image composition network with very high bandwidth and processing capabilities to support transmission and composition of overlaps [53].

2.4 Algorithms for General-Purpose Parallel Computers

According to the classification of Molnar et.al [53], the discussion about algorithms is divided into three sections. Most of these algorithms only support a simple shading model, as they are optimized for speed. It is difficult to compare the performance of these earlier works, because many factors can impact the performance, e.g. databases, rendering quality, hardware architecture, bandwidth. Therefore, it is meaningless to compare the triangles per second within different conditions. In this dissertation, some important figures not directly related to specific conditions are quoted, e.g. scalability, load balancing. overlap factors. These figures are the most favorable for each type of implementation, and are the ones typically reported.

2.4.1 Sort-first

With sort-first, the screen is subdivided and the regions are given to different processors. The screen partitioning methods can be broadly categorized as either static or adaptive.

The general static approach is to divide the screen into more sub-regions than the number of processors and assign the sub-regions to the processors in interleaved fashion. Region shapes have been based on scan lines [30], horizontal strips [17, 41, 74], vertical strips [74], and rectangular areas [11, 14, 41, 65, 76]. 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. The first interleaved partitioning algorithm was proposed by Fuchs et.al [28]. Figure 2.5 shows the processors are assigned regions in an interleaved pattern.

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

Figure 2.5: Processors Are Assigned Regions in an Interleaved Pattern.

Obviously, in this way, load balancing is difficult to be guaranteed, since it is still possible that most of the primitives are concentrated into one region. Then, the over-loaded processor will be a bottleneck of the system. In order to overcome this drawback, some techniques have been used, such as Level of Detail (LOD) [12] control and culling algorithms. Another problem is that a large number of partitions will introduce large redundant rendering. Primitives that overlap region boundaries must be processed in

multiple regions. This inefficiency is expressed in terms of the overlap factor (O), the number of regions covered by a typical primitive [53]. Both the overlap factor and primitive traffic are proportional to the total linear length of the region boundaries. Shown as Figure 2.6, it is intuitively easy to see, since having more boundaries will provide more opportunities for primitives to cross them. Therefore, a large number of partitions will dramatically increase the computation and communication overhead and do harm to the system scalability. Cox and Bhandari [14] investigated the relationships between region (bucket) sizes and overlap factors incurred in a bucket rendering system using static, grid-aligned rectangular regions . Although they considered alternative region arrangements, they didn't examine the extent to which they may reduce overlap. An appropriate number of partitions, namely granularity ratio, is important for static sort-first algorithms. For overhead considerations, a small granularity ratio is better, whereas having a larger granularity ratio would generally improve the load-balancing performance.

Figure 2.6: Bourdary Length and Number of Regions [58]

Adaptive approaches dynamically partition the screen space. there are a variety of solutions. Adaptive solutions offer the possible benefit of keeping the number of divisions to a minimum, but at the cost of increased overhead and complexity.

Samanta et.al divided adaptive decomposition approaches into three types, *Top-down, Bottom-up,* and *Optimization* [68]. Top-down approaches start from the screen-space as a whole and divide it recursively into regions based on estimated workloads. Bottom-up approaches start from a large number of predetermined small regions and combine them into larger regions that are then assigned to processors. Optimization approaches begin with some initial decomposition and assignment (e.g. a static one or the one from the previous frame) and adjust it to balance the workload by cutting out and reassigning smaller regions from existing partitions to meet some load balancing criterion.

Whelan's median-cut [74] method is a typical Top-down approach. Median-cut splits the screen into subregions based on the onscreen position of the centroids of each polygon. The algorithm recursively splits the screen (along the longer dimension of the given region) until the number of regions equals the number of processors. Since the algorithm is based only on the centroids of primitives, it cannot accurately take into account the onscreen area overlapped by the primitives. Therefore, it is difficult to achieve good and steady load balancing in this way.

Muller's mesh-based adaptive hierarchical decomposition (MAHD) algorithm is another Top-down decomposition approach [57]. Primitives are first tallied up according to how their bounding boxes overlaps a fine mesh. For each cell covered by a given primitive's bounding box, an amount proportional to the rendering costs for that primitive is tallied . Once all the primitives have been counted, the cells are added up to form a summed area table. Then, using this data table as a hint, screen space tiles are recursively split along their longest dimensions until the number of regions equals the

number of processors.

Whitman implemented a parallel rendering system on the BBN Butterfly [76], a non-uniform memory access (NUMA) shared-memory system. Whitman investigated several methods of screen-space subdivision. All use dynamic scheduling, also known as work-queue scheduling. His method is somewhat similar to Whelan's. The method starts with a regular rectangular decomposition with 40 regions per processor, and counts the polygons in each region. The final regions are made by starting with a single full-screen region, and then repeatedly dividing the region with the largest number of polygons in half until there are 10 regions per processor. This method can not automatically balance the workload among the processors. Dynamic task assignment is used to even out the processor load imbalance. Each processor works on a region at a time. When a processor is free and no more unassigned regions are available, the processor finds the processor that has the most work left undone, and takes half of it. Whitman experimented with different numbers of regions per processor. However, the resulting finer granularity of the regions means a higher overlap factor, resulting in more overhead for this method [58].

2.4.2 Sort-middle

Sort-middle is the most widely implemented in hardware systems. Sort-middle algorithms can be separated into two distinct categories: interleaved and tiled. Interleaved algorithms apply interleaved image parallelism. Each primitive is broadcasted to all rasterizers. SGI's Infinite Reality [55] is typical. Infinite Reality interleaves the screen across the rasterizers in 2-pixel wide vertical stripes, and broadcasts primitives to all the

rasterizers. It is quite bandwidth-intensive. The vertex bus bandwidth will ultimately limit the number triangles per second the system can process. Therefore, the system is difficult to scale.

Tiled algorithms use tiled image parallelism, sorting each primitive to only those rasterizers whose partition it overlaps. Thus, the system may scale to higher levels of performance. Some sort-middle tiled algorithms processe geometry data in primitives order such as Argus. Argus, a software parallel renderer built at Stanford, is a representative primitive order sort-middle tiled architecture [40]. Sort-middle tiled architecture may choose to process the primitives in tile order rather than primitive order. The tiles are processed sequentially, first processing all the primitives that overlap the first tile, then all the primitives that overlap the next tile, etc. The tile processing may also be parallelized, so that multiple tiles are processed simultaneously. Ellsworth report a polygon rendering algorithm which uses a load-balancing method that exploits the frame-to-frame coherence of interactive application [20]. Microsoft's Talisman system [71] divide the screen space into 32×32 tiles. Pixel-Planes 5 [29] is another hardware implementation of a tile order sort-middle tiled architecture.

2.4.3 Sort-last

The sort-last approaches defer sorting until the very last stage. Primitives are arbitrarily distributed to available processors. Each processor then performs both geometry processing and rasterization of its assigned primitives, regardless of where they fall within the raster image. The final stage is to compose (sort and merge) the raster images from all nodes to form the final image with depth information. Molnar et. al divided

sort-last algorithms into sparse merging and full-frame merging [53]. Sparse merging takes advantage of the observation that renderers in a sort-last system may generate pixels for only a fraction of the screen, and only these pixels need be merged. Full-frame merging takes advantage of the fact that merging a full frame from each processor is very regular and can be done using simple hardware.

The naive composition approach is to have a designated processor accept the contributions from all of the other processors, performing the appropriate Z-buffering or composing operations for each contribution. Obviously, in this way, the composing processor will become a bottleneck. With a large number of rendering processors, it will be overloaded. Ma et.al developed the Binary-Swap Composing algorithm [48]. It first distributes primitives arbitrarily among processors. Then, each processor renders its primitives into a whole frame image with Z-buffer. After that, as its name implies, each processor exchanges data with another processor and composes the image. The algorithm arranges the N processors into a conceptual hypercube with dimension $log_2 N$. Then, the processors loop over each dimension in the hypercube. In each step, each processor sends half of its remaining color and Z-buffer to the other processor in the current dimension, and composites its remaining buffer with the incoming data. One processor sends the top half of the remaining buffer, and the other sends the bottom half. The resulting smaller color-buffer and Z-buffer are then used in the next iteration. Figure 2.7 shows an example of the Binary-Swap Composing algorithm using four processors.

Lee et. al [45] developed a family of image composition algorithms for sort-last systems named Parallel Pipeline Composition. In this approach, the color-buffer and

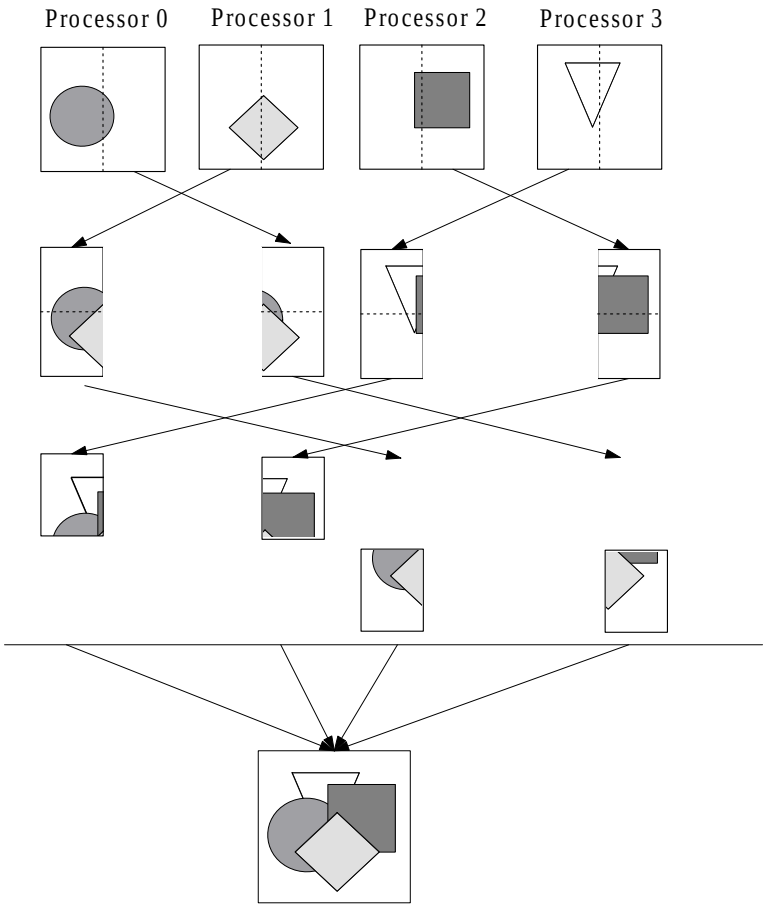


Figure 2.7: Binary-Swap Compositing algorithm with Four Processors.

Z-buffer at each processor is divided into N portions, where N is the number of processors. Processors are organized in circular ring and the sub-images are accumulated in a pipelined fashion along the ring with each processor involved in each stage. At the end, each processor holds a fraction of the final image. Figure 2.8 shows the procedure of parallel pipeline composition with four processors. In order to improve the performance of the Parallel Pipeline Composition, many optimization methods have been developed, including bounding box optimization, direct pixel forwarding, interleaved composition region and adaptive task scheduling.

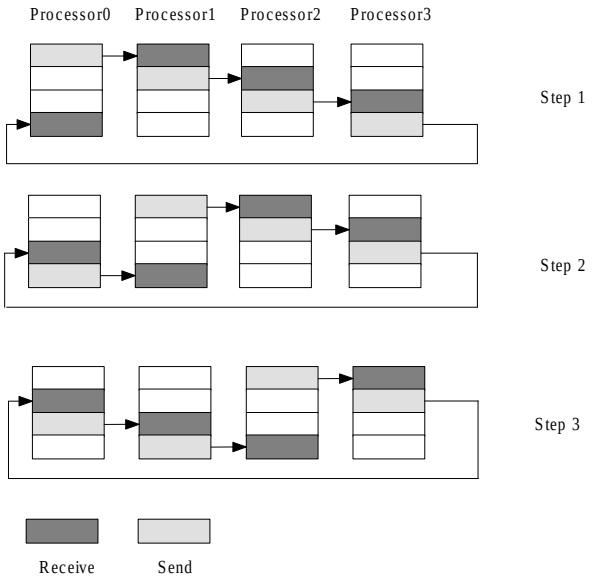


Figure 2.8: Parallel Pipeline Composition with Four Processors

Hinkenjann and Bues [6, 33, 34] presented a parallel rendering systems on commodity hardware. In this system, to reduce the network load, they developed a novel sort-last approach named Hierarchical Depth Buffer (HRD) [6, 33]. HRD organizes the depth buffer with a quadtree hierarchically. The non-leave nodes of the tree contain the overall maximum and minimum depth values(depth interval)of all its child nodes. The leaves of the HRD correspond to the values of the original depth buffer or to a small constant size subset. The hierarchies of the HRDs are sent by the rendering nodes from top to bottom on demand and are examined by the composition nodes from top to bottom. This allows the image composition nodes to decide, starting at the root of the HRD, which color buffers are overlapping or disjoint with respect to their depth. If their depth intervals are pair wise disjoint, only the color buffer with the nearest depth interval has to be considered. Overlapping intervals have to be compared recursively by traversing the HRD downwards to the leaves. At the leaves, a simple depth comparison is performed.

In this way, the rendering nodes needn't send the whole depth buffer to the composition nodes. It reduces the bandwidth requirements during image composition.

Nguyen et.al [59, 60] developed a partition algorithm named Image Layer Decomposition (ILD) that is similar to sort last in that it uses an object partition. However, unlike sort-last, for each frame, ILD repartitions the scene objects into P non-mutually occlusive subsets for a system with P nodes and assign each subset to a node for rendering. Because subsets are non-mutually occlusive, each node generates a coherent image layer; to compose the final image, ILD simply layers these image layers on top of one another according to the visibility order of the subsets (and thus do not need the Z-buffer generated at each node).

2.4.4 Others

Samanta et.al [67] introduced a hybrid sort-first and sort-last parallel rendering algorithm. It simultaneously decomposes the 2D screen into regions and the 3D polygonal model into groups and assign them to PCs to balance the load and minimize overheads [67].

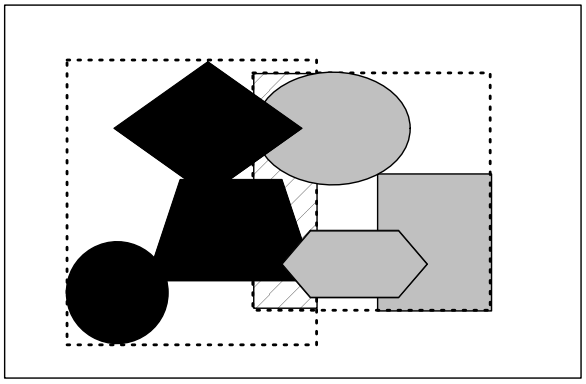


Figure 2.9: Samamta's Hybrid Partitioning Algorithm.

The key idea is that both 2D and 3D partitions are created together dynamically in a view-dependent context for every frame and the pixels overlapping both regions should be redistributed. In this case, 3D partitions will overlap each other in screen space. Shown in Figure 2.9, after partitioning by the hybrid algorithm, the black polygon region will overlap the region of grey polygon region. Therefore, the pixels in the overlapped regions (shaded region in Figure 2.9) will be redistributed for composition. Although it avoids redundant rendering, it has to perform pixel redistribution. When a large number of partitions is needed, the amount of overlapped regions will increase greatly and large communication cost is unavoidable. Therefore, it is only suitable for a cluster environment with high bandwidth connection and moderate number of processors. It is not suitable for large scale distributed environments.

2.5 Parallel Rendering Systems

Research on parallel rendering architecture began more than 10 years ago. The classification proposed by Molnar [53] is a milestone in the research of parallel rendering architecture. During this 10 years, some significant systems have been developed. They will be reviewed in two sections, hardware systems and software systems.

2.5.1 Hardware Parallel Rendering Systems

Most commercial high-end graphics systems adopt high speed bus and dedicated graphic chips. Sort-middle or sort-last architecture are applied in these systems. In this subsection, some typical systems will be introduced.

The Reality Engine [1] and Infinite Reality [55] from SGI are typical sort-middle

hardware systems. A Infinite Reality system includes 4 geometry processing engines, 4 rasterization engines and 8 display channels. After geometry processing, the data will be broadcast to the rasterization engines through the shared bus. It uses round-robin job assignment to schedule the tasks. SIMD array is adopted in the geometry processing engine. The system supports immediate-mode rendering. Infinite Reality is a powerful rendering engine. Its capability of geometry processing is 5 MTri/s and its pixel throughput is 7100 MPixle/s. Since it adopts sort-middle architecture, Infinite Reality requires very high data communication thoughput. This is based on the system bus. Therefore, the system bus becomes the bottleneck. When the polygon distribution on the screen space is unbalanced, especially, when most polygons overlap a small portion of screen space, the rasterization load-imbalancing will impact the system performance greatly.

The Pixel-Planes 4 [27] and Pixel-Planes 5 [29] systems use SIMD arrays for rasterization. The Pixel-Planes 5 front end consists of several Intel i860 processors. Each stores a portion of the model, and transforms it during each frame. After the transformation is complete, the rasterization is performed, with the screen which is subdivided into 128x128 pixel regions, the size of the SIMD array. The system firstly processes the regions with the most primitives. Each geometry processing engine sends the instructions to rasterize the primitives that overlap the region. When all geometry processing engines have sent their commands for a given region, the system begin to processe the region with the next smaller primitive count. This continues until all of the regions are completed. The Pixel-Planes 5 system uses the Phong shading and supports procedural textures. MIPMAP textures are also supported, but at a lower speed.

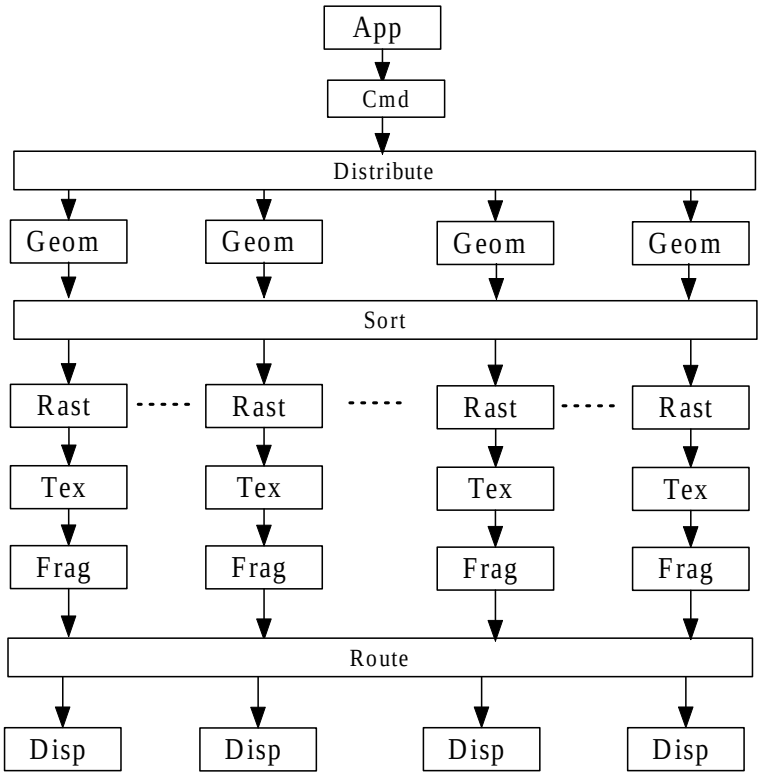


Figure 2.10: Infinite Reality Architecture [18]

PixelFlow [22, 54, 61] is an important hardware implementation of sort-last architecture. In this system, the geometry processing engine, as well as the rasterization engine is parallelized using SIMD arrays. Image composition is supported by a high speed composition network with 100 Gbits/s bandwidth. The high bandwidth guarantees the frame rates of high resolution image composition. Logic-enhanced memory technology is adopted to speedup the image composition in PixelFlow. PixelFlow also provides a shading language supporting procedural textures by which one can programme the rasterization engine [61]. It is very useful for generating high quality images.

Pomegranate [18, 19, 39] is a hardware simulation system developed by Eldridge et. al. They investigated the scalability of the rendering systems. In this system the

graphics pipeline is divided into 6 phases: command, geometry, rasterization, texture, fragment and display. Rendering processing is paralleled in each of the 6 phases.

- The command processors receive OpenGL [69] commands from an application, check the commands and parameters and pass them to the geometry processors.
- According to the current OpenGL status, the geometry processors transform, light, and clip the primitives. According to the bounding box of the primitives, the system send screen-space primitives to the corresponding rasterizers through the high speed networks.
- Each Rasterizer is in charge of several 64×64 pixels regions. The rasterizers perform a rasterization setup on these primitives, and convert them into untextured fragments.
- The texture processors map texture on the resultant fragments.
- The fragment processors receive textured fragments from the texture processors and merge them into the frame buffer.
- The display processors read pixels from the frame buffer and send them to a display.

The network allows each stage of each pipeline of the architecture to communicate with all the other pipelines at every stage. To the best of our knowledge, Pomegranate is the only hardware implementation adopting a sort-anywhere architecture.

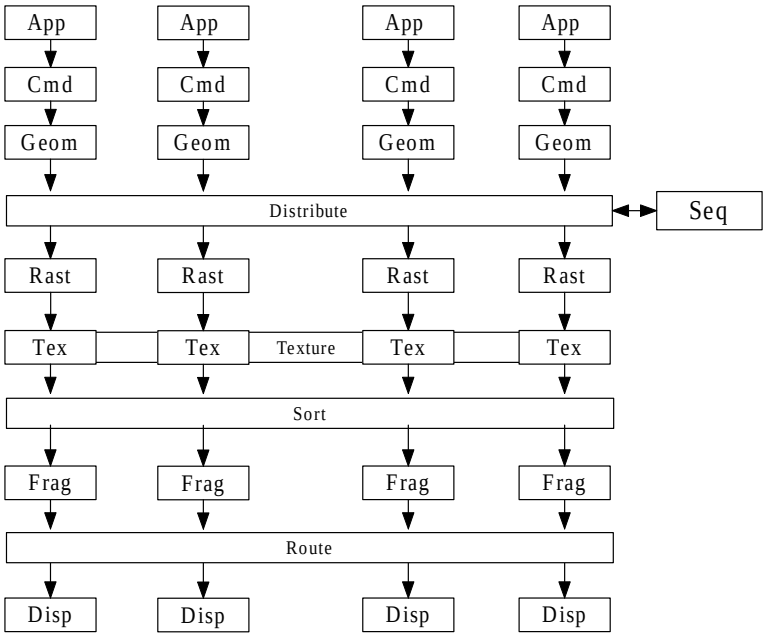


Figure 2.11: Pomegranate Architecture [18]

2.5.2 Software Parallel Rendering Systems

With the bandwidth of networks improving over the past ten years, research on software distributed parallel interactive rendering systems has been carried out. Most of the research is based on sort-first and sort-last architectures.

Ortega et. al. [62] implemented a sort-last algorithm on a MIMD system. In this system, three sets of virtual processors are needed. In the first set of virtual processors, each one is assigned one polygon. The processors process the polygons with transformation, lighting and scan conversion. The polygons is converted into spans. The spans are distributed to a second set of virtual processors, where the spans are clipped to the window and then are interpolated to calculate the pixels. Finally, the third set of virtual processors is in charge of composing, one processor for each screen pixel. All the other processors send all the samples that overlap this pixel to it. The

algorithm requires two communication steps, one for the transition from polygons to spans, and another for the transition from spans to pixels. The basic algorithm works well when the polygons are nearly the same size, displaying nearly a million triangles per second, but its performance drops by a factor of ten when some larger polygons are included.

WireGL [35, 36, 38] is the first sort-first cluster rendering system. It includes an efficient network protocol, a geometry bucketing scheme, and an OpenGL state tracking algorithm. It supports heterogeneous hardware platforms. WireGL divides the computational nodes into clients and rendering servers. WireGL replaces the OpenGL driver on the client machines, intercepts the OpenGL calls and sends the calls over a high-speed network to servers which render the geometry. WireGL classifies each OpenGL call into one of three categories: geometry, state, or special. Special commands, such as SwapBuffers, glFinish, and glClear, require individual treatment. Geometry commands are packed immediately into a global "geometry buffer". State commands are those that directly affect the graphics state, such as glRotatef, glBlendFunc, or glTexImage2D. The effects of state commands are recorded into a graphics context data structure. Each element of state has n "dirty" bits associated with it, where n is the number of rendering servers [5]. When the application executes a state command, all bits are set to 1, indicating that each server might need a new copy of that element and update the OpenGL state and the data are then copied into the local cache. Finally, the element is added to a list for processing during lazy update. Each rendering server receives OpenGL commands from remote clients and renders a portion of the screen space. Figure 2.12 shows the architecture of WireGL.

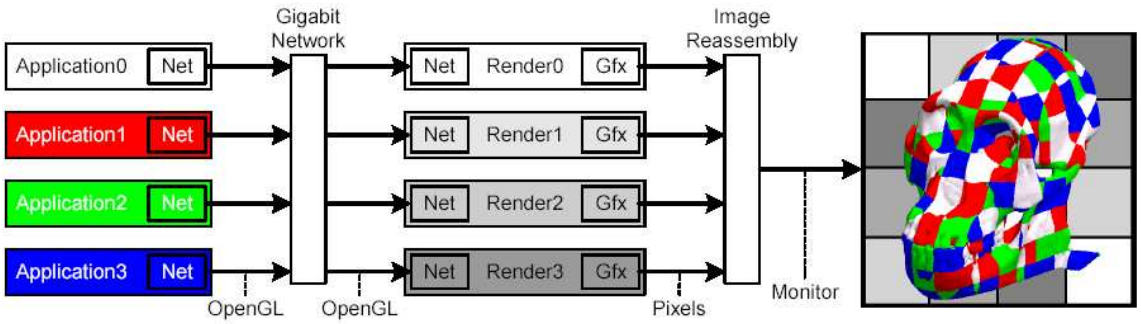


Figure 2.12: WireGL Architecture [36]

WireGL has good performance with 32 computational nodes (16 clients and 16 servers). However, because it uses sort-first architecture, it is difficult to scale to large numbers of rendering servers. Another limitation is that it needs high-speed connection. This system is supported by Myrinet. Therefore, distributed environments with plenty of computational resources and poor bandwidth connections are not promising for WireGL.

Princeton University has developed a multi-projector rendering system [13, 46, 66, 67, 68]. It adopts sort-first architecture. This system includes one client and eight rendering servers. The client partitions the geometry data regarding the bounding box of primitives and assigns them to the right rendering server. In order to achieve better load-balancing and cut down communication costs, Samanta et. al developed the sort-first and sort-last hybrid algorithm [67] mentioned in Section 2.4.4. They also investigated a data distribution strategy named k-way replication [66]. The key idea is to replicate every primitive in a scene k times among n rendering servers (where $k << n$). The k-way replication approach avoids full replication of the scene data on each rendering server. But, it can employ view-dependent load balancing algorithms (the sort-first and sort-last hybrid algorithm), since every primitive is available on more

than one processor. With k-way replication, they expect to achieve performance similar to the n-way replication, but with storage costs closer to the 1-way replication 3D model. This architecture is difficult to scale to a large number of rendering servers. Although it avoids redundant rendering, it has to perform pixel redistribution. When the number of partitions is very large, the pixel redistribution costs will impact the system performance greatly. Therefore, it is only suitable for a cluster environment with high bandwidth connection and a moderate number of processors. It is not suitable for large scale distributed environments.

Mesa is a open-source software implementation of OpenGL without using hardware accelerator at all. PMesa [49, 50, 51, 52] is a parallel version of Mesa. PMesa is built with sort-last architecture. The geometry data are arbitrarily assigned to rendering processors. Mitra et. al developed a general parallel compositing algorithm [49] which is integrated with both binary swapping compositiion and parallel pipeline composition. According to the number of composition processors, it divids the processors into serval groups. Inside the group, it performs parallel pipeline composition. Between groups, it runs binary sweeping composition. In this way, it not only keeps all processors at high utilization efficiency, but also cuts down the steps of composition to optimize the performance.

AnyGL [77] is a software implementation of sort-anywhere architecture. In this architecture, the system consists of four types of logical nodes, geometry distributing node, geometry rendering node, image composition node and display node. The first two types of logical nodes are combined as a sort-first graphics architecture while the others compose images. On the other hand, Yang et. al also developed a new state

tracking method based on logical time stamp for large-scale distributed graphics systems. Besides, three classes of compression are employed to reduce the requirement of network bandwidth, including command code compression, geometry compression and image compression. Because of the sort-anywhere architecture, processors have to communicate between each other in each rendering stage. Therefore, it requires high network bandwidth.

Others such as Aura [72], PGL[17] have achieved a set of valuable results in parallel rendering. Because their contributions are beyond our focus, they will not be introduced in detail.

2.6 Summary

This chapter takes a look at the space of application areas and sees 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 in a distributed computing environment. The fully-parallelized graphics architectures can be categorized into sort-first, sort-middle, and sort-last. Sort-first architectures have difficulty to maintain load-balance. They also introduce large amounts of redundant rendering. Sort-last architectures cannot satisfy the demand to generate very high resolution output images because they usually require a high bandwidth inter-connection. In sort-middle architectures, primitive communication appears to be a limiting factor. Commodity graphics accelerators do not generally provide high-performance access to the results of geometry processing through standard APIs. From the analysis, it is found that none of homogeneous architectures is suitable for the distributed interactive rendering applications with massive data sets

and large resolution output.

Our research on this problem leads to a hybrid architecture named sort-both which involves both sort-first and sort-last strategies. A typical characteristic of distributed computing environments is the large gap between the fast connection inside a cluster and slow connection between clusters. An application designed for uniform speed interconnects can lead to performance degradation. Sort-both architecture makes full use of this characteristic by hierarchically employing sort-first and sort-last at different levels.

Chapter 3

Overview of Sort-both Approach

Each architecture has its own set of advantages and disadvantages. We have outlined some of these above; for additional comparisons, one may refer to [53]. In this chapter, each architecture will be considered with regards to how well it can handle the demands for large primitive databases with high-resolution output in a distributed computing environment. Sort-both, a heterogenous architecture, will be presented and analyzed. Before analysis of the application suitability for each architecture, some basic issues should be introduced.

3.1 Basic Parallel Rendering Issues

- Task Decomposition

  There are two main strategies for task decomposition. In an object-parallel approach, tasks are formed by partitioning either the geometric description of the scene or the associated object space. Rendering operations are then applied in parallel to subsets of the geometric data, producing pixel values which must then be integrated into a final image. In contrast, image-parallel approaches reverse

this mapping. Tasks are formed by partitioning the image space, and each task renders the geometric primitives which contribute to the pixels which it has been assigned.

- Task Granularity

  Task granularity is considered the size of the basic workload unit. This workload unit may be defined to be an entire task, or it may be some smaller quantum, such as the number of instructions executed between communication events. Data decomposition is directly related with the task granularity. In rendering, for image parallelism, task granularity is the number of pixels allocated to a processor; for object parallelism, it is the number of primitives assigned to a processor. A computation is fine-grained if workload units are small, or coarse-grained if they involve substantial processing. A fine-grained decomposition includes one or a few data items in each partition, whereas a coarse-grained decomposition uses larger blocks of data. In a rendering context, a fine-grained task might compute the value of a single pixel, while a coarse-grained task might compute an entire frame in an animation sequence. Granularity often has a direct bearing on the efficiency of a parallel computation. Fine-grained computations generally incur more overhead for task scheduling and communication, but offer the possibility of more precise load balancing. Coarse-grained computations tend to minimize communication and scheduling overheads, but they are more susceptible to load imbalances and impose tighter limits on the amount of available parallelism.

- Load Balancing

  Load balancing is one of the most important issues in the design of parallel algo-

rithms. To achieve maximum performance, processors need to have roughly the same amount of work to do. 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 [58]. If primitives are evenly distributed among processors and they map evenly into the image space, then the rendering process is almost perfectly parallel. Unfortunately, real scenes are not this well-behaved. In many applications, the area of interest may be localized within the data space, and depending on the data partitioning scheme in use, this may correspond to a subset of the processors. In this case the objects to be rendered will not be evenly distributed. If the load imbalance is severe enough, it may be worthwhile to redistribute objects, either before or during the rendering process.

Usually load-balancing efficiency is measured 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. In rendering, the work load of a given processor is typically expressed as the number of primitives each processor must render.

- Scalability

  We consider not only how well a particular implementation of an architecture works, but also how that architecture's performance scales with increasing parallelism. In this dissertation, the research on scalability of distributed rendering system is divided into two parts: system size scalability and problem size scalability. System size scalability means the system performance scales with increasing

the number of the computational resources. In many fields, parallelism is employed not to solve the same problem faster, but instead to solve a bigger problem at the same speed [18]. Problem size scalability means that the time complexity and space complexity scale with increasing number of primitives. Communication costs and load imbalance both contribute negatively to scalability, and communication cost is more serious in distributed systems, since the high costs of data transmission also contribute to overheads in dynamic load balancing schemes.

3.2 Choosing a Parallel Strategy

Molnar et al. [53] has classified parallel rendering strategies as sort-first, sort-middle and sort-last. This classification is based on where the sort takes place in the translation from object coordinates to screen coordinates.

3.2.1 Sort-first

In sort-first algorithms, the image space is partitioned into non-overlapping 2D regions (i.e., tiles) and each processor does just enough geometry processing to determine the region of the raster image that a primitive will belong. The primitive is then sent to the appropriate processor to perform both the geometry processing and rasterization for that region of the raster image.

Sort-first is a promising architecture because of its low communication requirements. sort-first only needs to send finished pixels from the rasterizers to the frame buffers. This suggests that in sort-first systems, pixel communications issues are not a critical problem as the sort-middle and sort-last systems for very high-resolution output applications.

Another advantage is that the processors implement an entire rendering pipeline for a portion of the screen. These strong points benefit the system performance in distributed environments such as clusters and computational Grids. However, it has some disadvantages. It is susceptible to load imbalance because of the random distribution of the primitives in the image space. Another significant disadvantage is the redundant rendering work, which can be characterized by the overlap factor, the ratio of the total rendering work performed over the ideal rendering work required without redundancy. Since overlap factors grow with increasing numbers of processors, the scalability of sort-first systems is limited.

3.2.2 Sort-middle

Sort-middle algorithms are those that perform sorting and data redistribution at the obvious place - between geometry processing and rasterization. In a sort-middle algorithm, each geometry processing node computes screen coordinates for each primitive that has been assigned to it. It then determines the rasterizer to which it will send a primitive's scan line information, based on the raster image's partition among nodes.

Like sort-first, sort-middle algorithms are also susceptible to load imbalance of rasterizing processors due to non-uniformly distributed primitives. 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 limits the number of processors which can be added to a sort-middle system. This limitation is reached when the primitive communications network is saturated. Table 3.1 shows communication cost for one triangle.

Data Specification	Precision (bits)	Total(bits)
triangle vertices(x, y, z)	32	288
triangle normals (x, y, z)	32	288
vertex colors (r, g, b, a)	8	96
texture coordinates (u, v)	32	192
total		864

Table 3.1: Communication Cost for Each Triangle in Sort-middle Architecture

According to the triangle data size, the amount of bandwidth that is necessary to re-distribute a given number of on-screen primitives at interactive rates (30 frames/second) can be estimated. This is shown in Table 3.2.

Number of triangles	Bandwidth requirements(30 frame/second)
10 K	240 Mbits/second
500 K	12 Gbits/second
1 M	24 Gbits/second
5 M	120 Gbits/second
10 M	240 Gbits/second

Table 3.2: Bandwidth Requirements of Sort-middle

From this table, it can be known that sort-middle is best suited for tightly-coupled systems that use a fast, global interconnection to send primitives between geometry and rasterization processors, such as SGI Infinite-Reality [55]. For a distributed environment with slow interconnection, sort-middle is difficult to be employed.

Commodity graphics accelerators do not generally provide high-performance access to the results of geometry processing through standard APIs. Therefore, in a distributed environment, it is difficult to exchange fragments between different graphics accelerators.

3.2.3 Sort-last

Sort-last architectures defer sorting until the last stage. Primitives are arbitrarily distributed to available processors. Each processor then performs both the geometry processing and rasterization of its assigned primitives, regardless of where they fall within the raster image. The final stage then composes (sorts and merges) the raster images from all rendering processors to form the final image with depth information. The advantage of the sort-last algorithms is that, it is less prone to load imbalance and is easily scalable. The main disadvantage is that it usually requires an image composition network with very high bandwidth and processing capabilities to support transmission and composition of overlaps [53]. Table 4.1 shows limitations imposed by sort-last pixel communications. 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 composed. ("AA" is the anti-aliasing factor.)

Resolution	AA	Bandwidth requirements (30 frame/second)
512×512	16	7.5 Gbits/second
640×480	16	9 Gbits/second
800×600	16	13.7 Gbits/second
1024×768	16	22.5 Gbits/second
1280×1024	16	37.5 Gbits/second

Table 3.3: Sort-last Pixel Communication Requirements

It is obvious that the communication costs for sort-last architecture is far beyond the capability of the network bandwidth in a distributed rendering environment. Therefore, a homogenous sort-last architecture is not suitable for distributed rendering systems.

The above analysis shows that none of the three homogeneous strategies is par-

ticularly suited for rendering in a distributed environment. Sort-middle strategies are not suitable for distributed environments, because with standard APIs, it is difficult to access the results of geometry processing. Usually, the performance is too poor to responds in real-time. Another problem is that typical network bandwidth is too small to distribute every primitive during every frame. Therefore, sort-middle architectures are not adopted in this prototype system. Sort-first architectures perform well with a small number of partitions. Experiments show that the rendering efficiency drops to 50 percent when the number of partitions increases to 16 [67]. Sort-last strategies perform well with a low display resolution. When rendering at high display resolution, sort-last algorithms require very high network bandwidth and is not feasible with networked systems.

3.3 Sort-both Architecture

From the analysis above, it is shown that none of the homogeneous architectures is particularly suited for all conditions. In a distributed computing environment, the situation is much more complex. In most cases, it can be assumed that the bandwidth of inter-cluster connections is relatively lower than the bandwidth inside a cluster. Considering these limitations, we design a hierarchical architecture, named sort-both, involving both sort-first and sort-last architectures for distributed environments.

3.3.1 Sort-both Approach

For each frame of an interactive visualization session, the system proceeds in a three phase pipeline, as shown in Figure 3.1.

Figure 3.1: Sort-both Strategy

- Phase 1:

  The system executes a novel load balancing sort-first algorithm named Dynamic Pixel Bucket Partition (DPBP). According to the computational resource capability available in each cluster, it simultaneously decomposes the 3D primitives into K groups, assigning each group to a different rendering cluster, and partitions the pixels of the 2D screen into non-overlapping tiles, also assigning each tile to a different rendering cluster.

- Phase 2:

  The head-node of each cluster runs a sort-last algorithm to decompose the 3D primitives into N groups and distributes them arbitrarily to the available N rendering nodes. Each rendering node performs both the geometry processing and rasterization of assigned primitives. Finally, it composes the raster image tiles from all rendering nodes inside the cluster to form the final sub-image. Then, the

head node sends the complete sub-image to the display node.

- Phase 3:

  In the third phase, the display node receives all the sub-images from all of the rendering clusters, and assembles them (without depth comparisons) to form a final complete image in its frame buffer for display.

This system architecture has two important advantages. At the inter-cluster level, it uses a sort-first approach to achieve low communication cost. As with most sort-first algorithms, the number of partitions will impact scalability. In this way, because the partition number is small, the redundancy introduced by the sort-first one is relatively small. At the cluster level, a sort-last approach achieves good load-balancing and scalability. Since each image tile is a low-resolution sub-image, it is helpful to reduce the cost of pixel redistribution.

However, the limitations of sort-first and sort-last bring some problems to the sort-both architecture. Sort-first strategy is prone to load-imbalance. As mentioned in Chapter 2, static methods are difficult to guarantee load-balancing among rendering clusters. They also introduce large amounts of redundant rendering. Therefore, we develop a novel adaptive sort-first algorithm to maintain the load-balance. Another problem is from the large overhead of pixel redistribution for image composition in sort-last approaches. In sort-both architecture, the sort-fist partitioning is helpful to reduce the cost for pixel redistribution. However, when the original resolution is very large, pixel redistribution is still a big problem. Therefore, we develop a image composition scheme without depth information. In this way the pixel redistribution cost can be reduced farther.

3.3.2 Sort-first Algorithm

In this system, the DPBP algorithm is proposed to partition the screen on the inter-cluster level. The basic idea of DPBP is to dynamically cluster primitives into groups based on the overlaps of their projected bounding volumes in screen space. In this algorithm, firstly, it sorts the bounding boxes into pixel buckets, which are weighted with the number of primitives in it. It starts with the entire screen as a single tile and recursively divide the tile with axial lines into two sub-tiles each with the same number of primitives in them. The algorithm terminates when the number of sub-tiles is equal to the number of rendering clusters. Each sub-tile is then assigned to a cluster, which renders a sub-image containing all primitives in the bounding box that at least partially overlaps the extent of its tile. The clusters then read the resulting pixels back and send them to the image assembling machine so that they can be assembled into a complete image for display. It is easy to balance the workload among the rendering clusters in this way. This algorithm is quite efficient. Similar with most sort-first algorithms, the main overhead of DPBP is redundant rendering of objects overlapping multiple tiles. The DPBP algorithm includes three steps: preprocessing, sorting 2D bounding boxes into pixel buckets and scanning pixel buckets recursively.

3.3.3 Image Composition

In sort-last algorithm, according to the rendering capability available in each rendering node, the cluster head node evenly partition the workload assigned to the cluster again. A peer-to-peer sort-last scheme is implemented to compose images at the cluster level. In a traditional way, each rendering node renders the portion of primitives and generates

a full size sub-image. A composing algorithm inputs two images and iterates through pixel pairs (two pixels with corresponding locations in the input images). These pixels are read and their depth value compared so that the pixel closer to the viewer is drawn to the result image.

In order to take full advantage of computational resources, we also parallelize the image composition. Each node is in charge of a set of scan lines. After rendering is completed, each rendering node reads the pixels from its color buffer and its Z-buffer and sends them to the node which the scan lines have been assigned to be composed. Upon receiving pixels, the nodes compose them into their frame buffers. Finally, after all the sub-images have been fully composed, the clusters read the resulting pixels back and send them to display node. The running time of the sort-last composing algorithm is bounded by the size of the sub-image. Therefore, more rendering clusters lead to lower pixel redistribution cost in our sort-both architecture.

In our prototype system, we have developed novel sort-last image composition strategy, named Image Layer Composition with Alpha (ILCA) , which use alpha value instead of depth value. This saves significant time for accessing and transferring the depth-buffer. It will be introduced in Chapter 5.

3.4 Summary

In this chapter, we analyze three homogeneous architectures. It shows that none of them are fully suitable for distributed environments. Therefore, a novel hybrid architecture involving both sort-first and sort-last approaches is presented. This architecture, named sort-both, takes advantage of the typical characteristic of distributed computing envi-

ronments, a large gap between the fast connection inside a cluster and slow connection between clusters. In most cases, implemented with same algorithms, sort-both architecture is always superior to homogeneous architectures. However, in order to achieve better performance, we developed an efficient sort-first algorithm and a sort-last image composition method with less bandwidth requirement. The following chapter will introduce the sort-first algorithm in detail.

Chapter 4

Load-balancing Sort-first Algorithm

In sort-first approaches, the image space is partitioned into non-overlapping 2D regions and partitioning processor does just enough geometry processing to determine the region of the raster image that a primitive will belong. The primitive is then sent to the appropriate processor to perform both the geometry processing and rasterization for that region of the raster image. Figure 4.1 illustrates the sort-first approach.

Figure 4.1: Sort-first Approach

In sort-first algorithms, the choice of strategy for mapping screen regions to proces-

sors has a critical impact on the performance. A simple static 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 idle processors wait for the overloaded one to complete its job.

4.1 Basic Issues for Sort-first

In order to achieve high parallel efficiency, the work performed by the hardware must be balanced across the processors. Most load-balancing methodologies will introduce tradeoffs between the degree of load-balancing achieved and the amount of various overheads incurred. 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. The overheads include the cost of running the load-balancing algorithm and the overhead introduced by the load-balancing method.

4.1.1 Cost Estimation

The rendering cost of a region can be divided into geometry processing cost and the rasterization cost. The geometry processing cost directly depends on the amount of the primitives involved in the region. The rasterization cost is related to the number of the pixels involved in the region. We can characterize the rendering cost by the following equation:

$$C_{total} = N_{prim} \times C_{prim} + N_{pixel} \times C_{pixel} \tag{4.1.1}$$

C_{total} is the total cost of rendering a region. N_{prim} is the number of the primitive that involved in this region and C_{prim} is the cost of geometry processing per unit. N_{pixel} is the number of the pixels within the current region and C_{pixel} is the average rasterization cost per pixel. When we partition the screen, what we really care about is not the exact cost value, but the cost balance among the regions. Therefore, we can simplify the equation as:

$$\frac{C_{total}}{C_{prim}} = N_{prim} + N_{pixel}\frac{C_{pixel}}{C_{prim}} \tag{4.1.2}$$

E.g. in order to keep load balance, when we partition the screen, we must keep the value of $N_{prim} + N_{pixel} \times K$ is equal among the regions where $K = C_{pixel}/C_{prim}$. The K value varies from the rendering methods implemented.

One assumption can be made in cases where we have a highly pixel-parallel hardware rasterizer, the time elapsing in rasterization procedure is very small and no significant impact to the performance. Thus the rasterization cost is ignored and the only rendering cost is the geometry processing cost. That means when two regions involve same number of primitives, it can be assumed that they have the same resources cost. However, for computing systems without graphics hardware support, the rasterization cost should be considered as an important portion of the cost.

4.1.2 Overhead

For sort-first systems, the direct overhead of a load-balancing algorithm is entirely dependent upon the actual load-balancing algorithm chosen [58]. It is the cost of running the algorithm. Static algorithms may have no direct overhead. The adaptive algorithms have to dynamically compute the primitives distribution and partition the primitives.

The main overhead introduced by sort-first is the the extra work that results due to screen subdivision. Screen subdivision leads to redundant rendering which 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. This efficiency is characterized by the overlap factor.

4.1.3 Overlap Factor

Sort-first architectures will introduce redundant rendering since the primitives that cross region boundaries must be processed in multiple regions. This resource of inefficiency is expressed in terms of overlap factor, O. The overlap factor represents the number of regions overlapped by a typical primitive. John Eyles [53] first provided an equation to estimate the overlap factor. As shown in Figure 6.7, we assume the size of a screen region is $W \times H$ and the size of the 2D bounding box of a typical primitive is $w \times h$.

Obviously, the center of the bounding box can fall anywhere with respect to a region. It can be assumed that primitives have an equal probability of falling anywhere within a region. The probability of falling in a corner(the black area) is:

$$\frac{4(w/2)(h/2)}{W \times H} \tag{4.1.3}$$

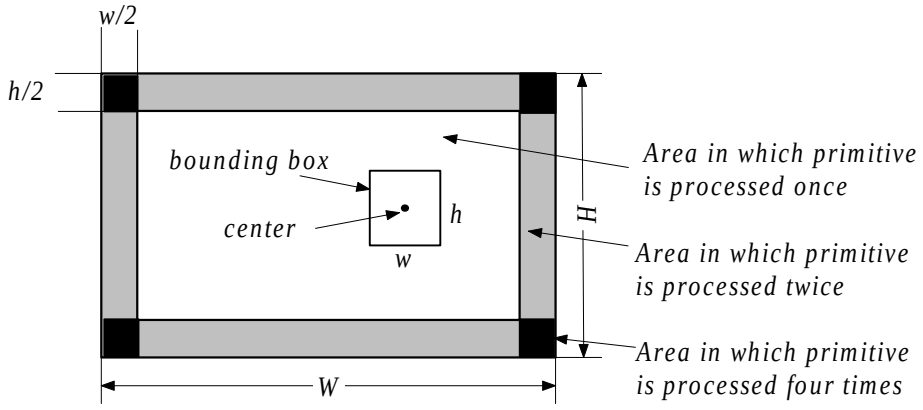


Figure 4.2: Overlap factor [53]

The probability of falling in a edge (the grey area) is :

$$\frac{2(w/2)(H-h)+2(h/2)(W-w)}{W \times H} \tag{4.1.4}$$

The probability of falling in the center area (the white area) is:

$$\frac{(W-w)(H-h)}{W \times H} \tag{4.1.5}$$

The overlap factor can be resulted by summing these probabilities weighted by the resulting overlap of a primitive falling in each of these regions.

$$O = (\frac{W+w}{W})(\frac{H+h}{H}) \tag{4.1.6}$$

4.2 Dynamic Pixel Bucket Partition (DPBP) Algorithm

4.2.1 Adaptive Sort-first Algorithm

Adaptive sort-first algorithms can be classified into two basic approaches. One approach is to fix the number of regions equal to the number of processors, but vary the shape

of each. Another statically partitions the screen into regions, but dynamically allocate them to processors. Various combinations of these 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) and minimum overlap factor.

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. The ideal shape is closer to square, meaning that both horizontal and vertical divisions are necessary [58]. Forcing the divisions to go all the ways across the screen is not necessary. 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.

4.2.2 New Adaptive Method: DPBP

The basic idea of DPBP is to dynamically cluster primitives (e.g. polygon or surface patches) into groups based on the overlaps of their projected bounding volumes in screen space. In this way, it is easy to balance the workload among the rendering nodes. At the start of each frame, a screen-space, axis-aligned bounding box (AABB) is computed for each group of primitives on the screen. These bounding boxes are sorted into pixel buckets according to their distribution in the screen space. Each single partition step, it splits the current region along the longer axis. Without loss of generality, it is assumed

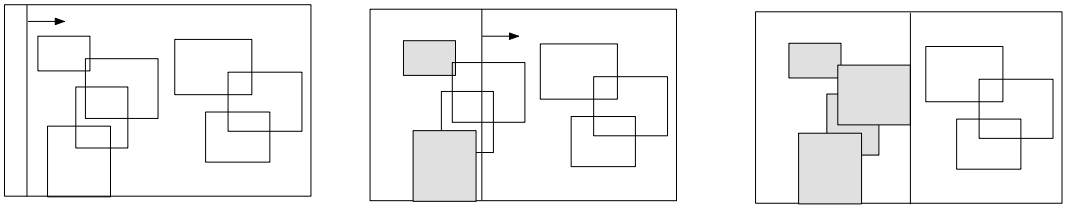


Figure 4.3: Example Execution of DPBP

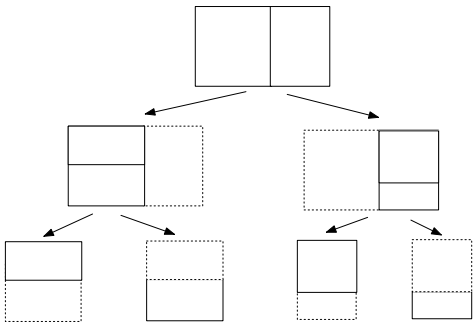


Figure 4.4: Example of Recursive Partitioning

that the width is larger than the height of the region. For each subdivision of the X dimension (subdividing the Y dimension is similar), the algorithm then moves a vertical partition line from left to right. At the same time, it estimates the rendering cost of the left and right half-spaces. The partition line moves pixel by pixel until it finds the location at which the rendering cost of the left region is equal to rendering cost of the right one. Figure 4.3 shows an example of execution of this step.The primitives are also partitioned into two groups. The screen partitioning continues recursively and alternately in the X, Y dimension, until exactly N sub-regions and N groups of primitives are formed, and one is assigned to each of the rendering processor. Figure 4.4 shows the an example of recursive partitioning into four sub-regions

4.2.3 Primitive Grouping & Bounding Volumes

In a massive geometry database, the number of primitives is usually much bigger than the number of rendering nodes. The primitives with large data size will be divided into a number of small primitives. It makes all the primitives have similar data size. In order to reduce the extra transformation burden in the partitioning node, as in [58, 66], primitive grouping method is introduced. In this way, multiple primitives are grouped together into a group. The algorithm keeps track of the simplified bounding volume around each group, rather than individual primitive. The advantage of this method is that, workload of running DPBP algorithm is reduced to only a fraction of the total number of primitives. Primitive sorting is also speed up. The master processor running DPBP examines which screen regions are overlapped by the bounding volume of the group and send that group of primitives to the appropriate rendering processors. The rendering processors transform the primitives within the group and render them as necessary. In addition, primitive grouping aslo reduces the memory bandwidth requirement of the partition node. Both computation and storage costs for managing the groups are amortized over the primitives in the group. However, primitive grouping also introduces some new questions of its own.

- What overhead results from using groups?

The tradeoff is the expense of the overhead from the group overlap factor. The estimation of the distribution will not be as accurate as using information about the individual primitives themselves. The higher the group-overlap factor, the more regions will be overlapped by each group. For example, a bounding box of a group overlaps multiple regions. For a particular region, it may be overlapped by the bounding box

of the group, but none of the primitives within the group actually overlap that region. However, the associated processor must transform and cull each of the primitives within the group to find this out.

- How should the primitives be grouped together?

The methods of grouping primitives can be classified into "unsorted", "partially sorted" and "sorted" strategies [58]. The unsorted strategy just arbitrarily takes every N primitives and places them into a group, where N is the selected group size. The partially sorted strategy firstly inserts the primitives from the input file into an octree data structure, then recursively traverse the octree cells to put every N primitives into a new group. The sorted strategy reads in the primitives and forms the groups by performing a 3D spatial subdivision. The algorithm subdivides the bounding box of the primitives repeatedly until the proper number of groups is achieved. Obviously, the sorted method resulted in the smallest average bounding box volume. Larger bounding volumes will lead to larger overlap factor. Because of these advantages, the sorted method was chosen. As the sorting is only done at database load time, this is a reasonable choice for a model with static structures. It can be processed off line. These methods have been discussed in [58] in detail.

- What kind of bounding volume should be adopted?

We must choose a bounding volume to use around each primitive group. There are various choices available. We list some samples in the order of increasing complexity and tightness of fit. They are spheres, axis-aligned bounding boxes (AABB), oriented bounding boxes (OBB), and multiple-slab intersection [4, 31]. Obviously, tightly fitted volumes can effectively decrease the group overlap factor. That means it is helpful to

decease the number of primitives that each processor must deal with as well as the amount of primitive communication. Since bounding volume tests must be executed for every primitive group for each frame, it is also important for the tests to be as simple and as fast as possible. In the model space, tightly fitted bounding volume is useful to reduce the overhead. The 3D bounding volumes can be calculated only one time for each session. However, in screen space, when DPBP partitioning the screen, the partition line aligns the X or Y Axis. Real-time performance is needed when calculating the 2D bounding box in the screen space. Therefore, we have chosen oriented bounding boxes in model space and axis-aligned bounding box in screen space. When DPBP is running, the oriented bounding boxes are projected to screen space. According to the coordinates of the eight vertexes in screen space, an axis-aligned bounding box can be obtained.

- How many primitives do we assigned in each group?

Assigning more primitives into each group would result in fewer overall groups to deal with, and having fewer groups would speed up the task decomposition process. On the other hand, assigning more primitives into a group would also likely increase the bounding volume and therefore the group overlap factor. The size of the group overlap factor is highly dependent on the grouping algorithm as well as on the ratio of the average group size in screen space to the average region size. Therefore, the number of primitives in each group is an influencing factor for the system performance. Mueller looked into this problem and further exploration are presented in his dissertation [58].

The hierarchical data structure shows its advantage in this situation. It is to implement a hierarchical database. Each big primitive group will be partitioned into several

smaller primitive groups. First, a big bounding box is used to evaluate the distribution of primitives. When the partition line crosses a big bounding box, it will be replaced by a set of smaller bounding boxes. Then the smaller primitive groups are assigned to the correct region respectively. In this way, the group overlap factor can be reduced.

Additional information on these topics can be found, respectively, in [58, 23].

4.2.4 Algorithm Description

We describe our Dynamic Pixel Bucket Partition (DPBP) algorithm in detail in this section. The DPBP algorithm involves three steps: Preprocessing, Sorting 2D Bounding Boxes, and Scanning Pixel Buckets.

- Preprocessing

  Initially, the geometry data should be processed with primitives grouping. To estimate the approximate distribution of primitive groups on-screen, a 3D bounding box marked with a weight value is used to approximate the primitives within it. The weight of a bounding volume is determined by the number of primitives included in it. This step is executed only once in the whole visualization and is illustrated in Figure 4.5.

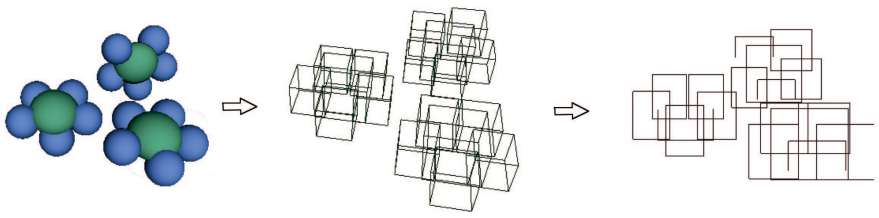


Figure 4.5: Geometry Processing for DPBP

- Sorting 2D Bounding Box

  In this step, the 3D bounding boxes are projected to generate the axis-aligned 2D bounding boxes. Then, according to the resolution of the frame, two arrays of pixel buckets are created. For example, if the frame size is 1024×768, the horizontal array includes 1024 buckets and the vertical array includes 768 buckets. Finally, the 2D bounding box is projected to both X-axis and Y-axis to generate two intervals. For example, if a 2D bounding box stretches horizontally from pixel number 20 to 135, then an interval [20,135] is generated and the left vertex is sorted into XBucket[20] and right vertex into XBucket[135]. If the whole 2D bounding box is out of the frame, it will be deleted. If one vertex of the interval is inside the frame and the other is out of the frame, the part of the interval outside the frame will be clipped. For instance, if a 2D bounding box overlaps the screen horizontally from pixel position -120 to 30, it is clipped and left vertex is sorted into X-Bucket[0] and the right vertex into X-Bucket[30]. This step is executed once for each frame. Figure 4.5 shows the 3D and 2D bounding boxes. Figure 4.6 shows an example of this step.

- Scanning Pixel Buckets

  The current region is scanned along the longer axis. Without loss of generality, it is assumed that the width is larger than the height of the region. The problem can be described as follows: For a set of intervals $\{I_i\}$, $X\_start \leq Left(I_i)$ and $Right(I_i) \leq X\_end$, where X_start is the left boundary of the current region, X_end is the right boundary of the current region, and $Left(I_i)$ and $Right(I_i)$ are left and right point of the interval respectively. The program is to find x_0,

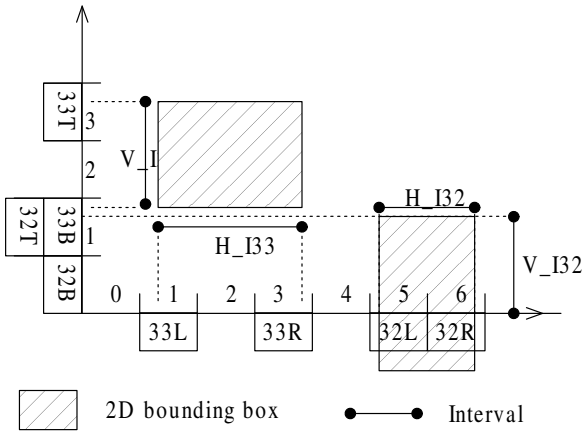


Figure 4.6: Interval Projection

$X\_start \leq x_0 \leq X\_end$, such that

$$\sum_{Left(I_l)<x_0} Weight(I_l) = \sum_{Right(I_r)\geq x_0} Weight(I_r) \tag{4.2.1}$$

During the scanning from $XBucket[X\_start]$ to $XBucket[X\_end]$, two lists to store the interval IDs lying in the left or right regions are maintained. Initially, the left list is empty and right list contains all the intervals, that is, $L \leftarrow \oslash$ and $R \leftarrow \{I_i\}$. Then, the pixel buckets are scanned from left to right. If a left point of interval is found in the bucket, the interval ID is added to left list. If a right point is found, we remove the corresponding ID from the right list. In this way the left list grows monotonically and right one shrinks monotonically and eventually a cross-over point is reached. At this point, a partition line that subdivides the current region is located and two lists of interval IDs are obtained.

This process is repeated until exactly n tiles (i.e., sub-regions) and primitive groups are formed, and each is assigned to one of the n rendering nodes. The

Partition algorithm is shown in Algorithm 1.

Algorithm 1 2D screen space partition algorithm

PARTITION($x_1,\ x_2,\ y_1, y_2,\ n$)

$XBucket$, $YBucket$: array stored numbers of primitives for X-axis and Y-axis
x_1, x_2,y_1, y_2: start and end point of X-axis and Y-axis of tile
n : number of partitions

```
BEGIN

IF (x2 − x1 > y2 − y1) THEN **** partitioning along X axis ****
    find x0, s.t.,
```

$$\frac{\sum_{x_1 \le i \le x_0} XBucket[i]}{\sum_{x_0 < i \le x_2} XBucket[i]} = \frac{\lfloor n/2 \rfloor}{\lceil n/2 \rceil}$$

```
    n1 ← ⌊n/2⌋
    n2 ← ⌈n/2⌉
    IF(n1 > 1) THEN
        call PARTITION(x1, x0, y1, y2, n1)
    ENDIF
    IF(n2 > 1) THEN
        call PARTITION(x0, x2, y1, y2, n2)
    ENDIF
ELSE
    ..... **** partitioning along Y axis****
ENDIF

END
```

4.2.5 Partition Rate Tree

In Algorithm 1, a split ratio of 1:1 is used in partitioning the screen space into two regions. In order to partition the screen into an arbitrary number of regions with arbitrary split ratios, an appropriate ratio should be determined for each split operation. This is achieved by constructing a minimum height binary tree with as many leaves as

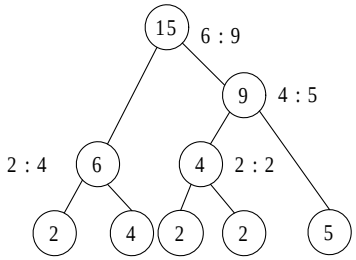


Figure 4.7: Partition Rate Tree

there are rendering nodes. The weights on the leaves are determined by the rendering performance of each rendering node. In the non-leaf nodes, the weight is calculated using the sum of the weights 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 child nodes corresponding to the next two splits, and so forth. The split ratio for any given split is determined by the ratio of the weights of the children of the corresponding node. Figure 4.7 shows a partition rate tree with five leaves.

4.2.6 Computational Complexity Analysis and Comparison

The preprocessing step is executed only once. It can be done off line, and therefore can be ignored. The procedure to sort m bounding boxes takes $O(m)$. In the third step, the recurrence is

$$T(m,n) = (T(k, \lfloor n/2 \rfloor) + T(m-k, \lceil n/2 \rceil)) + O(m) \tag{4.2.2}$$

$$T(m,n) \leq cm \lg n = O(m \lg n) \tag{4.2.3}$$

where $T(m,n)$ is the running time for Step 3; m is the number of 2D bounding boxes; n is the number of tiles. $0 \leq k \leq m-1$, because Step 3 produces two sub-problems with total size $m-1$. Therefore, the overall complexity of DPBP algorithm is $O(m+mlgn)$.

Compared with other sort-first algorithm, DPBP is more efficient. In MAHD, when a fine mesh is used to tally the primitives, for each cell covered by a given primitive's bounding box the system adds an amount proportional to the rendering cost for that primitive. The computation complexity for this step is $O(pm)$, where m is the number of bounding boxes and p is the average number of cells a bounding box overlaps. For the partitioning operation, the computation cost is $O(nm)$ [67]. Therefore, the total computation complexity of Mesh-based Adaptive Hierarchical Decomposition, or MAHD is $O(nm + pm)$ [58].

Similarly, compared with the hybrid algorithm mentioned in Section 2.4.4, the computation complexity of the hybrid algorithm is $O(mn + mlgm)$. The $mlgm$ term arises from the stage when the m bounding boxes are sorted according to their screen position, and the mn factor comes from the fact that $N-1$ cuts are made, while $O(m)$ bounding boxes are considered for each cut. Another advantage of DPBP in comparison with the hybrid algorithm is that there is no pixel redistribution among the rendering nodes.

4.3 Experiments and Results

In this section, some important factors about sort-first algorithms are examined. The purpose of these experiments is to answer the following questions:

1. How good is the load-balancing of the DPBP algorithm?
2. How does the method scale with respect to the number of processors?

4.3.1 Test-bed

The prototype system is built on the test-bed shown in Fig. 4.8 [1]. It contains several high-performance clusters and one visualization equipment. The task of sort-first task partitions runs in Parallel & Distributed Computing Center (PDCC). Tasks of Image Rendering run on all computing resources in the test-bed. The rendering resources are available in Asian Pacific Scientific & Technological Computing Center (APSTC), PDCC, BioInformation Research Center (BIRC), Nanyang Center of Supercomputing and Visualization (NCSV) and Institute of High Performance Computing (IHPC). Task of Image Composition runs in PDCC and end users can view final output on SMART Visualization Board located at PDCC.

Computing resources in each site include MPPs, NOWs and beowulf clusters. Processors inside NOWs are linked with ethernet while myrinet are used for Beowulf clusters. NOWs and Beowulf clusters can be reconfigured with user requirements. For example, Beowulf cluster in PDCC contains 20 processors. It can be reconfigured as 2 clusters with 10 processors each or 4 clusters with 5 processors each.

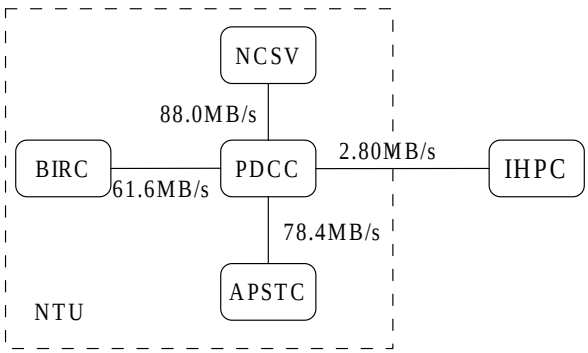


Figure 4.8: Topology of Test-bed

[1]The online information of the resources can be explored at: http://ntu-cg.ntu.edu.sg/ganglia/

Firstly, the effect of inter-cluster network bandwidth on performance is discussed. It is assumed that the frame in the application is displayed with resolution P pixels and b bits color. If the scene is rendered at least n frame per second, , the least requirement for network bandwidth is around $P \times b \times n$. Obviously, network between PDCC and IHPC is the bottleneck for high-resolution rendering at near real-time frame rates (10 frame/second). To reach real-time or near real-time requirement, experiments is carried on the test-bed excluding IHPC.

4.3.2 Load Balancing

This experiment is designed to examine the load-balancing of DPBP algorithm. The input objects for the test are the molecule models. The number of primitives ranges from 200,000 to 1,600,000. The following equation shows the performance metric used to evaluate whether workloads among various rendering processors are balanced.

$$workload\_imbalance = \frac{(\max\{(\frac{W_j}{DC_j}\} - \min\{\frac{W_j}{DC_j}\}) \times J}{\sum_{j=1}^{J} \frac{W_j}{DC_j}}$$

In this equation, J is the number of rendering processors (i.e. number of partitions). W_j is the workload assigned to processors R_j. DC_j is the computational capability of processors R_j.

The workload of each rendering processor is logged while rendering a sequence of frames in the test bed. We sampled the result one times every 10 frames. Figure 4.9 shows the results of experiments for a model which contains 600000, 800000, 1000000 and 1200000 primitives. The workload imbalance ranges from a minimum of only 0.4% to a maximum of 7.3%, and is never more than 10%.

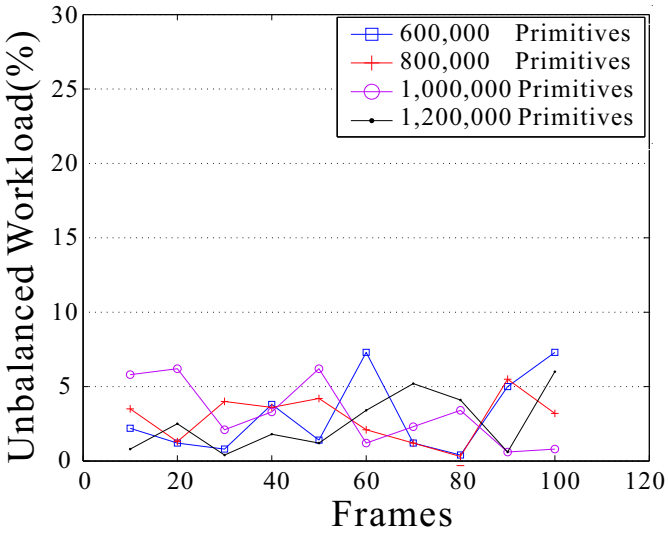


Figure 4.9: Load Balance

4.3.3 Efficiency

The second experiment for the DPBP algorithm is to investigate its efficiency. Efficiency of the DPBP algorithm is evaluated by the time cost when DPBP works on molecular models with different numbers of primitives. Figure 4.10 shows the time taken by DPBP algorithm with different number of primitives and different number of partitions. It also shows that with fixed number of partitions, time taken by DPBP algorithm increases linearly as the number of primitives increases. This matches well with the computational complexity analysis presented in Section 4.2.6.

4.3.4 Overlap Factor

The third experiment is to investigate the overheads introduced by the DPBP algorithm. Sort-first has a source of inefficiency: primitives that cross tile boundaries must be processed in multiple regions. In this experiment, the variation of the overlap factor is

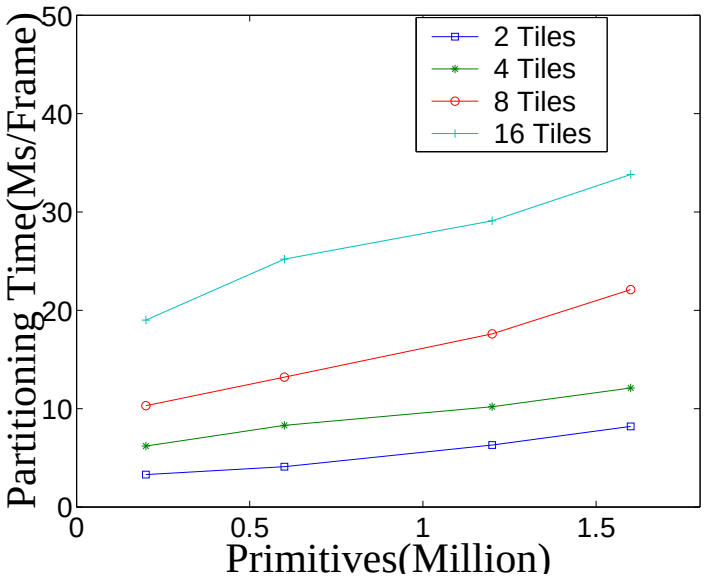


Figure 4.10: Efficiency

investigated by increasing the number of partitions. The primitives involved in each tile are summed. Then the result is divided by the number of the primitives inside the whole frame rendered. The following equation shows the real overlap factor in experiments:

$$overlap\_r = \frac{\sum_{i=1}^{J} W_j}{W'} \tag{4.3.1}$$

Where, J is the number of partitions , W_j is the number of primitives assigned to rendering processor R_j and W' is the number of total primitives (without redundancy) in the frame rendered.

The experimental results and the expected results are showed in the Figure 4.11. From these results, it can be found that the experimental results match the expected overlap factors closely. The overlap factor increases with the number of tiles.

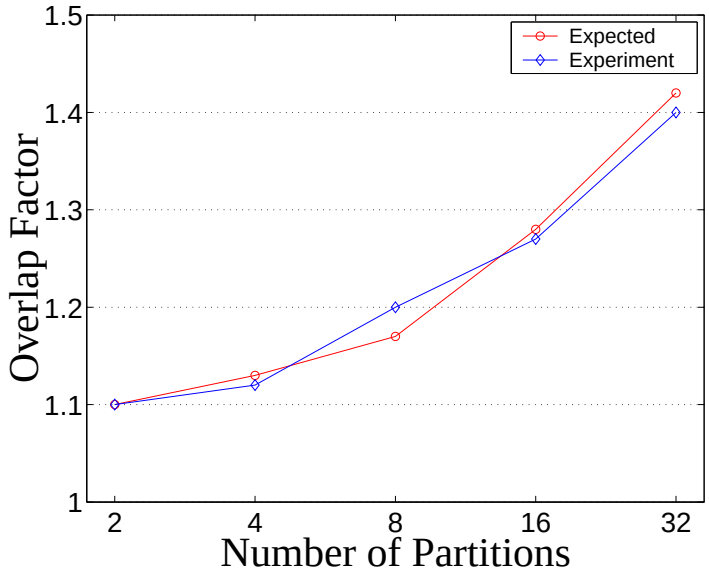


Figure 4.11: Overlap Factor

4.4 Conclusion

In this chapter, the important factors are discussed in the design and implementation of a sort-first graphics system. Load balancing and efficiency of sort-first algorithms have direct impact on the cost and the latency of the overall system. In sort-both architecture, sort-first will not introduce significant redundant rendering. DPBP, our novel sort-first algorithm is presented in detail. This algorithm recursively partitions the screen region and evenly partitions the primitives at the same time. This algorithm has promising applications for the distributed rendering systems.

Chapter 5

Sort-last Parallel Rendering with Alpha Composition

5.1 Basic Issues of Sort-last

Sort-last algorithms defer the data sort until the geometry processing and rasterization of all primitives are completed. The primitives are then evenly partitioned and each partition is assigned to a processor. After each processor finishes rendering its allocated primitives, the subimages created by the processors are merged into a final image. Figure 5.1 illustrates the sort-last approach.

Parallel rendering using sort-last yields a faster polygon per second rendering rate as data sets get larger. However, unlike the sort-first and sort-middle strategies, the performance of sort-last parallel rendering drops sharply as the resolution of the display increases [56]. That is because of the large communication cost of pixel redistribution.

Figure 5.1: Sort-last Approach

5.2 Image Layer Sort-last Strategy

In our novel sort-last strategy, data decomposition is based on the current viewpoint. The basic idea is to partition the scene primitives along the Z axis into P layers for a system with P rendering processors and assign each layer to a processor for rendering. To compose the final image, this strategy simply stacks these image layers one on top of another according to the visibility order of the layers (There is no Z-buffer comparison). The method includes following steps:

- According to the Z values of the geometry objects, partition the geometry objects into P groups.
- Each processor renders its assignment into a full size image.
- Merge the images from the rendering processors by drawing the images from back-to-front.

In this method, five essential problems should addressed.

1. A method for grouping primitives into aggregate objects
2. A method for estimating the rendering time of each object
3. Objects intersected by the front or back plane of multiple image layer volumes
4. An efficient on-line partitioning algorithm that can achieve good load balancing, taking into account both rendering and communication costs,
5. A method for merging images without depth value.

The first and second problems have been addressed in the Chapter 4. For objects overlapping two image layer volumes, they will be sent to both processors. When the processor rendering the overlapping object, the primitives which is outside the current image layer volume will be rejected. The primitives intersected by the font or back plane of the volume will be clipped. This problem will be discussed in the next section. The efficient online decomposition algorithm is similar to the DPBP algorithm. It is also a divide-and-conquer algorithm. However it only partitions the data along the Z axis. DPBP is partition in screen space(X, Y axis). In order to merge images without depth value, we draw the image layers from back to front. However, this does not solve all the problems, such as a hole in the object. Therefore, alpha channel is used to merge images, and will be discussed in Section 5.4.

5.3 Data Decomposition with Image Layer

The data decomposition algorithm for image layers is similar to DPBP. It dynamically cluster primitives into groups based on the overlaps of their projected bounding volumes in the Z axis. As in DPBP, to estimate the approximate distribution of primitives, a 3D bounding box marked with a weight value is used to approximate the number of primitives within it. The weight of a bounding volume is determined by the number of primitives included in it. The 3D bounding boxes are projected to the Z axis to generate the axis-aligned intervals. The vertices of the intervals are sorted by its Z coordinates. Then the sorted list of vertices is scanned from left to right and is partitioned into P parts. Similar to DPBP, some objects will overlap the partition lines. It means that these objects are intersected by the partition plane. They will be assigned to both groups. According to the Z coordinates of the partition lines, the view frustum will be partitioned into P image layer volumes. An image layer volume is a clip volume which is bounded by six clip planes. Figure 5.2 shows an example of partitioning a view frustum into three image layer volumes. All the image layer volume have the same left, right, top and bottom planes with the view frustum but different front (near) and back (far) planes.

A clip plane can be defined as $(p_1\ p_2\ p_3\ p_4)$, where p_i are the coefficients of a plane equation in object coordinates. The coefficients of the plane in view coordinates can be obtained by applying the inverse of the current model-view matrix to these coefficients. $(p'_1\ p'_2\ p'_3\ p'_4) = (p_1\ p_2\ p_3\ p_4)M^T$. All points with eye coordinates $(x_e\ y_e\ z_e\ w_e)^T$ that satisfy the following inequation lie in the half-space defined by the plane; points that do not satisfy this condition lie in another half-space.

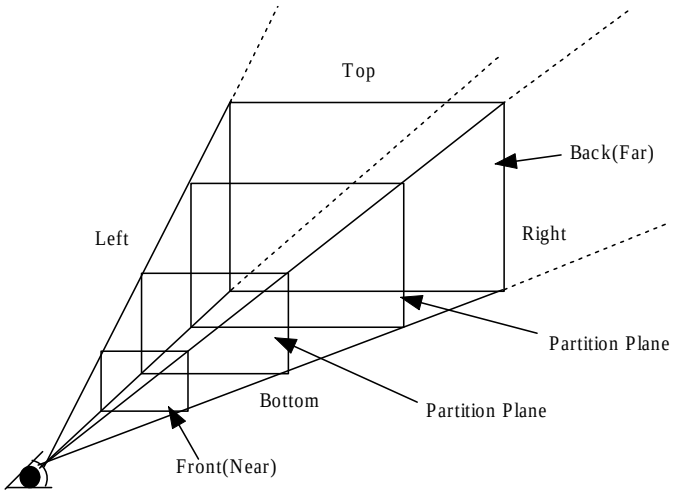


Figure 5.2: View Frustum and Image Layer Volume

$$\begin{pmatrix} p'_1 & p'_2 & p'_3 & p'_4 \end{pmatrix} \begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \end{pmatrix} \geq 0 \tag{5.3.1}$$

In this way, primitives can be identified whether they are inside the volume. If the primitive under consideration is a polygon, it is passed if every one of its edges lies entirely inside the clip volume or is either clipped or discarded otherwise. If it happens that a polygon intersects the clip volume's plane, the polygon edges should be clipped. However, as polygon connectivity must be maintained, these clipped edges are connected by new edges that lie along the clip volume's boundary. Thus, clipping may require the introduction of vertices into a polygon. Edge flags are associated with these vertices so that edges introduced by clipping are flagged as boundary (edge flag TRUE), and so that original edges of the polygon that have cut off at these vertices retain their original flags.

All these testing and clipping operations are supported by OpenGL and accelerated by graphic hardware. Therefore, for implementation, a small tip is useful. Since each

image layer volume has the same left, right, top and bottom planes with the view frustum but different front (near) and back (far) planes, instead of calculating the six clip planes, the rendering node can just reconfigure the near and far clip planes. In the eye coordinates, the near and far plane are paralleled with the image projection plane. It is (0,0,1,-d) where d is the distance from the eye to the near or far plane. The value of d can be obtained when partitioning the view frustum into image layer volumes.

Figure 5.3 shows an example of data decomposition into image layer volumes. In this sample, the geometry data is partitioned into two parts. The pyramid overlaps the partition plane. It will be assigned to both sides. Thus, the set of geometries that lies in the front image layer volume includes a small teapot, a sphere and the intersected pyramid. The set of geometries lying in the back image layer volume includes the pyramid, a big teapot, a cylinder and a cone.

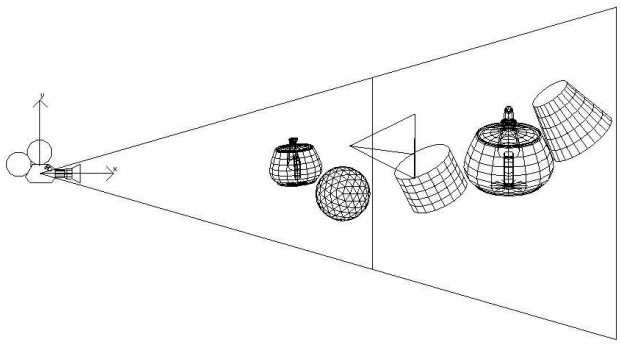


Figure 5.3: Data Decomposition with Image Layer Volume(left)

Figure 5.4 shows the perspective view of the scene. The view frustum is partitioned into two image layer volumes. The front image layer volume has the same front(near), left, top, right and bottom clip plane as the view frustum. Its back clip plane is the partition plane. The back image layer volume has the same back(far), left, top, right

and bottom clip planes as the view frustum. Its front clip plane is the partition plane.

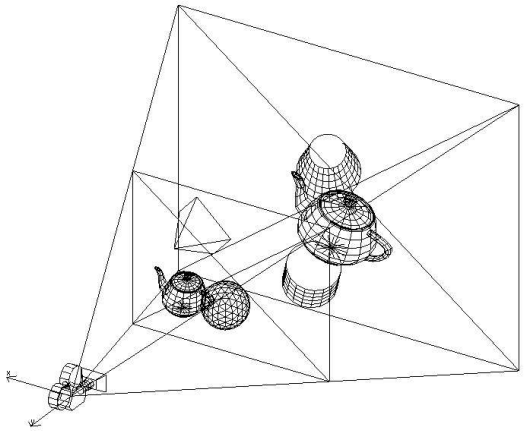


Figure 5.4: Data Decomposition with Image Layer Volume(perspective)

After decomposition, the system sends the set of data with its image layer volume (the front and back clip plane) to a rendering node. Each rendering node generates a full size image of the assigned geometry data. Figure 5.5 and Figure 5.6 show the rendering results of front and back image layer volumes.

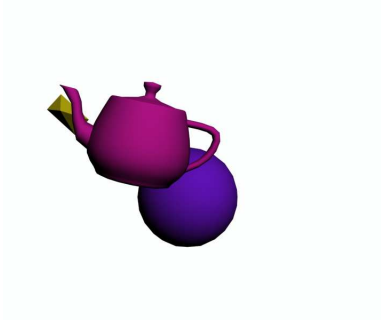


Figure 5.5: The Rendering Result of the Front Image Layer Volume

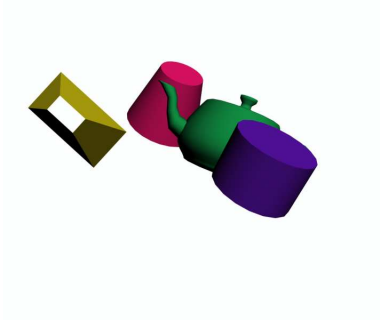


Figure 5.6: The Rendering Result of the Back Image Layer Volume

5.4 Peer-to-peer Image Composition with Alpha

In traditional sort-last algorithms, each rendering node usually renders its portion of primitives and generates a full size image with a Z-buffer. The Z-buffer contains the Z (depth) value for the point in the scene which is drawn at that pixel. Assume that the negative Z-axis points away from the viewer, so that the smaller the Z value, the farther away from the viewer. A composing algorithm inputs two images and iterates through pixel pairs (two pixels with corresponding locations in the input images). These pixels are read and their Z-values compared so that the pixel closer to the viewer is drawn to the result image. However, in a geographically distributed computational environment, this method is quite expensive. Usually, tightly coupled graphics systems are integrated with high bandwidth bus, e.g., PixelFlow system offers a point-to-point composition network bandwidth of 6.4 GB/s [22]. In a geographically distributed system, the Ethernet bandwidth is 100 Mbits/second and Myrinet is 1 Gbits/second. These are much lower than the requirements of high resolution image composition. Therefore, an image composition using Alpha instead of Z-buffer is proposed in this section.

In section 5.3, the geometry data are decomposed into image layer volumes according to their distribution along Z axis. Each image layer volume is rendered into an image layer. Then, the image layers are merged by drawing the images from back-to-front. However, this method does not always lead to a right result. For example, Figure 5.7 shows an image with both letter A and B. A is further than B from the viewpoint. If A and B are rendered into different image layers and merged by drawing the images from back-to-front, shown as Figure 5.8, the merged result is not correct. Some background color pixels in B which should be transparent obstruct A incorrectly.

Figure 5.7: Image Compostion with Z buffer

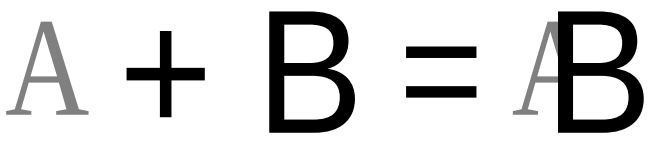


Figure 5.8: Image Compostion without Z buffer

One solution to this problem is to transfer active pixels. This means, the pixels generated by geometry data in the upper layer is transferred and merge to the lower image layer. The pixels which should be transparent will not be transferred. This method has some obvious disadvantages. It introduces much computational overhead for selecting and encoding active pixels. Reading and writing fragmental pixels is much slower than reading and writing successive pixel block, though they have the same data size. These disadvantages are critical for interactive distributed rendering systems. Therefore, transferring active pixels is out of our consideration.

In our solution, alpha channel is used to merge the image layers. The alpha value of a transparent pixel is 0. The alpha value of an opaque pixel is 1. The alpha value of a translucent pixel arranges from 0 to 1. Therefore, when writing the pixels of upper image layer to lower image layer bound with an alpha channel, the correct result can be achieved. It is important to merge image layers in the order from back to front.

Another advantage of this method is that it is easy to render the effect of the translucent materials such as glass, water and so on. Figure 5.9 shows the last image composed by Figure 5.5 and Figure 5.6.

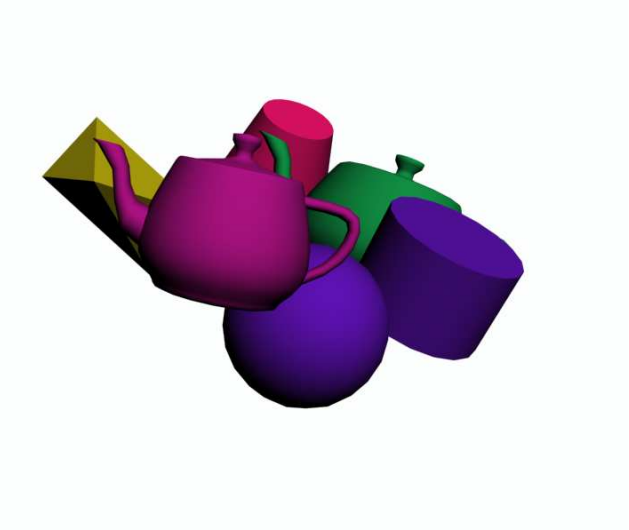


Figure 5.9: Image Composition with Alpha Channel

In order to take full advantage of the computational capability to speed up image composition, an peer-to-peer image composition scheme is developed. This scheme is similar to Lee's parallel pipeline composition [45]. Because it is based on the composition with Z-buffer, Lee's approach need not follow the order from back to front. However, our method should take it into account.

In this method, each node composes a portion of the image. The processor at each stage directly sends a sub-image to the processor which is in charge of composing that portion of the final image. The sending sequence is ordered to avoid link contention. This sequence should also maintain the composition order from back to front. If the composition strictly follows back-to-font order, it is difficult to avoid the link contention. An alternative method is to compose the adjacent image layers one by one. For example,

there are four image layers, L_0, L_1, L_2, and L_3 in the order of from back to front. Firstly, the image layer L_1 can be merged with L_2 by drawing L_2 on the top of L_1, the result marked as L_1. Then, image layer L_0 merges with L_1 by drawing L_1 on top of L_0, the result marked as L_0. Lastly, merges image layer L_0 merges with L_3 by drawing L_3 on top of L_0, the result marked as L_0. In this way, the correct result also can be achieved. The pseudocode is shown as Algorithm 2.

Algorithm 2 Composition Sequence Algorithm

N : Number of processors
$L_{i,k}$: The $k-th$ sub-image of $i-th$ image layer
P_i : The $i-th$ Processor which hold L_i image layer and is in charge of composing the $i-th$ sub-image

```
BEGIN

  FOR j = 1 TO N − 1
     P_i sends L_{i,(i−j)modN} to P_{(i−j)modN}
     P_i receives L_{((i+j)modN),i} from P_{(i+j)modN}
     P_i compose L_{i,i} with received L_{((i+j)modN),i}

  ENDFOR

END
```

For example, Figure 5.10 shows an example of image composition with 4 image layers. In the first step, P_0 send its $L_{0,3}$ to P_3 and receives $L_{1,0}$ from P_1. In the second step, P_0 send its $L_{0,2}$ to P_2 and receives $L_{2,0}$ from P_2. In the third step, P_0 send its $L_{0,1}$ to P_1 and receives $L_{3,0}$ from P_3. Similarly, the pixel information exchanges and compositions occur in this fashion for P_1, P_2 and P_3.

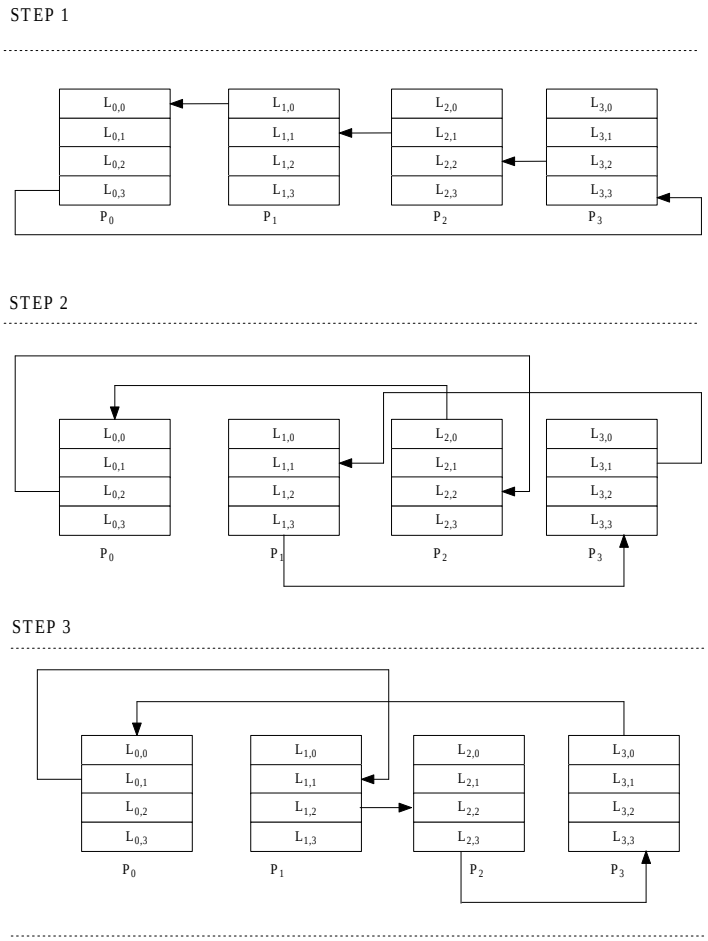


Figure 5.10: Image Compostion Sequence with 4 Image Layers

5.5 3D Overlap Factor

In our prototype system, the rendering workloads are parallelized at the inter-cluster level by the sort-first strategy, DPBP. They are also parallelized inside each cluster by the sort-last strategy, Image Layer Composition with Alpha, called ILCA. In the projection coordinates (the 3D space after viewing transformation), the primitives are sorted not only in 2D screen space(X, Y coordinates) but also in depth (Z coordinate). That means, the primitives are sorted in 3D. In contrast to other sort-last strategies, this method introduces redundant rendering resulting from overlap factor. This is similar

with the sort-first algorithms. Combining the overlap factors in screen space and in depth, a 3D overlap factor can be achieved. The 3D overlap factor describes the number of image layer volumes overlapped by a typical primitive.

An equation to estimate the 3D overlap factor can be derived as follows:

In Figure 5.11, we assume the size of an image layer volume is $W \times H \times D$ and the size of the 3D bounding box of a typical primitive is $w \times h \times d$.

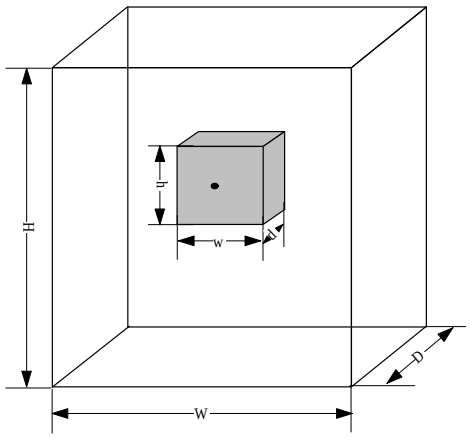


Figure 5.11: A 3D Bounding Box of a Typical Primitive within an Image Layer Volume

It can be assumed that primitives have an equal probability of falling anywhere within an image layer volume.

Shown as Figure 5.14, the probability of falling in a corner should be:

$$O_c = \frac{8(w/2)(h/2)(d/2)}{W \times H \times D} \tag{5.5.1}$$

Shown as Figure 5.13, the probability of falling in a edge should be:

$$O_e = \frac{4(w/2)(d/2)(H-h) + 4(h/2)(d/2)(W-w) + 4(w/2)(h/2)(D-d)}{W \times H \times D} \tag{5.5.2}$$

Figure 5.12: A 3D Bounding Box of a Typical Primitive within an Image Layer Volume

Figure 5.13: A 3D Bounding Box of a Typical Primitive within an Image Layer Volume

Figure 5.14: A 3D Bounding Box of a Typical Primitive within an Image Layer Volume

Shown as Figure 5.14, the probability of falling in a face should be:

$$O_f = \frac{2(w/2)(D-d)(H-h) + 2(h/2)(D-d)(W-w) + 2(w/2)(H-h)(D-d)}{W \times H \times D} \tag{5.5.3}$$

Shown as Figure 5.15, the probability of falling in a center area should be:

$$O_m = \frac{(W-w) \times (H-h) \times (D-d)}{W \times H \times D} \tag{5.5.4}$$

The overlap factor can be calculated by summing these probabilities weighted by the resulting overlap of a primitive falling in each of these volumes.

$$O_{3D} = 8O_c + 4O_e + 2O_f + O_m = \frac{(W+w) \times (H+h) \times (D+d)}{W \times H \times D} \tag{5.5.5}$$

Figure 5.15: A 3D Bounding Box of a Typical Primitive within an Image Layer Volume

5.6 Comparison with 2D Overlap Factor

As derived in Section 4.1.3, for homogeneous sort-first systems, the overlap equation is :

$$O_{2D} = (\frac{W' + w}{W'})(\frac{H' + h}{H'}) \tag{5.6.1}$$

where the size of a typical screen tile is $W' \times H'$ and the size of the 2D bounding box of a typical primitive is $w \times h$.

In order to compare with the 3D overlap factor, some assumptions are made as follows:

- All primitives are evenly distributed.
- The static partition strategy is used for sort-first and sort-last algorithms. All the screen partitions have the same size $W' \times H'$. All the image layer volumes have the same size $W \times H \times D$
- There are N rendering nodes in K clusters. Each cluster has n rendering nodes.

$K \times n = N$.

- In each dimension, the proportions between a typical image layer volume and a typical 3D bounding box are same.

$$\frac{w}{W} = \frac{h}{H} = \frac{d}{D} = k$$

- In each dimension, the proportions between a typical screen tile and a typical 2D bounding box are same.

$$\frac{w}{W'} = \frac{h}{H'} = k'$$

The 3D overlap equation can be simplified as follows:

$$O_{3D} = \frac{(W+w) \times (H+h) \times (D+d)}{W \times H \times D} = (1+k)^3 \tag{5.6.2}$$

The 2D overlap equation can be simplified as follows:

$$O_{2D} = \frac{(W'+w) \times (H'+h)}{W' \times H'} = (1+k')^2 \tag{5.6.3}$$

For homogeneous sort-first strategy, the image size $S = W' \times H' \times N$. For sort-both strategy, the image size $S = W \times H \times K$. Then, we have $k' = n^{1/2}k$.

Comparing the 3D overlap factor with 2D overlap factor,

$$\Delta = O_{3D} - O_{2D} = (1+k)^3 - (1+k')^2 = k(k^2 + (3-n)k + 3 - 2n^{1/2}) \tag{5.6.4}$$

Because $k > 0$, we focus on the following equation:

$$\Delta' = (k^2 + (3-n)k + 3 - 2n^{1/2}) \tag{5.6.5}$$

Solving the equation gives:

$$k_1 = \frac{n - 3 - \sqrt{n^2 - 6n + 8\sqrt{n} - 3}}{2}$$

$$k_2 = \frac{n - 3 + \sqrt{n^2 - 6n + 8\sqrt{n} - 3}}{2}$$

From this result, it can be found that: $\Delta < 0$, when $k_1 < k < k_2$, and $n > 3$, $k_1 < 0$ and $1 < k_2$. As shown in Figure 5.16, when each cluster has more than 3 rendering processors and the size of the 3D bounding box is smaller than the image layer volume, the 3D overlap factor will be smaller than the 2D overlap factor. When $n = 1$, we can get $k = k'$ and $\Delta = 0$. That means 2D overlap factor is same as 3D overlap factor when each cluster only has one node inside. In this situation, the sort-both architecture is same as the sort-first architecture. The result is reasonable.

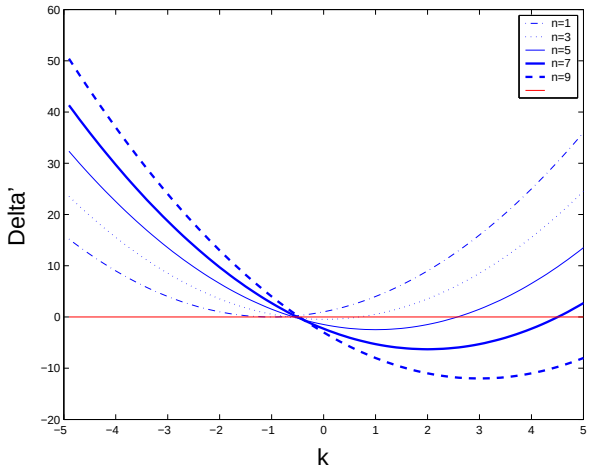


Figure 5.16: Delta

Figure 5.17 shows that k_1 is always smaller than 0, and k_2 increases linearly with n. With $n = 9$, even when the bounding box is 108 times larger than the image layer volume, the 3D overlap factor is still smaller than the 2D overlap factor with the

same number of partitions. Therefore, generally, sort-both architecture introduces less redundant rendering than a homogeneous sort-first architecture.

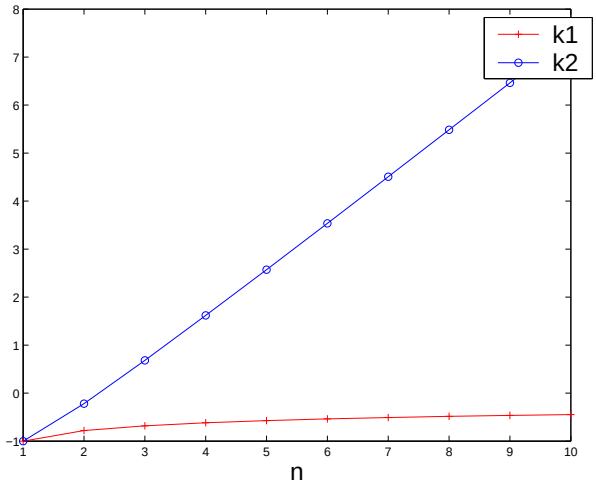


Figure 5.17: Solutions with Different n

5.7 Experiments and Results

In this section, we present the figures showing the summary results of Image Layer Composition with Alpha (ILCA).

Obviously, ILCA is not a free lunch. The tradeoff of a relatively small pixel redistribution cost is the redundant rendering. As discussed above, when a primitive intersects two image layer volumes, it will be processed twice. Therefor, performance of ILCA depends on various factors, such as, the bandwidth of interconnection, the geometry data size, the primitives' density in the 3D space, the number of the rendering nodes and so on. However, in a distributed rendering environment, bandwidth is critical to the performance. It directly affects the image composition performance. Therefore, we examine the image composition cost of ILCA in this section.

The first experiment compared ILCA with a general sort-last method which applys depth comparison. This method arbitrarily partitions and distributes the geometrical primitives to the various rendering processors. Each rendering processor renders a full size image with a portion of the geometrical primitives. These images are composed with the depth to generate the final image. As for ILCA, these images are composed with peer-to-peer parallel method.

In these experiments, we use 64 rendering nodes rendering the model with 1,453,290 polygons into a sequence of images with 800×600 resolution. Figure 5.18 shows the result statistics of both ILCA and the normal sort-last method running 20 frames. As shown in this figure, the image composition performance of ILCA is better than the normal sort-last method, because of the lower pixel redistribution cost.

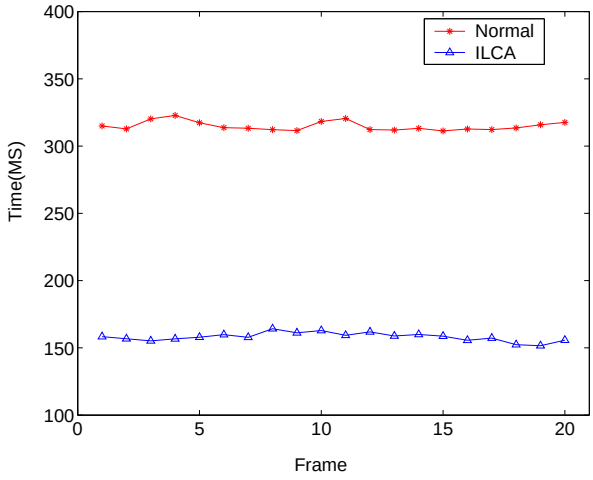


Figure 5.18: Image Composition Performance Comparison

According to the conclusion from [53], the performance of sort-last methods depends much on the image resolution. The second experiment examines the impact of resolution on the performance of ILCA. Similar as the first experiment, this experiment also uses 64 rendering nodes rendering the model with 1,453,290 polygons into a sequence of

images with different resolutions. For each resolution, we record the time for 50 frames and compute the average time per frame. Figure 5.19 presents the image composition results of ILCA and normal sort-last method together. It is obviously, ILCA has much better performance than the normal sort-last method with higher resolution. At high resolution, the pixel redistribution become the dominant overhead for sort-last architectures. Compared with normal sort-last methods, ILCA can effectively reduce this overhead.

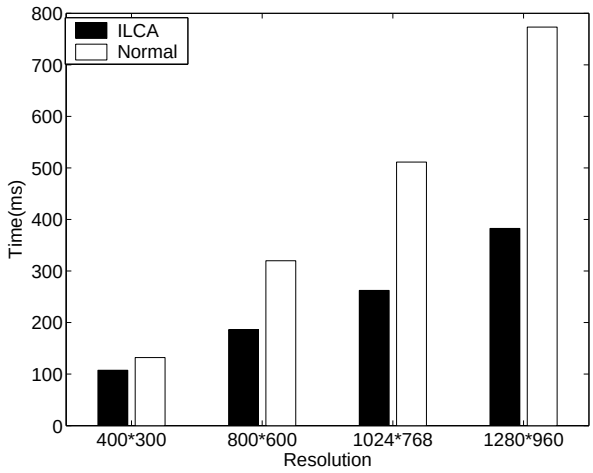


Figure 5.19: Impact of Image Resolution

5.8 Conclusion

In this chapter, we examined the important issues of sort-last architectures. In a distributed computing environment, low bandwidth connection is always the bottleneck of the sort-last systems. Therefore, a sort-last architecture is normally suitable only for low resolution rendering. We have developed a novel sort-last algorithm in our research. It is implemented at the cluster level which renders small size sub-images. In order to reduce bandwidth requirements, in this algorithm, the geometry data is decomposed

into multiple image layer volumes along the Z axis. Each rendering processor generates an image layer with the assigned primitives. To compose the final image, this strategy simply stacks these image layers one on top of another according to the visibility order of the layers. In this way, the system does not need to access the depth buffer which is a time consuming operation. It also does not transfer depth buffer data. This reduces the bandwidth requirements significantly.

Chapter 6

Sort-both Parallel Rendering

6.1 Sort-both Architecture

None of the homogeneous architectures is a clear winner under all conditions; rather, each is potentially useful for some set of applications and implementation constraints. In a distributed computing environment, the situation is much more complex. In most cases, it can be assumed that the bandwidth of inter-cluster connections is relatively lower than the bandwidth inside a cluster. Therefore a hybrid architecture, named sort-both, involving both sort-first and sort-last architectures, is designed for distributed environments. Figure 6.1 shows the hierarchical architecture of sort-both.

Sort-both is composed of N graphics pipelines with in K rendering clusters as depicted in Figure 6.1. Each pipeline is composed of six stages: command, geometry, rasterization, texture, fragment and display. The client receives rendering tasks from an application. It decomposes the tasks evenly and passes them to the rendering clusters (sort-first partitioning). The head node of each cluster receives tasks from client and decomposes them again (sort-last partitioning). Rendering processors receive the tasks from the head nodes and transform, light, and clip the primitives. Then the rendering

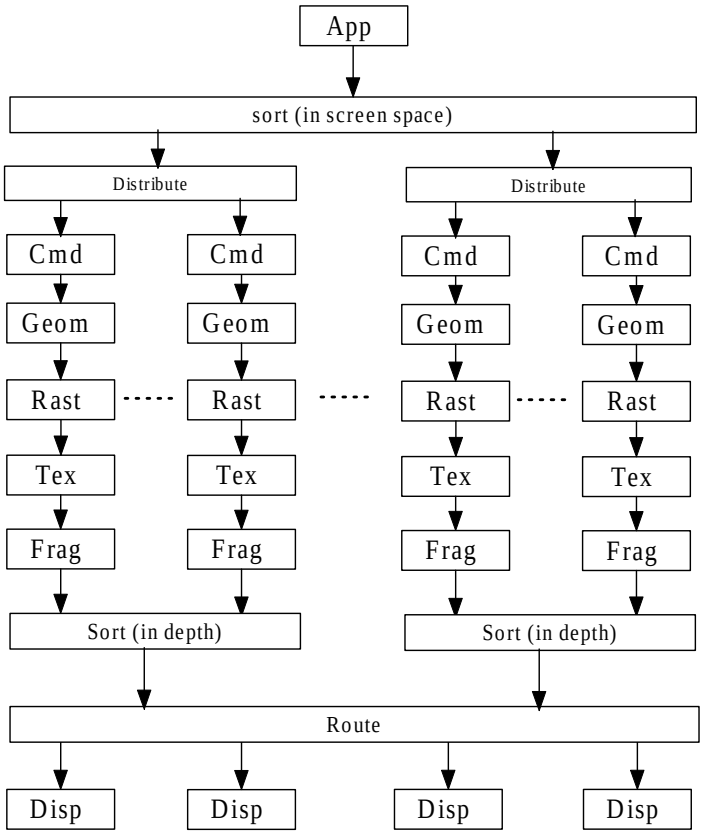


Figure 6.1: Sort-both Architecture

processors perform rasterization setup on these screen-space primitives, and scan convert them into untextured fragments. The rendering processors texture the resultant fragments and merge them into the frame buffers. the pixels in the frame buffers will be redistributed through an image composition network. The network performs parallel image composition to generate a sub-image. Each cluster generates a sub-image and total K sub-images will be assembled together to generate a final image.

In order to make sort-both a viable architecture in distributed computing environments, we address the major difficulties in creating a sort-both implementation. As the discussion in Chapter 4 and Chapter 5, we investigate an efficient load balancing algo-

rithm for sort-first and a low overhead image composition approach for sort-last. These efforts have two purposes. One is to keep load-balancing among rendering clusters. The other is to reduce the pixel redistribution cost and advance the performance of image composition.

6.2 An Implementation on Computational Grids

Shown as Figure 6.1, sort-both is a parallel rendering strategy based on the rendering pipeline. Mapping the sort-both strategy onto the distributed rendering architecture is an important problem related to performance. It is helpful to achieve maximum computational potential and bandwidth utilization in the environment.

In this section, we suggest a possible implementation on Computational Grids [26]. A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities. The motivations of distributing rendering tasks over computational Grids are numerous, such as:

- Geographical Distribution

  As the demands placed on visualization environments increase, it becomes more difficult to address all requirements on a single computing platform or for that matter in a single location [25]. Resources for scientific visualization, e.g., instruments for data collection, high performance computers for data processing and high-end visualization equipments, may be geographically distributed and belong to different institutes. Computational Grids provide distributed computing technologies to manage resources shared in this large scale distributed environment.

- Flexibility

  Networked systems in which computers communicate only via network protocols, allow computational resources to be added to or removed from the system easily; and they can even be heterogeneous. On computational Grids, the system can share these resources dynamically. It makes the system much more flexible to meet different requirements with easy scalable capacity.

- Low Cost

  High end graphics systems are extremely expensive. In Grid environments, commodity hardware components are used. Thus, the system is less expensive, more flexible, and tracks technology better than other systems with custom hardware [9]. Since the resources are shared by different users dynamically on computational Grids, for each user, the cost is relatively lower.

we describe three phases mentioned in Chapter 2 in the general context of a Grid which consists of non-homogeneous computing resources connected by a wide-area network. It usually consists of a cluster of clusters. Computing resources located in geographically distributed sites, are shared in the environment. A typical characteristic of this architecture is the large gap between the fast connection inside a cluster and slow connection between clusters. An application designed for uniform speed interconnects can lead to performance degradation.

In a parallel rendering system, obviously, the sort-first strategy requires much less bandwidth than the sort-last strategy [53]. Therefore, in this system, phase 1 and phase 3 are mapped onto Grid level and phase 2 is mapped onto cluster level (also see Figure 3.1). Namely, the sort-first processes, including partitioning and composing run

on Grid level and the sort-last rendering processes, including partitioning, drawing and composing, run on the cluster level (see also Figure 6.2). Shared memory is used for communication between sort-first and sort-last processes.

Communication interface provides functionalities of sending, receiving and buffering messages from other nodes. we adopt MPICH, a portable message passing interface standard (MPI) [24] implementation, as the basis for the implementation of the interface for several reasons: (1) MPICH provides a free of charge, general purpose message passing library that can save us the burden of implementing communication facilities, (2) MPICH is portable to a wide range of platforms, including shared-memory machines, loosely-coupled workstation clusters and even individual PCs. The usage of MPICH enables our prototype to be running on various environments. To make full use of resources in a hierarchical distributed computing environment, the system needs to link different MPI libraries in two levels. In the Grid environment, MPICH-G2 [42] based processes run on the display node, the sort-first task partitioning node and the head node of each rendering cluster for data exchange. It passes rendering requirements from GUI to partitioning node, sends partition information to head nodes of the rendering clusters and collects sub-images from the head nodes for display. Inside the cluster, MPICH-P4 is used to transfer data between rendering processes. The head node partitions the data received from Grid level with a sort-last strategy. MPICH-P4 passes the results to the rendering processes and passes sub-images between rendering processes for sub-image composition (with depth comparison). In the head node, two communication processes are running. One based on MPICH-G2 belongs to the Grid environment and the other based on MPICH-P4 belongs to the local cluster. Shared memory supports

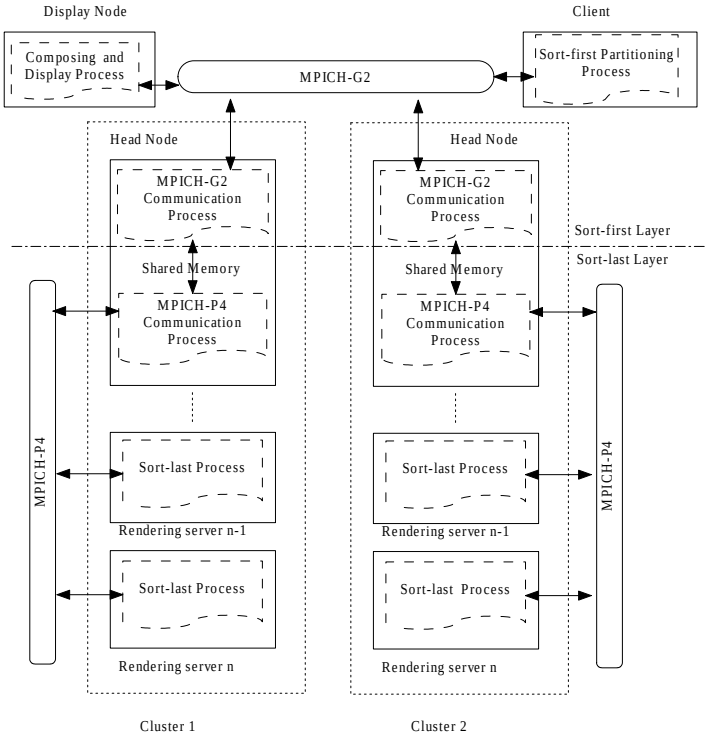


Figure 6.2: Archtecture of the Software Module

the communication between the two processes in the head node. The MPICH-G2 based processes write sort-first partition results to the shared memory and read the rendering results (sub-images) from the shared memory. The MPICH-P4 based processes read sort-first partition results from the shared memory and write the rendering results to the shared memory. Figure 6.2 shows the hierarchical architecture on Grids.

6.3 System Scalability

This study investigates the scalability of the sort-both architecture as the number of rendering nodes increases. System scalability involves two concerns, server scalability and client scalability.

6.3.1 Client Scalability with System Size

The first experiment examines the client scalability. A model with 1,453,290 polygons are rendering into images with 1280 $\times$ 960 resolution. The renderer is implemented with OpenGL. The client time of sort-first, sort-last and sort-both is recorded respectively. Each session renders 50 frames and the average is adopted. Figure 6.3 shows the scalability of the client when increasing the number of the rendering nodes. The client runs DPBP sort-first algorithm to partition the image into K subregions, where K is the number of clusters. As discussed in Chapter 4, the running time of DPBP algorithm grows linearly with K. From Figure 6.3, it is clear that sort-last needs no client time at all. The client time needed by sort-both architecture is much smaller than a homogeneous sort-first architecture. Although the DPBP algorithm is adopted by both of them, the performance on the clients is different. For sort-both architecture with N rendering nodes in K clusters, only K subregions are needed to be partitioned into. However, for homogeneous sort-first architecture, the DPBP algorithm has to partition an image into N subregions. In this way, compared with a homogeneous sort-first architecture, the performance of sort-both architecture on the client is improved.

6.3.2 Analysis of Server Time

The server time of the sort-both architecture is one of the key factors in evaluating performance. The server time is the time interval from receiving data from the client to generating the final sub-image. In Figure 3.1, it is the time consumed by rendering clusters on phase 2. It consists mainly of rendering time and pixel redistribution time. Other small costs are ignored in our discussion. In order to analyze the server time, the

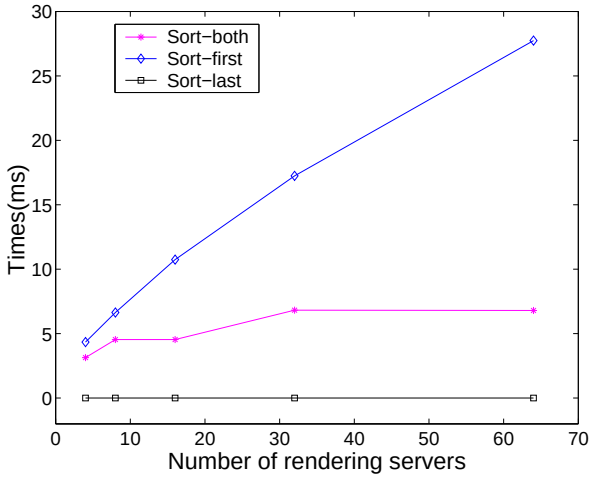


Figure 6.3: Client Scalability

following simplifying assumptions are made:

- All objects are evenly distributed.
- The 2D bounding boxes of objects are square of equal size in screen space. The edge of the square is of length w.
- There are P total pixels on screen and N rendering nodes in K clusters. Each cluster has n rendering nodes. $K \times n = N$.
- Each rendering node is assigned a square tile with area of P/K.
- There are M net primitives of the model.
- All the rendering nodes have same performance. Primitives throughput of a rendering node is DC primitives/second and composition throughput is C pixels/second.
- Network bandwidth inside clusters is B bits/second.

From the overlap factor equation first derived by John Eyles of UNC [53], the work load, total number of primitives to be rendered over the clusters is

$$W = M(1 + w\sqrt{\frac{K}{P}})^2 \tag{6.3.1}$$

The total rendering time is given by

$$\frac{M(1 + w\sqrt{\frac{K}{P}})^2}{N \times DC} \tag{6.3.2}$$

Each pixel contains a 32 bits color value and a 32 bits depth value. The pixel redistribution overhead is

$$\frac{P}{K \times C} + \frac{P \times 64}{K \times B} \tag{6.3.3}$$

We can get the total rendering time:

$$T(K, N) = \frac{P}{K \times C} + \frac{P \times 64}{K \times B} + \frac{M(1 + w\sqrt{\frac{K}{P}})^2}{N \times DC} \tag{6.3.4}$$

If we implement ILCA in the sort-both architecture, the total rendering time should be:

$$T(K, N) = \frac{P}{K \times C} + \frac{P \times 32}{K \times B} + \frac{M(1 + w\sqrt{\frac{K}{P}})^3}{N \times DC} \tag{6.3.5}$$

From this equation, we can find an important feature of sort-both architecture - flexibility. By adjusting the value of K and N, the system performance can meet various requirements properly. This is quite useful for Grid resource allocation. By this

equation, the system can evaluate how many rendering nodes within how many clusters should be arranged to achieve the quality of rendering service required by users.

The second experiment examines the scalability of the rendering nodes. The experiment renders a model with sort-first, sort-middle and sort-last strategy, respectively. The image size is 1280 × 960. The seed up factors are recorded in the Table 6.1. Figure 6.4 and Figure 6.5 show the scalability of the rendering nodes using models with 681,160 polygons and 1,453,290 polygons respectively. The experimental results are quite encouraging. For 64 rendering nodes, the speed up factors of sort-both are 38.6 and 43.2, respectively. These speedups compare favorably to homogeneous sort-first or sort-last architectures. In the case of the model with 1,453,290 polygons, the sort-first architecture speed up factors is 24.3 and sort-last architecture speed up factor is 11.0. It owes much to the flexibility of the architecture. It is helpful in achieving low overlap rendering and low pixel redistribution overhead.

Data	Nodes	Speedup		
		sort-first	sort-last	sort-both
681,160	4	3.2	3.5	3.3
	8	6.1	5.3	5.5
	16	10.7	9.2	12.8
	32	15.9	11.6	23.3
	64	23.6	12.7	39.1
1,453,290	4	3.5	3.6	3.6
	8	6.4	5.1	6.8
	16	11.2	7.8	14.8
	32	16.7	9.7	27.6
	64	24.3	11.0	43.2

Table 6.1: Rendering Speedup for Image 1280 × 960

Comparing with Figure 6.4, Figure 6.5 can be found that the speedup factor of sort-both architecture is more close to ideal one. Therefore, sort-both architecture also

scales with the number of primitives.

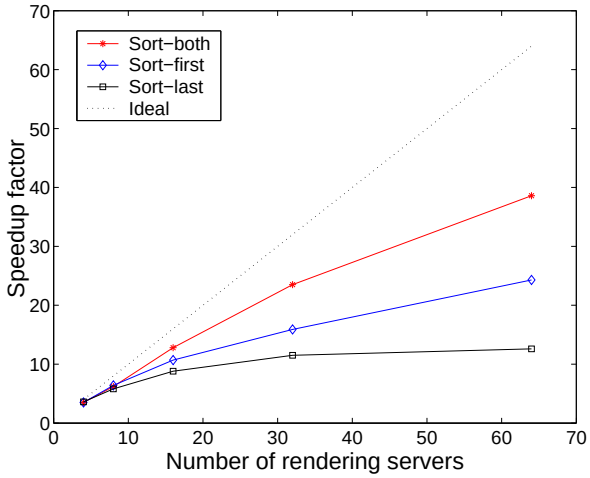


Figure 6.4: Rendering Speedup with 681,160 Polygons

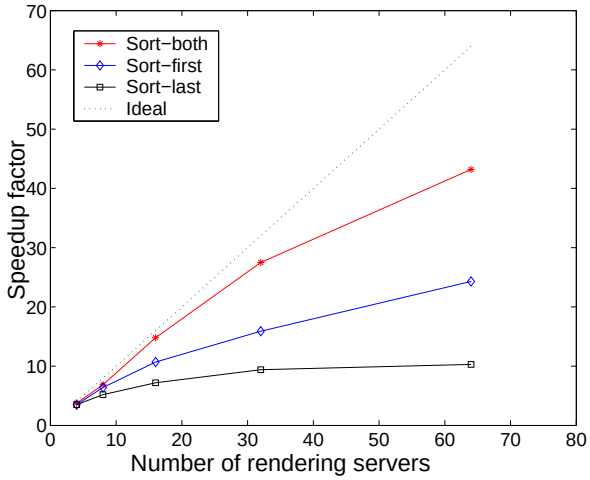


Figure 6.5: Rendering Speedup with 1,453,290 Polygons

6.3.3 Comparison with Sort-first Architecture

In this investigation, we compare sort-both strategy to the sort-first strategy mentioned above. From Figure 6.6, it can be found that with small number of rendering nodes, the sort-first architecture is a little bit more efficient than the sort-both one because of the

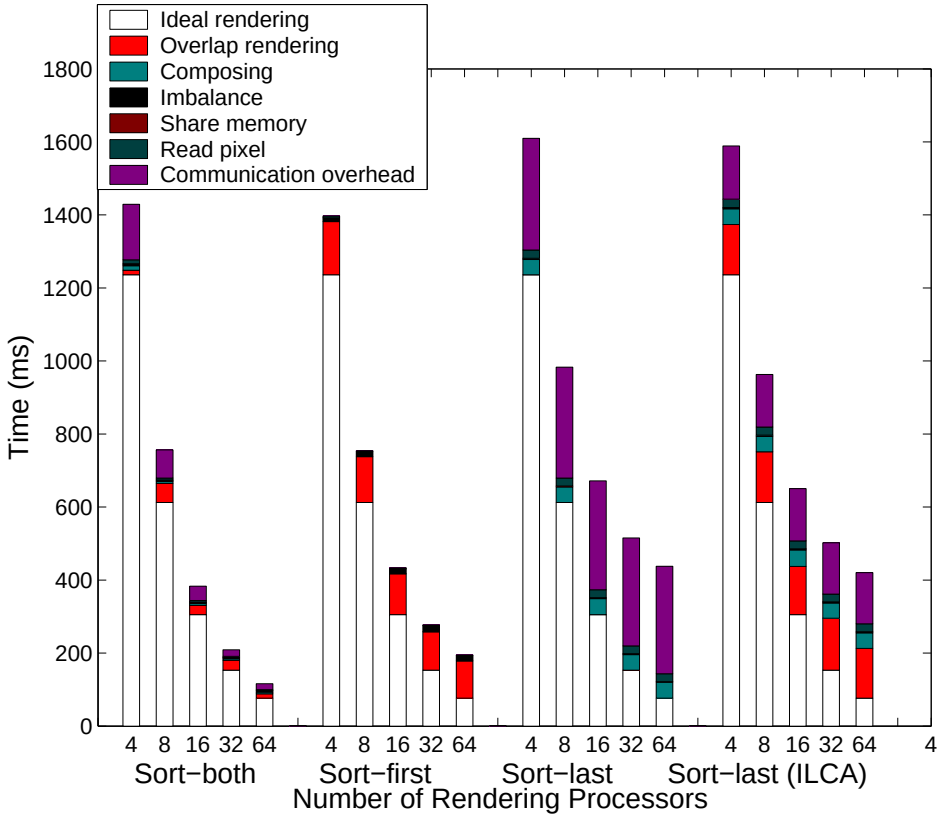


Figure 6.6: Rendering Times for Sort-first, Sort-both, and Sort-last

small overlap rendering introduced. However, these redundant rendering time become a larger percentage of the total server time as the overlap factor grows with increasing the number of rendering nodes. In contrast, the sort-both architecture introduces a small amount of redundant overlap rendering with large number of rendering nodes. Figure 6.7 shows the overlap factor of sort-first and sort-both architectures rendering 1,453,290 polygons with 1024×768 resolution. From Figure 6.4 and 6.5, it can be found that the speedup factors of sort-both is much better than sort-first when more than 8 rendering nodes are involved in the system.

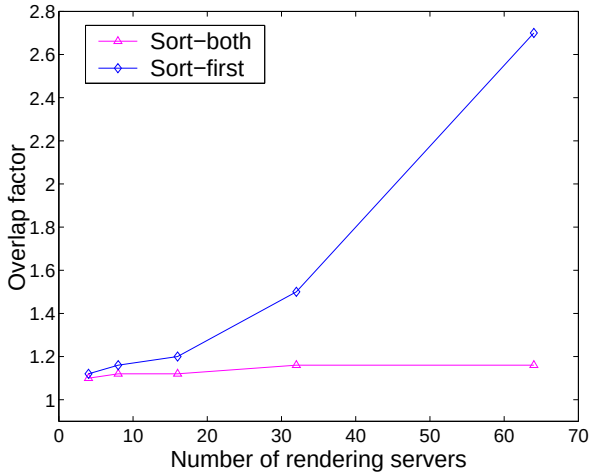


Figure 6.7: Overlap Factor

6.3.4 Comparison with Sort-last Architecture

In the third study, the sort-both architecture is compared with the sort-last architecture. In this comparison, the sort-last architecture is integrated with the same algorithm used in sort-both architecture. The main bottleneck of sort-last architecture is pixel redistribution overhead.

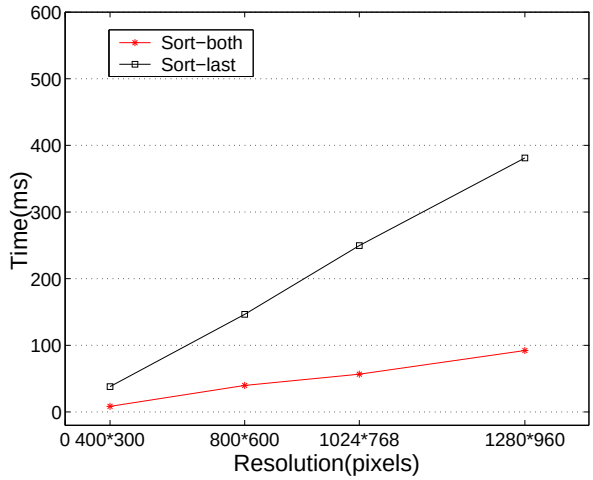


Figure 6.8: Pixel Redistribution Overhead

Figure 6.6 shows that, in sort-last architectures, the communication overhead for

pixel redistribution becomes a significant percentage of the total rendering time as the number of rendering nodes is increased. However, the communication overhead for pixel redistribution of sort-both architecture remains a small portion of the total server time. It benefits from the small portion of the image each cluster is required to render. Figure 6.8 shows the pixel redistribution overhead of sort-both and sort-last architectures rendering the model with 1,453,290 polygons in various resolutions. Both of them use ILCA sort-last method.

6.3.5 Impact of Increasing Image Resolution

In the fourth study, we analyze the effect of increasing image resolution on sort-both architecture. Firstly, consider the scalability of the display node. The bandwidth requirement grows linearly when increasing the image resolution. This impact on the display nodes exists irrespective of sort-first, sort-last and sort-both architectures.

Secondly, consider the rendering time. There are fundamental differences among sort-first, sort-last and sort-both. For sort-first, the dominant overhead is independent of image resolution. As a result, sort-first appears most appropriate for very high resolution images [53].

Sort-last and sort-both architectures are definitely impacted by increasing the image resolution. The pixel redistribution overheads of the sort-last architecture grows with P, where P is the resolution. However, the pixel redistribution overheads of the sort-both architecture grows with P/K, where K is the number of clusters used. Shown as Figure 6.8, with the same resolution, sort-last architectures consume much more time than the sort-both one for pixel redistribution. This means that the impact of increasing image

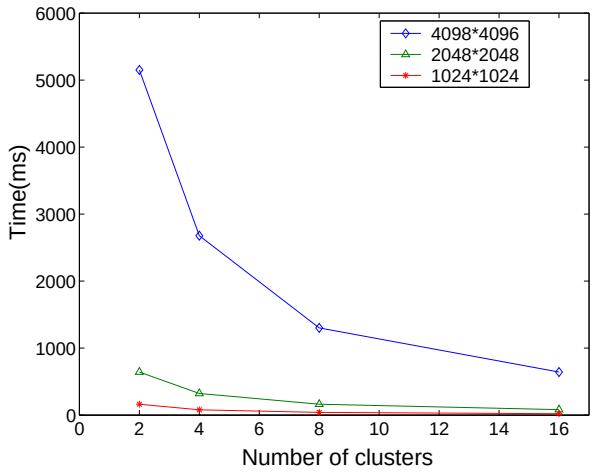


Figure 6.9: Pixel Redistribution Overhead When Increasing the Resolution

resolution can be reduced by increasing the number of clusters and is clearly shown in Figure 6.9. This figure shows the variation of the pixel redistribution overhead with increasing the number of clusters.

6.3.6 Impact of Resource Mapping

Sort-both is a flexible strategy. If all the nodes are in one cluster, sort-both becomes a homogenous sort-last strategy. If all the nodes are in different clusters, it is a homogenous sort-first strategy. Therefore, the sort-both strategy can adjust task structure of resource mapping to various hierarchical distributed environments. The resource mapping results can affect the performance of the system with the sort-both architecture. Therefore, the system should estimate the performance before rendering to optimize the task structure of resource mapping. According to the available bandwidth and available computational capability, the system estimates the number of rendering nodes needed to achieve the required rendering service (such as frame rate, image resolution and geometry data size) using Equation 6.3.4 (or Equation 6.3.5). In this equation, the

pixel redistribution overhead can be decreased by increasing K, the number of clusters. However, it will increase the cost of redundantly rendering objects that overlap multiple tiles. It is important to strike a practical balance between these trade-offs. E.g., Figure 6.10 shows the server time for rendering 681,160 polygons in 1024×768 resolution with 32 rendering nodes. In this experiment, the number of rendering nodes is fixed but can be configured into varying number of clusters. The rendering systems has the best performance when 8 clusters with 4 rendering nodes each are used.

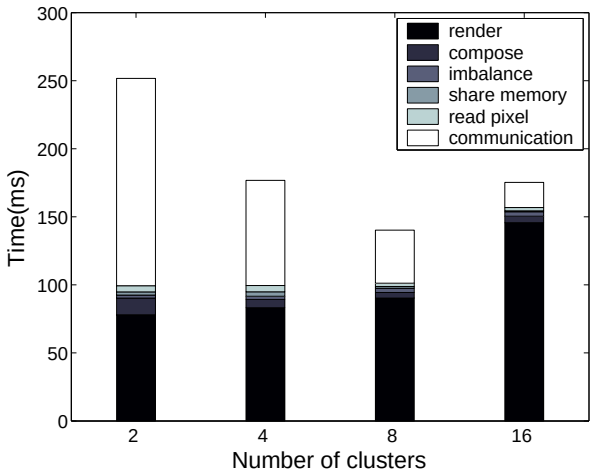


Figure 6.10: Impact of Resource Mapping

In this case, in a homogenous sort-first architecture, the communication data size is around 3.1 MB per frame. In homogenous sort-last architecture, the data size of pixel redistribution is around 200 MB per frame. However, in this experiment, with a sort-both architecture, it is still 3.1 MB per frame for inter-cluster communication (sort-first). The data size is 25 MB per frame for inside cluster communication (sort-last). If ILCA is implemented, the data size is 12.5MB per frame. Therefore, an optimized task structure of resource mapping is important to improve the rendering performance.

6.4 Summary

This chapter presents our novel architecture for distributed rendering environments, named sort-both. It involves both sort-first and sort-last strategies. The advantage of this architecture is obvious. Compared with homogeneous sort-first architectures, it introduces less overhead because of lower overlap ratio. Compared with sort-last, it requires less communication cost for pixel distribution. Each processor implements entire rendering pipeline for a portion of the screen. Sort-both architecture is flexible and scalable.

Chapter 7

Conclusion and Discussion

Scientific visualization for massive data sets is important in many disciplines, e.g., vehicle simulation, architectural walk-through, computer-aided design and molecular dynamics [2, 3]. It requires that the system must always respond with updated output within a certain small fixed amount of time [58]. They can require hundreds of MFLOPS of floating-point performance and gigabytes per second of memory bandwidth, far beyond the capabilities of a single processor [53]. As a consequence many researchers have looked into speeding up the computation by parallelizing rendering tasks in a distributed computing environment [36, 38, 59, 67]. Parallel graphics architecture is fundamentally a problem in efficiently managing object and image parallelism, and the transition between them. This transition between object and image parallelism is the "sort" in parallel graphics architecture, and creates the necessity for communication in any such architecture [19]. Our choice of a sort-both architecture was driven by our desire for a scalable architecture in distributed computing environments. In this dissertation, we demonstrate that sort-both is a viable architecture providing excellent scalability.

In the next sections, we will look at some issues which are related to our research but out of our focus in this dissertation.

7.1 Discussion

7.1.1 Various Primitives

In this dissertation, our research focuses on polygon primitives because polygon is the most popular primitives in many applications. However, for various reasons, other primitive types are also needed in some applications because of their special advantages. For instance, NURBS can model perfect complex surface patch for CAD/CAM applications. The triangle strip can provide compressed description for geometry data.

The number of triangles is arbitrary in one triangle strip. Therefore, in some cases, a very large database only consist of a few triangle strips. Then, it is difficult to distribute and load-balance this work since the task granularity is too big. The obvious solution to this problem is to make the task granularity smaller. Namely, we can subdivide large triangle strips into a number of smaller triangle strips. The system can partitions and distributes work load based on these smaller triangle strips. Each time a triangle-strip is split, two vertices must be duplicated in order to provide the continuity for the new strip. After subdivision, the triangle strips can work with sort-both as well.

Surface patches use a relatively small number of control points to model complex curved surfaces. Compared with triangle data, surface patches are very economical. 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 [43]. The tessel-

lation process is a complicated factor when surface patches are brought into a parallel graphics system. There are two ways to handle the tessellation and redistribution in parallel rendering systems. A single processor can tessellate and redistribute the tessellation results, polygons. Another method is that the patches can be redistributed to each processor performing the tessellation separately. Each approach has tradeoffs. Tessellating a patch with single processor can reduce the overall amount of computation. However, an expense of additional communication is introduced. Any time the patch be re-tessellated, the polygons have to be re-distributed. This makes the rendering algorithm very expensive. The second method has no such problems since each processor tessellates the patches by itself. This avoids the redistribution of polygons, but increases the total amount of computation required. Another problem is that, in order to avoid cracks or seams cross boundaries, all the processors must be assured to perform the same tessellation algorithm. Since the polygon-distribution information is not available until after tessellation, the second method may not lead to an accurate estimate of the amount of rendering work to be done. Compared with the second method, the first one can be easy to keep load balancing. It can do the tessellation first, then perform the necessary task partitioning based on the generated polygons. In [58], more discussion is presented.

7.1.2 Massive Data Sets

In the absence of shared address space, it can be both difficult and overhead-consuming to manage the partitioning or distribution of the 3D model data among processors and the dynamic replication of data on demand [66]. We have therefore chosen to replicate the entire 3D model on all processing nodes for now, to eliminate the data management

problem and focus on the partitioning issues. However, this clearly restricts our current implementation to rendering static models and limits the size of the models that can be rendered. Since dynamic data distribution and replication will introduce runtime overheads, to build a truly scalable approach, we must examine methods if they can perform data management efficiently.

To solve this problem, Mueller gives a solution by exploring frame to frame coherence in interactive rendering [58]. In interactive applications, it can be expected 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. With this optimization, the number of primitives redistributed can be reduced to a small fraction of the number of on-screen primitives.

Samamta's solution is named k-way replication [66]. The key new idea is to replicate every primitive in a scene k times among n rendering nodes (where $k \ll n$). This approach avoids full replication of the scene data. But, it can employ view-dependent load balancing algorithms, since every primitive is available on more than one processor. With k-way replication, the system performance is similar to n-way replication, but with storage costs closer to 1-way replication.

7.1.3 Immediate-Mode Database

In this dissertation, the discussion is based on the retained-mode database. However, there are certain applications which must take immediate-mode database as input. An example is a scientific visualization application that is generating its graphics data base

in real time when a simulation is progressing.

In a serial immediate-mode application, a single stream of OpenGL commands is passed from the host to the graphics system. Each OpenGL command can be classified into one of the three categories: geometry, state, or special commands, such as SwapBuffers, glFinish, and glClear [36]. In a parallel rendering system, it is obvious that the stream must be subdivided and distributed among the graphics processors for efficient processing. There are two ways to perform sorting and distribution. It can be integrated into the application level. Namely, the data that graphics systems received is already partitioned. Another method is that the application arbitrarily partitions and distributes the command stream to the rendering processors with approximately equal portions of the command stream in each processor. However, the data in stream are not sorted. Then, the rendering processors sort the primitives received based on their location in screen space. Then each primitive is redistributed to the right rendering processor. This method is similar to sort-middle. It introduces a large amount of additional communication cost. These issues are fully explored in [23, 58].

Most of the load-balancing methods suggested for retained-mode database apply equally well to immediate-mode database [58]. When using immediate-mode database, 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, because using the current frame's information would increase latency significantly and require large amounts of buffering space. However, using last frame information may not be able to accurately estimate the distribution information.

Since primitives are always generated newly, they must be completely resorted and

redistributed for each frame. An immediate-mode system will require much more processor-to-processor communications bandwidth than a retained-mode system.

7.1.4 Future Investigation

After many years Moore's law continue to apply to the development of micro-processors. However, in some cases such as GPUs the rate of performance increase is much faster than Moore's law. This provides both challenges as well as opportunities to the research on distributed rendering environments.

Recently, research interest in high resolution displays has led to tiled displays. A tiled display includes a number of standard displays arranged in an array and are configured as one large logical display. Stanford's "Interactive Mural" [37], NCSA's Tiled Wall and Princeton's Scalable Display Wall [10] are examples of such displays. The resolution of tiled displays is usually around 4K x 4K. Such high resolution displays are beyond the capability of being driven by single processor graphics systems. Therefore, distributed rendering systems should provide to be useful for tiled displays. The scalability of the sort-both architecture makes it a suitable candidate for tiled display rendering systems. Tiled display systems can benefit a lot from its small pixel distribution cost. However, tiled display also introduces some constraints, such as fixed tile size. The load-balancing sort-first algorithms are not helpful in this case because of the variable-size screen partitioning. Balancing the workload in tiled display systems is an interesting topic for further discussion.

From single pipeline to multi-pipeline, from PCI (Peripheral Component Interconnect) to AGP (Accelerated Graphics Port), from 1M RAM to 256M RAM, from no

antialiasing to super-sampling antialiasing, the development of the GPU is very significant. Recently, nVidia developed SLI (Scalable Link Interface) which takes advantage of the increased bandwidth of the PCI Express bus architecture. PCI Express is an I/O interconnect bus standard (which includes a protocol and a layered architecture) that expands on and doubles the data transfer rates of the original PCI. An nVidia SLI system with a PCI Express motherboard that supports multiple physical connectors is capable of having multiple nVidia-based PCI Express graphics cards plugged into if. Joined by the nVidia SLI connector, the multiple graphics cards drive a single display. Since the throughput of PCI Express reaches 2.5Gb/s per channel direction, it can speedup the pixel redistribution of the sort-last algorithm. An ideal performance can be achieved by applying sort-both on a cluster of SLI systems. At cluster level, sort-first is applied; then sort-last is applied inside each node of the SLI GPUs.

With the increase of network bandwidth, more and more computing resources can be shared for visualization in Grid environments. Within the next few years, it may be possible that thousands of computing nodes within dozens of cluster connected by gigabit links serve one computational intensive application. What is the optimum graphics architecture for such a huge distributed system? Many factors will have an influence on the choice of architecture, e.g., GPU's pipeline, the shading model. There will be many opportunities for further exploration in this area. Sort-both may be just the first step towards huge distributed rendering systems.

7.2 Conclusion

In this dissertation we have examined the field of parallel computer rendering systems, with focus on the sort-both architecture. Our goal is to find an architecture that is well-suited for handling very high-resolution output, very large data sets and distributed computing resources. Among the choices of sort-first, sort-middle, and sort-last, it is found that none of them is suitable for a distributed rendering system. A hybrid architecture named sort-both is developed which integrated with both sort-first and sort-last strategies. We have demonstrated this thesis by offering a description of how one would efficiently implement sort-first and sort-last strategies in the sort-both architectures and also by providing a comparison between sort-both and competing architectures. In our comparison we briefly looked at homogeneous sort-last. This architecture is desirable for its excellent scalability properties when dealing with large data sets. However, it suffers from bandwidth constraints when one desires to produce high-resolution output. We then examined homogeneous sort-first. With this architecture, pixel communication bandwidth is not a large issue, but redundant rendering could be a potential problem when rendering very large data sets with a large number of rendering processors. This left the door open for sort-both, an architecture that offers lower communication requirements than homogeneous sort-last architectures and lower overlap factor than homogeneous sort-first architectures.

In our search to find an efficient load-balancing algorithm for sort-first, both static and adaptive load-balancing methods are discussed. 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 DPBP, a new adaptive load-balancing algorithm that provides a good work distribution with very low overhead.

Our research on sort-last methods results in a novel sort-last strategy named ILCA. This method partitions the primitives into different image layer volume according to the primitives' distribution along the Z axis. When composing the final image, the image layer are merged together with the alpha value instead of the Z value. In this way, the bandwidth requirements for sort-last composition are reduced greatly. This sort-last method also introduces redundant rendering for the primitives overlap multi image layer volumes. Therefore, a 3D overlap factor is discussed in the sort-both architecture. The result shows that the 3D overlap factor is usually smaller than a 2D overlap factor. Sort-both architecture introduce less redundant rendering than homogeneous sort-first architecture.

Compared with sort-first and sort-last architectures, sort-both is a scalable, flexible fully parallel rendering architecture for high resolution output, very large data sets and distributed computing resources.

Author's Publications

1. H. Zhu, T. K. Y. Chan, L. Wang, and C. R. Jegathese, "A Distributed 3D Rendering Application for Massive Data Sets" IEICE Transaction of Information and Systems. E87-D(7): 1805-1812, 2004.

2. H. Zhu, T. K. Y. Chan, L. Wang, W. Cai, and S. See, "A Distributed Molecular Visualization System Running on Computational Grids" Journal of Future Generation Computer Systems. 20(5): 727-737 2004.

3. H. Zhu, L. Wang, T. K. Y.Chan, W. Cai, and S. See, "A Distributed Rendering Environment for Massive Data on Computational Grids" Procs of IEEE P2P 2003.

4. H. Zhu, T. K. Y. Chan, L. Wang, W. Cai, and S. See, "DPBP:A Sort-First Parallel Rendering Algorithm for Distributed Rendering Environments" Procs of Cyberworld 2003.

5. H. Zhu and T. K. Y. Chan, "A Framework for a Real-time Distributed Rendering Environment" Procs of International Conference on Scientific & Engineering Computation, pp 838-845 2002.

6. H. S. Tan , H. Zhu, and H. Zhou, "V.O.L.U.M.E.: A Web-Centric Database System for Building Interactive 3D Constructivist Learning Environments" Temasek

Engineering Journal, Vol. I, pp. 3-8 2002.

7. H. Zhu, H. S. Tan, and E. H. Tay, " A Database-Supported Visualization for Supply Chain Simulations" Procs of the Global Supply Chain Management Symposium 2001, pp. 36-45 2001.

8. H. Zhu, H. S. Tan, and H. Zhou, "An Internet Database for Supply Chain Visualization" Procs of International Conference on Integrated Logistics, Singapore, pp.123-128 2001.

9. H. S. Tan, H. Zhu, and H. Zhou, "Knowledge Construction in Education: A Web-database for building Interactive 3D Environments" Procs of Asia Pacific Advanced Network Conference 2001, pp. 120-124 2001.

10. H. Zhu, H. S. Tan, and H. Zhou , "The Combination of Internet Database and Virtual Environment for Manufacturing Visualisation" Procs of the Automation Science and Technology Application Conference 2001, pp. 85-91 2001.

Zhu Huabing (Mr.)

Research Student (Ph.D candidate)

School of Computer Engineering

Nanyang Technological University

Bibliography

[1] K. Akeley. Realityengine graphics. *Computer Graphics (SIGGRAPH 93 Proceedings)*, (27):109–116, 1993.

[2] D. Aliaga, J. Cohen, A. Wilson, E. Baker, H. Zhang, C. Erikson, K. Hoff, T. Hudson, W. Stuerzlinger, R. Bastos, M. Whitton, F. Brooks, and D. Manocha. Mmr: An integrated massive model rendering system using geometric and image-based acceleration. In *Proceedings of the Symposium on Interactive 3D Graphics (I3D)*, pages 199–206, 1999.

[3] D. Aliaga, J. Cohen, A. Wilson, H. Zhang, C. Erikson, K. Hoff, T. Hudson, W. Stuerzlinger, E. Baker, R. Bastos, M. Whitton, F. Brooks, and D. Manocha. A framework for the real-time walkthrough of massive models. Technical Report TR 98-013, UNC, March 1998.

[4] J. Arvo and D. Kirk. *A Survey of Ray Tracing Acceleration Techniques*, chapter 6 in An Introduction to Ray Tracing. Academic Press, New York, 1989.

[5] I. Buck, G. Humphreys, and P. Hanrahan. Tracking graphics state for networked rendering. In *Proceedings of the 2000 Eurographics/SIGGRAPH Workshop on Graphics Hardware.*, pages 87–95, Interlaken, Switzerland, 2000.

[6] M. Bues, A. Hinkenjann, T. Olry, and S. Schupp. Attacking the network bottleneck of parallel rendering with commodity hardware. In *Proceedings of 6th Symposium on Virtual Reality*, Ribeirao Preto, Brazil,, 2003.

[7] P. Bui-Tong. Illumination for computer generated pictures. *Communications of the Association for Computing Machinery*, 18(6):311–317, 1975.

[8] A. Burke and W. Leler. Parallelism and graphics: An introduction and annotated bibliography. In S. Whitman, editor, *SIGGRAPH Course Notes: Parallel Algorithms and Architectures for 3D Image Generation*, pages 111–140, 1990.

[9] H. Chen, Y. Chen, A. Finkelstein, T. Funkhouser, K. Li, Z. Liu, R. Samanta, and G. Wallace. Data distribution strategies for high-resolution displays. *Computers & Graphics*, 25(5), Oct 2001.

[10] H. Chen, R. Sukthankar, G. Wallace, and K. Li. Scalable alignment of large-format multi-projector displays using camera homography trees, 2002.

[11] M. Chen, G. Stoll, H. Igehy, K. Proudfoot, and P. Hanrahan. Simple models of the impact of overlap in bucket rendering. In *Proceedings of 1998 Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 105–112, 1998.

[12] J. H. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, 1976.

[13] W. Corra, J. Klosowski, and C. Silva. Out-of-core sort-first parallel rendering for cluster-based tiled displays. In *Proceedings of Eurographics PGV 2002*, Sept 2002.

[14] M. Cox. Architectural implications of hardware-accelerated bucket rendering on the pc. In *Proceedings of 1997 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 25–34, 1997.

[15] T. W. Crockett. Design considerations for parallel graphics libraries. Nasa cr-194935, Institute for Computer Applications in Science and Engineering, 1994.

[16] T. W. Crockett. Parallel rendering. In *SIGGRAPH Course Notes: Parallel Graphics and Visualization Technology*, 42, pages 150–207, 1998.

[17] T. W. Crockett and T. Orloff. A mimd rendering algorithm for distributed memory architectures. In *Proceedings of Parallel Rendering Symposium, ACM Press*, pages 35–42, 1993.

[18] M. Eldridge. *Designing Graphics Architectures around Scalability and Communication*. PhD thesis, Stanford university, 2001.

[19] M. Eldridge, H. Igehy, and P. Hanrahan. Pomegranate: A fully scalable graphics architecture. *Computer Graphics (Proceedings of ACM SIGGRAPH 2000)*, 2000.

[20] D. A. Ellsworth. A new algorithm for interactive graphics on multicomputers. *Computer Graphics and Applications*, 14(4):33–40, July 1994.

[21] D. A. Ellsworth. *Polygon Rendering for Interactive Visualization on Multicomputers*. PhD thesis, University of North Carolina at Chapel Hill, 1996.

[22] J. Eyles, S. Molnar, J. Poulton, T. Greer, A. Lastra, N. England, and L. Westover. Pixelflow: The realization. In *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 57–68, Los Angeles USA, 1997.

[23] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, Reading, Massachusetts, 2nd edition, 1990.

[24] MPI Forum. *The Message Passing Interface (MPI) Standard*. http://www-unix.mcs.anl.gov/mpi/.

[25] I. Foster, J. Insley, G. von Laszewski, C. Kesselman, and M. Thiebaux. Distance visualization: Data exploration on the grid. *IEEE Computer Magazine*, 32(12):36–43, 1999.

[26] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, Los Altos, CA, 1999.

[27] H. Fuchs, J. Goldfeather, J. Hultquist, S. Spach, J. Austin, F. Brooks, J. Eyles, and J. Poulton. Fast spheres, shadows, textures, transparencies, and image enhancements in pixel-planes. *Computer Graphics (Proceedings of SIGGRAPH 85)*, 19(3):111–120, July 1985.

[28] H. Fuchs and B. Johnson. An expandable multiprocessor architecture for video graphics. In *Proceedings of 6th ACM-IEEE Symposium on Computer Architecture*, pages 58–67, 1979.

[29] H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, and L. Israel. Pixel-planes 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories. *Computer Graphics (Proceedings of SIGGRAPH 89)*, 23(3):79–88, 1989.

[30] P. N. Glaskowsky. Advanced 3D chips show promise. *Microprocessor Report*, 11(8):5–7, 1997.

[31] S. Gottschalk, M. Lin, and D. Manocha. Obbtree: A hierarchical structure for rapid interference detection. *Computer Graphics (Proceedings of SIGGRAPH 96)*, pages 171–180, 1996.

[32] H. Gouraud. Continuous shading of curved surfaces. *IEEE Transactions on Computers*, C-20(6):623–629, 1971.

[33] A. Hinkenjann and M. Bues. Network aware parallel rendering with pcs. In *Proceedings of ACM SIGGRAPH International Conference on Virtual Reality Continuum and its Applications in Industry*, pages 33–37, Singapore, 2004.

[34] A. Hinkenjann, M. Bues, T. Olry, and S. Schupp. Mixed-mode parallel real-time rendering on commodity hardware. In *Proceedings of 5th Symposium on Virtual Reality*, Fortaleza, Brazil, 2002.

[35] G. Humphreys, I. Buck, M. Eldridge, and P. Hanrahan. Distributed rendering for scalable displays. In *IEEE Proceedings of Supercomputing 2000*, Oct 2000.

[36] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan. Wiregl: A scalable graphics system for clusters. *ACM Trans. Computer Graphics (Proceedings of SIGGRAPH 2001)*, pages 129–140, 2001.

[37] G. Humphreys and P. Hanrahan. A distributed graphics system for large tiled displays. In *Proceedings of IEEE Visualization'99*, 1999.

[38] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. *ACM Trans. Computer Graphics (Proceedings of SIGGRAPH 2002)*, 21(3):693–702, 2002.

[39] H. Igehy. *Scalable Graphics Architectures: interface & architecture*. PhD thesis, Standard Univeristy, 2000.

[40] H. Igehy, G. Stoll, and P. Hanrahan. The design of a parallel graphics interface. *Computer Graphics(Proceedings of SIGGRAPH 98)*, pages 141–150, July 1998.

[41] M. Kaplan and D. P. Greenberg. Parallel processing techniques for hidden surface removal. *Computer Graphics (SIGGRAPH 79)*, 13(2):300–307, July 1979.

[42] N. Karonis, B. Toonen, and I. Foster. Mpich-g2: A grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing.*, 63(5):551–563, 2003.

[43] S. Kumar. *Interactive Rendering of Parametric Spline Surfaces*. PhD thesis, University of North Carolina at Chapel Hill, 1996.

[44] R. Latham. Advanced image generator architectures. In *Image VII conference tutorial*, June 1994.

[45] T. Y. Lee, C. S. Raghavendra, and J. B. Nicholas. Image composition schemes for sort-last polygon rendering on 2d mesh multicomputers. *IEEE Transactions on Visualization and Computer Graphics*, 2(3):202–217, 1996.

[46] K. Li, H. Chen, Y. Chen, D. W. Clark, P. Cook, S. Damianakis, G. Essl, A. Finkelstein, T. Funkhouser, A. Klein, Z. Liu, E. Praun, R. Samanta, B. Shedd, J. P. Singh, G. Tzanetakis, and J. Zheng. Early experiences and challenges in building and using a scalable display wall system. *IEEE Computer Graphics and Applications*, 20(4):671–680, 2000.

[47] L.McMillan. *An Image-Based Approach to Three-Dimensional Computer Graphics*. PhD thesis, University of North Carolina at Chapel Hill, 1997.

[48] K. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh. Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics and Applications*, 14(4):59–68, July 1994.

[49] T. Mitra and T. Chiueh. A breadth-first approach to efficient mesh traversal. In *Proceedings of 13th ACM SIGGRAPH/Euro-graphics Graphics Hardware Workshop*, August 1998.

[50] T. Mitra and T. Chiueh. Implementation and evaluation of parallel mesa library. In *Proceedings of IEEE International Conference on Parallel and Distributed Systems*, December 1998.

[51] T. Mitra and T. Chiueh. Dynamic 3D graphics workload characterization and the architectural implications. In *Proceedings of 32nd ACM/IEEE Annual International Symposium on Microarchitecture*, November 1999.

[52] T. Mitra and T. Chiueh. Three dimensional graphics architecture. *Current Science: Special Section on Computational Science*, 78(7):101–109, April 2000.

[53] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications: Special Issue on Rendering*, 14(4):23–32, 1994.

[54] S. Molnar, J. Eyles, and J. Poulton. Pixelflow: High-speed rendering using image composition. *Computer Graphics (Proc. SIGGRAPH 92)*, 26(2):231–240, 1992.

[55] J. S. Montrym, D. R. Baum, D. L. Dignam, and C. J. Migdal. Infinitereality: A real-time graphics system. *Computer Graphics (SIGGRAPH 97)*, pages 293–302, Aug.

[56] K. Moreland, B. Wylie, and C. Pavlakos. Sort-last parallel rendering for viewing extremely large data sets on tile displays. In *Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphics*, pages 85–92, San Diego, California USA, 2001. IEEE Press.

[57] C. A. Mueller. The sort-first rendering architecture for high-performance graphics. In *Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 75–85, Monterey, California, United States, 1995. ACM Press.

[58] C. A. Mueller. *The Sort-First Architecture for Real-Time Image Generation*. PhD thesis, University of North Carolina at Chapel Hill, 2001.

[59] T. D. Nguyen, C. Peery, and J. Zahorjan. Drrrraw: A prototype distributed 3D real-time rendering toolkit for commodity clusters. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, May 2001.

[60] T. D. Nguyen and J. Zahorjan. Image layer decomposition for distributed rendering on nows. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, May 2000.

[61] M. Olano and A. Lastra. A shading language on graphics hardware: The pixelflow shading system. *Computer Graphics (Proc. of SIGGRAPH 98)*, July 1998.

[62] F. A. Ortega, C. D. Hansen, and J. P. Ahrens. Fast data parallel polygon rendering. In *Proceedings of Supercomputing93*, page 709C718, Portland, Oregon, Nov. 1993.

[63] P. Padmos and M. Milders. Checklist for outside-world images of simulators. In *Proceedings of International Training Equipment Conference and Exhibition (ITEC)*, pages 2–14, Luxembourg, April 1992.

[64] M. Regan, G. Miller, S. Rubin, and C. Kogelnik. A real-time low-latency hardware light-field renderer. *Computer Graphics (SIGGRAPH 99 Proceedings)*, 33:287–290, 1999.

[65] D. R. Roble. A load balanced parallel scanline z-buffer algorithm for the ipsc hypercube. In *Proceedings of Pixim 88*, pages 177–192, 1988.

[66] R. Samanta, T. Funkhouser, and K. Li. Parallel rendering with k-way replication. In *Proceedings of IEEE 2001 Symposium on Parallel and Large-Data Visualization and Graphics*, San Diego, USA, October 2001.

[67] R. Samanta, T. Funkhouser, K. Li, and J. Pal Singh. Hybrid sort-first and sort-last parallel rendering with a cluster of pcs. In *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, Interlaken, Switzerland, August 2000.

[68] R. Samanta, J. Zheng, T. Funkhouser, K. Li, and J. P. Singh. Load balancing for multi-projector rendering systems. In *Proceedings of 1999 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, Los Angelos, California, August 1999.

[69] M. Segal and K. Akeley. *The OpenGL Graphics System: A Specification(version 1.4)*. OpenGL, 2003. Editor: Jon Leech.

[70] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker. A characterization of ten hidden-surface algorithms. *ACM Computing Surveys*, 6(1):1–55, 1974.

[71] J. Torborg and J. T. Kajiya. Talisman: Commodity realtime 3D graphics for the pc. In *Proceedings of SIGGRAPH 96*, pages 353–363, August 1996.

[72] T. van der Schaaf, L. Renambot, D. Germans, H. Spoelder, and H. Bal. Retained mode parallel rendering for scalable tiled displays. In *Proceedings of 6th annual Immersive Projection Technology (IPT) Symposium.*, Orlando Florida, March 2002.

[73] A. Watt and M. Watt. *Advanced Animation and Rendering Techniques*. ACM Press and Addsion-Wesley, 1992.

[74] D. S. Whelan. *Animac: A Multiprocessor Architecture for Real-time Computer Animation*. PhD thesis, California Institute of Technology, 1985.

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

[76] S. Whitman. Dynamic load balancing for parallel polygon rendering. *IEEE Computer Graphics and Applications*, 14(4):41–48, July 1994.

[77] J. Yang, J. Shi, Z. Jin, and H. Zhang. Design and implementation of a large-scale hybrid distributed graphics system. In *Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization*, pages 39–49, 2002.