RAY COHERENCE IN PARALLEL VOLUME RENDERING

by

Brendan Moloney

B.Sc. Cum Laude, University of Arizona, 2005

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
Summer 2008

APPROVAL

Name:	Brendan Moloney
Degree:	Master of Science
Title of thesis:	Ray Coherence in Parallel Volume Rendering
Examining Committee:	Dr. Richard Zhang Chair
	Dr. Torsten Möller, Senior Supervisor Associate Professor, Computing Science
	Dr. Daniel Weiskopf, Co-Supervisor Professor, Computer Science Universität Stuttgart, Germany
	Dr. Alexandra Fedorova, SFU Examiner Assistant Professor, Computing Science
Date Approved:	June 24, 2008

Abstract

Ray coherence, meaning all processing along each ray is local to a single machine, is achieved in our parallel volume rendering environment by using a workload distribution scheme that divides the image space. This allows one to avoid the compositing stage when performing standard volume rendering in a parallel rendering pipeline. More importantly, there are a number of existing algorithms for volume rendering that either benefit from or require ray coherence when being adapted to a parallel environment. We discuss several of these algorithms and adapt and implement two of them, our own improved visibility culling technique to speed up rendering when occlusion occurs and a volumetric shadowing technique that produces more realistic and informative images. We also present novel algorithms for providing a consistent load balancing and efficiently loading and rendering pieces of a subdivided data set, addressing two of the major issues for data scalable image space distributions.

Keywords: distributed computing; sort first parallelization; volume rendering; visualization; load balancing; occlusion; shadow; ray coherence

Subject Terms: computer graphics; parallel processing (electronic computers); parallel algorithms; three-dimensional imaging; high performance computing; visualization data processing

To Elizabeth, for her unwavering support.

"Man's mind, once stretched by a new idea, never regains its original dimensions."

— *Oliver Wendell Holmes*

Acknowledgments

Firstly, I must thank my two senior supervisors Torsten Möller and Daniel Weiskopf. Like the proverbial good cop bad cop duo, they simultaneously encouraged me to explore unknown territory while reminding me of the need to produce some practical and precise results. They have also shown me that it is possible to know more than a little about a lot of different subjects, inspiring me to embiggen my breadth of knowledge.

Members of the GrUVi lab, both past and present, have been a great source of entertainment and enlightenment. All of the professors that I have interacted with (whether through a class, a teaching assistant position, or just casually) have been both friendly and helpful.

Most importantly, I must thank my parents for giving me the encouragement and opportunities that have gotten me to where I am today.

This work was funded in part by the Natural Sciences and Engineering Research Council (NSERC) of Canada. I would like to thank the following sources for data sets used in the illustrations and experiments:

- Dr. Christof Rezk-Salama, University of Siegen, Germany and Dr. Michael Scheuering, Siemens Medical Solutions, Forchheim, Germany for the fish data set.
- Brown & Herbranson Imaging, Stanford Radiology, and The Rosicrucian museum, for the mummy data set.
- Lawrence Livermore National Laboratory for the Richtmyer-Meshkov data set.
- The National Library of Medicine for the visible human data set.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Volume rendering is characterized by the type of data it works on: a continuous field (usually interpolated from samples) over a three dimensional space. Visualizing such data poses a number of challenges, from providing an effective visual mapping to harnessing enough computational power to accurately render the data at an interactive frame rate. The focus of this thesis is on the latter problem, specifically on how to use multiple processing units to interactively render data sets that are too large or require too much processing for a single unit.

The partitioning strategy used to distribute the rendering workload and data set among the processing units can limit the types of algorithms that are applicable within the parallel rendering environment. In particular, many image space algorithms cannot be efficiently adapted to work with a partitioning strategy that does not keep the data and processing along each ray local to a single processing unit. In this thesis we explore the advantages and challenges of utilizing a number of such algorithms in a parallel environment.

1.1 Motivation

Many scientific simulations and measurements result in enormous volumetric data sets. Volume rendering is an essential tool for visualizing and gaining insight from such data. The process of exploring volumetric data can also benefit greatly from volume rendering, but only if the user can interactively alter the viewing conditions.

To perform interactive volume rendering, even of small data sets, requires a tremendous amount of computational power. In the past this has been the domain of super computers

– utilizing many processors in a parallel environment. More recently Graphics Processing Units (GPUs) have provided a cost efficient method of rendering small to medium sized data sets at interactive frame rates. This has allowed single workstations to do the tasks once reserved for super computers. However, larger data sets still require more memory and processing power than a single GPU can provide. Thus we have come full circle in the sense that we must once again use multiple processing elements, only now they are often GPUs instead of CPUs.

All high performance GPUs have their own dedicated high speed memory to maximize the bandwidth available to the processing core. In a parallel environment, this extra layer of memory further complicates the problem of distributing the data set while simultaneously distributing the rendering workload evenly. Regardless of this complication, the focus of this thesis is on multi-GPU systems.

There are two main reasons for this choice. First, the computational power provided by GPUs is unrivaled at their price point. As of early 2008, a top of the line dual core GPU has a theoretical limit of approximately one TFLOPS and costs under 500 US$. In comparison, a top of the line quad core CPU has a theoretical limit of only 48 GFLOPS at a cost of over 1000 US$. The second reason is that the computational power of GPUs is growing significantly faster than that of CPUs, and this trend shows no signs of abating.

While much work has been done on volume rendering with multiple GPUs. algorithms which require or benefit from data locality along a ray have been under utilized. These algorithms can provide tremendous speed ups through visibility culling, more informative images through sophisticated lighting models that include shadowing effects, more accurate and consistent load balancing, and potentially many other benefits. The downside is the need to load data during the rendering process when the data set is too large to be replicated across all of the processing units.

1.2 Volume Rendering Preliminaries

Preliminary work on volume rendering came from attempts to model clouds and other gaseous phenomena [5, 30]. Later work began to focus on volume rendering as a tool for visualizing scientific data sets [10, 60]. Regardless of the application, the foundation is the same.

The propagation of light through participating media is a subset of the field of radiative

transport theory [8]. When modeling natural phenomena it may be important to model complex effects such as fluorescence and Rayleigh scattering. For visualizing scientific data sets the effects are usually limited to emission, absorption, and some limited form of scattering.

In this section we give a high level overview of the fundamentals of volume rendering. For a more in depth discussion of this topic we refer our readers to the work of Hege [14].

1.2.1 The Volume Rendering Integral

Conceptually, we can think of a volume as a particle cloud where each particle can emit, absorb, and scatter (reflect) light. The scattering incidents can be further classified, based on whether they reflect light along a ray toward the eye (in-scattering), or reflect light away from a ray toward the eye (out-scattering). In the most precise terms, this would include global illumination and shadowing effects. In practice, in-scattering is often limited to first hit reflections from external sources towards the eye, and out-scattering is often ignored completely.

Radiance is the fundamental measure used for computing light transport, due largely to the fact that it is constant along the length of a ray (through a vacuum). Radiance is defined as the radiative energy Q per unit projected area $A_\perp$ per solid angle Ω per unit of time t:

$$L = \frac{dQ}{dA_\perp d\Omega dt} \tag{1.1}$$

Volumetric data is usually defined by a set of interpolated functions over a three dimensional domain. Each function gives us some property of the particle cloud. For example, we could have a function $L(r)$ which gives the amount of radiance being emitted along the direction of the ray r for each portion of the volume. The integral equation for the total radiance accumulated between two points on the ray, r_1 and r_2, is given in Equation 1.2.

$$L_r = \int_{r_1}^{r_2} L(r)dr \tag{1.2}$$

This emission only model is quite efficient to compute, but lacks the depth cues provided by attenuation along the ray towards the eye. When a ray of light travels through a cloud of particles, the amount of light that makes it to the other side depends on the distance the ray travels through the cloud, the density of the particles, and the ability of the particles to reflect and absorb light. For volume rendering, the latter two parameters can be combined

into one function $\kappa(r)$ which describes the opacity at each point along a ray. The change in radiance along a ray with an absorption only model can then be expressed as:

$$dL = -\kappa(r)Ldr \tag{1.3}$$

Which we can rearrange and integrate between the points r_1 and r_2 to get the final radiance L_{r2} in terms of the initial radiance L_{r1} and the opacity function $\kappa(r)$:

$$\int_{r_1}^{r_2} \frac{dL}{L} = \int_{r_1}^{r_2} -\kappa(r)dr \tag{1.4}$$

$$\ln\left(\frac{L_{r2}}{L_{r1}}\right) = \int_{r_1}^{r_2} -\kappa(r)dr \tag{1.5}$$

$$L_{r2} = L_{r1}e^{\int_{r_1}^{r_2} -\kappa(r)dr} \tag{1.6}$$

The negative of the exponent of the last term in Equation 1.6 is often called the optical depth. We abbreviate the optical depth between two points r_1 and r_2 as:

$$\tau(r_1, r_2) = \int_{r_1}^{r_2} \kappa(r)dr \tag{1.7}$$

We can then express the emission and absorption model for volume rendering as in Equation 1.8. This is what is called the Volume Rendering Integral. Although it does not explicitly contain terms for scattering, the in-scattering and out-scattering effects can be added into the emission function $L(r)$. In this equation the eye point would lie at the point r_2 and the initial radiance from behind the volume is given by L_{r1}.

$$L_{r2} = L_{r1}e^{-\tau(r_1,r_2)} + \int_{r_1}^{r_2} L(r)e^{-\tau(r,r_2)}dr \tag{1.8}$$

1.2.2 Compositing Approximation

For all but a few special cases, Equation 1.8 cannot be solved analytically and thus numerical methods must be used. For an emission absorption model without scattering, simple ray integration will suffice. We must discretize the ray r into a number of intervals in order to perform the numerical integration. If we discretize r into n intervals, then we can compute the radiance at position r_k from the radiance at r_{k-1} as follows:

$$L_{r_k} = L_{r_{k-1}}e^{-\tau(r_{k-1},r_k)} + \int_{r_{k-1}}^{r_k} L(r)e^{-\tau(r,r_k)}dr \tag{1.9}$$

We define two useful abbreviations θ_k and β_k:

$$\theta_k = e^{-\tau(r_{k-1}, r_k)} \tag{1.10}$$

$$\beta_k = \int_{r_{k-1}}^{r_k} L(r) e^{-\tau(r, r_k)} dr \tag{1.11}$$

The term θ_k is the transparency between two sample points. Transparency is a value between zero (when the optical depth is infinity) and one (when the optical depth is zero). Both θ_k and β_k are generally approximated by assuming they are constant in the vicinity of a sample or linearly varying from one sample to the next. Using these terms we construct a discrete recursive description of the volume rendering integral (for simplicity, we assume r_n is the sample closest to the eye) in Equation 1.12. For the base case β_0 equals the initial radiance from behind the volume and θ_0 is zero.

$$L_r(r_n) = L_r(r_{n-1})\theta_n + \beta_n = \sum_{k=0}^{n} \beta_k \prod_{j=k+1}^{n} \theta_j \tag{1.12}$$

If we consider the inverse of transparency, opacity $\alpha_k = 1 - \theta_k$, and we note that β_k is equivalent to a pre-multiplied color c_k (ignoring the intricacies of mapping a spectral power distribution to a color) then we can reformulate this equation in terms of the alpha compositing over operator [43] as follows:

$$L_r(r_n) = \sum_{k=0}^{n} c_k \prod_{j=k+1}^{n} (1 - \alpha_j) = c_n \text{ over } c_{n-1} \text{ over } c_{n-2} \ldots \text{over } c_1 \text{ over } c_0 \tag{1.13}$$

1.2.3 Classification

Generally the data has no intrinsic optical properties, and thus we must assign colors and opacity values based on some attributes of the data. The mapping of data attributes to optical properties is called classification, and the function which defines this mapping is called the transfer function.

Using an appropriate transfer function is vital to obtaining an informative rendering. The goal when designing a transfer function is to highlight the regions of interest while minimizing occlusion of such regions. Achieving this goal is rarely easy, and often unintuitive.

The most common scenario is a data set represented by a scalar field and a one dimensional transfer function which assigns optical properties to each scalar value. Recently

there has been research into multi-dimensional transfer functions which also classify based on the gradients of the scalar field [20]. This approach can help classify boundaries between materials, but at the cost of added complexity both in transfer function design and rendering.

Generally the transfer function is user specified, though there has been some work on automating the process to some degree [18]. Interactive editing of the transfer function is a crucial part of effectively exploring volume data sets; often the user will not know what transfer function is appropriate for visualizing a structure until they get some visual feedback.

Even with an appropriate transfer function, there are a number of factors that affect the quality of the rendered image. One important factor is the stage in the rendering pipeline in which the classification occurs. If the classification occurs before interpolation (pre-classification) then the resulting image can be blurry and inaccurate when compared to classification after data interpolation (post-classification).

Another factor is the bandwidth of the transfer function. The maximum frequency of a data set with a transfer function applied is the product of the maximum frequency of transfer function and the maximum gradient of the data set [3]. The explosion of the Nyquist rate when using high frequency transfer functions can be avoided by using pre-integrated transfer functions [12]. The idea is to compute the volume rendering integral for all combinations of scalar values over the length of one sampling distance. This removes the dependency on the Nyquist rate of the transfer function, at the cost of losing some interactivity when editing the transfer function.

1.2.4 Partitioning Strategy for Parallel Volume Rendering

There are two main reasons for using multiple processing units to render volumetric data. The first is that the amount of processing required might take too long to achieve interactive frame rates, and the second is that the data itself might be too large to fit into the local memory of a single unit. Parallel workload distributions that address the former issue can be called "performance scaling" and distributions that address the latter issue can be called "data scaling". Often it is difficult to balance both of these goals reliably for all viewing conditions.

A variety of methods have been proposed for distributing a rendering workload among a number of machines. Molnar et al. [31] classify these into groups based on where in

the rendering pipeline primitives are sorted in regards to viewing conditions. In sort first methods the screen space is divided into regions and object space primitives are sorted into these regions and distributed before rendering. In sort last methods the object space primitives are distributed, processed, and rasterized independently. Then overlapping pixels are sorted in depth order and composited together. For the special case of volume rendering, the object space primitives are arrays of volume data. These arrays generally require little to no processing before rendering, but they do have significant storage requirements.

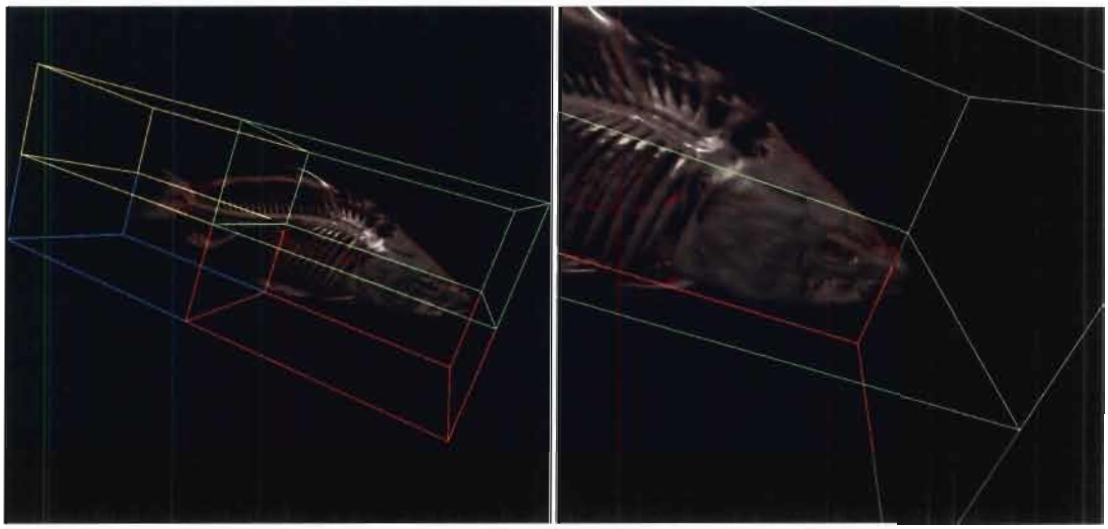

Figure 1.1: An example of an object space distribution scheme with four nodes. The left image shows a global view of the data set with each node using a different color when rendering the bounding box of their respective portions of the object space. The right image shows a zoomed in view which illustrates the problem of load balancing with a static object space distribution (only the green and red nodes are doing work).

Recently there has been a focus on sort last distributions due to their ideal data scaling. Even with a simple static distribution such an approach provides very good data scaling and reasonable performance scaling when the data set is viewed globally. However, as illustrated in Figure 1.1, once the user starts to zoom in to look at smaller features of the data set a static data distribution is no longer sufficient. Additionally, the performance scaling of such a distribution can often be hampered by the need to transmit and blend intermediate images over the network for the alpha compositing process.

A sort first distribution does not need to alpha composite intermediate images, and thus can provide better performance scaling in certain scenarios. The main drawback to sort

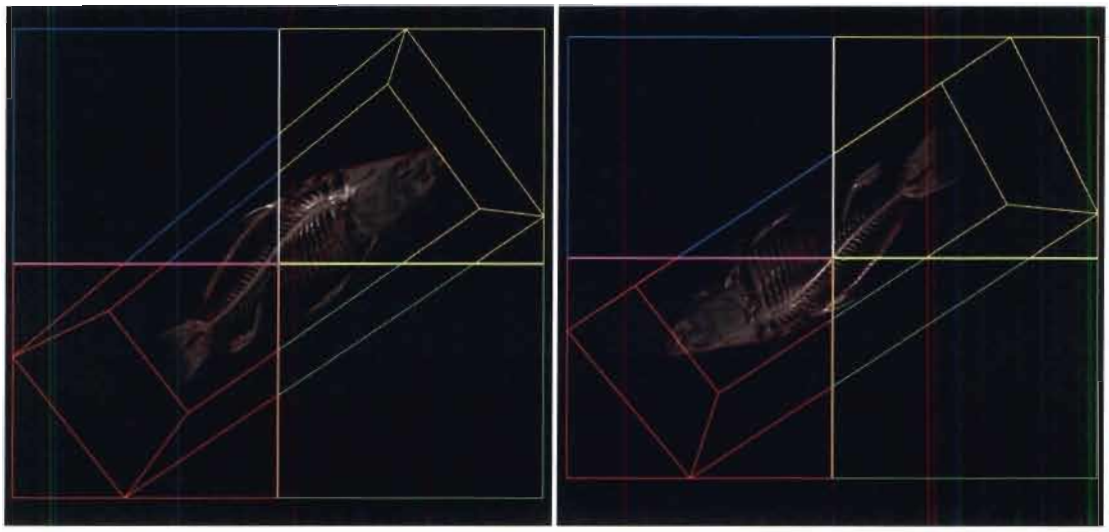

Figure 1.2: An example of an image space distribution scheme with four nodes. Each node colors their image space bounding rectangles and the bounding box of the volume with a different color. For the two viewpoints used to make the images, the data that each node needs to render is completely different.

first approaches is the difficulty of achieving data scalability. As illustrated in Figure 1.2, for different viewing conditions each node may require completely different parts of the data to render their respective portions of the image space. While the total size of the volume data is always going to be much larger than the size of the image data, as long as we have frame-to-frame coherence the amount that needs to be loaded on any single frame can be quite small. It is also possible to cache and asynchronously load the volume data, while the compositing has to be done after rendering and before sending the final results to the display unit.

With a sort first distribution, all the processing and information along a ray is local to a single unit. When rays are local to a single machine we say they are coherent. If view rays from the eye position are coherent then the compositing stage of the parallel rendering pipeline can be skipped and we can perform accurate visibility culling. If shadow rays along the light direction are coherent then interactive volumetric shadowing can be achieved. In fact, the ray coherence provided by sort first distributions could play a key role in adapting many volume rendering algorithms to work efficiently in a parallel environment.

1.3 Goals and Contributions

The majority of the state of the art research on parallel volume rendering using multiple GPUs has focused on sort last decompositions. Our goal is to show that sort first methods cannot only achieve data scalability at a cost that is often less than the cost of alpha compositing in sort last methods, but they can also allow for efficient parallelization of a number of existing volume rendering algorithms that would otherwise be intractable. The reason for these benefits is the property of ray coherence, which can often eliminate some of the communication and synchronization that would otherwise be required by the algorithms.

In Chapter 5 we discuss a number of volume rendering algorithms that would benefit from using a parallel workload distribution that provides ray coherence. We implement and discuss two of the algorithms in great detail. The first algorithm we implement is an improvement on an existing method of doing occlusion culling on the GPU. Our improved algorithm incurs almost no overheads when there is little occlusion while achieving superior performance when occlusion does occur. More importantly, we show that the culling efficiency of a sort first parallel distribution greatly outperforms a sort last distribution due to ray coherence. The second algorithm we implement uses a hybrid sort first and sort last distribution which allows us to adapt an image based volumetric shadowing algorithm to a parallel environment. We achieve ray coherence along the shadow rays by performing a sort first distribution of the light's image space. This essentially gives us a sort last distribution from the camera's point of view, which requires us to composite the intermediate images in order to get the final result.

We also address three of the major challenges to the scalability of sort first methods. A novel load balancing algorithm that gives a consistently well balanced distribution, even without frame-to-frame coherence, is provided in Chapter 4. The increased accuracy and consistency compared to existing load balancing algorithms allows for larger data sets to be rendered using a data scalable sort first approach. We introduce a technique which uses slice templates to eliminate a major bottleneck for slice based rendering on subdivided data sets in Chapter 3. This allows us to subdivide our data into smaller pieces which reduces the amount of data that is replicated among processing units and allows us to perform more accurate culling of empty portions of the classified data. Finally, in Chapter 3 we also explore a proximity based caching scheme that reduces sudden dips in performance associated with data loading.

On top of these algorithmic contributions, a detailed performance analysis of each stage in the parallel rendering pipeline on an inexpensive cluster of workstations is given in Chapter 6. For each stage of the pipeline, we compare the advantages and disadvantages of distributing the image space versus the object space. Most importantly, we expose the utility of a sort first distribution for not only standard volume rendering but also adapting many existing algorithms to a parallel environment. In Chapter 7 we draw our final conclusions and highlight possibilities for future research.

Chapter 2

Previous Work

This chapter provides a review of previous literature related to the work presented in later chapters. Section 2.1 gives an overview of some methods used to evaluate the volume rendering integral. Section 2.2 reviews the state of the art in interactive volume rendering on a GPU. Finally, Section 2.3 explores previous work on rendering in a parallel environment.

2.1 Evaluating the Volume Rendering Integral

There are two general classes of algorithms for evaluating the volume rendering integral, categorized by their method of processing the data. Image order algorithms, i.e. ray casting, process the data along rays originating from the eye point (or rays along the view direction for orthographic projections). Object order algorithms are flexible in the order they process data, the only restriction being that they use a valid compositing order to combine results from the object space primitives.

2.1.1 Image Order Algorithms

The most intuitive manner of evaluating the volume rendering integral is through ray casting. For each pixel, one or more rays are cast into the volume, each one evaluating the volume rendering integral as it takes discrete steps through the scalar field. Any known filter can be used to reconstruct the field, but when interactivity is desired trilinear interpolation is most common.

A number of acceleration schemes [25] have been devised to speed up this process,

including methods of minimizing the processing of parts of the volume that are transparent (commonly referred to as empty space leaping) or occluded (commonly referred to as early ray termination). Even with such techniques, large data sets require many CPUs and a high speed interconnect to reach interactive frame rates.

2.1.2 Object Order Algorithms

Object order algorithms are useful due to their additional flexibility in terms of the order in which the data is processed. This additional flexibility can allow for better performance or more complicated light transport models.

Early object order algorithms such as splatting [58, 59] and shear-warp factorization [24] achieved better performance than ray casting but at the cost of image fidelity. Revised versions of these algorithms [53, 35] attempted to increase image fidelity while maintaining the advantage in performance. For direct rendering of an irregular mesh, cell projection [48] is often used but it can be hindered by an expensive visibility sorting stage.

Slicing algorithms sample the volume along one planar slice at a time. If the slices are view aligned then the resulting image will be equivalent to one produced through ray casting with an orthogonal projection. For a perspective projection the spacing between slices along the rays varies with the angle between the rays and the normal of the slices. This can cause rendering artifacts (although often unnoticeable) if the opacity is not corrected for the variation in distance between samples. Slicing was designed as a means of computing the volume rendering integral on a GPU. The tremendous throughput afforded by GPUs allowed for an order of magnitude increase in performance.

2.2 GPU Accelerated Volume Rendering

Modern GPUs have the ability to load data into textures, and then process that data concurrently using many vector units. In essence they are the rebirth of vector processors, but the important difference is that the boom of the video game industry has driven GPU prices down well below prices of traditional CPUs. The cost of all this processing power is a restricted programming model that requires new approaches to implementing some existing algorithms.

On GPUs, three dimensional textures provide hardware accelerated trilinear interpolation of a scalar field at the flip of a switch. Some kind of proxy geometry can then be

rendered to sample the texture. When the geometry is rendered, each vertex is passed into a vertex shader program which can manipulate the position and other attributes before passing the vertex down the pipeline. Next the polygons are rasterized which generates a bunch of fragments (pixels with additional information such as depth and texture coordinates). For each fragment a fragment shader program is executed which can access textures and fragment attributes and compute a color for that fragment. Finally, in the blending stage the fragments that are output can then be composited with values already in the frame buffer. A more detailed look at the GPU pipeline can found in Appendix A.3.

Three dimensional texture slicing [9, 6] is an easy way to interactively perform a high quality volume rendering of small to medium sized data sets. Slices are drawn so that they are parallel to the view plane in front-to-back or back-to-front order, sampling the texture at those locations and then being composited into the frame buffer.

For older GPUs without support for three dimensional textures, two dimensional textures that correspond to slices of the data can be used. In two dimensional slicing there are three fixed sets of slices, one for each orthogonal axis. At render time the set of slices corresponding to the axis most aligned with the viewing direction is used. This can actually be faster than three dimensional texture slicing, but it requires three times the texture data and artifacts occur when the viewing direction is near the boundary between the orthogonal axes.

A slicing technique called half angle slicing [20, 61] produces a single set of slices that can be used to render the data from two different points of view. The direction perpendicular to the slices is chosen so that it is the half vector of the two view directions if they both lie in the same hemisphere, or the half vector of one view direction and the inverse of the other if they lie in opposite hemispheres. By varying the slice spacing based on the angle between the view directions, a consistent sampling distance can be maintained along the rays parallel to the view directions.

By using half angle slicing and alternating between rendering each slice from the point of view of the camera and the point of view of the light source, a shadowing effect can be produced [20, 61]. When the slice is drawn from the camera's point of view, it samples the result from the rendering of the previous slice from the light's point of view to determine the light attenuation. The slices must always be rendered in front-to-back order for the light but the camera must render the slices back-to-front when it is in the opposite hemisphere. Forward scattering effects can also be approximated by taking multiple samples at random offsets on the previous slice [21].

Early volume ray casting algorithms for the GPU used a multi-pass approach due to the limited number of instructions that could be executed in a shader program on older GPUs [22, 45]. These multi-pass approaches also allowed them to update the depth buffer between each pass, and set it to kill fragments above some opacity threshold using depth culling. This approach to early ray termination is applicable to any iterative front to back algorithm, and has been adapted for data sets that have been subdivided into bricks [46].

Newer GPUs allow for single pass ray casting [49] which allows for effects like reflection, refraction, and self shadowing isosurfaces. If full spectral information is used instead of just RGB, effects like selective absorption and dispersion [52] can be achieved. The single pass algorithms must rely on the shader programs dynamic branching support for early ray termination, which can be less efficient than depth culling. Ray casting is also generally slower than sliced based rendering, but by a narrowing margin.

Subdividing the data into bricks is a popular method of empty space leaping on GPUs due to the fact that it can reduce not only the amount of computations but also the amount of texture memory required. Bricking has been used both for slice based rendering [54] and ray tracing [36]. More accurate methods of reducing the computations on empty voxels [19, 46] exist, but do not save any additional texture memory.

Bricking techniques have also been used to perform out of core rendering through memory paging [54] and volume roaming [7], reduce the angular dependency on texture performance [57], and enable level of detail techniques [56]. In parallel rendering it has been used for data distribution [4] and load balancing [36, 29].

2.3 Parallel Rendering

Rendering a high quality image is almost always computationally expensive. Given the high demand for computer graphics, it is no surprise that parallelism has been widely exploited to speed up rendering times. There are a number of different considerations to be made when parallelizing the rendering of polygonal mesh data versus volume data. However, there are enough similarities in some areas to warrant a look at previous work in both endeavours.

Much of the early research on parallel rendering focused on specific network topologies and custom architectures. This includes mesh connected networks [37], hypercube networks [33], and custom parallel rendering systems like Pixel-Flow [13] and the more specialized Cube-4 architecture [42] (which evolved into the VolumePro add-in board from Mitsubishi

[41]). While these custom solutions can offer some considerable benefits, the low price and high flexibility of commodity workstations with GPUs and a simple bus type network has shifted the focus of research.

There have been several attempts at creating a software framework for parallel rendering. Some of the best known examples are Chromium [17] and FlowVR [2]. Chromium manipulates the streams of OpenGL commands using filters so that certain commands are being routed to each unit. FlowVR avoids the many complexities of OpenGL by instead distributing its own resources which define geometry, textures, and shaders. These frameworks provide a means to parallelize many existing graphics applications with minimal effort. They also tend to provide an easy way to set up things like display walls and head tracking. The tradeoff is that it can be difficult to incorporate application specific optimizations.

The sort last method probably is the most common for parallel volume rendering. One of the primary research topics for sort last algorithms is how to efficiently transfer and composite the intermediate images. Binary swap [27] and direct send [15, 11] compositing schemes are easy to implement and do a fair job of distributing the compositing workload among the render nodes. SLIC [51] improves direct send compositing primarily through better load balancing and scheduling. Hardware solutions to the compositing problem are also available [50, 26, 39] but they are expensive alternatives.

Sort first methods for parallel volume rendering either replicate the data set across all render nodes [1] or transfer data over the network [4]. Algorithms that replicate the full data set on each GPU can only scale performance, but not the maximum data set size. Algorithms that transfer data over the network, or cache data locally, can allow for data sets larger than the memory available to a single processing unit.

Neumann [38] compares the communication costs for sort first versus sort last volume rendering. He provides a detailed analysis of the communication overheads incurred by each approach as well as some insight into their advantages and disadvantages in terms of load balancing, performance for different viewing conditions, and compatibility with different network topologies. He concludes that dynamic sort first distributions can have much worse communication costs than sort last. However this does not consider the possibility of asynchronously loading and caching data or avoiding loading of occluded data.

There are many other methods of distributing a rendering workload that are less common than sort first or sort last. Temporal distributions assign different frames of an animation to different groups of processing units. This provides perfect load balancing but introduces

undesirable lag in interactive applications. When rendering for a stereo display it is possible to divide the work of rendering for each eye. Sort middle techniques use a static distribution of the primitives which each unit transforms into camera space and then routes to the unit responsible for rasterizing the corresponding portion of the screen space. GPUs make this option unappealing since the transform and rasterization of primitives is hardwired together. A number of hybrid methods which combine elements of different distributions have also been devised. Hybrid sort first and sort last [47] tries to minimize the amount of compositing and data loading by dynamically adjusting overlapping partitions in both the image and object space.

Load balancing is an important research subject for parallel volume rendering, as the overall performance is limited by the slowest component. Load balancing algorithms can be classified as static or dynamic. A static load balancing algorithm partitions the data once while a dynamic one can update on each frame. Most static algorithms partition the image or data into many more pieces than there are processing units and then each processing unit is assigned several of these pieces to render. A typical example of this for sort first algorithms is assigning alternating scan lines to different units. Overpartioning the image space causes much more data replication when using a data scalable sort first approach and overpartitioning the object space causes much higher compositing overheads. If overpartitioning is not used, static load balancing tends to perform quite poorly for some viewing conditions (particularly in the case of sort first).

Dynamic load balancing tends to assign each unit a single partition and thus avoids the problems associated with overpartitioning. Each processing unit's partition is updated as the camera moves in order to maintain a good load balancing. A common technique uses the relative performance of each rendering node in the previous frame. This has been done with sort last algorithms [36, 29] that subdivide the volume into bricks and reassign bricks to nodes that had a higher frame rate (less workload) in previous frames. Despite being sort last, these methods require volume data to be sent over the network or cached locally. Sort first algorithms have also used this method of load balancing [1], redistributing the image space instead of the object space. Any such method relies on frame-to-frame coherence and thus cannot guarantee any tight bounds on the level of load balancing.

For sort first rendering of polygonal meshes there are existing methods that do not rely on frame-to-frame coherence [23], but these techniques do not directly translate to volume rendering. The existing method which is most similar to the one we propose is the mesh

based adaptive hierarchy decomposition (MAHD) [34]. The cost of rendering portions of the screen is estimated and then split evenly using a hierarchical distribution. The method of computing the cost measure is completely different since they are working with polygonal meshes instead of volumes. Our cost measure is computed at a much finer resolution since we are doing the computations on GPUs and distributing the overhead among the processing units.

Chapter 3

Data Scalable Sort First Rendering

Sort first parallel volume rendering can achieve ideal performance scaling when the full data set fits into the memory of a single GPU. To achieve data scaling one must subdivide the data and swap the pieces to and from the GPU as the camera moves. This presents a number of challenges which we address here, but first we discuss the key differences between sort first and sort last rendering. Parts of this chapter originally appeared in the work of Moloney, Weiskopf, Möller, and Strengert[1] [32].

3.1 The Parallel Rendering Pipeline

In Figure 3.1 we illustrate a generic parallel rendering pipeline, as well as the paths that several algorithms take. The red and blue paths through the pipeline correspond to the sort first and sort last algorithms respectively. There are then three possible points to loop back to in each frame. The dotted line shows the path taken by the sort last algorithm when static load balancing is used and by the sort first algorithm when the data is smaller than the memory of a single GPU. The solid line shows the path taken by the data scalable sort first algorithm and the sort last algorithm with dynamic load balancing when the data does fit into system memory. When the data does not fit into RAM, both algorithms must take the path shown with the dashed line.

One obvious advantage to sort first techniques is that the blending stage can be skipped entirely. The cost of blending intermediate images can be quite large, particularly when

[1] ©Eurographics Association 2007; Reproduced by kind permission of the Eurographics Association.

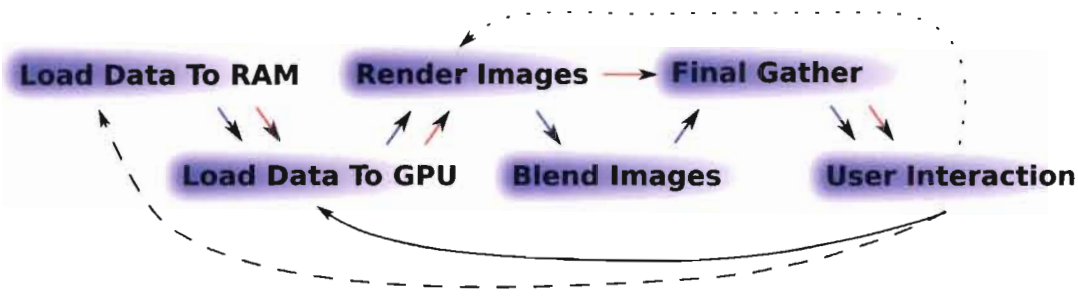

Figure 3.1: A generic overview of the parallel DVR pipeline, with several possible paths traced through it.

a high latency network like ethernet is used. However, sort first algorithms are inherently bad at distributing the data set. If we want to render a data set that is larger than the amount of memory available on the GPU, then we must loop all the way back to the data distribution stages of the pipeline and potentially load some data before we can render the next frame. While the total size of the volumetric data is larger than total size of the intermediate images, only a small portion of the volumetric data needs to be transfered when the coherence with the previous frame is high. Volume data loading can also be cached and communicated in parallel to the rendering process while compositing must be done sequentially after rendering. Finally, occluded portions of the data could potentially not be loaded, reducing the overhead further.

A more important advantage of the sort first rendering is the ability to adapt a number of algorithms that are not efficient (or sometimes even feasible) with a sort last approach. Algorithms that require information from a previously rendered sample on a ray may require too much synchronization when the rays are split up among different rendering nodes. Sort first rendering can keep one set of rays local to each machine and thus allow for such algorithms to be utilized efficiently.

Sort last rendering achieves data scalability without swapping data but only if it uses a static load balancing scheme. Static load balancing can work acceptably well for sort last algorithms, provided the camera is always viewing the full data set and the size difference due to perspective projection is small. These caveats are less likely to hold true once the data sets get larger and the user becomes more likely to zoom in and study features that

are small in proportion to the full data set. Sort last rendering with dynamic load balancing requires data to be redistributed as the view point changes. When the camera is zoomed out and rotating the amount of data redistributed is going to be less than for sort first rendering, but when zoomed in they will both require similar amounts of data to be transfered.

When the whole data set can fit in the memory of a single GPU, sort first will always be more efficient due to the lack of compositing. As the data set grows in size, at some point a sort last distribution will become faster. Where this cross over occurs depends on many parameters, including the data layout, image resolution, point of view, frame-to-frame coherence, number of samples, cost per sample, bytes per voxel, network performance, and transfer function. Advanced compression methods which reduce the data size and increase the fragment cost could shift the threshold upwards when a transfer function with medium to high opacity is used. This is due to the more efficient visibility culling possible in sort first algorithms. which shows increasing benefits as the fragment cost goes up.

3.2 Bricking

We divide our data set into a uniform grid of evenly sized bricks. We do this once, when the data is loaded, based on a user defined parameter for the size of the bricks. At the same time we compute a bit mask for each brick, corresponding to the scalar values that occur within that brick. This allows us to quickly and accurately cull bricks in the same manner as used in [7]. Each rendering node loads only the bricks intersected by its view frustum, as illustrated by a 2D example in Figure 3.2. By subdividing the data set in this manner, we can reduce the amount of data that must be replicated among the render nodes by reducing the brick size.

When choosing a brick size, we must balance the benefits of having a finer granularity in object space and the increased overheads from having a larger number of bricks. A finer granularity reduces data replication between rendering nodes along shared planes of the nodes' view frustums. This replication is illustrated by a 2D example in Figure 3.2. However, there is a per brick memory overhead since adjacent bricks must share one data value at every point on their border so that the trilinear interpolation is consistent across bricks. When using a pre-integrated transfer function [12], two data values must be replicated so that you can access the values for the back sides of the slabs at the boundary. When bricks are culled based on the transfer function, having a finer granularity can allow us to perform

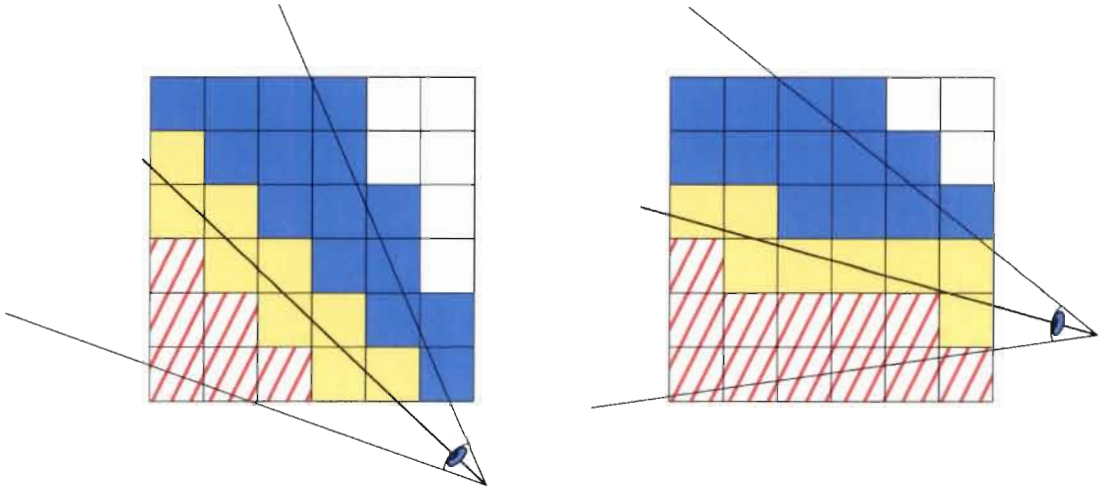

Figure 3.2: A 2D illustration of how bricking allows data scalability albeit with a memory overhead. The red hatched bricks are loaded into textures by the node with the left view frustum, the solid blue bricks are loaded by the node with the right view frustum, and the solid yellow bricks must be loaded by both. As the viewpoint changes (left vs right image) the bricks required by each node can change.

a more accurate culling, thus reducing the rendering workload and the amount of data that must be loaded. A hierarchy of brick sizes has often been used to help balance these factors but in turn has it's own associated overheads (in particular memory usage).

The most significant per brick performance overhead (when using a slice based rendering engine) is the increased number of vertices that must be generated on the CPU, and sent to the GPU, for the proxy geometry of each brick. To tackle this issue we devise a novel technique that computes a single vertex 'template' for each frame, which can be used to render every brick of the same size. This reduces the amount of computation on the CPU as well as the amount of data that must be transferred to, and stored on, the GPU.

3.2.1 Slice Template

We found that our rendering times were heavily CPU limited when rendering several hundreds or thousands of bricks. This was because hundreds of slice vertices were being computed for every brick for each frame. Since the slices intersect all of the bricks at the same angle, the only information that is potentially different for each brick is an offset in the view direction that determines where the first slice intersects the brick. Since the number of

slices for each brick differs by at most one, we can compute a slice template by expanding the brick along the view direction by one slice distance, and then use the vertices at the intersection points of this expanded brick and the slice planes. Figure 3.3 illustrates this concept.

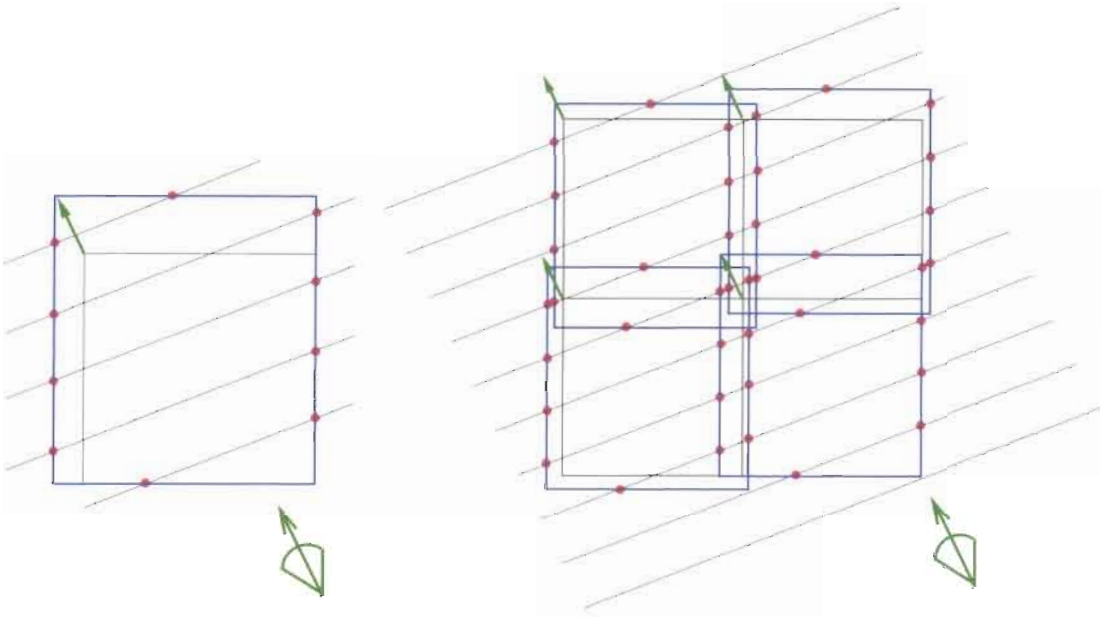

Figure 3.3: An illustration of how the slice templating works. ©Eurographics Association 2007; Reproduced by kind permission of the Eurographics Association.

We can use this slice template to render any brick by simply computing the offset along the view direction for the first intersection, and then translating the templated slice vertices along the direction opposite of the view direction. Since the vertices themselves never change, they can be loaded onto the GPU as a vertex buffer object once at the beginning of each frame. The pre-integration texture coordinates are computed in a vertex shader program on the GPU. Since the templated slices are larger than the actual bricks, we apply user defined OpenGL clip planes to kill any fragments that lie outside of the actual brick.

In order to minimize the number of fragments that we need to kill, we compute one template for each brick size. We use a single brick size, except at the boundaries when the size of the data is not evenly divisible by the size of the bricks. This means that we need at most eight templates. If there are many brick sizes (due to adaptive bricking for instance) then a few templates of different sizes could be chosen. Each brick could then be rendered with the smallest template which completely covers the brick.

3.3 Caching

By dividing the object space into bricks and caching the ones that intersect the view frustum, we can achieve data scalability with a sort first algorithm. This caching requirement imposes an additional overhead as data might need to be transferred to the GPU or system memory during the rendering. However, the amount of data that needs to be loaded should decrease as the number of processing units increases. Loading can often be done asynchronously in a predictive manner, and in a number of scenarios data loading is unavoidable (eg. time varying data, out of core rendering, dynamic load balancing).

The amount of bricks that need to be loaded on any given frame depends on the size of the frustum relative to the bricks, the level of frame to frame coherence, the viewing conditions, and the method of caching bricks. Since the size of the frustums decrease as we add processing units, the number of bricks that need to be loaded also decreases as long as the width and height of the frustums are more than those of a couple of bricks. Frame-to-frame coherence is usually a fairly safe assumption in interactive volume rendering, with the exception being time varying data, which has to be loaded on every frame anyway. When the camera is rotating around the volume from a distance, the amount of data loading is much higher than when the camera is zooming in or panning.

We experiment with two caching schemes. The simplest method of caching bricks is to load them as they intersect the frustum, and once memory runs out start swapping bricks out in least recently used (LRU) order. The LRU method is simple to implement but cannot take advantage of asynchronous loading and it suffers from sudden spikes in the amount of loading that must be done in a frame. If we can assume frame to frame coherence, then we can try to predict which bricks will be needed in upcoming frames and load them ahead of time. The simplest way of doing this is to cache bricks that are in close proximity to the frustum but not yet intersecting. For the proximity caching method we approximate the frustum with a cone and record the distance from the center of each brick to the surface of the cone. We then swap out the bricks that are farthest away from the frustum and pre-cache the bricks that are closest to the frustum. We have a user specified limit on how many bricks can be pre-cached in a frame. A more advanced prediction technique may favor bricks on a certain side of the frustum based on the movement of the previous frames.

Currently there are no GPUs that are capable of asynchronously loading and rendering at the same time. Transfers to the GPU are only asynchronous on the CPU side, which is of

little benefit since we have no substantial work to do on the CPU between distributing the image space and rendering. It is probable that future GPU hardware will have this ability since the manufacturers are striving to make them more general purpose. Even with the current hardware, there is still a benefit to predictive caching to the GPU. We can stabilize performance by spreading the loading costs across multiple frames instead of having large spikes in the amount of data that needs to be loaded in a single frame.

We assume that each processing unit has access to enough system RAM to hold the entire data set, because we only have a single layer of caching between local system memory and GPU memory. For a shared memory environment this in not a limitation, but in a distributed memory environment a second layer of caching would be required if the data set is too large to replicate in the system memory of each node. The second caching layer would swap data in from network or storage devices to system memory. The bandwidth from network and storage is likely to be significantly less than the bandwidth to the GPU, but the amount of memory available for caching would be much larger and the loading could be done asynchronously while the GPU if performing rendering. The same predictive caching methods could be used, but the goal would be to take maximum advantage of the asynchronous loading rather than just reducing spikes in the amount loaded per frame.

Chapter 4

Consistent Dynamic Load Balancing

Sort first algorithms have the potential to provide more accurate load balancing than sort last algorithms due to the finer granularity available in image space versus object space. In this chapter we present a novel dynamic load balancing scheme to give consistent results regardless of how much coherence there is between frames. We do this by working strictly with data from the current frame and taking advantage of the processing power of GPUs. This approach was originally described by Moloney, Weiskopf, Möller, and Strengert [32], and the following chapter is based on their work[1].

Since we are using a slice based rendering engine, good load balancing is equivalent to having each GPU render the same number of fragments. Thus we need an estimate of how many fragments contribute to each pixel. If there is no early ray termination, this is directly proportional to the total length of the ray through a given pixel, that lies within some brick.

4.1 Calculating Per Pixel Rendering Cost

An important aspect of our algorithm is the use of GPUs to compute the per pixel cost contributed by thousands of bricks in a timely manner. We can trade off the accuracy for speed by using approximation techniques for the per pixel cost and a lower resolution pixel cost map. We can get further speed ups by distributing the computations among processing

[1] ©Eurographics Association 2007; Reproduced by kind permission of the Eurographics Association.

Figure 4.1: An illustration of the results for the three different methods of computing the per pixel rendering cost for a single brick. The brightness of each pixel represents the calculated rendering cost. From left to right: accurate method, backface method, and splatting method.

units. We provide an overview of how the cost is computed here, with details about the implementation in Appendix A.4.

We present three methods of computing the rendering cost for the full image space, each with varying levels of accuracy and computational cost. The methods differ in the manner they compute the rendering cost for a single brick, but all three methods use additive blending to sum up the contributions of individual bricks. A visual comparison of the results generated by the three methods for a single brick is given in Figure 4.1 and the results for a full data set is shown in Figure 4.4.

The first method is completely accurate. For an initial attempt at the accurate method we tried rendering the front faces' depth into a buffer and then rendering back faces and taking the difference of the depths. This was too slow for significant numbers of bricks because it requires two passes for each brick. Instead, we compute the distance between the front and back faces in a single pass by rendering the back faces and intersecting a ray from the position each fragment to the eye with the front faces.

If the number of bricks is quite small, then the accurate method has about the same overhead as the other methods. For a large number of bricks, the accurate method is about twice as slow as the other two methods on our target architecture (NVIDIA 6800). However, on the latest generation NVIDIA 8800 this method is as fast as the two approximating methods and thus it would always be the method of choice.

The second method splats a sphere for each brick, with a diameter equal to the longest

diagonal of the brick. For each brick we render a quad, textured with this spherical footprint, at the brick's center with a width and height equal to the diameter of the sphere. In order to accommodate multiple brick sizes, the fragment shader scales the values from the texture by the size of the brick. This method will obviously assign a rendering cost outside of the brick's image space footprint, and even within the brick's footprint the assigned rendering cost will be inaccurate. This method is quite straightforward to implement, can be much faster than the accurate method, and actually gives good load balancing results under certain viewing conditions.

The third and final method we implemented draws only the back faces of each brick, with an estimated rendering cost assigned to each vertex and then linearly interpolated for each fragment. We calculate two costs, one for vertices on the silhouette and one for vertices inside the brick's footprint. Vertices inside the brick's footprint are assigned a higher cost than vertices on the silhouette. In reality the cost on the silhouette is always zero, but this can cause the interpolated cost to drop off too fast since we are not considering the positions of the front faces at all. If the view direction is almost perpendicular to a face of the brick then the cost near the silhouette will be relatively high and the cost for the interior will be relatively low. On the other hand, if the view direction is almost parallel to one of the brick's diagonals then the cost near the silhouettes will be close to zero and the cost in the interior will be relatively high.

It is also possible to distribute the computation of the rendering cost among the processing units. Trying to do this in a manner that balances the computational load on each rendering node leads to a chicken and egg scenario: we are trying to load balance the computation of our load balancing. If we can assume frame-to-frame coherence then we can just use the screen distribution from the previous frame, otherwise the best we can do is evenly tile the image space bounding box of the volume.

4.2 Distributing the Image Space

Once we have computed the rendering cost for each pixel, we want to use this information to update the image space decomposition in a manner that distributes the workload evenly. We divide the screen using a set of rows that each potentially have a different number of columns. Each portion of the screen associated with a row and column pair is assigned to one processing unit. We utilize this type of distribution due to it's compatibility with parallel

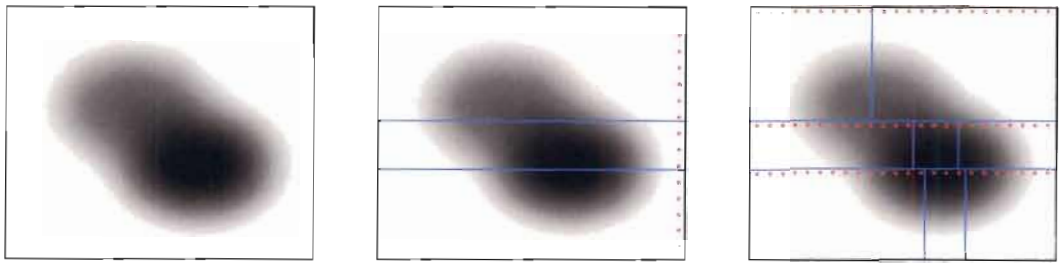

Figure 4.2: An illustration of the basic method used to divide the screen space into 8 pieces. The left most image shows a distribution of rendering cost on the screen (darker pixels are more expensive). The middle and right images illustrate the row and column splitting procedure. The red dotted lines highlight the rows and columns of the SAT that must be searched to find the split lines. The resulting split lines between rows and columns are shown in solid blue.

environments that have a non-power of two number of processing units. Additionally, since the decomposition of the screen is only a two level hierarchy it allows us to efficiently parallelize the load balancing computations.

We find the divisions for the rows first with each one getting a portion of the rendering workload proportional to the number of processing units (columns) in that row. Then we split each row up evenly into the appropriate number of columns. We illustrate this process in Figure 4.2. To do this efficiently we need to be able to quickly compute the total rendering cost for an area of the image space. Therefore, we compute a Summed Area Table (SAT) of the rendering cost, which allows us to query an area of the image space for its total rendering cost in constant time. Finding a line to split the screen space can then be done with a simple binary search along the row or column of the SAT that is perpendicular to the splitting line. This approach has been used previously for sort first rendering of polygonal meshes [34].

We use an axis aligned bounding box in the image space to reduce the number of pixels that are read back from the GPU and processed in the SAT computation. If we are not parallelizing the computation of the rendering cost, then the master node computes the per pixel cost and SAT for the full image space. The master node then computes the image space distribution and broadcasts the result to all other nodes.

For a parallel computation of the rendering cost, each processing unit computes the per pixel rendering cost and SAT for their respective portion of the image space. A master unit

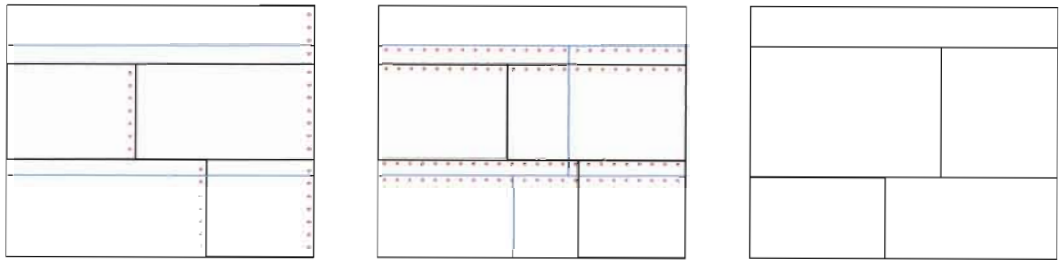

Figure 4.3: The left image shows the screen decomposition used to distribute the render cost computation among 5 nodes (black lines). Only the right most column of each nodes SAT (corresponding to the columns of pixels highlighted with a red dotted lines) need to be used to find the row split lines (solid blue lines). The middle image shows the rows of SAT information needed to find the column split lines in a similar fashion. The right image shows the image decomposition for the next frame.

gathers data from the SATs of all other units in order to compute the screen distribution which is then broadcast back out to the slaves. As illustrated in Figure 4.3 we only need to combine the information from the right most column of each node's SAT to compute the row split lines. To compute the column split lines, we need the SAT information along the top most row of each nodes screen space and any row intersected by a row split line. The communications take place in two stages. First the master unit gathers the top most row and right most column of SAT data from all units before computing and broadcasting the row split lines. Secondly, all units that have row split lines running through their screen space send the SAT information along those rows to the master node. The master node can then compute and broadcast the column split points.

Of course we do not need to divide the rendering cost evenly. If we have a heterogeneous parallel environment it may be desirable to give some processing units more work than others. This can easily be achieved by keeping track of how long it takes each unit to process the workloads they have been given, and then using this performance measure to decide how much of the total workload each unit should be given.

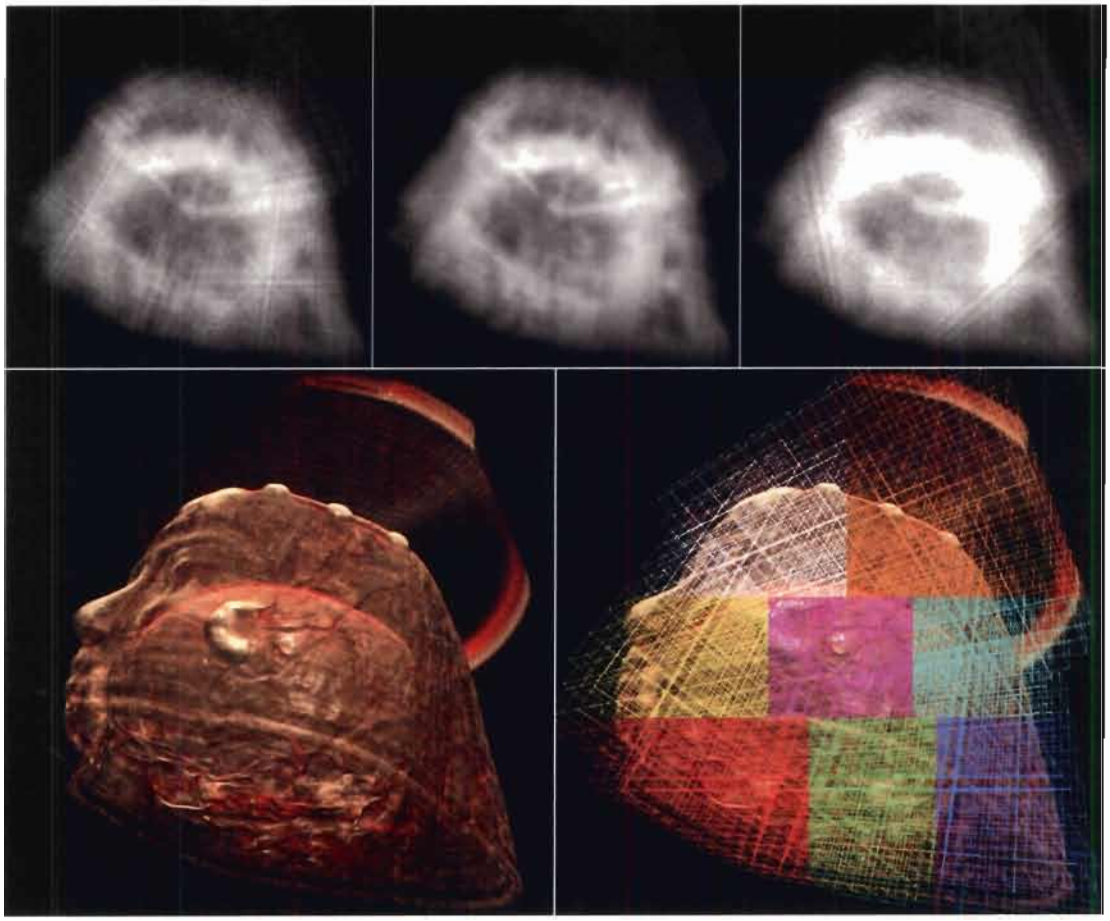

Figure 4.4: The top row of images shows the calculated rendering cost (brightness normalized for print) for the rendering of the mummy head shown in the bottom row of images. From left to right on the top row we have the results generated by the backface, accurate, and splatting methods. The bottom row shows the rendered image with and without the brick outlines drawn. Each of the eight processing units uses a unique color for the brick boundaries so that the screen decomposition is visible.

Chapter 5

Ray Coherent Algorithms in Parallel Rendering

The main reason for us exploring sort first approaches to data scalable parallel volume rendering is due to their compatibility with many volume rendering algorithms. In this chapter, we discuss our implementation of two such algorithms as well as some other possible candidate algorithms. All of the algorithms benefit from keeping the information and processing along each ray local to a single processing unit. The load balancing strategy discussed in Chapter 4 is an example of such an algorithm, but in this chapter we focus on existing algorithms for volume rendering on a single processing unit that could benefit from ray coherence when adapted to a parallel environment.

The first algorithm, discussed in Section 5.1, takes advantage of the locality of all the information along a viewing ray in order to cull occluded parts of the data. Each processing unit in a sort last distribution can only cull data from its local portion of the data set. This can be very inefficient, imagine the case where one unit's portion of the object space is completely occluded by data on the other units. The second algorithm, discussed in Section 5.2, is an image space shadowing algorithm that alternates between rendering from the light's and camera's point of view. To parallelize this algorithm we must do a sort first decomposition from the light's point of view, so that all the information along the shadow rays is available locally. From the camera's point of view, we then need to composite the intermediate images like in a sort last rendering pipeline. Lastly, in Section 5.3, we discuss other existing algorithms that could benefit from ray coherence when being adapted to a

parallel environment.

5.1 Occlusion Culling

It has long been observed that many of the fragments processed when rendering a volume do not contribute anything to the final image. Typically these fragments are separated into two groups: fragments that have zero opacity (empty fragments) and fragments that are occluded by one or more fragments which have a total opacity of one (occluded fragments). While we already have a simple (but crude) method of skipping empty fragments by culling empty bricks, we have not yet exploited occlusion.

We provide two methods of avoiding processing of occluded parts of the data. The first is a simple adaptation of an existing approach [22, 45, 46] which uses the depth culling ability of GPUs to speed up the processing of occluded fragments. The "culled" fragments still have some, though greatly reduced, processing cost and per brick overheads cannot be avoided. The second method builds on the first by using the new occlusion query feature on GPUs to test if entire bricks are completely occluded. Loading and rendering of the occluded bricks can be avoided for that frame, but the queries themselves incur a significant overhead.

5.1.1 Killing Fragments with Depth Culling

The depth culling feature on GPUs was originally designed to speed up rendering of occluded fragments in surface based rendering. The basic idea is to not only store the color of fragments in the render target, but also some information about the depth of the fragments. Fragments that are behind a surface that was already rendered could then be quickly culled by comparing their depth values to the depth values already stored in the render target. Occlusion in volume rendering is far more complicated as fragments almost never have an opacity of one, and thus we can only say something is occluded once we have composited together a number of fragments whose total opacity approaches one.

In the multi-pass ray casting approaches [22, 45] the authors were forced to switch render targets periodically due to the limited number of operations that could be performed in a shader on older GPUs. However, one benefit to this switching was that it gave them an opportunity to do an extra pass which set the depth buffer so that fragments behind opaque pixels would be culled. To do this they simply mapped the result that had been rendered

thus far as a texture onto a full screen quad at the near clip plane, and any pixels beneath an opacity threshold were killed in the shader. The pixels that were not killed (the opaque ones) would then set the value in the depth buffer to the depth of the near clip plane. With the depth test set to cull fragments with a depth value greater than what is stored in the buffer, this would depth cull fragments that would project to opaque pixels.

Periodically doing an extra pass to update the depth buffer incurs an overhead proportional to the number of updates (and to a lesser degree, the number of pixels updated). Ruijters et al. do an update once for every brick in a subdivided data set [46] by rendering the front faces of each brick's bounding box into the depth buffer and killing pixels in the same manner described above. Since many bricks do not overlap at all in image space, we have found that it is beneficial to update the depth buffer even less frequently. Therefore, we render a chunk of bricks at a time, and update the depth buffer in between each chunk. A smaller chunk size potentially results in a more accurate occlusion culling but also a larger overhead.

While reducing the number of update passes is going to have the biggest effect on performance, we also try to minimize the cost associated with each update pass. To do this, we do not change the render target (as is required in multi-pass raycasting) but instead just disable color output for the update pass. Also, we do not need to render a full screen quad for each update but rather we can just render a quad which covers the bricks in the last chunk. To do this we just keep track of an approximate bounding box in the image space as we render each chunk.

A front to back ordering of the bricks in a data set is not unique, and which ordering we choose affects our early ray termination algorithm. Since we cannot capture any occlusion happening between bricks in the same chunk, we would like the bricks in a chunk to be spread out over the image space rather than overlapping. We can achieve this by generating our front to back order slab by slab, where we choose the set of slabs perpendicular to the axis most aligned with the view direction. We find the first brick in our front to back order by finding the set of dividing planes (between the bricks in the slab) that our view point lies in between. Starting from this point we can build our front to back order by simply iterating outwards along the rows and columns of the slab. We can use the same order in the following slabs.

Using these basic optimizations, we try to determine an appropriate chunk size. The ideal chunk size depends on the data set, the transfer function, the viewing angle, and even

the type of GPU we are using. Optimizing for all these parameters on the fly is intractable. Instead we try to find a good value for the general case. A good chunk size should have little overhead when no occlusion occurs, and at the same time it should be close to optimal performance when occlusion does occur.

5.1.2 Culling Bricks with Occlusion Queries

We can use the above method for killing fragments in conjunction with the occlusion query feature on GPUs to cull full bricks which are completely occluded. Occlusion queries allow a program to know how many fragments were actually rendered (passed the depth and stencil test) for a group of primitives. Thus if we were to render the bounding box of a brick (with the depth buffer setup as described above) and we get a fragment count of zero, then we know that the brick can be skipped entirely.

There is however a much larger overhead for performing occlusion queries compared to just depth culling. The nature of the overhead is also quite complex as it depends not just on the number of queries made, but also how the queries are dispersed through the rendering process. Dispatching a small number of queries at even intervals during the rendering process can result in a greater overhead than dispatching a large number of queries in rapid succession. This behaviour seems to indicate that the occlusion queries disrupt the flow of data through the GPU resulting in reduced throughput. In order to reduce the overhead from the occlusion queries we must then try to reduce the number of chunks of bricks that we perform the queries for, rather than just reducing the number of bricks we query in each chunk.

If we have frame-to-frame coherence, we can use the culling results from previous frames to guide our choice of which chunks to dispatch occlusion queries for. As a simple heuristic we allow the user to specify a threshold which defines the minimum percentage of bricks that must be culled in a chunk for the speedup to outweigh the overhead. The threshold value can be determined by the ratio of the overhead to the time it takes to render a brick that is getting culled by the depth test. Each time we perform the queries we count the number of bricks that get occluded in each chunk and the chunks that surpass the threshold will be queried again in the next frame.

If we only query chunks that surpass the threshold then the results for the other chunks will go stale over time. To rectify this, we use a second parameter for the probability of querying a chunk which is below the threshold. The discrete probability distribution for the

age of a chunk is a geometric distribution, so the expected value for the age of chunk is $\frac{1-p}{p}$ where p is the probability that a chunk is queried. As an example, we could set the query probability to ten percent and then the expected value for the age of the queries would be nine frames.

5.2 Volumetric Shadowing

Shadowing effects can provide an additional depth cue to a user exploring a volumetric data set. In the past this was done by creating a corresponding shadow volume which describes the amount of light arriving at any point in the data. Computing such a shadow volume is expensive and must be done every time that the light position or transfer function changes. The ability to interactively change the light position and transfer function is key to efficient volume exploration. Also, shadow volume approaches can suffer from attenuation leakage due to insufficient resolution.

A new image space approach to volume shadowing that avoids these problems has since been proposed [20, 61]. Instead of rendering the slices so that they are aligned with the camera, they are instead aligned with the half angle between the camera and the light. This allows the same slice to be rendered from both the camera's and light's points of view. We can then render the volume slice by slice, with each slice being rendered first from the camera's point of view, and then from the light's. When the next slice is rendered from the camera's point of view, the previous result from the light's point of view is mapped as a texture. The opacity of this texture then tells us how much light has been attenuated thus far. This approach allows for interactive updates of the light and transfer function, requires far less memory, and avoids issues with attenuation leakage. By combining this image space shadowing algorithm with a hybrid partitioning scheme, we are able to perform interactive shadowed volume rendering on data sets which are too large to fit on a single GPU.

5.2.1 Hybrid Partitioning

In the same way that we exploited the coherence of viewing rays for performing visibility culling, we can use a sort first distribution of the light's image space in order to make the light rays coherent on each processing unit. The screen space for the light map is divided into regions and the corresponding frustums of each unit are intersected against the bricks. The bricks that intersect the light frustum must also be rendered from the camera's

point of view. Obviously, when the camera's view is not perfectly aligned with the light's, the intermediate images produced by each node will overlap. Therefore, as in sort last partitioning, we require a compositing stage to combine the samples along the viewing rays and create the final image. A two dimensional version of this hybrid partitioning scheme is illustrated in the left image of Figure 5.1.

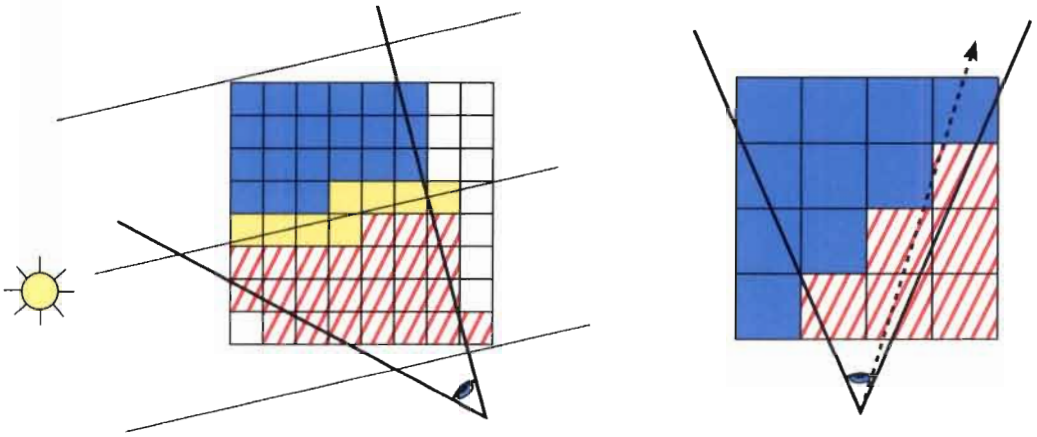

Figure 5.1: The left image shows a two dimensional illustration of the hybrid partitioning. The light frustum is divided into two pieces and the solid blue bricks are rendered by the unit with the left frustum while the red hatched bricks are rendered by the unit with the right frustum. The bricks that are solid yellow must be clipped against the shared plane of the two frustums and each unit renders their respective portions. The right image shows how the staircasing effect that occurs if the bricks are not clipped can create cycles in the compositing order. A viewing ray (shown with the dashed black line) passes from one unit's set of bricks to the other's and back again.

The processing units cannot just render their portions of the data brick by brick as we have done for standard volume rendering. For many viewing conditions there is no ordering of the bricks that will give correct compositing results for both the light and the camera. While this could potentially be overcome by rendering sets of bricks into different buffers and then combining the results, this would add significant complexity and computational overhead. Instead we render the data slice by slice by consecutively rendering the pieces of each slice from each of the bricks it intersects. This incurs a significant overhead since we must change some of the rendering state, such as the current texture, for every piece of every slice. Due to this additional overhead, the ideal brick size is much larger for shadowed rendering compared to standard volume rendering.

Bricks that are shared by neighboring processing units must be dealt with some how. It is usually not possible to just assign whole bricks to one node or the other and have a valid compositing order. Although the bricks themselves are convex, the set of bricks that are intersecting a unit's frustum are likely to have concavities due to staircasing. As shown in the right image of Figure 5.1, this creates cycles in the compositing order whenever viewing rays cross from one unit's set of bricks into another's and then back into the first unit's set again. Therefore we clip the bricks with the frustum planes to create convex pieces, and have each unit render their respective portions of the shared bricks. The clipping can be seen in Figure 5.2 which shows the results from a shadowed rendering with two processing units.

5.2.2 Direct Send Compositing

For the compositing stage we choose direct send compositing, due to its simplicity and efficiency when handling non power of two numbers of nodes. Binary swap compositing requires some processing units to remain idle for the first compositing stage if the number of units is not a power of two. For our method of distributing the light's screen space, both the number of rows and the number of columns in the screen space distribution would have to be powers of two in order to have no idle units during binary swap compositing. This is because, in order to avoid cycles in the compositing order, we must composite the images from the units that belong to the same row in the image space distribution before we can composite the results from the different rows.

For direct send compositing, each unit is assigned a portion of the screen for which they will perform all of the compositing calculations. Each unit must read back the full screen space and then do an all-to-all communication that scatters the portions of the screen that are not being composited locally and gathers from all units the portion of the screen that is being composited locally. Each unit can then composite the intermediate results gathered from the other units in depth order.

We do not take advantage of the potential sparsity of the image data. That is, we assume every pixel (inside the image space bounding box of the full volume) needs to be transferred and composited. A run length encoding scheme would eliminate this issue, but we do not pursue this as it has been well studied. If the image space bounding box of the volume contains P pixels and we have N processing units, then each unit will send, receive, and composite $\frac{N-1}{N}P$ pixels. This means that as the number of processing units increases, we

converge towards a fixed compositing cost which is linearly related to the number of pixels. One pragmatic issue with the scalability of image compositing is that the total number of pixels being transferred over the network increases at a rate of $(N-1)P$. This can potentially cause reduced performance as the number of units increases due to congestion on the network.

We minimize the compositing cost by overlapping the read back of pixels from the GPU, the transfer of pixels over the network, and the compositing computations. The result is that our entire compositing time is reduced to slightly more than the time it takes to communicate the pixels over the network, since that is the bottleneck in our parallel environment. For a detailed description of how we implemented our overlapped compositing we refer our readers to Appendix A.5. Even with this optimization, the compositing cost on gigabit ethernet can become quite significant for moderate sizes.

5.3 Other Potential Algorithms

Most algorithms which process the data along a set of rays are going to benefit from ray coherence when adapted to a sort first parallel distribution. Even standard emission absorption volume rendering performance can benefit from ray coherence through reduced communication overheads. Algorithms that benefit from reduced synchronization overheads are likely to be the most interesting. Acceleration techniques like occlusion culling can just avoid synchronization at the cost of accuracy. For rendering algorithms like shadowed volume rendering this is not an option, and the synchronization required for doing shadowed rendering with an object space distribution would leave the processing units idle for most of each frame.

Spectral effects like inelastic scattering and selective absorption [40, 52] can be used to make rendered images more realistic and informative. Nested structures of interest within the volume can be made visually distinct while maintaining surface features by having each one scatter and absorb different frequencies of light. These effects depend of the spectrum of light traveling along the rays, which changes as the ray steps through the volume. Since each rendered sample requires information gained from processing the preceding samples along the ray, the synchronization overhead is simply too high when rays are split among processing units.

Visualization techniques that utilize depth peeling, such as opacity peeling [44] and

feature peeling [28], are an alternative for visualizing nested structures within the volume data. These techniques split the volume into layers based on some criteria which is evaluated as the rays pass through the volume. In opacity peeling the criteria is a threshold for the accumulated opacity of rays. Feature peeling uses a more sophisticated criteria that looks for transitions in the scalar values along each ray. Both approaches require information along the ray to be available locally if adapted to a parallel environment.

Using an image based metric for level of detail techniques [55] results in higher quality images than those acquired with a level of detail algorithm using an object based metric. The reason that image based metrics are superior, is that they can take the visibility of a brick into consideration when choosing its level of detail. The downside is that the image based metric must be periodically recomputed as the viewpoint changes, which can be an expensive task. In a parallel environment, a ray coherent workload distribution would allow each processing unit to compute the image space metric for the bricks they are rendering without any additional communication or synchronization.

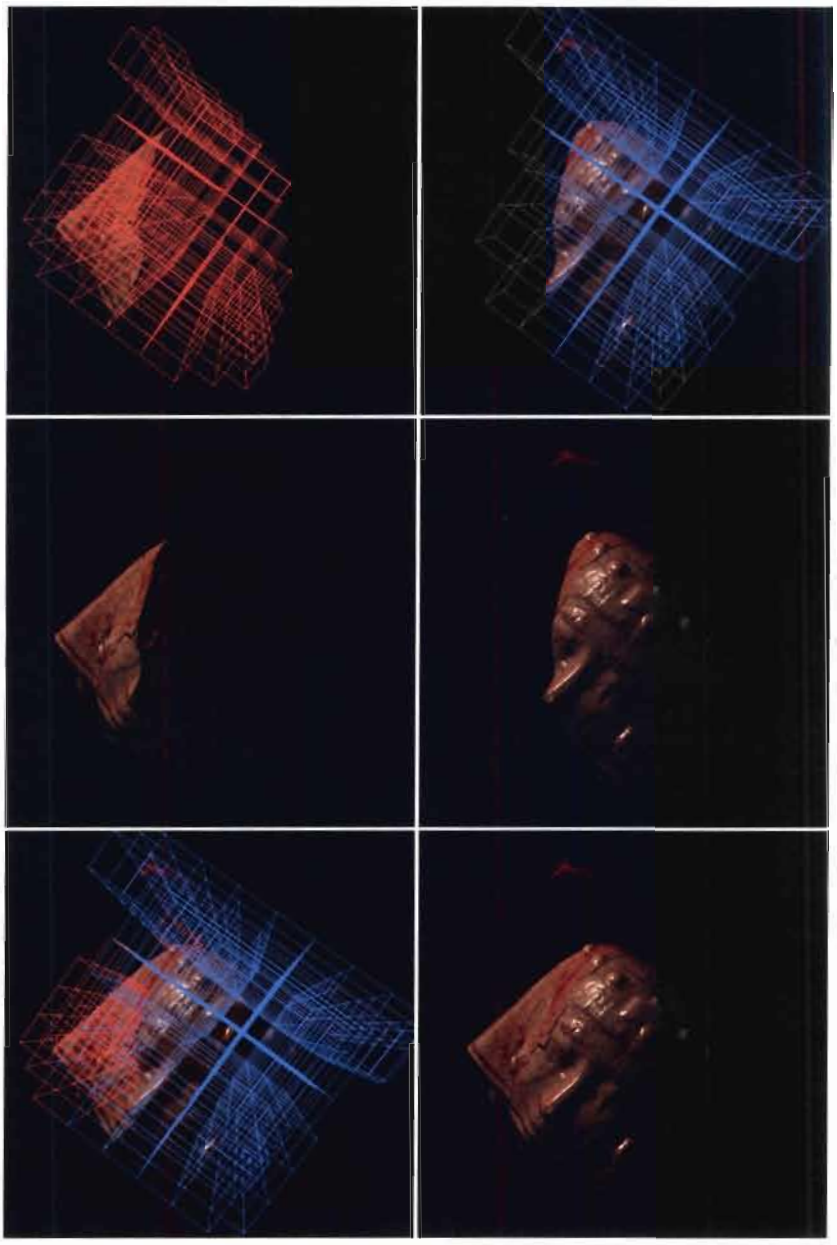

Figure 5.2: An example of shadowed volume rendering on two processing units. The first row shows the intermediate images for the two units with the brick outlines being drawn. The second row shows the intermediate images without the brick outlines shown. The last row shows the final image, with and without the brick outlines.

Chapter 6

Results

One of the important contributions of this thesis is the detailed look at the various bottlenecks in parallel volume rendering. For an overview of both the software and hardware in our parallel rendering system we direct the reader to Appendix A.1. The reader can also refer to Appendix A.2 for details about the data sets that are used for the experiments.

The three most costly parts of the parallel rendering pipeline are: rendering (Section 6.1), data loading (Section 6.2), and compositing (Section 6.3). We study each of these costs independently as well as how they interact and change when we use a sort first vs sort last distribution. The results for our novel load balancing algorithm, with a comparison to existing approaches, is given in Section 6.4. The two ray coherent algorithms we adapted for using in a parallel environment, visibility culling and shadowing, are studied in Sections 6.5 and 6.6 respectively. Finally we look at the overall performance achieved by our rendering system on a number of large real world data sets in Section 6.7.

6.1 Baseline Rendering Performance

The most important factor in the performance of our parallel rendering system is the rendering performance of the individual processing units. Since we need to subdivide the data set into bricks for data scalable sort first rendering, we experiment with how rendering performance is affected by the size of the bricks. Using small bricks improves the rendering performance by providing better cache coherence and a finer granularity for the empty space leaping. However, large numbers of bricks reduce the rendering performance due to per brick overheads.

6.1.1 Per Brick Overheads

Existing methods for rendering a bricked data set with view aligned slices generate the proxy geometry for every brick individually. Once the number of bricks enter the hundreds or thousands, the rendering performance can become limited by this overhead. Our templated slicing technique discussed in Section 3.2 minimizes the overhead associated with rendering each brick.

Since the overhead we are targeting corresponds to the number of bricks, not the image size, we use a 128^2 view port and 5 different brick sizes on the same 256^3 volume. In Table 6.1 the templated slice technique is shown to outperform the standard slicing technique by as much as a factor of seven. The plot in Figure 6.1 shows that while the performance scales linearly in the number of bricks both with and without templates, the slope is much greater without templates. It's important to note that although the per brick overhead is greatly reduced by using slice templates, it can still be significant if the number of bricks is very large.

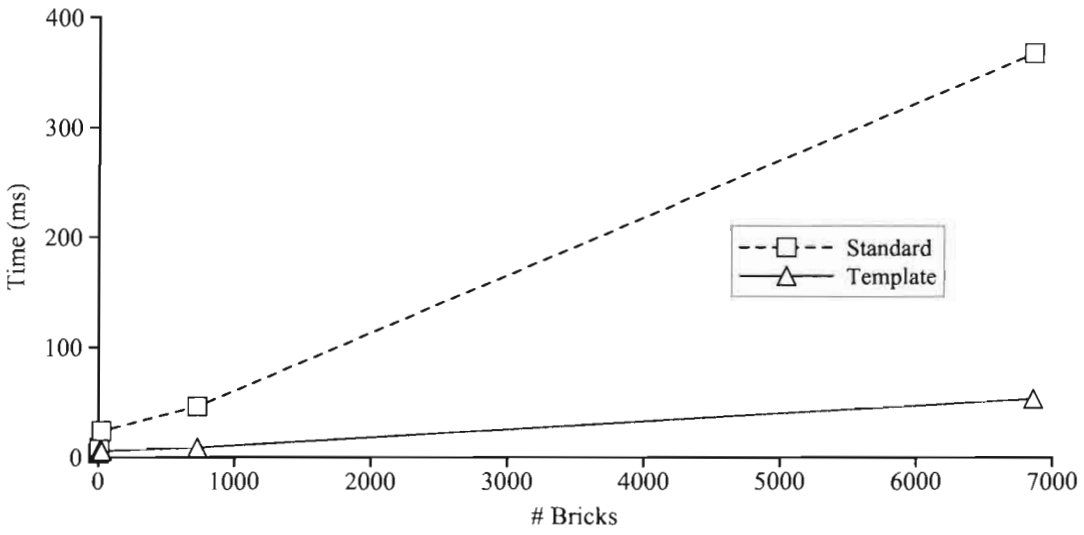

Figure 6.1: A graph of the performance of the templated slicing technique compared to the standard slicing technique. We render the 256^3 radial data set into a 128^2 view port. We use a variety of power-of-two brick sizes, which results in the data set being subdivided into different numbers of bricks. No acceleration techniques are used, so the transfer function is irrelevant.

Table 6.1: A table of the performance of the templated slicing technique compared to the standard slicing technique. We render the 256^3 radial data set into a 128^2 view port. We use a variety of power-of-two brick sizes, which results in the data set being subdivided into different numbers of bricks. No acceleration techniques are used, so the transfer function is irrelevant.

	Rendering Time (ms)		
# Bricks	Standard	Templated	Speed Up
1	3.78	3.77	1.00
9	9.05	5.23	1.73
25	24.21	5.84	4.15
729	46.41	9.21	5.04
6859	367.70	53.33	6.89

6.1.2 Fragment Processing and Data Throughput

When all of the data needed for rendering is resident in texture memory, there are five essential factors in the performance of standard volume rendering of a bricked data set: the number of fragments being processed, the amount of processing that must be done for each fragment, the amount of data that must be accessed for each fragment, the level of cache coherence, and the per brick overheads.

The level of cache coherence on the GPU depends on the memory footprint of the brick textures, and whether or not the driver does any additional processing to order three dimensional chunks of data in a more linear fashion (eg. space filling curves). For NVidia 6800 GPUs, only textures with power-of-two (POT) dimensions are processed in the driver to improve their caching performance for different viewing angles. Textures with non-power-of-two (NPOT) dimensions have the same performance when looking down the z-axis but much worse performance when looking from any other angle. This is clearly illustrated in Figure 6.2 where we plot the rendering times for each frame of an animation which starts off looking down the z-axis and then does one full rotation around the y-axis.

In order to relate rendering performance to the size of the bricks that we subdivide the data into, we find the average performance for a number of brick sizes when rendering a 256^3 volume that is being rotated around the y-axis. We do this twice for a texture with one byte per sample and show the results in Figure 6.3. For one set of results we

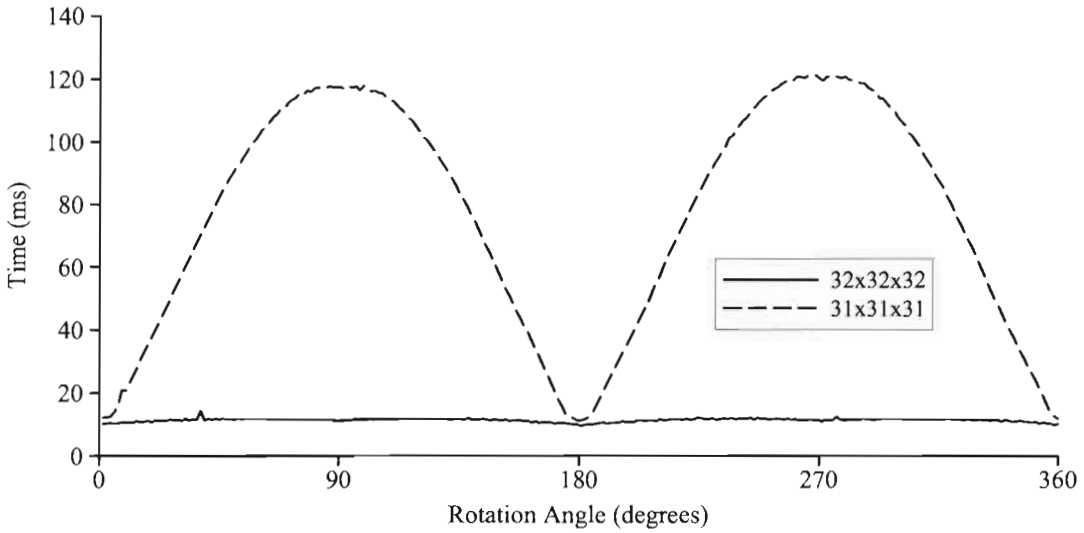

Figure 6.2: The difference in rendering performance for a power-of-two and non-power-of-two brick size. We render the 256^3 radial data set into a 256^2 view port with the slicing distance equal to the sampling distance. We are not using any acceleration techniques.

are rendering approximately one fragment for every sample (256^2 view port and the slicing distance equals the sampling distance) and for the other set of results we are rendering approximately eight fragments per sample (512^2 view port and the slicing distance is double the sampling distance). We can see that POT brick sizes drastically outperform NPOT brick sizes. For POT brick sizes, both 32^3 and 64^3 sizes show a performance benefit due to the fact that they are small enough to fit in cache. However once the brick size is reduced to 16^3 the bricking overheads cause the performance to plummet. Once we start to sample the data set with a larger number of fragments, the benefits from the bricks fitting into cache are reduced (although we still see some benefit for 32^3 and 64^3 brick sizes). The difference between POT and NPOT performance is also reduced when rendering more fragments, but remains significant.

In Figure 6.4 we show results for the same set of tests run on a texture with four one byte components per sample. For POT sizes we consistently see slightly worse performance as we decrease the brick size, which indicates that the caching benefits have been reduced to the point that they are less than the costs from bricking overheads. The difference between POT and NPOT textures has also been reduced drastically. When rendering eight fragments per sample, a brick size of 32^3 gives only thirty nine percent more throughput than a brick size

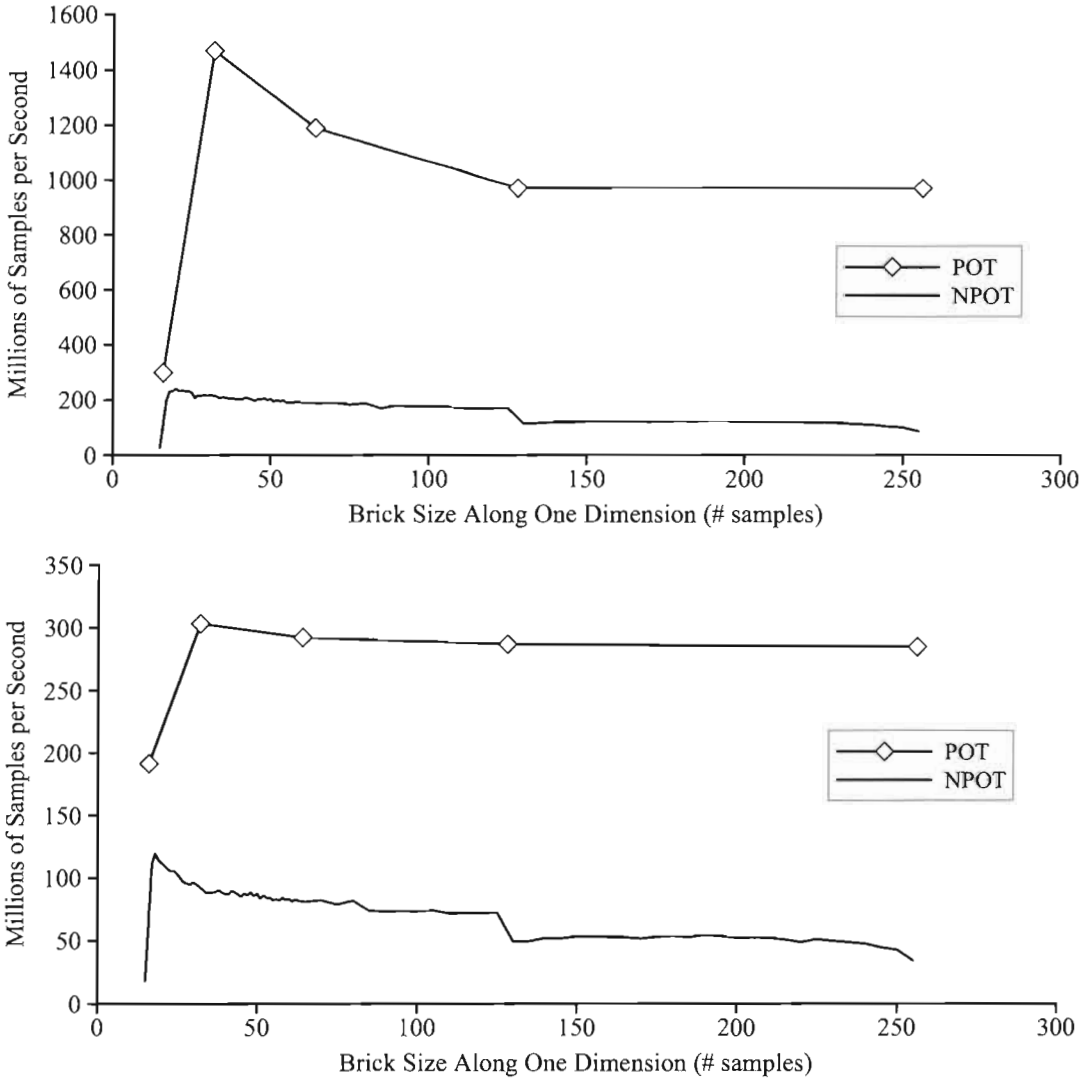

Figure 6.3: Rendering throughput for a texture with a single one byte component per sample and various brick sizes. We render the 256^3 radial data set without any acceleration techniques. The top graph shows the throughput when approximately one fragment is drawn for every voxel and the bottom graph shows the same thing when approximately eight fragments are drawn for every voxel. The scaling of the y-axis is different so that details are visible.

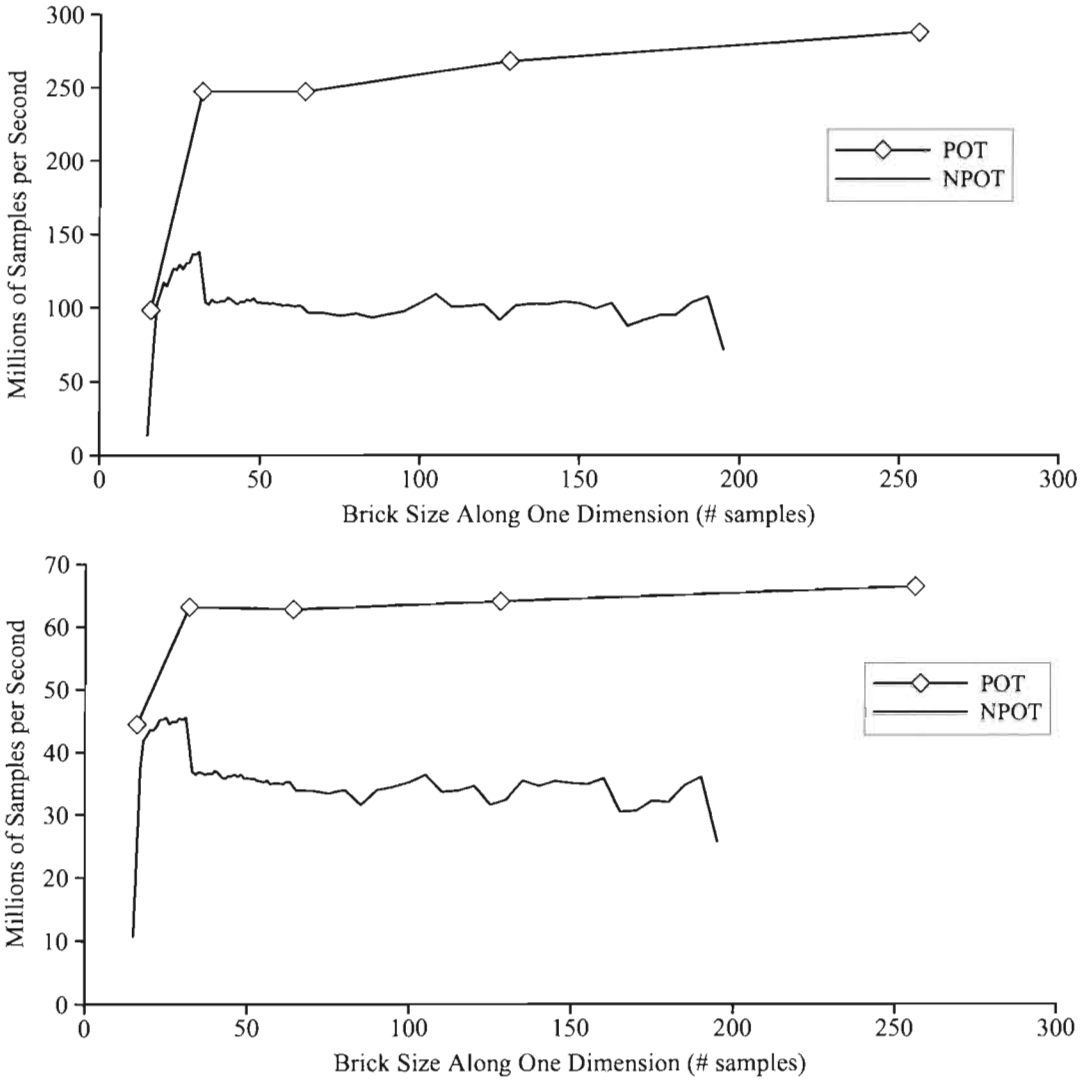

Figure 6.4: Rendering throughput for a texture with four one byte components per sample and various brick sizes. We render the 256^3 radial data set without any acceleration techniques. The top graph shows the throughput when approximately one fragment is drawn for every voxel and the bottom graph shows the same thing when approximately eight fragments are drawn for every voxel. The scaling of the y-axis is different so that details are visible.

of 31^3. For comparison, when the texture had only one byte per sample the difference was over three hundred percent.

6.1.3 Empty Space Leaping

While smaller brick sizes can show an improvement in performance for some scenarios due to cache coherence, a much greater improvement can be achieved by culling empty bricks. We say a brick is empty when all of its data values are mapped to zero opacity by the transfer function. Generally the smaller the brick size the more accurate the culling. However, the level of culling depends heavily on the layout of the data set and the current transfer function. We experiment with the mummy head data set and the transfer function used in Figure 6.18.

In Table 6.2 we list the culling performance for a number of brick sizes. While the total number of samples rendered monotonically decreases with the brick size, the number of samples stored can actually increase. This is due to the duplicated samples at brick boundaries and the data set not being evenly divisible by the the brick size. The latter problem is especially pronounced for larger brick sizes when the size of all bricks is fixed (the third column in the table). If instead we allow bricks at the boundary to be smaller (just rounded up to the next power-of-two in each dimension) then this problem can be alleviated (the fourth column in the table). The problem with variable brick sizes is that it can make it more complicated and costly to swap textures of different sizes between system memory and GPU memory.

For this setup, we get over double the performance using a brick size of 32^3 compared to rendering the data set as one large brick. This is because of the cache coherence shown in the previous section as well as the reduction in the number of samples rendered. Using the even smaller brick size of 16^3 results in a further reduction in the number of samples rendered, but the performance is worse than that of one large brick since the per brick overheads dominate the rendering time.

6.2 Data Loading

Data scalable sort first rendering requires some amount of data is loaded to the GPU when the camera moves. The amount of loading depends on the size of the frustum relative to the size of the bricks and the level of frame-to-frame coherence. How much of an impact

Table 6.2: The accuracy and performance of empty space leaping with various POT brick sizes. The frames per second results are for rendering the mummy head data set into a 512^2 view port with the distance between slices equal to the distance between samples.

	Millions of Samples			
Brick Size	Rendered	Stored (fixed)	Stored (variable)	FPS
16^3	42.24	80.15	79.53	6.2
32^3	57.85	79.89	79.89	15.4
64^3	82.53	108.00	104.07	10.4
128^3	102.03	171.97	112.20	7.9
256^3	103.46	218.10	136.84	7.9
512^3	104.04	134.22	134.22	7.1

the loading has on performance depends on the bandwidth available from system memory to GPU memory and the caching algorithm used.

6.2.1 Bandwidth to Texture Memory

The two factors that determine the bandwidth to texture memory are the size of the textures being loaded and their format. We use brick sizes in the range of 32^3 to 64^3 with both four component and single component textures. For the four component textures we use the BGRA format since this is the internal format for eight bit textures on NVidia GPUs. For single component textures we use the ALPHA format. We test four different POT brick sizes for the four component textures (32^3, $32 \times 32 \times 64$, $32 \times 64 \times 64$, and 64^3) and two for the single component textures (32^3 and 64^3). For the NPOT tests we subtracted one from each of the dimensions of these brick sizes. The results are given in Figure 6.5.

Since NPOT textures are not processed by the GPU driver before upload, it is possible to avoid copying the texture into the driver's memory and load NPOT textures directly to the GPU by using pixel buffer objects (PBOs). This results in almost double the bandwidth in the best case. However, for very small textures the bandwidth can actually be worse with PBOs. Four component textures always outperform single component textures, but by a much larger margin for NPOT brick sizes. The ideal texture size is always 256KB regardless of the texture format.

There is almost an inverse relationship between the rendering performance and loading

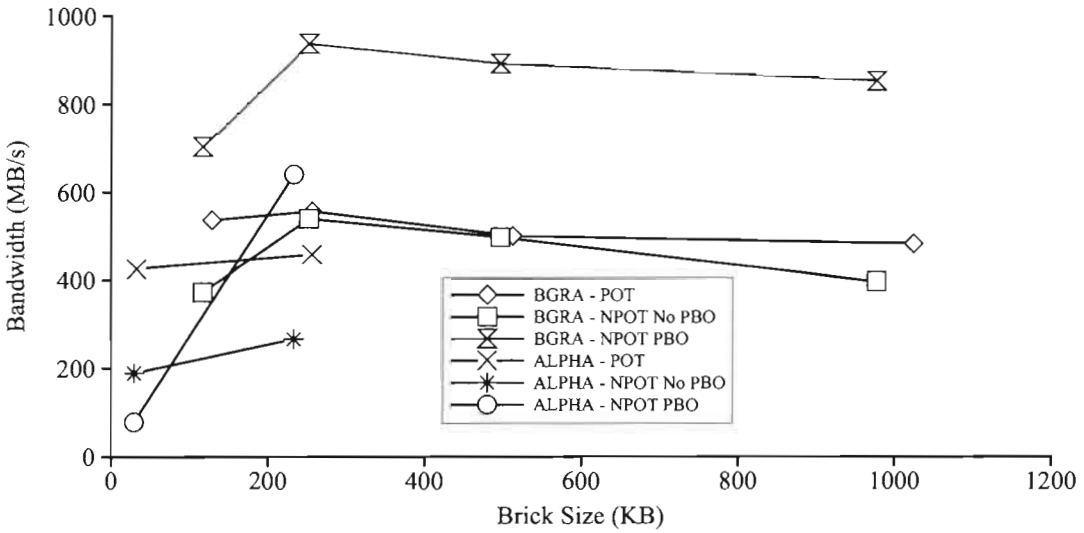

Figure 6.5: Available bandwidth to texture memory for various brick sizes and formats.

performance for the various texture formats. While POT brick sizes and single component textures achieve superior rendering performance, NPOT brick sizes and four component textures are capable of achieving significantly better loading performance. Since we have a static data set the rendering performance is more important even with a data scalable sort first distribution. If the data set was time varying or required out-of-core rendering, then the data loading bandwidth may become more relevant.

6.2.2 Caching Performance

Provided that we have frame-to-frame coherence, the average number of bricks loaded on a frame will be quite low. However, the number of bricks being loaded on any single frame can be quite high. This is because the loading occurs in spurts where many bricks are loaded on one frame and then none are loaded on the next several frames. This is undesirable when trying to interactively explore a data set because of the sudden slow down when a spurt of loading occurs. To combat this we try loading some of the bricks in close proximity to the frustum on frames where the loading requirements are small. This will result in more bricks being loaded in total but in a more consistent fashion.

We compare the proximity caching algorithm to the naive LRU caching algorithm that just loads bricks as they intersect the frustum. We render the mummy head data set with

four component textures and a bricks size of 32^3. This results in 320MB of data after culling empty bricks. We set the maximum amount of texture memory to be used as a buffer by each render unit to be 230MB. For these tests we use a recorded animation of a user exploring a data set. The animation includes rotation, panning, and zooming motions. The results are compiled into Table 6.3.

We can see that even with a preloading threshold as low as five bricks per frame there is a large reduction in the number of frames where a spurt of loading occurs. With a threshold of fifteen bricks per frame the loading spurts are almost eliminated. With four component textures and a brick size of 32^3 it takes 3.5 milliseconds to load 15 bricks. Finally, we can see that when we increase the number of rendering nodes the data loading requirements decrease in tandem with the size of each render unit's frustum.

Table 6.3: A comparison of the simple LRU caching algorithm and the proximity caching algorithm. The threshold value is the limit on the number of bricks being preloaded in the proximity caching algorithm. The results from using four and nine render units are shown. We take the average number of frames above the threshold among all the render units. Results are for the mummy head data set with a brick size of 32^3.

	Average Percentage of Frames above Threshold			
	Four Render Units		Nine Render Units	
Threshold	LRU	Proximity	LRU	Proximity
5	11.65%	2.25%	10.53%	1.54%
10	6.52%	0.59%	5.67%	0.37%
15	4.38%	0.09%	3.33%	0.09%
20	2.96%	0.06%	2.25%	0.05%

6.3 Compositing

Generally speaking, compositing is a process of combining multiple images into a single image. The two types of compositing we are interested in are alpha compositing (or blending) and final gather compositing. Blending takes a number of images and combines them in depth order using the over and under compositing operators. Final gather compositing simply takes a number of images and tiles them together to get a single larger image. While sort first rendering algorithms at most need to perform final gather compositing, sort

last algorithms and the hybrid algorithm we propose in Section 5.2 must do both types of compositing.

6.3.1 Blending

In Figure 6.6 we plot the performance of our synchronous and asynchronous implementations of direct send compositing for a one mega pixel image. We also plot the time needed for just the communication portion of the synchronous implementation. As expected, the asynchronous implementation is much more efficient as it incurs just a small overhead on top of the communication time. This overhead comes from the reduction in bandwidth due to the fragmentation of the packets of data and the fact that synchronization is reduced but not eliminated.

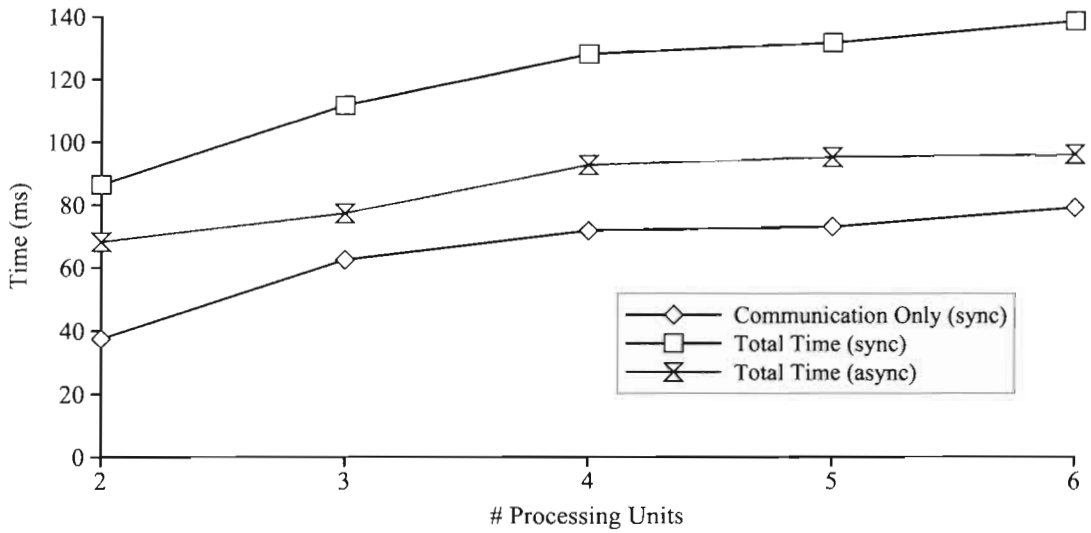

Figure 6.6: A comparison of the performance of our synchronous and asynchronous implementations of direct send compositing for a one mega pixel image.

Since the blending cost converges to a constant value as we increase the number of processing units, we show the image scaling performance for our asynchronous implementation using six processing units in Figure 6.7. The computation time is linear in the number of pixels as expected, but even with a moderate image size of one mega pixel we are already limited to a maximum frame rate of about ten frames per second. If we also consider the final gather time, the maximum frame rate is even lower.

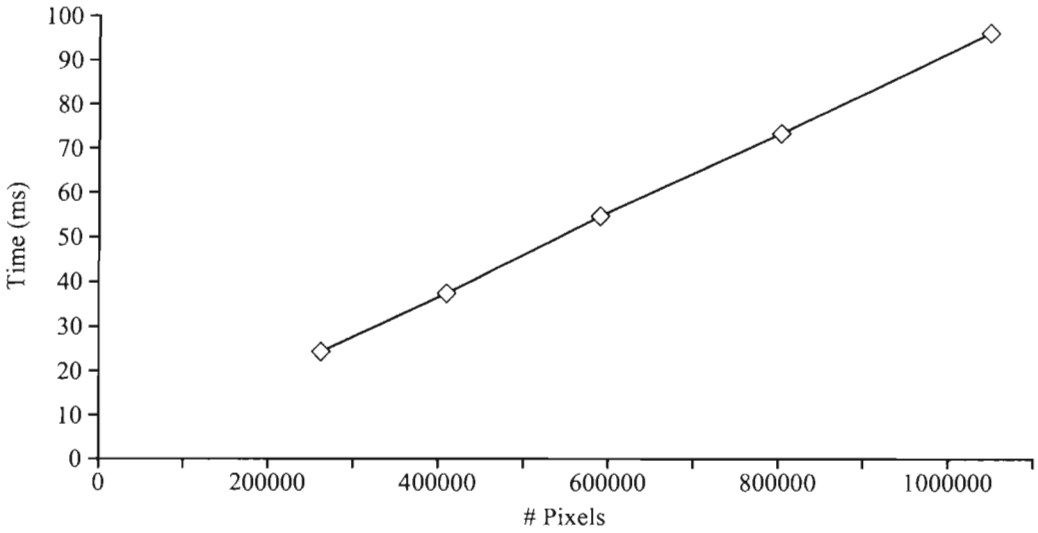

Figure 6.7: The scaling behaviour, in relation to the image size, of our asynchronous compositing implementation with six processing units.

6.3.2 Final Gather Compositing

In our parallel environment we have a single machine that acts as the view client. This machine handles user interaction by sending the appropriate requests to the rendering units and then receiving and displaying the resulting image(s). The images can either be gathered into a single packet on the head render unit and then sent to the view client all at once or the view client itself can receive the individual packets from each render unit. If the network connection between the render units has a much lower latency than the connection to the view client then the former method could be preferable. In our parallel environment the network connection is always gigabit ethernet and so we use the latter method to reduce the total number of pixels transmitted.

Since the total amount of data being sent over the network for the final gather stage does not increase as we add processing units, we are unlikely to run into congestion problems when scaling up our rendering cluster. The difference between doing a final gather with six units and one unit is at most a few milliseconds. The performance of the final gather stage then depends solely on the bandwidth and latency of network. Since our view client is connected to the same switch as the render units, the latency is negligible. Our experiments show about 95 MB/s bandwidth with TCP/IP over gigabit ethernet. For a one mega pixel

image with four bytes per pixel it would take about forty-two milliseconds to complete the final gather. Just like for the blending stage, compression could improve this performance considerably.

6.4 Load Balancing Results

No matter how accurate a load balancing algorithm is, it is only useful if the computational cost is relatively small compared to the rendering cost. Much like the rendering process itself, computing the per pixel rendering cost for our load balancing algorithm is an embarrassingly parallel task. This allows us to compute a per pixel cost both quickly and accurately by utilizing the immense processing power of GPUs. We can trade off accuracy in favor of computational efficiency by reducing the resolution we compute the pixel cost at and using approximate techniques to compute the cost of a brick. We reduce the overhead even further by distributing the computations among processing units. In this section, we compare how these parameters affect the resulting overheads and quality of our load balancing algorithm.

6.4.1 Computation Time

For a data set that has not been subdivided, that is to say the load balancing is computed for a single brick, there is essentially no difference in the performance of our three methods for computing the per pixel cost. However, as shown in Figure 6.8, when the number of bricks increases we see an increasing margin between the performance of the accurate method and the two approximate methods. On more recent GPU architectures this margin does not exist and thus the accurate method should always be used, but for our target architecture the cost of using the accurate method is likely to be too high when the number of bricks is large.

We show the image scaling results for all three methods of computing the per pixel rendering cost with 729 bricks in Figure 6.9. We can see that the image scaling results are quite similar for all three methods, with the splatting method being slightly worse due to it drawing outside the bricks' image space footprints. The accurate method becomes more attractive as the resolution increases since the relative difference between accurate and approximate methods decreases. For any of the three methods, the overhead becomes quite significant as we approach a resolution of one mega pixel. In order to further reduce the processing time, we can distribute the load balancing computations among the processing

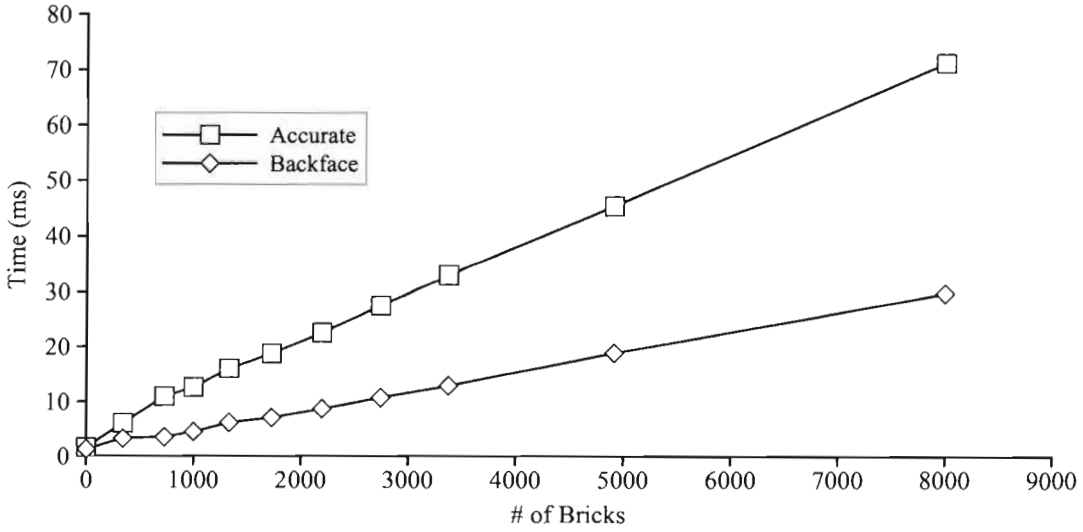

Figure 6.8: The overheads for the accurate and backface methods of computing the per pixel rendering cost. We compute the cost at a resolution of 128^2 and vary the number of bricks. The results for the splatting method are very similar to the backface method and thus are not shown.

units.

In Figure 6.10 we show how the performance of our load balancing algorithm scales for both the accurate and backface methods. We process 9072 bricks and compute the cost at a resolution of one mega pixel. We plot a line for just the computation performance as well as the total performance (computation plus communication and synchronization). The communication time is consistently less than two milliseconds for this resolution, regardless of how many processing units are used. However the synchronization cost from load imbalance adds over two milliseconds to this overhead. Since the computation is much faster for the backface method, the communication and synchronization overhead has a larger impact on the scaling performance.

6.4.2 Load Balancing Quality

In order to quantify how well the load balancing works, we take the difference between the render times of the fastest and slowest processing unit for each frame. We then average this over all frames and normalize by the average render time. The result is a measure of the deviation in performance among units as a percentage of the average render time.

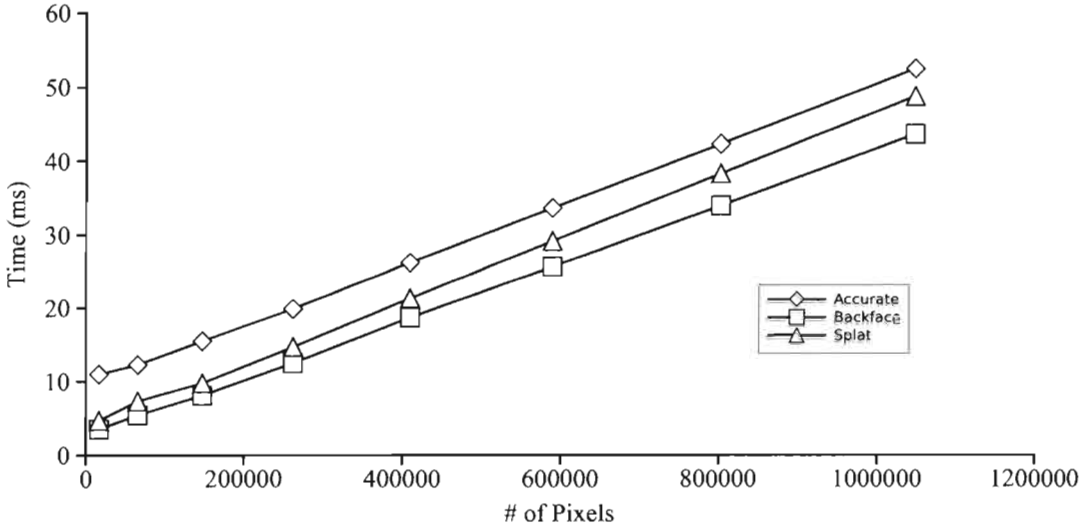

Figure 6.9: The image scaling behaviour for all three methods of computing the per pixel cost. We render 729 bricks and vary the resolution.

For these measurements we render the mummy head data set with a brick size of 64^3 to a one mega pixel image using six processing units. In Table 6.4 we show the results for all three methods of computing the pixel cost, each with two different resolutions and two different animations. The resolution for the load balancing computation is relative to the image resolution, meaning that the 'full' resolution is 1024^2 and the 'quarter' resolution is 512^2. The rotation animation views the full data set and rotates around the x-axis, while the zoom animation zooms in on the data set and then zooms back out.

Table 6.4: The deviation of render times among nodes, as a percentage of the average render time. We use two different types of animations and two resolutions for all three methods of computing the rendering cost.

	Average Render Time Deviation		
Animation (Resolution)	Accurate	Splatting	Back Face
Rotation (Full)	13.7%	10.8%	17.3%
Rotation (Quarter)	14.0%	10.9%	17.4%
Zoom (Full)	10.3%	9.1%	14.7%
Zoom (Quarter)	10.6%	9.5%	15.2%

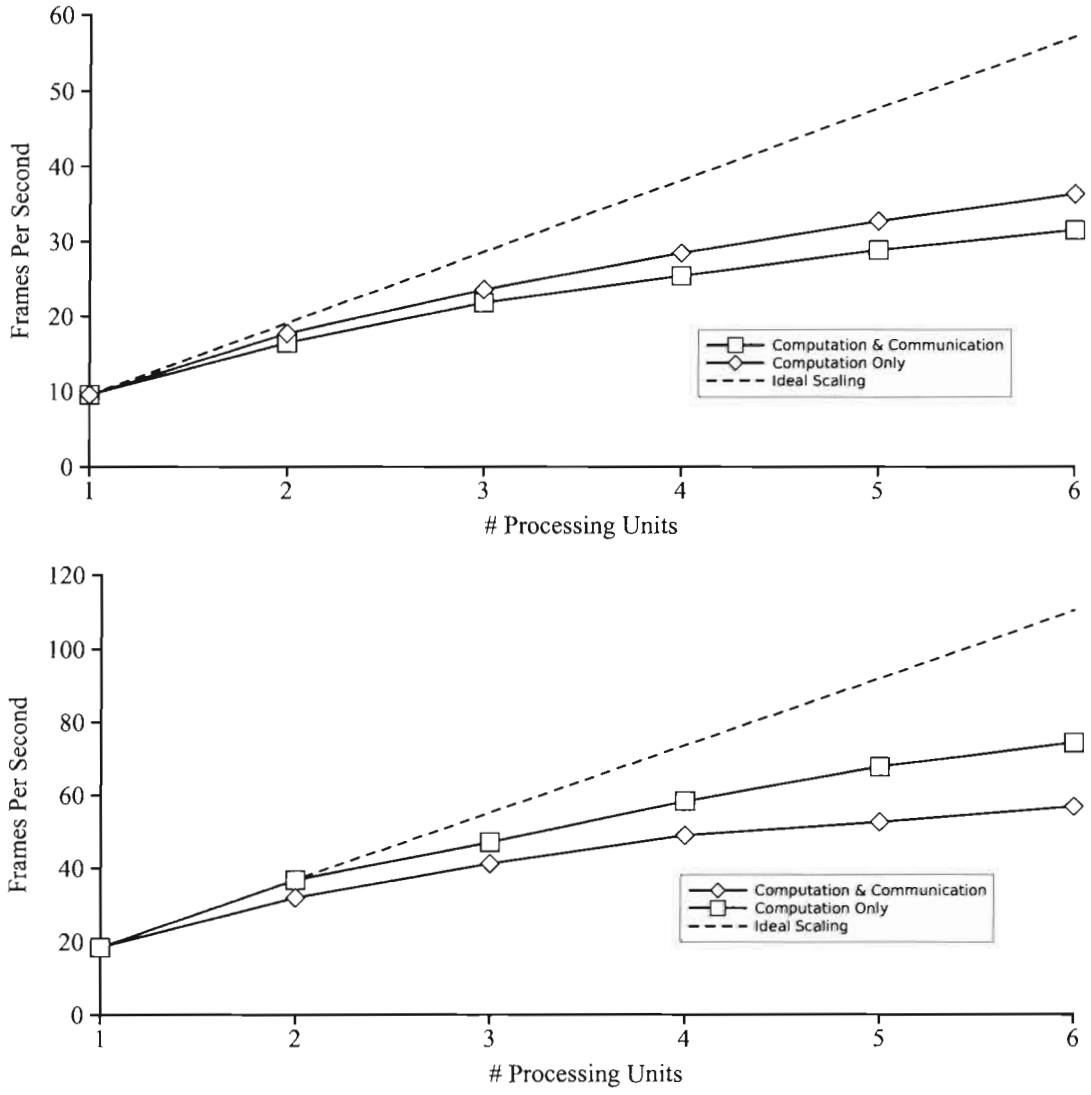

Figure 6.10: Performance scaling results for the accurate method (top) and the backface method (bottom). We compute the load balancing for 9702 bricks at a 1024^2 resolution. The scaling of the y-axis is different so that details are visible.

Unexpectedly, the splatting method does a slightly better job than the accurate method for load balancing. This is possibly due to a better balancing of the brick overheads, since the accurate method only gives an accurate measure of fragment processing costs. The lower resolution did not hurt the load balancing much for any method. The backface method gives worse performance than the other two, but it should still be better than load balancing based off of timing results from previous frames.

Lastly, we compare the quality of our load balancing algorithm against a simple method of load balancing that uses the relative performance of each unit in the previous frames. We compare the two algorithms in scenarios with varying levels of frame-to-frame coherence. We can vary the level of frame-to-frame coherence by taking a recorded animation that has fairly good coherence and skipping some number of frames in the animation. The more frames we skip, the lower the coherence will be. We use the same recorded animation that we used for the caching results since it includes a variety of viewing conditions. We render the mummy head data set into a 1024^2 view port with six processing units. For our load balancing algorithm we use the backface method and a quarter resolution for the pixel cost. The results are listed in Table 6.5.

Table 6.5: A comparison of our load balancing algorithm (cost based) to an algorithm that uses the performance of each unit in previous frames (performance based). The results are listed as a percentage of the average render time.

	Average Render Time Deviation	
Frames Skipped	Cost Based	Performance Based
0	$17.0\% \pm 12.7\%$	$22.8\% \pm 23.3\%$
2	$17.6\% \pm 13.0\%$	$46.9\% \pm 34.0\%$
4	$18.2\% \pm 13.6\%$	$60.6\% \pm 34.2\%$

Even when we are not skipping any frames in the animation, our cost based load balancing gives a better workload distribution than the performance based load balancing. As we decrease the level of frame-to-frame coherence (by skipping frames of the animation) the difference becomes even greater. The standard deviation of the quality of the load balancing also shows that our algorithm is more consistent in all scenarios. For this configuration, the amount of time each processing unit spent computing the load balancing for our algorithm is just over seven percent of the time each unit spent on the rendering stage. Therefore

we achieve a more consistent and better quality load balancing, regardless of the level of frame-to-frame coherence, for a small overhead in processing time.

6.5 Visibility Culling Results

The novel aspect of our approach to visibility culling is the ability to fine tune the performance so that there is minimal overheads when there is little to no occlusion and maximal performance when there is occlusion. For the fragment culling we have one essential parameter, the number of bricks to render between updates of the depth buffer. We refer to the groups of bricks rendered between updates as 'chunks'. For the brick culling there are two parameters: the threshold value for what percentage of bricks in a chunk need to be culled to obtain a net increase in performance, and the number of chunks we should randomly test so that the previous culling results do not go stale. Once we have experimentally fixed these parameters, we compare the efficiency of the visibility culling when used with a sort first versus sort last workload distribution.

6.5.1 Fragment Culling Performance

The amount of overhead incurred from updating the depth buffer depends on the number of updates performed and the image resolution. To test for the overhead we render the mummy head data, divided into 4536 bricks, with a transfer function that causes no occlusion and vary the number of updates to the depth buffer in a frame. We plot the difference between these runs and a run with zero updates to the depth buffer in Figure 6.11 for two different image resolutions. For thousands of bricks, updating the depth buffer for each brick would cause a significant overhead when no occlusion is happening. The negative overhead for small numbers of updates is unintuitive, but is probably due to the update passes flushing the the graphics pipeline at opportune moments.

Since the bricks are processed in an order that distributes consecutive bricks across the image plane, updating the depth buffer less frequently should not hurt the culling efficiency much. To test this we run the same tests as above but with a transfer function that does cause occlusion to occur (the same high opacity transfer function shown in Figure 6.18). First we examine the culling efficiency by using occlusion queries to count the number of fragments that pass the depth test when the bounding box of each brick is rendered. We plot the percent of fragments culled for varying numbers of updates with an image size of

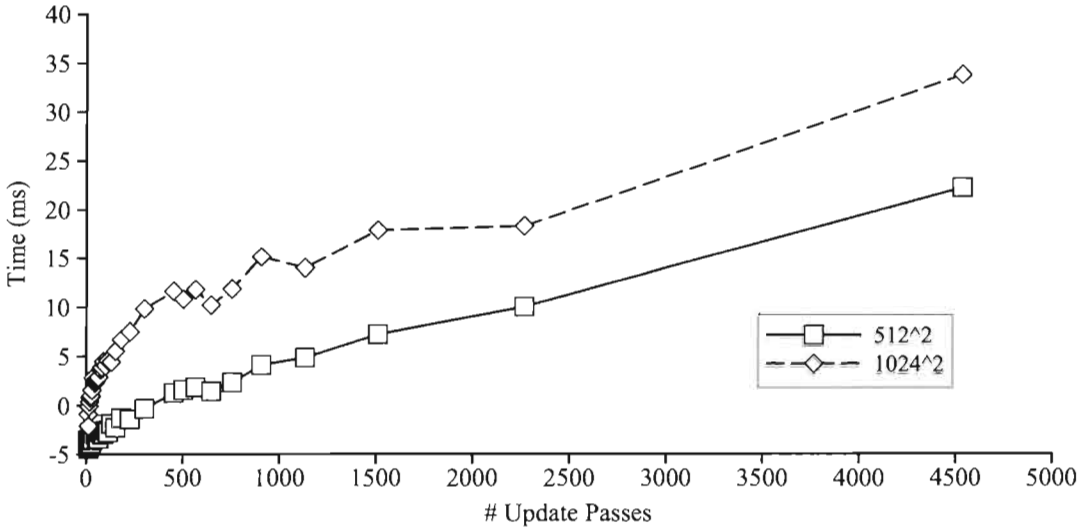

Figure 6.11: The amount of overhead incurred from updating the depth buffer for two different image resolutions. We render 4536 bricks with a transfer function that does not cause occlusion and vary the number of update passes in a frame from one to one for every brick. The overhead is calculated by taking the total time and subtracting the time from a run that does zero updates to the depth buffer.

1024^2 in Figure 6.12. The difference between updating the depth buffer 4536 times (once for every brick) and updating it eleven times (once for every four hundred bricks) is less than four percent.

Finally we look at the speed up achieved from the occlusion culling for different numbers of updates to the depth buffer. We use the same rendering parameters as in the last test and plot the resulting speed up in Figure 6.13. The peak of ninety-eight percent speed up comes from using eighteen updates (two-hundred and fifty bricks rendered between each update of the depth buffer). Doing as many as thirty-six updates achieves a ninety-four percent speed up in this scenario, and may be a more appropriate choice when more occlusion is occurring. Even greater speedups can be achieved when more slices are rendered or more expensive fragment shaders are used. For this test, the difference between the peak performance and the performance achieved when doing an update for every single brick is just over a hundred milliseconds. This difference is about three times the overhead we recorded for doing an update for every brick when there is no occlusion happening. This indicates that the updates to the depth buffer are stalling the pipeline, and these stalls have a larger effect

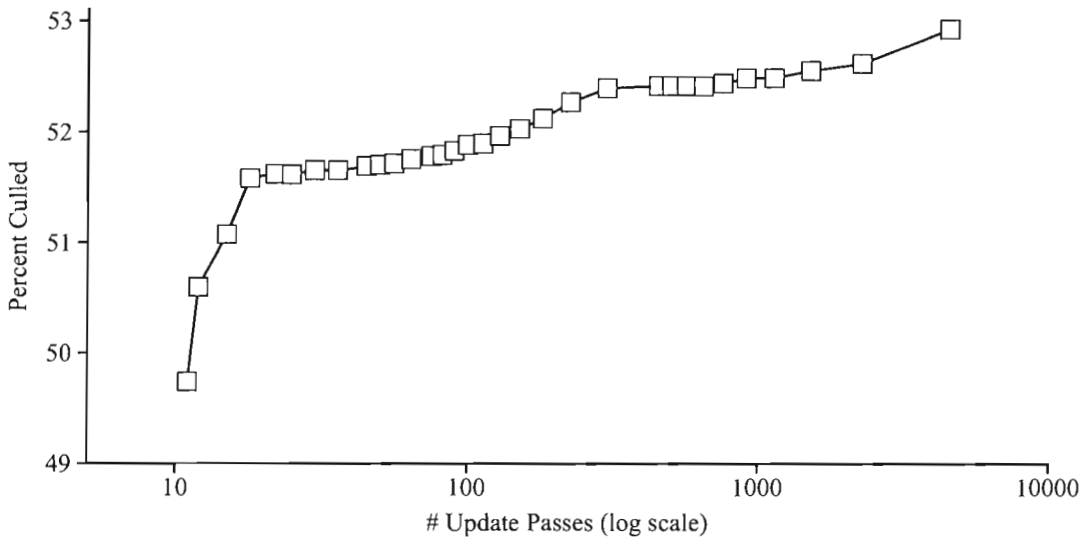

Figure 6.12: The percentage of fragments killed using different numbers of updates to the depth buffer in a frame. We render the mummy head data set as 4536 bricks with a high opacity transfer function and count how many fragments are killed by the depth test when we render the bounding box of each brick.

when occlusion is happening (since bricks are being rendered at a higher rate).

6.5.2 Occlusion Query Performance

The amount of overhead incurred from performing occlusion queries depends primarily on how many batches of queries are made, and to a lesser degree how many queries are in those batches. We experimentally quantify these overheads using the same testing methodology as we did in the previous section. We render the mummy head data set as 4536 bricks with a transfer function that does not cause any occlusion into a 1024^2 view port. We use two different chunk sizes, one hundred and two hundred bricks, and we vary the number of chunks that we perform the occlusion queries for. We plot the resulting overheads in Figure 6.14. It is clear that while the number of queries made in each batch has an effect on the overhead, the number of batches has a much greater effect.

To determine the threshold value for what percentage of bricks in a chunk need to be culled in order to see a net increase in performance, we need to know how long it takes to render a brick that is completely culled by the depth test. The time it takes to render a brick that is culled by the depth test depends on many rendering parameters including:

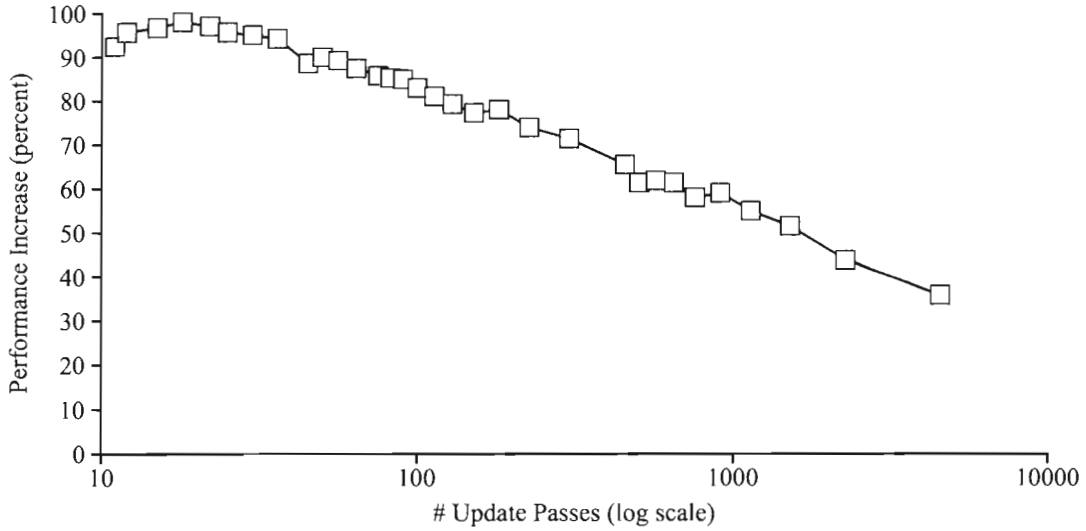

Figure 6.13: The performance increase achieved by the occlusion culling for different numbers of updates to the depth buffer. We render the mummy head data set as 4536 bricks with a high occlusion transfer function. We render into a 1024^2 image with the slice distance equal to half the sample distance.

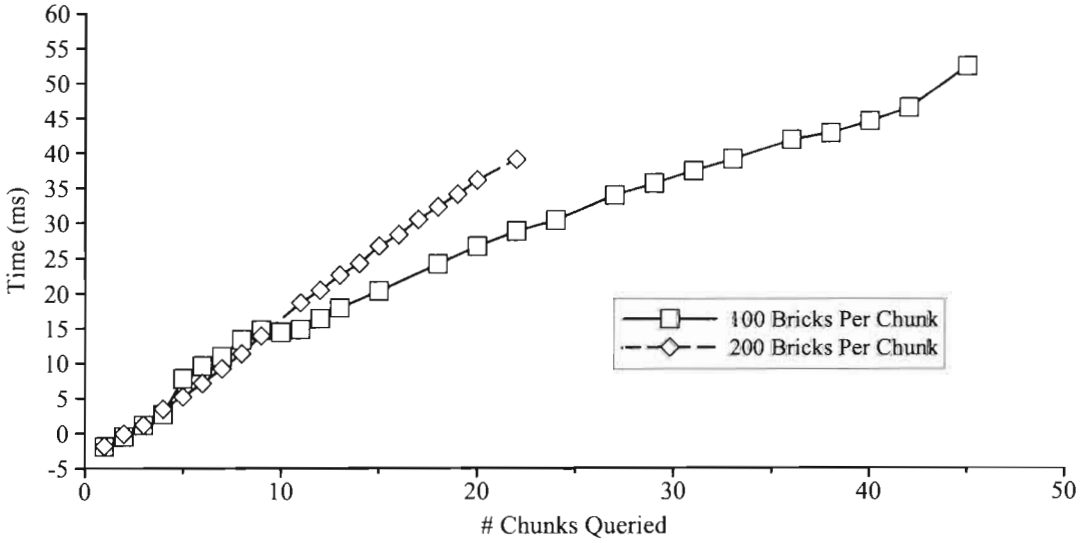

Figure 6.14: The amount of overhead incurred from occlusion queries. We use the mummy head data set divided into 4536 bricks and 1024^2 image. We use two different chunk sizes, 100 bricks (forty-four chunks total) and 200 bricks (twenty-two chunks total). We vary the number of these chunks that we query in each frame. We plot the difference between the time for the runs when queries are performed and a runs where they are not.

the image resolution, the number of slices, the number of components in the texture, and the number of samples in the brick. In Table 6.6 we show the rendering time and the computed threshold for a number of different rendering parameters. The threshold values are computed for a chunk size of two hundred for the 32^3 brick size and twenty-five for the 64^3 brick size. The higher the threshold is, the smaller the performance gain that can be achieved. In the cases where the threshold is above one hundred percent, the speed up from skipping culled bricks cannot overcome the overhead of performing the queries.

Table 6.6: A table of the time it takes to render a brick that is completely culled by the depth test, and the resulting threshold for what percentage of bricks need to be culled in a chunk to overcome the query overheads, for a variety of rendering parameters. The slice ratio parameter is the ratio of the distance between slices versus the distance between samples. The threshold is computed for a chunk size of one hundred for the 32^3 brick size and ten for the 64^3 brick size.

Rendering Parameters					
Brick Size	Image Size	Slice Ratio	# Components	Time (ms)	Threshold
32^3	512^2	0.5	1	0.012	80%
32^3	512^2	0.5	4	0.026	38%
32^3	512^2	0.125	1	0.022	44%
32^3	512^2	0.125	4	0.055	18%
32^3	1024^2	0.5	1	0.012	75%
32^3	1024^2	0.5	4	0.028	32%
32^3	1024^2	0.125	1	0.034	26%
32^3	1024^2	0.125	4	0.062	14%
64^3	512^2	0.5	1	0.098	53%
64^3	512^2	0.5	4	0.211	25%
64^3	512^2	0.125	1	0.177	30%
64^3	512^2	0.125	4	0.414	13%
64^3	1024^2	0.5	1	0.095	58%
64^3	1024^2	0.5	4	0.208	27%
64^3	1024^2	0.125	1	0.236	23%
64^3	1024^2	0.125	4	0.466	12%

The remaining parameter, the percentage of chunks to randomly query, should be set to the highest value that does not exceed an acceptable overhead for the application. When the average frame rate is higher, then the amount of overhead that is acceptable will be lower. However, a higher frame rate will often result in more frame-to-frame coherence and

thus fewer chunks need to be randomly queried in a frame. If we are breaking the data set up into about forty chunks, then randomly querying ten percent of the chunks would result in an overhead of approximately four to five milliseconds. This would also mean that the culling results for the chunks should not be much older than ten frames, which should be acceptable as long as the frame-to-frame coherence is not extremely low.

Finally, we look at how much performance we can gain from using the occlusion queries. The threshold value must be low to see a significant performance increase. Therefore we use four component textures and a slicing distance that is one eighth of the sampling distance. We render a cropped version of the mummy head data set ($510 \times 510 \times 291$) with a brick size of 32^3 into a 1024^2 image. We use the same high occlusion transfer function that we used in earlier tests, which gives us 1680 non-empty bricks. For this scenario, we find that a chunk size of one hundred gives us the best performance from killing occluded fragments. We use a threshold value of fourteen percent (see Table 6.6) for determining which chunks to query, and we randomly query ten percent of the chunks in each frame. Using these parameters, the performance is increased by just over six percent compared to just killing occluded fragments. This increase is quite small compared to the other acceleration strategies, but with rendering algorithms that have a higher per brick cost the benefit would be magnified. It is also possible to avoid loading bricks that are culled by the queries, which could be a significant benefit in some scenarios. With multi-resolution volume rendering techniques, the results from the occlusion queries could also be used to help choose a resolution for bricks that are not completely occluded.

6.5.3 Sort First vs Sort Last

Our last set of experiments for the visibility culling focuses on the effect of using a sort first versus sort last workload distribution. While we expect the sort first approach to achieve a similar speed up regardless of the number of processing units, we expect the speed up to rapidly diminish as we add processing units to a sort last distribution. We use the mummy head data set with a brick size of 32^3, but we don't cull empty bricks since we only have simple static load balancing for the sort last approach. We use the performance based load balancing for the sort first experiments since our pixel cost load balancing does not account for occlusion. We use an animation where the full data set is visible and is rotated around the y-axis. These viewing conditions should minimize the load imbalance for the static sort last distribution. We set the slice distance to be one eighth of the of the sample distance

and render into a 1024^2 view port. The results for using one to six processing units is given in Figure 6.15.

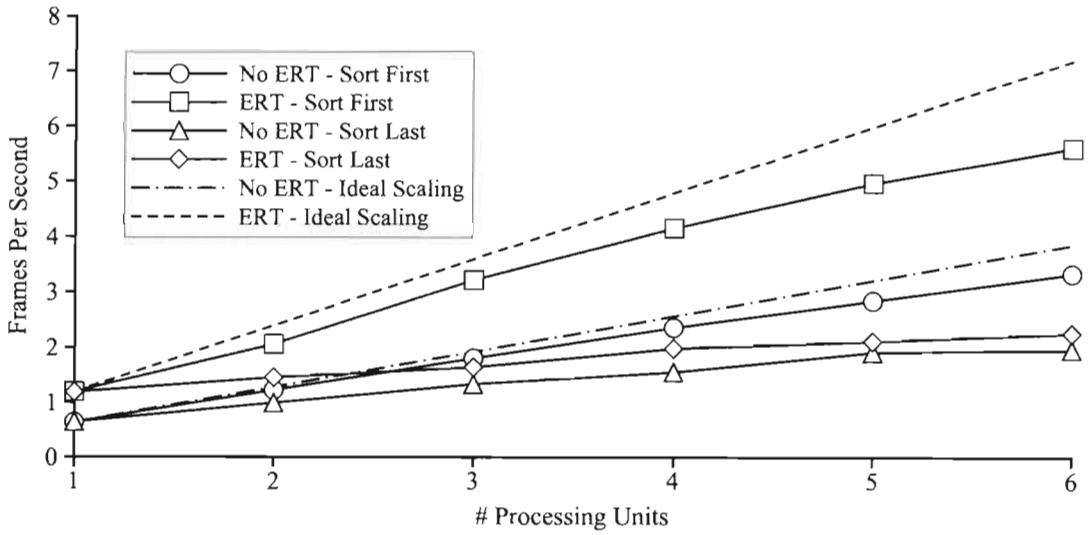

Figure 6.15: The performance scaling for sort first and sort last distributions, both with and without Early Ray Termination (ERT).

The sort first distribution shows some variation in the amount of speed up from the visibility culling, but remains within twenty percent of what is achieved on a single unit. In comparison, the sort last distribution loses as much as seventy-five percent of the speed up. Clearly this is a significant advantage for sort first workload distributions when occlusion is occurring.

6.6 Shadowed Rendering Results

Subdividing the data into bricks is necessary to achieve data scalability, however the shadowed rendering algorithm must render the data one slice at a time rather than one brick at a time. This means that each slice that we render must be rendered as a collection of smaller pieces from all the bricks that the slice intersects. For each piece of each slice we are required to change some of the rendering state such as the texture that is bound and the transformation matrix. This incurs a much greater per brick overhead, which limits us to a larger brick size than we have used in the previous sections.

When we are parallelizing the computation another complication arises; we need to avoid

rendering the parts of the data that are outside the light's frustum when we render from the camera's point of view. If we are using slice templates, then our only option is to discard some of the fragments in the fragment shader. This is far from ideal since the discarded fragments have the same processing cost as the rendered fragments. If we don't use the slice templates, then we can either utilize the user specified clip planes in OpenGL to kill fragments early in the pipeline or we can clip the slice geometry against the frustum as the slices are generated. The latter option would provide the best performance but we use the former option since it is much simpler to implement and the performance difference should be minimal as long as the bricks are not too large.

6.6.1 Bricking Overheads

To test the bricking overheads, we render the mummy head data set with a variety of brick sizes. We do this both with and without culling empty bricks to see the total overhead and improvement from the culling. In Figure 6.16 we show the results for both a 512^2 and 1024^2 view port using a slicing distance that is equal to the sampling distance. For the smaller image resolution we are limited to a brick size of 128^3 before we see a large drop in performance. For the larger image size there is a greater benefit from empty space leaping and so we can use bricks as small as 64^3. With different data sets and transfer functions it is possible to see a net increase in performance from culling empty bricks, but the gain is likely to be small for all but the most extreme circumstances.

6.6.2 Scaling

Next we would like to look at how well the performance scales when we use multiple processing units to render the same data set. In Figure 6.17 we show the scaling results for a brick size of 64^3 and 128^3 for the same data set as above being rendered to one and two mega pixel images. For the smaller image resolution the larger brick size performs slightly better, and the scaling for both brick sizes maxes out at five processing units. For the larger image resolution the smaller brick size performs slightly better and we continue to see some improvement all the way up to six processing units.

While the scaling is far from ideal, this is the only available method for speeding up this rendering algorithm and for applying it to data sets that are larger than the texture memory on a single GPU. The various overheads of our parallel shadowing method are a

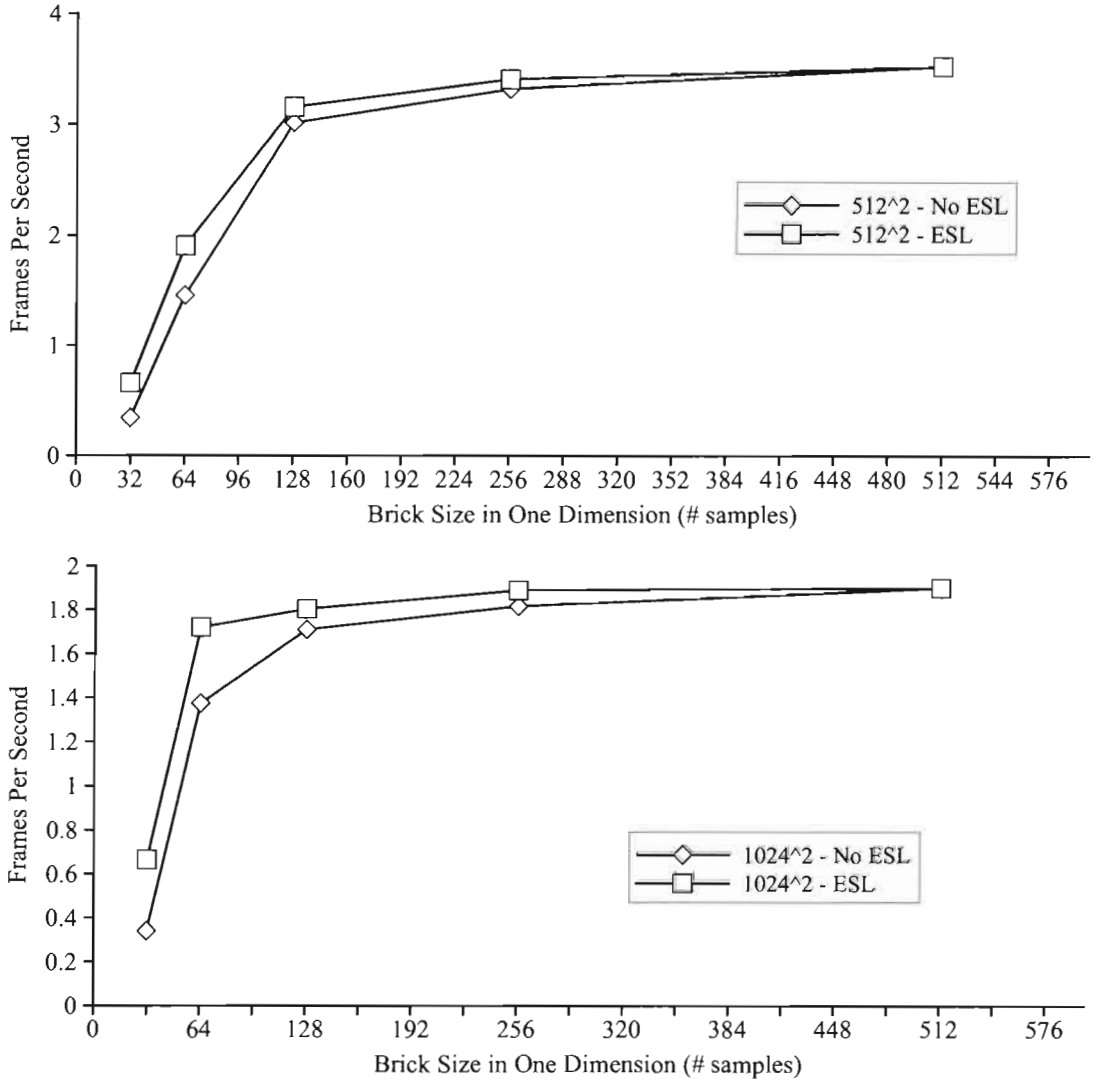

Figure 6.16: The performance of shadowed volume rendering on the mummy head data set subdivided into different brick sizes. Two image resolutions are shown as well as results with and with out culling empty bricks (ESL). The distance between slices is equal to the distance between samples. The scale of the y-axis is different so that details remain visible.

large part of why we achieve such poor scaling, but more important is the fact that the main overhead (switching the render target twice for every slice) is not reduced when we use multiple processing units. We should see better scalability on newer graphics hardware that reduces the overhead from render target switching. A better network interconnect would also help by reducing the compositing overheads.

6.7 Overall Performance

Our primary interest in the previous sections of this chapter was to isolate the impact of different overheads and acceleration techniques. In this chapter we explore the total performance of our rendering system for a number of large data sets being rendered at high resolutions. We start by giving an overview of the experiment setup and then present the results.

6.7.1 Experiment Setup

We use the larger portion of the mummy data set, the visble male data set, and the Richtmyer-Meshkov data set for these experiments. Each data set has two different transfer functions which we use in the experiments. One transfer function has a relatively high opacity and thus produces isosurface like images while the other has a relatively low opacity and produces more cloud like images. The transfer functions are designed so that the same number of bricks are culled for both the high and low opacity versions. The transfer functions for the mummy, Richtmyer-Meshkov, and visible male data sets can be seen in Figure 6.18, Figure 6.19, and Figure 6.20 respectively.

For the mummy data set we choose a brick size of 32^3, which results in 5571 visible bricks for our chosen transfer function. For the Richtmyer-Meshkov and visible human data sets we choose a brick size of 64^3, which results in 1662 and 2265 visible bricks for their respective transfer functions. The total amount of texture memory required for storing the visible bricks is then 696 MB, 416 MB, and 566 MB. We allocate 211 MB of memory for the texture cache of each GPU, giving us a total of 1266 MB of texture space. The largest data set (after culling) we could render with this set up is just over fifty percent of the total texture memory available. While sort last would allow us to essentially use all the available texture memory, the GPUs are not capable of rendering that much data at interactive frame rates and a good quality.

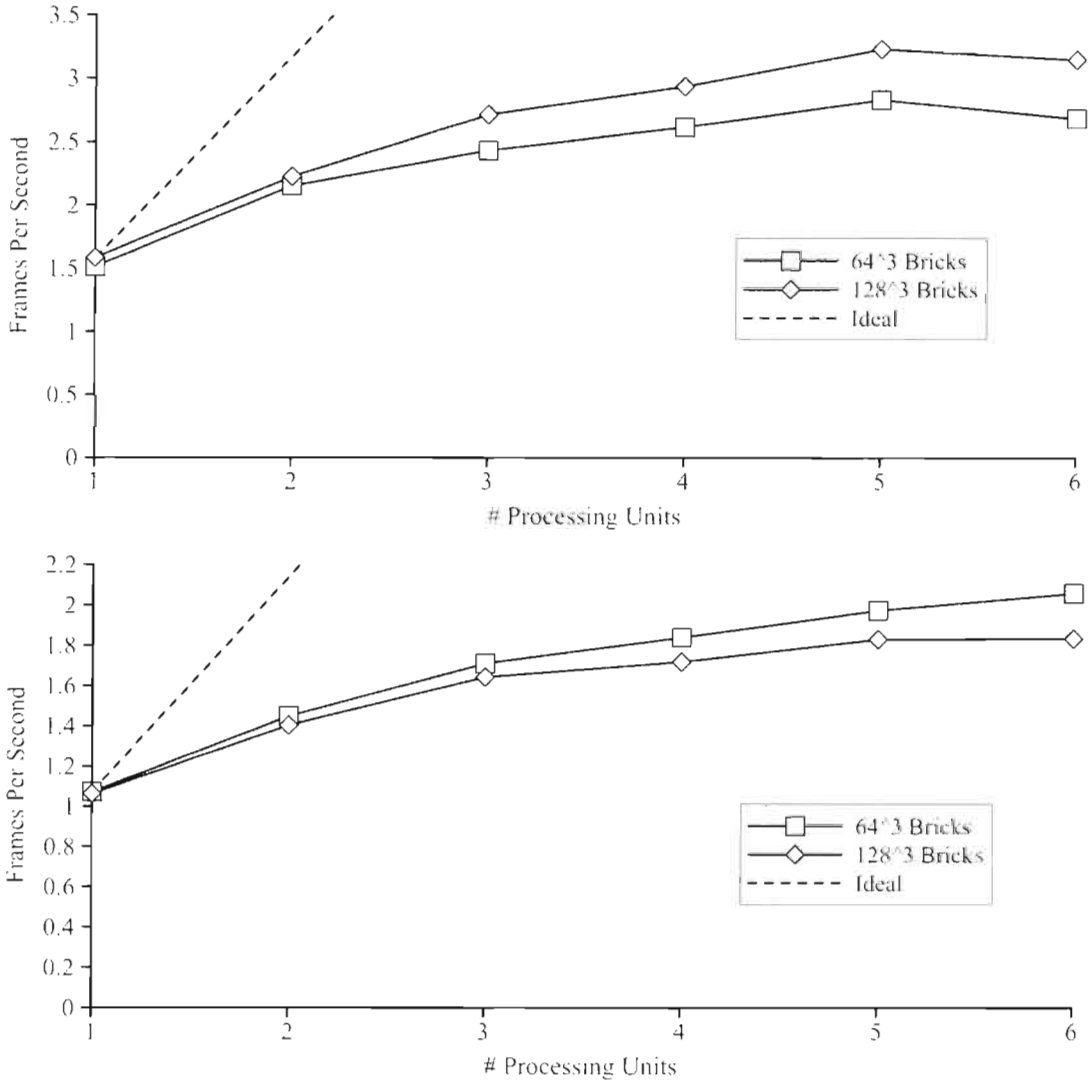

Figure 6.17: The scaling of the shadowed rendering for up to six processing units. The top graph shows the scaling for a one mega pixel image and the bottom graph shows the results for a two mega pixel image. On each graph we show two curves, one for a brick size of 64^3 and one for a size of 128^3. The scaling of the y-axis is different to preserve details.

To test the performance of our rendering system, we use four different rotation animations and seven different zooming animations. The rotation animations rotate the camera around all three of the major axes as well as the vector average of all three axes. The camera is placed at a distance that tries to maximize the size of the data on the screen while keeping the frustum culling to a minimum. The zooming animations start at seven different positions that are far away from the data and then zoom in towards the origin and then zoom back out to the starting positions. The starting positions lie on the three major axes, the three half angles between the major axes, and the diagonal between all three major axes. For all of the animations, we run them at a variety of speeds so that the effects of data loading can be observed.

We render into one mega pixel (1024^2) and two mega pixel (1792×1170) images with the slice distance set to be half the largest sampling distance. We cull occluded fragments using the depth test with twenty updates to the depth buffer for each frame. We use our pixel cost based load balancing technique with the back face method for calculating the cost and the resolution set to one quarter of the image resolution. The pixel cost load balancing cannot account for occlusion, but it allows us to consistently render larger data sets than what is possible with the performance based load balancing. With the performance based load balancing it is not uncommon for the screen distribution to jump around and require a processing unit to render more data than it can store in texture memory. We precache at most seven and a half megabytes in each frame, which corresponds to thirty bricks with single component 64^3 textures and sixty bricks with four component 32^3 textures. For these texture dimensions we have 458.5 MB/s and 537 MB/s of bandwidth, giving maximum precache loading times of about sixteen and fourteen milliseconds.

6.7.2 Results

Since the amount of loading depends on the frame-to-frame coherence, the performance of any data scalable sort first system will decrease when an animation is sped up. We show this drop in performance for the visible human data set in Figure 6.21 by averaging the rotation and zooming animations separately for each animation speed. As expected, the rotations experience a larger performance drop than the zooming animations. The visible human has the largest relative drop in performance among the three data sets, but it is still a relatively small at eighteen percent.

While the loading does not have that much of an impact on the average performance, it

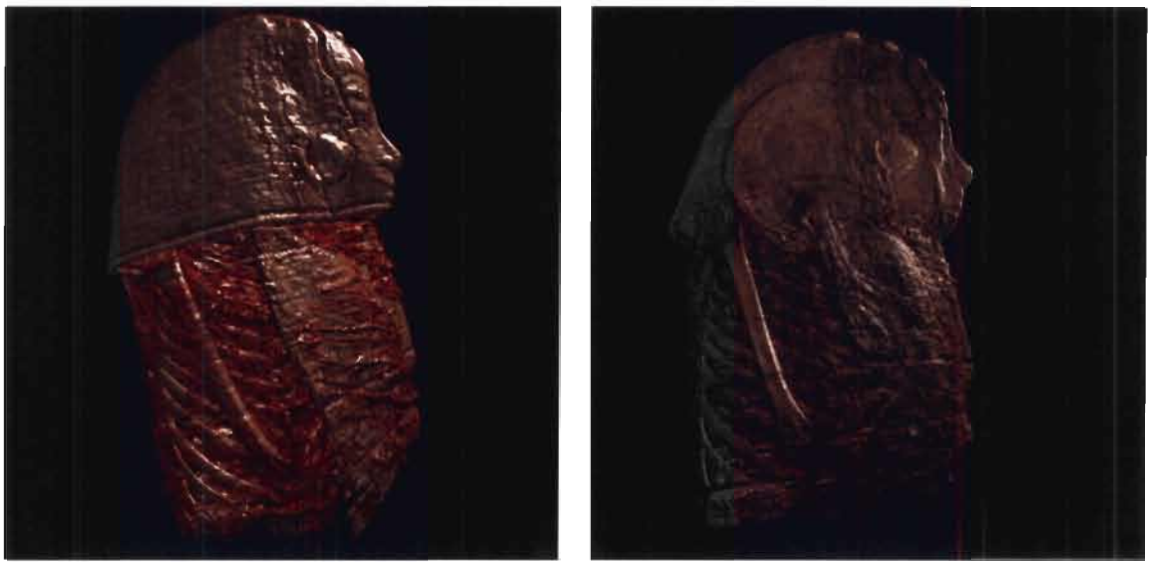

Figure 6.18: The mummy data set with its high and low opacity transfer functions. Gradients are loaded into the textures with the scalar data and used to perform lighting computations.

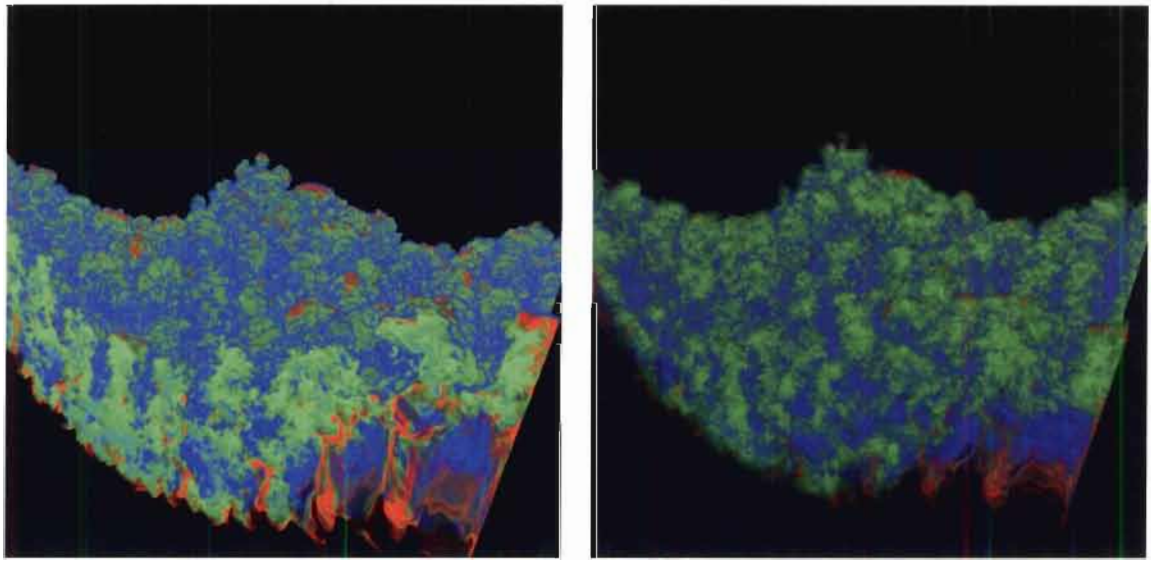

Figure 6.19: The Richtmyer-Meshkov data set with its high and low opacity transfer functions.

Figure 6.20: The visible male data set with its high and low opacity transfer functions.

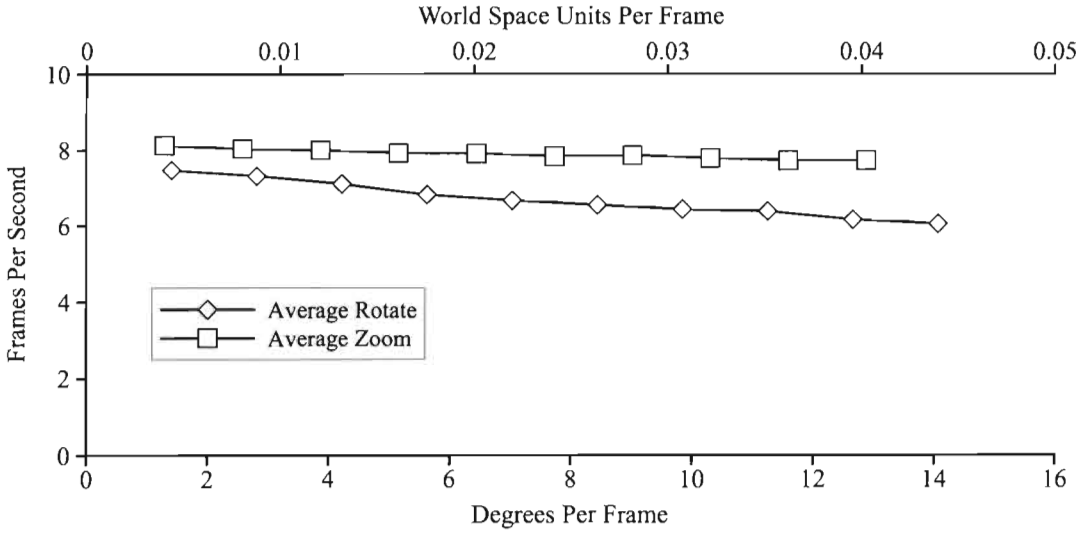

Figure 6.21: A graph of the average performance when rotating and zooming the visible male data set at different speeds. These results are for the high opacity transfer function and a one mega pixel image.

can cause sudden performance hitches when there are spikes in the amount of bricks loaded. We look at the loading behaviour of the Richtmyer-Meshkov and mummy data sets since they have the smallest and largest texture memory footprints respectively. The results are shown for the Richtmyer-Meshkov in Figure 6.22 by plotting the maximum number of bricks loaded by any processing unit in each frame of an animation. The animation we use rotates the camera twice around the x-axis, which is illustrated by the series of images beneath the graph. We plot three different speeds of rotation. For the slowest speed, the amount of loading is consistently beneath the threshold of thirty bricks per frame. The medium speed loads the threshold amount on most frames but never goes above it. For the fastest speed we start to see spikes in the loading when the bricks become the most condensed on the screen.

The loading results for the mummy data set are given in Figure 6.23. Since the mummy data set is rendered with four bytes per voxel, we are preloading one quarter of the voxels compared to the other data sets. Since the mummy data set is also right at the limit of how large of a data set can be rendered with our cluster, there is less memory available to each processing unit for caching. These issues cause much larger spikes in the loading for the mummy data set. We still see the largest spikes when the data condenses in the screen space, just like the Richtmyer-Meshkov data set. However, we now start to see some spikes at the medium rotation speed. It should also be noted that a constant speed rotation is fairly unlikely in the real world. Our caching algorithm works best for short bursts of fast rotation which are broken up by slower rotations and zooming animations.

Finally, we show a detailed break down of the average performance for all data sets, transfer functions, and image resolutions in Figure 6.24. These results are the average of the rotation animations at a speed of seven degrees per frame. The killing of occluded fragments results in about double the performance for the high occlusion versus low occlusion transfer functions. The rendering time is significantly larger for the four component textures, as is expected based on the results in Section 6.1.2. The load balancing and data loading take comparable amounts of time, and both are quite small relative to the rendering and the final gather time. The frame buffer read back is essentially inconsequential on PCI-E and scales linearly with the number of processing units when doing sort first.

If we were to add more processing units we would expect all of the overheads to remain the same or go down. The rendering time still dominates the total time, and would continue to shrink as we add processing units. Therefore we would expect to continue to see

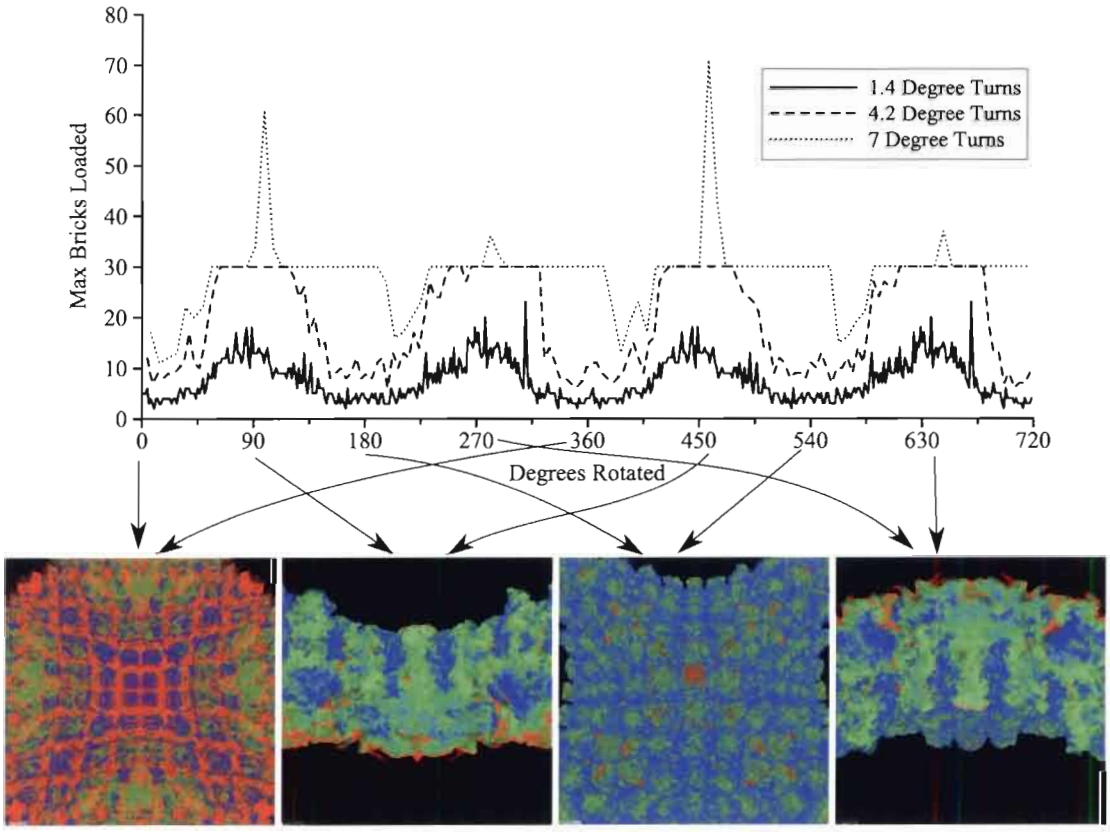

Figure 6.22: The maximum amount of bricks loaded among all processing units for each frame of rendering the Richtmyer-Meshkov data set. There are three plots corresponding to three different speeds of rotation around the x-axis. Underneath the plot, we show the frames of the animation corresponding to the major ticks on the x-axis.

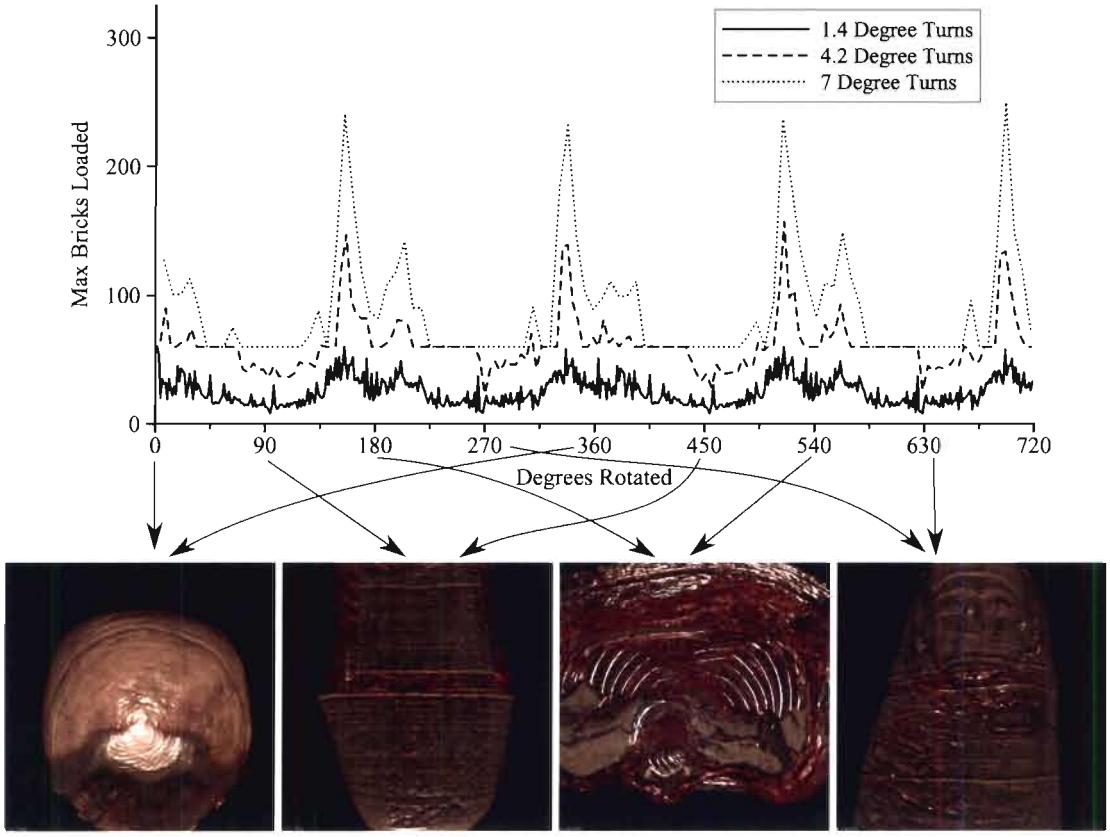

Figure 6.23: The maximum amount of bricks loaded among all processing units for each frame of rendering the mummy data set. There are three plots corresponding to three different speeds of rotation around the x-axis. Underneath the plot, we show the frames of the animation corresponding to the major ticks on the x-axis.

performance scaling for these data sets as we add many more processing units. Eventually the compositing time will dominate the total time when using gigabit ethernet. This is true for both sort first and sort last approaches, but the total compositing time is much worse for sort last.

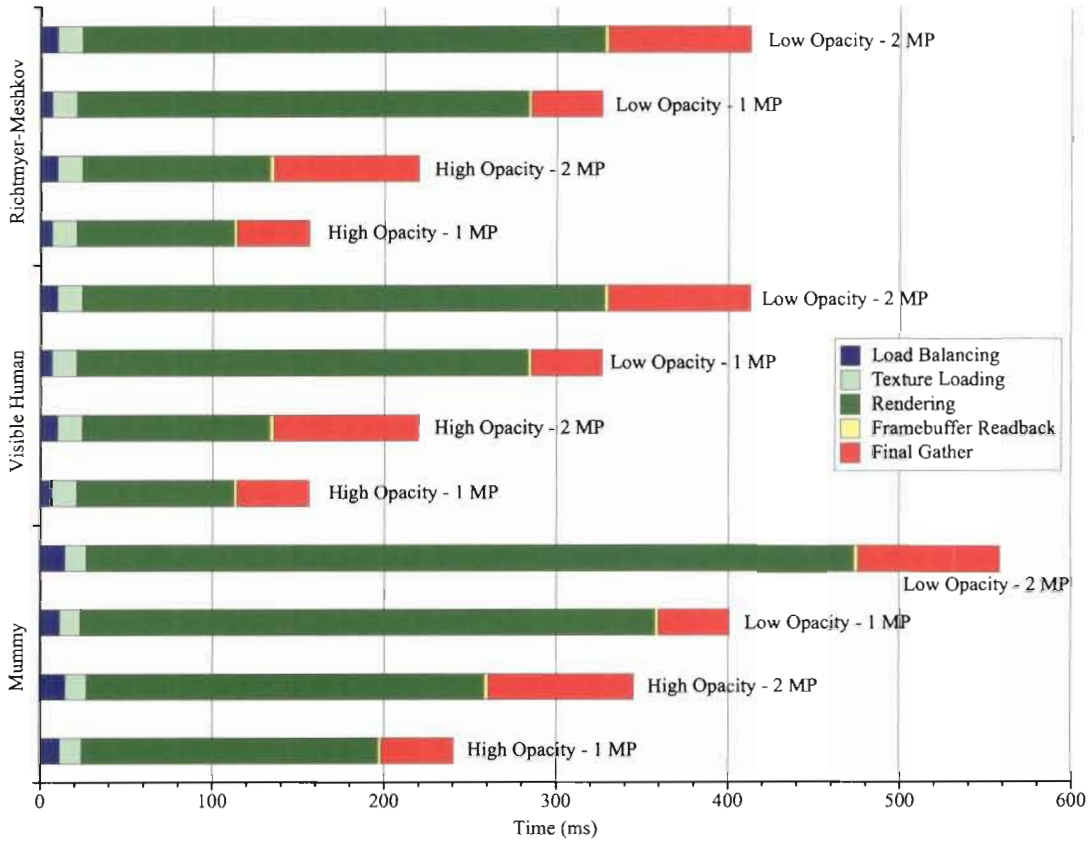

Figure 6.24: A detailed break down of how the processing time is split up among the different stages of the parallel rendering pipeline. All four experiments are shown for all three data sets and a rotation speed of seven degrees per frame.

This demonstrates the viability of a sort first distribution for data scalable volume rendering. Our algorithm for killing occluded fragments gives almost a two fold increase in performance when occlusion occurs, and essentially no overhead when occlusion does not occur. Our consistent load balancing algorithm allows for larger data sets to be rendered compared to performance based load balancing. The proximity caching algorithm we use minimizes spikes in data loading and thus provides a more consistent overhead. Finally, both the load balancing and data loading take up a relatively small portion of the total

time for each frame.

Chapter 7

Conclusions

The utility of sort first workload distributions for parallel volume rendering has been demonstrated successfully. Due to concerns of data scalability, the overwhelming majority of the state of the art work on parallel volume rendering has focused on sort last distributions. We have shown that the data loading overhead in sort first distributions can be much smaller than the compositing overhead in sort last distributions for many scenarios. Three important issues for data scalable sort first rendering have been addressed: how to efficiently render a subdivided data set, how to handle the loading of pieces of a subdivided data set, and how to guarantee a balanced distribution of the work load. More importantly, we have shown how the locality of the data and processing along rays afforded by a sort first distribution can allow for efficient adaptations of many existing volume rendering algorithms to a parallel environment.

The templated slicing technique described in Chapter 3 allows the data set to be divided into smaller pieces without the generation of slice vertices becoming the bottleneck. This allows a much finer granularity for any rendering algorithm that requires a subdivided data set. For our purposes, this means we can cull empty portions of the data set more accurately and reduce the memory overhead in our data scalable sort first distribution.

The simple proximity caching technique described in Chapter 3 has been demonstrated to dramatically reduce sudden spikes in loading for our data scalable sort first distribution. While the proximity caching actually causes more data loading overall, it is preferable in interactive applications to have a consistent overhead. We have demonstrated that this consistent overhead can be much smaller than the overhead incurred from compositing when using a sort last distribution. While we have not addressed the issue of data loading

to system memory, the same technique could be used. The bandwidth over the network or from disk would be lower than what is available to the GPU, but the size of the cache would be much larger and it would be possible to perform the loading asynchronously while the GPU performs the rendering.

The load balancing algorithm that is presented in Chapter 4 has been shown to give a better and more consistent distribution of the workload than the existing method of using the performance results from previous frames. Even in situations where there is relatively good frame-to-frame coherence, our algorithm outperforms the alternative. As the level of frame-to-frame coherence decreases, our algorithm outperforms the alternative by an increasing margin. The ability to provide a consistently good load balancing is going to be even more vital in rendering environments that have a larger number of processing units.

Our optimized visibility culling algorithm, which we discussed in Chapter 5, has achieved over three times the speed up of existing approaches when a high occlusion transfer function is used. Simultaneously, we have essentially eliminated the overheads from the visibility culling when a low occlusion transfer function is used. We have demonstrated the decrease in the efficiency of visibility culling when a sort last distribution is used, due to the lack of ray coherence.

The parallel version of an existing shadowing algorithm, which was also discussed in Chapter 5, allows for data scalable interactive shadowed volume rendering. Our hybrid workload distribution scheme keeps the rays from the light source local to a single machine. Without this ray coherence, there would be too much synchronization required for an efficient parallel algorithm. We also demonstrate an efficient method of performing the compositing in our hybrid distribution. By using direct send compositing we make sure that no processing unit is ever idle during the compositing stage. Additionally, our asynchronous implementation of the compositing reduces the total compositing time to just slightly more than the time it takes to communicate the intermediate images over the network.

In addition to the visibility culling and shadowing algorithms, in Chapter 5 we also discuss a number of other rendering algorithms and visualization techniques which could benefit from the ray coherence afforded by a sort first distribution. In the past, almost all research on parallel volume rendering has focused on the standard emission and absorption lighting model (or the even simpler emission only model). With the advent of affordable GPUs, we have seen a dramatic increase in the processing power available to each processing unit. This increased processing power allows for more complicated lighting models and

visualization techniques. Adapting these new algorithms to a parallel environment requires a new set of considerations when choosing the method of distributing the workload. We have demonstrated that ray coherence is a vital consideration for many of these new algorithms.

7.1 Future Work

There are many interesting avenues for expanding upon the work in this thesis. Many of the algorithms presented can be expanded to include new functionality or adapted to a new application. We attempt to highlight both the pragmatically useful as well theoretically interesting future directions in this section.

The load balancing algorithm presented in Chapter 4 could play an important role in parallel rendering of time varying data. Frame to frame coherence cannot be assumed when rendering time varying data and thus the performance results from previous frames cannot be used for load balancing. Since the pixel cost load balancing algorithm we have presented works strictly with data from the current frame, it will work just as well for time varying data as it does for static data. Loading data is also unavoidable when rendering time varying data, and thus it would cease to be an overhead that is particular to sort first approaches.

While we have provided a variety of techniques for speeding up our load balancing algorithm at the cost of some accuracy, we have not provided a method of reducing the number of bricks that are processed. This could be achieved by using a octree data structure so that a single larger brick could be used in place of a group of smaller bricks. The only downside would be some implementation overhead, but the benefits could be significant for data sets with large numbers of bricks. A simpler alternative would be to just compute the load balancing with a coarser bricking everywhere.

The occlusion culling techniques that were described in Chapter 5 could show even greater benefits when used with more expensive rendering techniques. This includes out of core rendering, compressed volume rendering, rendering with higher order interpolants, and much more. Compressed rendering is an especially attractive pairing since it could significantly reduce the amount of data that needs to be loaded for the sort first approach.

The visibility results from the occlusion queries could be used in a number of interesting ways. When performing multi resolution rendering, the visibility results could help determine the appropriate resolution to render a brick at. Visibility results could also be used to weight the pixel cost computed for the load balancing so that occlusion is accounted for in

the load balancing.

The proximity based caching algorithm that we use in this paper could potentially be improved by considering the camera movement in the previous frames and trying to predict where the camera will move in the next frame. The caching overhead could also potentially be reduced by waiting to load bricks until after the occlusion queries are done, so that the loading of occluded bricks can be skipped. This is especially appealing since the frames that have the spikes in loading overhead (when the data compresses in the screen space) are also usually the frames that have the most occlusion. Ultimately, the caching performance would need to be tested with a user study to see if the caching scheme is suitable for real world use.

Appendix A

System Details

A.1 System Overview

Our system consists of a number of processing units, each with its own CPU and GPU, which act together as a render server. A view client can then connect to the server and provide a configuration file which specifies the data set and rendering parameters to use. Once a connection is established, the render server waits for render request packets which specify the updated viewing conditions. When a render request arrives it is distributed among the processing units which each do their respective parts of the workload before relaying a portion of the final image back to the view client for display. Once the next user input is made on the view client the whole process repeats.

Our parallel environment consists of a cluster of workstations, each with an Intel Xeon Processor (NetBurst architecture)in the range of 2.8 to 3.2 GHz and at least two gigabytes of RAM. The workstations are connected with gigabit ethernet through a switch with flow control to ease congestion. Each workstation has one NVidia Geforce 6800 Ultra with 256MB of memory connected over a PCI-E bus. Our system could also be used on multipipe machines (multiple GPUs on a single SMP) by creating an individual X11 server and render process for each GPU.

Our implementation is written in C++ for processing done on the CPU and the OpenGL Shading Language (GLSL) for the processing done on the GPU. We use the MPICH2 library for communication among rendering processes and TCP/IP for communication between the rendering processes and the view client. We use the GLUT library to display the final results and handle user interaction on the view client.

A.2 Volumetric Data Sets

We use a total of four real world data sets in this thesis, both for illustrative and experimental purposes. The fish data set that is used in Chapter 1 to illustrate different workload distribution methods is a CT scan of a Karpfen fish. The mummy head data set that is used for illustrative purposes in Chapter 4 and Chapter 5 is also a CT scan, but of a mummy inside of a sarcophagus. In all of the illustrative renderings we load gradients into the textures and perform Phong illumination.

We also use the mummy head data set extensively for testing. The mummy head data set is one of many equally sized portions ($510 \times 510 \times 400$) of the full mummy data set. This data set requires a pretty small brick size to cull empty space accurately for most transfer functions. We use this data set in many of the experiments where we want to isolate a particular overhead since it is small enough to be replicated among all of the GPUs if one byte per voxel is used, yet it is still large enough to require some significant processing power.

For tests where the results are not dependent on the under lying data (no culling empty bricks and no occlusion culling) a synthetic data set (256^3) is used where the values decrease radially from the center. This radial data set is used simply because it is small enough to fit onto a single GPU even when there are four bytes per voxel and the sample spacing is isotropic.

For the overall performance experiments we combine the top two portions of the mummy data so that we have the head and torso ($510 \times 510 \times 800$). There is a visible discontinuity between the portions since they have not been registered, but this does not effect our results. We load the gradients into the textures for this data set so that the size increases to 793 MB and we can see the effect of using four bytes per voxel. Just like the smaller version, the amount of empty bricks that can be culled for this data set is relatively small for most transfer functions.

We also use a cropped version of the visible male data set ($2048 \times 1024 \times 611$) which is an MRI scan of the human cadaver. The cropped version includes the head and most of the torso. With one byte per voxel this data set is 1222 MB but there are quite a few bricks around the head that are likely to be culled.

Finally, we use a single time step from a data set which shows how two fluids mix during the Richtmyer-Meshkov instability. This instability occurs when a shock wave passes

through the two layers of fluid. The scalar value is a measure of entropy at each point in the simulation. We down sample the time step by a factor of eight ($1024 \times 1024 \times 960$) by simply averaging the values of the voxels. With one byte per voxel, this data set is 960 MB before culling. The Richtmyer-Meshkov data set allows for lots of culling with most transfer functions since the fluid interface is the interesting portion and it is quite contained spatially.

A.3 GPU Pipeline

We show an illustration of the pipeline for a modern programmable GPU in Figure A.1. The boxes represent processing stages, and those with a dashed outline represent processing stages that the application programmer has no direct control over. The ellipses represent data types on the GPU, each of which can be accessed and written to in a limited number of ways. The connecting arrows show the flow of data through the pipeline and what data types can be read from and written to by each processing stage. The dashed horizontal line separates the data and processing on the CPU from that which is resident on the GPU.

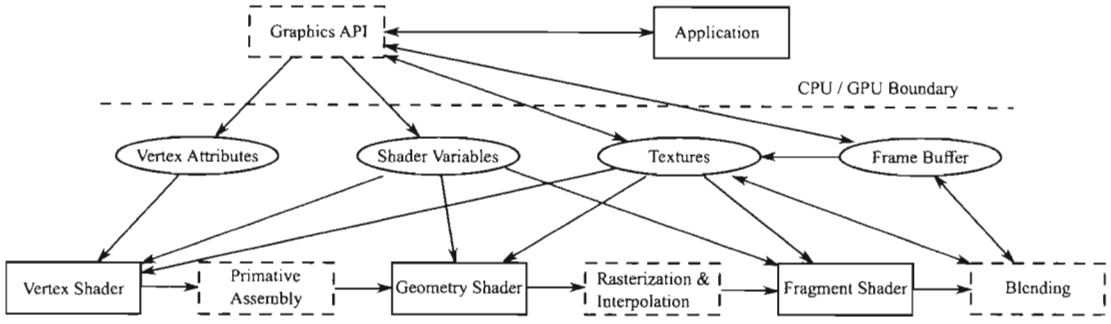

Figure A.1: A high-level overview of the pipeline on a modern programmable GPU.

Traditionally, all communication with the GPU is done through a graphics API (OpenGL or DirectX). Although the graphics APIs can be circumvented with new tools like NVIDIA's CUDA and ATI's CTM, this is mainly of interest to people doing general purpose computations on GPUs. Since we are doing graphics computations we utilize the traditional path through the OpenGL API, and we use the associated terminology in our discussion.

Since GPUs are designed for data parallel tasks, the computations are data driven. The vertex shader program is executed once for every incoming vertex, the geometry shader program is executed once for every incoming primitive, and the fragment shader program

is executed once for every incoming fragment. The output of each stage is fed as input to the next stage; transformed vertices are grouped to form primitives and primitives are rasterized to get fragments. In the blending stage, the output from the fragment shader can be combined with the value currently in the render target (either a texture or the frame buffer) using a number of fixed functions. Only the output from the fragment shader can be read back by the application or fed back into the GPU for another iteration. The geometry shader is a relatively new addition to the pipeline and is not available on our target hardware, thus we do not utilize it at all.

There are several different methods of communicating data between the CPU and the GPU. Computations are initiated by issuing rendering commands for groups of vertices. Each vertex must at least have position information, but other attributes like color and texture coordinates can also be supplied. Non-standard vertex attributes can be specified as shader variables. These attributes can be read and modified in the vertex shader before the vertices are grouped together based on the specified primitive type. The rasterization stage can then interpolate the vertex attributes across the primitives and feed the interpolated values into the fragment shader. Values that are constant across a primitive can also be supplied through shader variables by specifying them as 'uniform' rather than 'varying'.

Textures provide a more flexible means of communication as they can be read by the shader programs in an almost arbitrary fashion. Textures are essentially one, two, or three dimensional arrays with an associated interpolation method. Nearest neighbor and linear interpolation can be provided by the hardware at little to no cost, but higher order interpolations can be computed manually in the shader programs if desired. Initially shader programs only had read access to texture data, but it is now possible to redirect the output of the fragment shader into a texture instead of the frame buffer. This allows for much more efficient iterative computations since the output of one iteration could be read in directly during the next iteration without copying data from the frame buffer to a texture.

A.4 Computing Per-Pixel Cost on the GPU

In order to accurately represent the cost of rendering a pixel we must account for potentially hundreds of bricks that project to each pixel as well as the variation in the thickness of each brick over the image plane. Clearly using a standard fixed precision on eight bit buffer to store the pixel cost on the GPU is not going to be sufficient. Using the OpenGL frame

buffer object (FBO) extension, it is possible to render into a sixteen or thirty-two bit floating point buffer. We opt for a single component sixteen bit floating point buffer since this should provide sufficient resolution as well as both superior performance and better support on older hardware compared to a thirty-two bit buffer. All three methods of computing the pixel cost utilize additive blending to sum up the contributions of the individual bricks.

For the accurate method, we use a simple set of shaders which we provide the code for in Figure A.2. We perform the ray intersections in object space since the normals of the brick faces are aligned with the major axis. The vertex shader simply computes the view direction in object space and passes it along with the position in object space to the rasterization stage for interpolation. The fragment shader receives these interpolated values as well as two uniform variables which describe which faces are front facing and the scalar components of the plane equations for those faces. A single addition and division is then sufficient for finding the the distance along the view ray to the intersection with the plane. We have to compute at most three intersections and take the minimum distance among them. Finally, we normalize the result by multiplying it with $\frac{1}{\sqrt{3}}$ before outputting it to the render target.

The backface method does not require custom shader programs and instead uses the fixed function pipeline. The vertex weights are simply passed to the GPU as the color attributes, which are then linearly interpolated across the triangles of the faces. In the situation where only one vertex is inside the silhouette of the brick, one must be careful to triangulate the faces so that the interior vertex is part of both of the triangles that make up that face. If the vertex is only part of one of the two triangles, the other triangle will have no weight assigned to it. An alternative would be to compute a bilinear interpolation along each face in a set of custom shader programs, but this would defeat our goal of making this method as fast as possible.

The splatting method computes a spherical footprint and loads it into a texture. A different type of footprint may be appropriate in some scenarios. For example, with an orthographic projection the footprint could be computed using the accurate method and then replicated for all of the other bricks as a splat. We load the footprint into a 32^2 texture so that the resolution is sufficient while still being able to fit into the cache of the GPU. The fragment shader is essentially the same as the fixed function pipeline, except that it scales the texture values down based on the size of the brick.

```
uniform vec3 eyePosOS;
varying vec3 posOS;
varying vec3 viewDirOS;

void main(){
  posOS = gl_Vertex.xyz;
  viewDirOS = normalize(posOS - eyePosOS);
  gl_Position = ftransform();
}
```

```
uniform ivec3 isFront;
uniform vec3 dComps;
varying vec3 posOS;
varying vec3 viewDirOS;

void main(){
  float t = 2.0;

  if(isFront.x == 1)
    t = min((posOS.x + dComps.x)/viewDirOS.x, t);
  if(isFront.y == 1)
    t = min((posOS.y + dComps.y)/viewDirOS.y, t);
  if(isFront.z == 1)
    t = min((posOS.z + dComps.z)/viewDirOS.z, t);

  gl_FragColor = vec4(0.57735 * t, 0, 0, 0);
}
```

Figure A.2: The vertex (top) and fragment (bottom) shader programs for the accurate method of computing the per-pixel cost.

A.5 Asynchronous Direct Send Compositing

Compositing over gigabit ethernet can be quite expensive. It is important to try to minimize communication costs by overlapping communications and processing with each other. We give the psuedocode for our asynchronous compositing in Algorithm 1. We split the code up into four different functions for better readability. The first function shows the main structure of the algorithm and the other three are helper functions. The only input to the main function asyncDirectSend() is *depthOrder* which is a list of the ranks of the processing units in front to back order.

The first helper function is getPacket(). This function tries to get the packet of pixels at *packetIndex* from the unit *rank* and store them in the array *buf*. The function returns a boolean indicating if the get was successful. If the pixels are not local, then a non-blocking receive is used to try to get the packets from *rank*. If the pixels are local, then they are read back from the frame buffer and the function returns true.

The other two helper functions, updateSender() and updateReceiver(), are used to keep track of which packets of pixels we are currently sending and receiving as well as the rank of the units that have or need those packets. Both functions take the current rank and packet count for sending and receiving as the first two parameters and updates them. If the current rank is equal to negative one then the functions know that this is the initial iteration and so they reset their static counters (for how many processing units have finished communications) to zero. Both functions also take *depthOrder* as an parameter which they use to update the current rank we are communicating with. We start by sending to all of the units behind us and receiving from all of the units in front of us in the depth order. Once we reach the end of *depthOrder* going one direction we then go the other direction (sending to the units in front, receiving from behind). The function updateReceiver takes one additional parameter *backToFront* which is set to true when we are receiving from units behind us and false otherwise. Both functions return true when all of the communications are complete.

We also assume that each unit has access to its local frame buffer. Each unit can start a asynchronous frame buffer read by calling the non-blocking function startFrameBufRead(). If the read is not complete when endFrameBufRead() is called then it will block at that point. The function compositeImages() simply blends two packets of pixels together (the order is determined by the parameter *backToFront*). The processing units can communicate with

each other asynchronously with the nonBlockingSend() and nonBlockingRecv() functions. Depending on how these functions are implemented in the message passing library, it may be necessary to loop through all of the unfinished sends at the end and wait for them to finish. Finally, we assume that each unit has its own rank stored in $myRank$ and the total number of packets of pixels per unit stored in $packetsPerNode$.

The main function asyncDirectSend() starts by initializing some state variables for keeping track of which packets are being communicated and who they are being communicated with. Then it simply enters a loop where it initiates a frame buffer read of the packet it is going to send, tries to receive and composite a packet into its intermediate buffer, and finally finishes reading the packet from the frame buffer and sends it to the appropriate destination. We try to overlap each of these actions by performing them asynchronously.

Algorithm 1 Asynchronous Compositing Algorithm

```
1:  function asyncDirectSend(list depthOrder):
2:    {Initialize the sending and receiving information}
3:    bool backToFront
4:    int sendRank = -1, recvRank = -1, sentPackets = 0, recvPackets = 0
5:    bool sendDone = updateSender(sendRank, sentPackets, depthOrder)
6:    bool recvDone = updateReciever(recvRank, recvPackets, depthOrder, backToFront)
7:
8:    {Enter loop reading back, receiving, compositing, and sending packets of pixels}
9:    pixel * recvBuf, sendBuf, compositeBuf
10:   while !sendDone or !recvDone do
11:     if !sendDone then
12:       startFrameBufRead(sendRank, sentPackets, sendBuf)
13:     end if
14:     if !recvDone then
15:       bool success = getPacket(recvRank, recvPackets, recvBuf)
16:       if success then
17:         compositeImages(recvBuf, compositeBuf, backToFront)
18:         recvDone = updateReciever(recvRank, recvPackets, backToFront, depthOrder)
19:       end if
20:     end if
21:     if !sendDone then
22:       endFrameBufRead()
23:       nonBlockingSend(sendRank, sentPackets, sendBuf)
24:       sendDone = updateSender(sendRank, sentPackets, depthOrder)
25:     end if
26:   end while

27: function getPacket(int rank, int packetIndex, array buf):
28:   if rank == myRank then
29:     startFrameBufRead(myRank, packetIndex, buf)
30:     endFrameBufRead()
31:     return true
32:   else
33:     return nonBlockingRecv(rank, packetIndex, buf)
34:   end if
```

```
1: function updateSender (int sendRank, int sentPackets, list depthOrder):
2:   static int sentCounter
3:   if sendRank == -1 then
4:     sentCounter = 0
5:   else
6:     sentPackets += 1
7:   end if
8:   if sentPackets == packetsPerNode or sendRank == -1 then
9:     sentCounter += 1
10:    if sentCounter >= numNodes then
11:      return false
12:    end if
13:    int sendPos = depthOrder.find(myRank) + sentCounter
14:    if sendPos < numNodes then
15:      sendRank = depthOrder.at(sendPos)
16:    else
17:      sendRank = depthOrder.at(numNodes − sentCounter − 1)
18:    end if
19:  end if
20:  return true

21: function updateReciever (int recvRank, int recvPackets, bool backToFront, list depthOrder):
22:   static int recvCounter
23:   if recvRank == -1 then
24:     recvCounter = 0
25:   else
26:     recvPackets += 1
27:   end if
28:   if recvPackets == packetsPerNode or recvRank == -1 then
29:     recvCounter += 1
30:     if recvCounter > numNodes then
31:       return false
32:     end if
33:     int recvPos = depthOrder.find(myRank) − recvCounter
34:     if recvPos > 0 then
35:       recvRank = depthOrder.at(recvPos)
36:       backToFront = true
37:     else
38:       recvRank = depthOrder.at(recvCounter − 1)
39:       backToFront = false
40:     end if
41:   end if
42:   return true
```

Bibliography

[1] F. R. Abraham, W. Celes, R. Cerqueira, and J. L. Elias. A load-balancing strategy for sort-first distributed rendering. In *Proc. Computer Graphics and Image Processing (SIBGRAPI)*, pages 292–299, Washington, DC, USA, 2004. IEEE Computer Society.

[2] J. Allard and B. Raffin. A shader-based parallel rendering framework. In *Proc. IEEE Visualization (Vis)*, pages 127–134, Los Alamitos, CA, USA, 2005. IEEE Computer Society Press.

[3] S. Bergner, T. Möller, D. Weiskopf, and D. J. Muraki. A spectral analysis of function composition and its implications for sampling in direct volume visualization. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1353–1360, 2006.

[4] E. W. Bethel, G. Humphreys, B. Paul, and J. D. Brederson. Sort-first, distributed memory parallel visualization and rendering. In *Proc. IEEE Symp. Parallel Large-Data Visualization and Graphics (PVG)*, page 7, Washington, DC, USA, 2003. IEEE Computer Society.

[5] J. F. Blinn. Light reflection functions for simulation of clouds and dusty surfaces. In *Proc. ACM SIGGRAPH*, pages 21–29, New York, NY, USA, 1982. ACM Press.

[6] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Proc. IEEE Symp. Volume Visualization and Graphics (VolVis)*, pages 91–98, New York, NY, USA, 1994. ACM Press.

[7] L. Castanie, C. Mion, X. Cavin, and B. Levy. Distributed shared memory for roaming large volumes. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1299–1306, 2006.

[8] S. Chandrasekhar. *Radiative Transfer*. Courier Dover Publications, New York, NY, USA, 1960.

[9] T. J. Cullip and U. Neumann. Accelerating volume reconstruction with 3d texture hardware. Technical Report TR93-027, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, 1993.

[10] R. A. Drebin, L. Carpenter, and P. Hanrahan. Volume rendering. *ACM SIGGRAPH Computer Graphics*, 22(4):65–74, 1988.

[11] S. Eilemann and R. Pajarola. Direct Send Compositing for Parallel Sort-Last Rendering . In *Proc. EG Symp. Parallel Graphics and Visualization (PGV)*, pages 29–36, Aire-la-Ville, Switzerland, 2007. Eurographics Association.

[12] K. Engel, M. Kraus, and T. Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proc. ACM SIGGRAPH/EG Workshop Graphics Hardware (HWWS)*, pages 9–16, New York, NY, USA, 2001. ACM Press.

[13] J. Eyles, S. Molnar, J. Poulton, T. Greer, A. Lastra, N. England, and L. Westover. Pixelflow: the realization. In *Proc. ACM SIGGRAPH / EG Workshop on Graphics Hardware (HWWS)*, pages 57–68, New York, NY, USA, 1997. ACM.

[14] H.-C. Hege, T. Höllerer, and D. Stalling. Volume rendering: Mathematical models and algorithmic aspects. Technical Report TR-93-07, Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB), Berlin, Germany, 1993.

[15] W. M. Hsu. Segmented ray casting for data parallel volume rendering. In *Proc. Symp. Parallel Rendering (PRS)*, pages 7–14, New York, NY, USA, 1993. ACM Press.

[16] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan. Wiregl: a scalable graphics system for clusters. In *Proc. ACM SIGGRAPH*, pages 129–140, New York, NY, USA, 2001. ACM Press.

[17] 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 Transactions on Graphics*, 21(3):693–702, 2002.

[18] G. Kindlmann and J. Durkin. Semi-automatic generation of transfer functions for direct volume rendering. *Proc. IEEE Symp. on Volume Visualization (VVS)*, pages 79–86, 1998.

[19] T. Klein, M. Strengert, S. Stegmaier, and T. Ertl. Exploiting Frame-to-Frame Coherence for Accelerating High-Quality Volume Raycasting on Graphics Hardware. In *Proc. IEEE Visualization (Vis)*, pages 223–230, Los Alamitos, CA, USA, 2005. IEEE, IEEE Computer Society Press.

[20] J. Kniss, G. Kindlmann, and C. Hansen. Multi-dimensional transfer functions for interactive volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):270–285, July 2002.

[21] J. Kniss, S. Premoze, C. Hansen, and D. Ebert. Interactive translucent volume rendering and procedural modeling. In *Proc. IEEE Visualization (Vis)*, pages 109–116. Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[22] J. Kruger and R. Westermann. Acceleration techniques for gpu-based volume rendering. In *Proc. IEEE Visualization (Vis)*, page 38, Washington, DC, USA, 2003. IEEE Computer Society.

[23] T. M. Kurç, H. Kutluca, C. Aykanat, and B. Özgüç. A comparison of spatial subdivision algorithms for sort-first rendering. In *Proc. International Conference and Exhibition on High-Performance Computing and Networking (HPCN)*, pages 137–146, London. UK, 1997. Springer-Verlag.

[24] P. Lacroute and M. Levoy. Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transform. *ACM SIGGRAPH Computer Graphics*, 28(4):451–458, 1994.

[25] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, 1990.

[26] S. Lombeyda, L. Moll, M. Shand, D. Breen, and A. Heirich. Scalable interactive volume rendering using off-the-shelf components. In *Proc. IEEE Symp. Parallel Large-Data Visualization and Graphics (PVG)*, pages 115–121, Piscataway, NJ, USA, 2001. IEEE Press.

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

[28] M. M. Malik, T. Möller, and M. E. Gröller. Feature peeling. In *Proc. ACM Graphics Interface*, pages 273–280, New York, NY, USA, 2007. ACM.

[29] S. Marchesin, C. Mongenet, and J. Dischler. Dynamic load balancing for parallel volume rendering. In *Proc. EG Symp. Parallel Graphics and Visualization (PGV)*, pages 43–50, Aire-la-Ville, Switzerland, 2006. Eurographics Association.

[30] N. Max. Light diffusion through clouds and haze. *Computer Vision, Graphics, and Image Processing*, 33(3):280–292, 1986.

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

[32] B. Moloney, D. Weiskopf, T. Möller, and M. Strengert. Scalable sort-first parallel direct volume rendering with dynamic load balancing. In *Proc. EG Symp. Parallel Graphics and Visualization (PGV)*, pages 45–52, Aire-la-Ville, Switzerland, 2007. Eurographics Association.

[33] C. Montani, R. Perego, and R. Scopigno. Parallel volume visualization on a hypercube architecture. In *Proc. ACM Workshop on Volume Visualization (VVS)*, pages 9–16, New York, NY, USA, 1992. ACM.

[34] C. Mueller. The sort-first rendering architecture for high-performance graphics. In *Proc. Symp. on Interactive 3D Graphics (SI3D)*, pages 75–84, New York, NY, USA, 1995. ACM.

[35] K. Mueller, T. Möller, and R. Crawfis. Splatting without the blur. In *Proc. IEEE Visualization (Vis)*, pages 363–370, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[36] C. Müller, M. Strengert, and T. Ertl. Optimized volume raycasting for graphics-hardware-based cluster systems. In *Proc. EG Symp. Parallel Graphics and Visualization (PGV)*, pages 59–66, Aire-la-Ville, Switzerland, 2006. Eurographics Association.

[37] U. Neumann. Parallel volume-rendering algorithm performance on mesh-connected multicomputers. In *Proc. ACM Symp. on Parallel Rendering (PRS)*, pages 97-104, New York, NY, USA, 1993. ACM Press.

[38] U. Neumann. Communication costs for parallel volume-rendering algorithms. *IEEE Computer Graphics and Applications*, 14(4):49–58, Jul 1994.

[39] J. Nonaka, N. Kukimoto. N. Sakamoto, H. Hazama, Y. Watashiba, X. Liu, M. Ogata, M. Kanazawa, and K. Koyamada. Hybrid hardware-accelerated image composition for sort-last parallel rendering on graphics clusters with commodity image compositor. In *Proc. IEEE Symp. Volume Visualization and Graphics (VolVis)*, pages 17–24, Washington. DC, USA, 2004. IEEE Computer Society.

[40] H. Noordmans, H. van der Voort, and A. Smeulders. Spectral volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 6(3):196–207, Jul-Sep 2000.

[41] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler. The volumepro real-time ray-casting system. In *Proc. ACM SIGGRAPH*, pages 251–260, New York, NY. USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[42] H. Pfister and A. Kaufman. Cube-4—a scalable architecture for real-time volume rendering. In *Proc. IEEE Symp. on Volume Visualization (VVS)*, pages 47–54, Piscataway, NJ, USA, 1996. IEEE Press.

[43] T. Porter and T. Duff. Compositing digital images. In *Proc. ACM SIGGRAPH*, pages 253–259, New York, NY, USA, 1984. ACM.

[44] C. Rezk-Salama and A. Kolb. Opacity Peeling for Direct Volume Rendering. *Computer Graphics Forum*, 25(3):597-606, 2006.

[45] S. Roettger, S. Guthe, D. Weiskopf, T. Ertl, and W. Strasser. Smart hardware-accelerated volume rendering. In *Proc. EG Symp. on Data Visualisation (VisSym)*, pages 231 238, Aire-la-Ville, Switzerland, 2003. Eurographics Association.

[46] D. Ruijters and A. Vilanova. Optimizing GPU Volume Rendering. *Winter School of Computer Graphics*, 14, 2006.

[47] R. Samanta, T. Funkhouser, K. Li, and J. P. Singh. Hybrid sort-first and sort-last parallel rendering with a cluster of pcs. In *Proc. ACM SIGGRAPH / EG Workshop on Graphics Hardware (HWWS)*, pages 97–108, New York, NY, USA, 2000. ACM.

[48] P. Shirley and A. Tuchman. A polygonal approximation to direct scalar volume rendering. In *Proc. ACM SIGGRAPH*, volume 24, pages 63–70, New York, NY, USA, 1990. ACM Press.

[49] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. A simple and flexible volume rendering framework for graphics-hardware-based raycasting. In *Proc. EG Workshop on Volume Graphics*, pages 187–195, Aire-la-Ville, Switzerland, 2005. Eurographics Association.

[50] G. Stoll, M. Eldridge, D. Patterson, A. Webb, S. Berman, R. Levy, C. Caywood, M. Taveira, S. Hunt, and P. Hanrahan. Lightning-2: A high-performance display subsystem for PC clusters. In *Proc. ACM SIGGRAPH*, pages 141–148, New York, NY, USA, 2001. ACM Press.

[51] A. Stompel, K.-L. Ma, E. B. Lum, J. Ahrens, and J. Patchett. SLIC: scheduled linear image compositing for parallel volume rendering. In *Proc. IEEE Symp. Parallel Large-Data Visualization and Graphics (PVG)*, pages 33–40, Washington, DC, USA, 2003. IEEE Computer Society.

[52] M. Strengert, T. Klein, R. Botchen, S. Stegmaier, M. Chen, and T. Ertl. Spectral volume rendering using GPU-based raycasting. *The Visual Computer*, 22(8):550–561, 2006.

[53] J. Sweeney and K. Mueller. Shear-warp deluxe: the shear-warp algorithm revisited. In *Proc. EG Symp. on Data Visualisation (VisSym)*, pages 95–104, Aire-la-Ville, Switzerland, 2002. Eurographics Association.

[54] X. Tong, W. Wang, W. Tsang, and Z. Tang. Efficiently rendering large volume data using texture mapping hardware. In *Proc. EG / IEEE TCVG Symp. on Visualization (VisSym)*, Berlin / Heidelberg, Germany, 1999. Springer.

[55] C. Wang, A. Garcia, and H.-W. Shen. Interactive level-of-detail selection using image-based quality metric for large volume visualization. *IEEE Transactions on Visualization and Computer Graphics*, 13(1):122–134, 2007.

[56] M. Weiler, R. Westermann, C. Hansen, K. Zimmermann, and T. Ertl. Level-of-detail volume rendering via 3d textures. In *Proc. IEEE Symp. on Volume Visualization (VVS)*, pages 7–13, New York, NY, USA, 2000. ACM Press.

[57] D. Weiskopf, M. Weiler, and T. Ertl. Maintaining constant frame rates in 3d texture-based volume rendering. In *Proc. Computer Graphics International (CGI)*, pages 604–607, Washington, DC, USA, 2004. IEEE Computer Society.

[58] L. Westover. Interactive volume rendering. In *Proc. Chapel Hill Workshop on Volume Visualization*, pages 9–16, New York, NY, USA, 1989. ACM Press.

[59] L. Westover. Footprint evaluation for volume rendering. In *Proc. ACM SIGGRAPH*, pages 367–376, New York, NY, USA, 1990. ACM Press.

[60] P. L. Williams and N. Max. A volume density optical model. In *Proc. ACM Workshop on Volume Visualization*, pages 61–68, New York, NY, USA, 1992. ACM Press.

[61] C. Zhang and R. Crawfis. Shadows and soft shadows with participating media using splatting. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):139–149, 2003.