Volume Ray Casting Techniques and Applications using General Purpose Computations on Graphics Processing Units

by

Michael Romero

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Engineering

Supervised by

Professor Dr. Muhammad Shaaban
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
June 2009

Approved by:

Dr. Muhammad Shaaban
Thesis Advisor, Department of Computer Engineering

Dr. Roy Melton
Committee Member, Department of Computer Engineering

Dr. Joe Geigel
Committee Member, Department of Computer Science

Dr. Reynold Bailey
Committee Member, Department of Computer Science

Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title:

Volume Ray Casting Techniques and Applications using General Purpose Computations on Graphics Processing Units

I, Michael Romero, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

Michael Romero

Date

Dedication

For my family, for their love and support,
and for sharing with me the best times of my life.

Acknowledgments

I would like to thank my primary advisor Dr. Muhammad Shaaban for sparking my interest in the field of computer architecture. His instruction has led to my passion in the field, giving me satisfaction and fulfillment in my work. Second I would like to thank Dr. Roy Melton for his invaluable assistance and attention to detail with this thesis. His availability and guidance have proven very helpful in the course of writing and development. Next I would like to thank Dr. Joe Geigel for providing many of the necessary and thought-provoking resources used in the development of this thesis. His selection of literature has helped shape many of the concepts explored in this work. Finally I would like to thank Dr. Reynold Bailey for both his instruction in the field of computer graphics, and his remarkable dedication towards his students. His commitment to teaching and concern for students has been inspirational.

Abstract

Volume Ray Casting Techniques and Applications using General Purpose Computations on Graphics Processing Units

Michael Romero

Traditional 3D computer graphics focus on rendering the exterior of objects. Volume rendering is a technique used to visualize information corresponding to the interior of an object, commonly used in medical imaging and other fields. Visualization of such data may be accomplished by ray casting; an embarrassingly parallel algorithm also commonly used in ray tracing. There has been growing interest in performing general purpose computations on graphics processing units (GPGPU), which are capable exploiting parallel applications and yielding far greater performance than sequential implementations on CPUs. Modern GPUs allow for rapid acceleration of volume rendering applications, offering affordable high performance visualization systems.

This thesis explores volume ray casting performance and visual quality enhancements using the NVIDIA CUDA platform, and demonstrates how high quality volume renderings can be produced with interactive and real time frame rates on modern commodity graphics hardware. A number of techniques are employed in this effort, including early ray termination, super

sampling and texture filtering. In a performance comparison of a sequential versus CUDA implementation on high-end hardware, the latter is capable of rendering 60 frames per second with an impressive price-performance ratio heavily favoring GPUs.

A number of unique volume rendering applications are explored including multiple volume rendering capable of arbitrary placement and rigid volume registration, hypertexturing and stereoscopic anaglyphs, each greatly enhanced by the real time interaction of volume data. The techniques and applications discussed in this thesis may prove to be invaluable tools in fields such as medical and molecular imaging, flow and scientific visualization, engineering drawing and many others.

Contents

List of Tables

List of Figures

Glossary

block

A grouping of threads dynamically allocated to the next available multiprocessor. 15

GPGPU

General-Purpose computing on Graphics Processing Units. 1

grain size

The amount of work dynamically partitioned among multiprocessors, synonymous with threads per block. 47

grid

A collection of thread blocks executing the same kernel. 15

kernel

The code that each thread executes on the graphics device, the entry point for computation on the device. 14

multiprocessor

A cluster of computational resources on the graphics device. There may be one or several depending on the device. 19

multiprocessor occupancy

The ratio of the number of active warps on a multiprocessor to the maximum number of active warps. Determined by devices compute model, threads per block, registers per thread and shared memory usage. 50

ray casting

An embarrassingly algorithm for direct volume rendering. 23

thread

The most fundamental level of the CUDA computational hierarchy, where several perform computations in parallel. 15

transfer function

A function used to corollate a density with a color and opacity. 34

voxel

Discrete volume element, like a three dimensional pixel. 3

warp

A group of 32 threads from the same thread block starting at the same program address on the multiprocessor. 19

Chapter 1

Introduction

Technology trends and advances in graphics techniques (such as those found in modern video games and simulations) have led to a need for extremely powerful dedicated computational hardware to perform the necessary calculations. Graphics hardware companies such as AMD/ATI and NVIDIA have developed graphics processors capable of massively parallel processing, with large throughput and memory bandwidth typically necessary for displaying high resolution graphics. However, these hardware devices have the potential to be repurposed and used for other non-graphics-related work, or programmed outside the bounds of traditional graphics APIs. This is commonly refered to as "GPGPU," or "General-purpose computing on graphics processing units." A number of frameworks allows for graphics devices to be repurposed and programmed in a general fashion, such as Stanford's Brook programming language, AMD's CTM (Close To Metal) technology, and NVIDIA's programming interface known as CUDA (Compute Unified Device Architecture). Utilizing graphics devices to execute massively parallel algorithms will yield a significantly large speedup over sequential implementations on conventional CPUs, essentially transforming them into high

performance computing nodes. Figure 1.1 illustrates the potential computational throughput of NVIDIA's discrete graphics devices in contrast with CPU offerings from Intel.

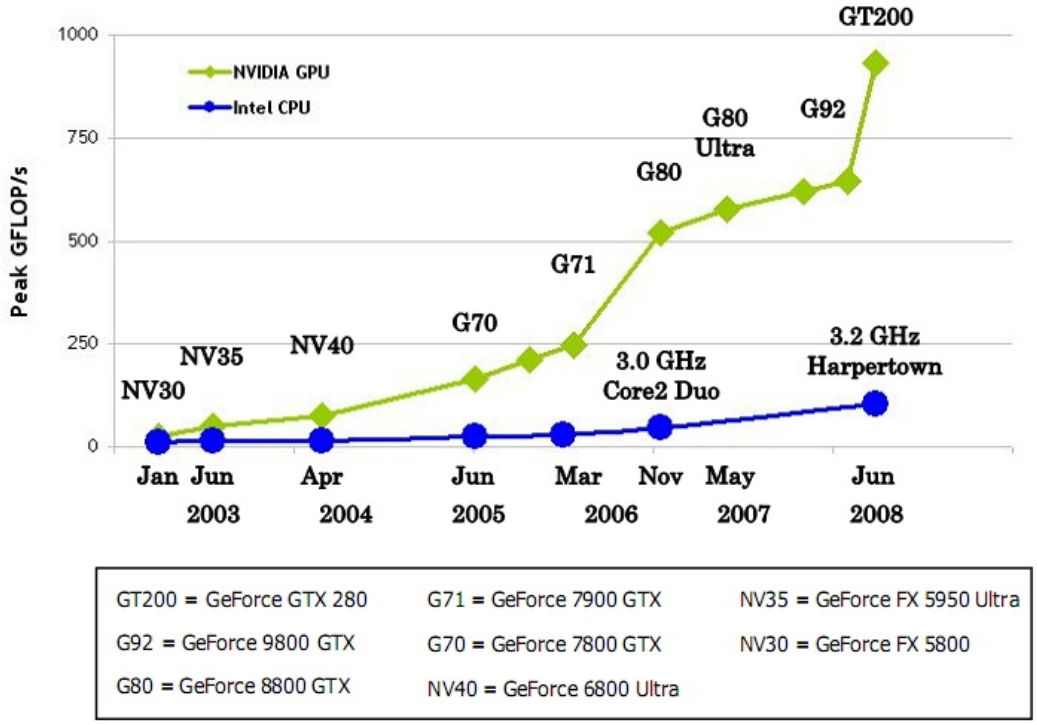


Figure 1.1: GFLOPS Performance Comparison of Modern GPUs vs. CPUs. Courtesy of NVIDIA [10].

With the expanded capabilities of graphics processors that programming frameworks like CUDA afford, it is possible to achieve performance that has previously been out of reach. Real-time or interactive manipulation of data is a common use of these technologies, which is often beneficial to the exploration and understanding of these particular data sets. Many excellent candidates exist for GPGPU implementations, but the work to be done

by these applications must be parallelizable; ideally, embarrassingly parallel. Target applications include audio and video processing, physics simulations, flow dynamics, digital signal processing, computer vision, weather modeling, neural networks, medical imaging and many more. In the case of medical imaging, it is desirable to reconstruct models obtained from CT (computed tomography) scans, and view the results. The data obtained from a CT scan is "voxel" data, and requires a certain rendering technique in order to view and interpret the results.

Volume rendering describes a specific paradigm of visualizing three-dimensional data. A 3D model is typically constructed from a polygon mesh which defines only the surface of the model. With volume rendering however, the data describing the model defines the interior volume of the model as well, so the internal details of a 3D object may be assigned "optical properties" such as color and opacity. Volumetric data is comprised of "voxels" or volume elements, and can be viewed a number of ways. One option is to approximate and render a polygon mesh from the voxel data. One of the most common approximation methods is the "marching cubes" algorithm. This algorithm and other approximation algorithms allow the visualization of the approximate surface structure of a volume, however they do not allow visualization of the volume's interior, and are not suitable for the purposes of this thesis. Direct volume rendering allows the visualization of the interior of objects, where the exact voxel data is rendered by assigning

optical properties to each voxel element in the scene. Direct volume rendering can be achieved by a number of techniques including volume ray casting [15] or plane composing algorithms such as texture-based rendering [4] or shear-warping [7]. Additionally, various optimization and quality enhancement techniques exist, such as early ray termination, volume segmentation, empty space skipping, anti-aliasing, interpolation of voxel data, illumination models and more. Figure 1.2 shows an approximated volume rendering of a buckyball represented as a polygon mesh by using the marching cubes algorithm. Figure 1.3 shows a direct volume rendering of a buckyball achieved using the ray casting algorithm. Comparing these two images, it is clear that more information can be gleaned by assigning color and opacity to the density of the volume and visualizing the volumes interior than strictly visualizing the approximated surface of the volume.

This thesis focuses on implementing direct volume rendering techniques using commodity graphics cards, specifically the volume ray casting approach, though other approaches may benefit from the results of this work. NVIDIA's CUDA was chosen as the target GPGPU platform, and this document will discuss the approach to implementing a volume ray casting system using this platform, the advantages and speedups obtained, as well as difficulties encountered. This work's unique contributions lie in the implementation of some of the visual and performance enhancements on the CUDA platform, development of an extensible and easily manageable framework for ray casting, and particularly in the advancement of volume rendering

applications made possible by these enhancements such as the interactive rendering of multiple volumes and stereoscopic anaglyphs. The volume ray casting software, all of the effects achieved, and applications demonstrated are capable of running with interactive performance on commodity off the shelf devices, including laptops, and across Microsoft Windows and Mac OS X operating systems.

This thesis has been organized to best place the volume rendering system developed in context. Chapter 2 discusses the motivation behind volume rendering, and possible applications of this rendering technique. Chapter 3 provides the necessary background information to understand how CUDA may be used to exploit volume rendering algorithms. Chapter 4 examines common volume rendering concepts, the existing efforts in the field, and the potential that exists to improve on these efforts. Chapter 5 describes the implementation details, features and novel contributions of the volume rendering system developed. Chapter 6 shows the performance characteristics of the volume renderer by listing the framerate of the system while varying a number of factors. Chapter 7 analyzes the quality, performance and effectiveness of each applicable volume rendering technique or application. Chapter 8 contains ideas and considerations for the future of the system developed in this thesis and volume rendering in general. Finally, chapter 9 offers closing remarks on the effectiveness of the system developed.

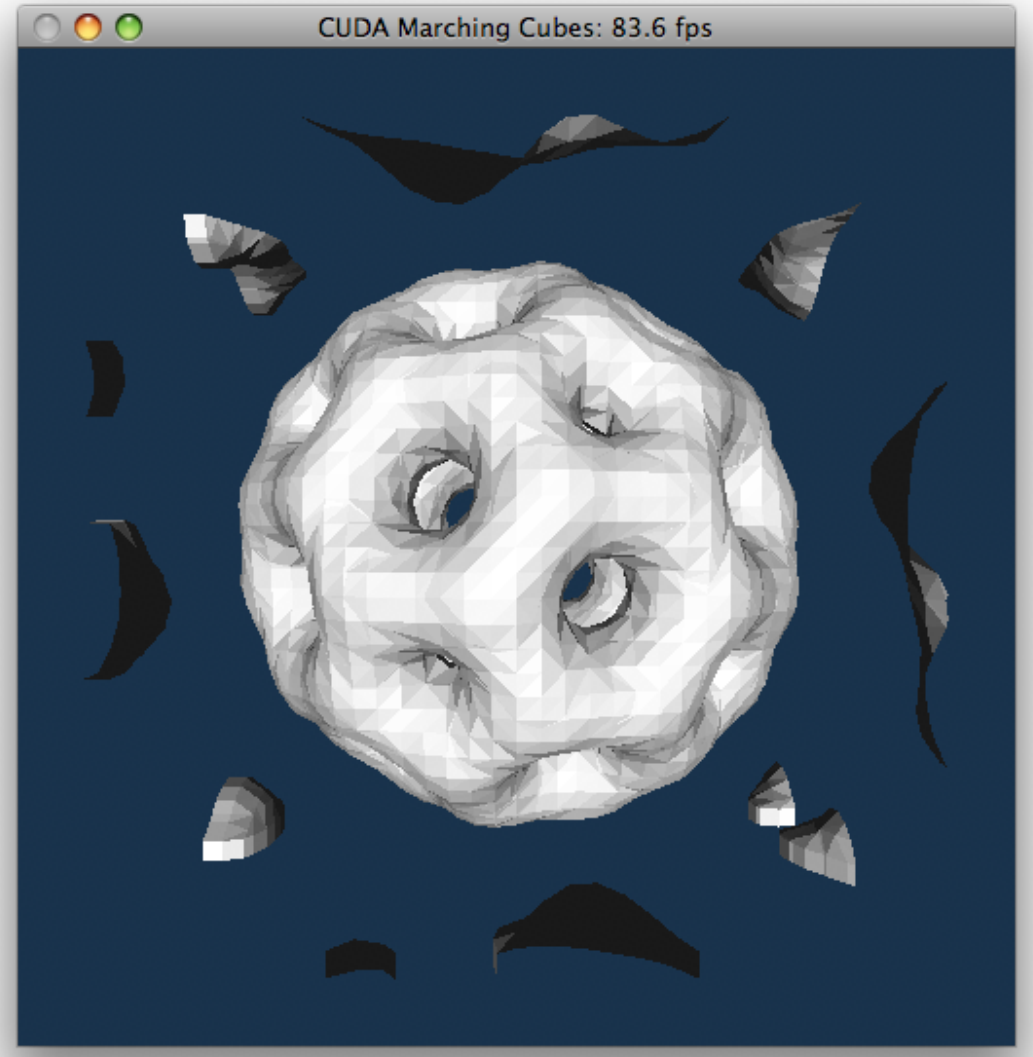


Figure 1.2: A buckyball volume rendered as a polygon mesh using the marching cubes algorithm. This rendering was produced using the marching cubes example application in the NVIDIA CUDA SDK.

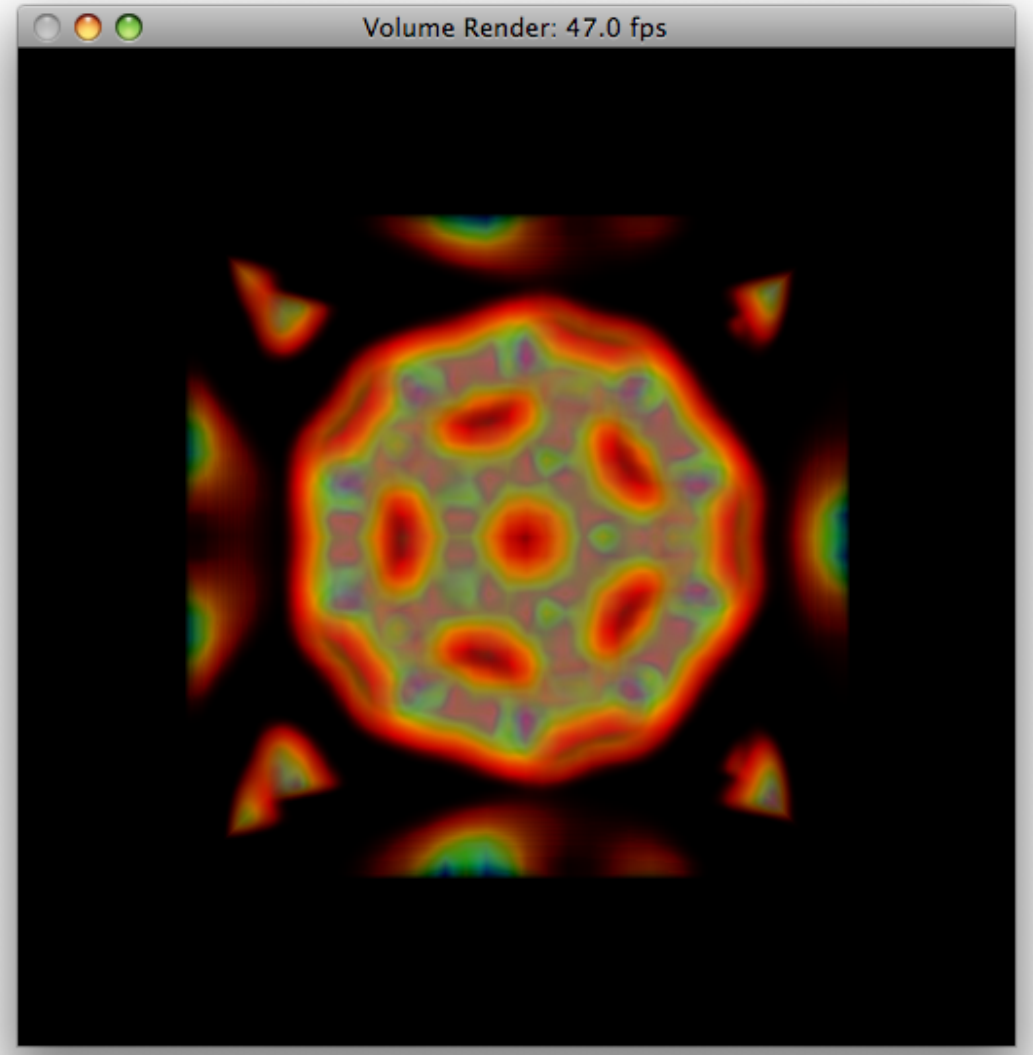


Figure 1.3: A buckyball volume rendered using volume ray casting, a direct volume rendering algorithm. This rendering was produced using the volume rendering example application in the NVIDIA CUDA SDK.

Chapter 2

Motivation

To place the volume rendering efforts of this thesis in the proper context, it is necessary to examine the technologies available capable of performing volume rendering, and the potential applications for volume rendering. When the potential applications are considered, ideal technologies for these applications may be identified.

2.1 Technology

In most common modern day implementations of visualizing three-dimensional models on a computer, a polygon consisting of a mesh of triangles is used to represent the figure to be displayed. This polygon mesh describes the surface of the model, and various techniques and special effects are applied to add detail and light to a scene described by polygons. Volumetric rendering differs from traditional visualization technique by using a 3D model represented by "voxel" or "volumetric elements" rather than a polygon mesh. Voxel representations describe a 3D model as a set of values on a regular grid existing in 3D space. In other words, a voxel is a discrete point in 3D

space, and these voxels describe the interior as well as exterior surface of the model. This differs from polygon meshes, where the mesh describes a series of points on the surface of the model but not the interior of the model. This fundamental difference is one of the primary advantages of volumetric rendering over traditional rendering schemes: voxel-based models contain information about their volume.

Current generation specialized graphics hardware is targeted specifically for rendering polygon meshes, and the performance is more than adequate to render scenes interactively in real-time. Volume rendering systems exist in a variety of fields. Particular interest in volume rendering comes from the medical imaging field, because body scans obtained from patients are volumetric in nature, and it is very desirable to investigate these body scans interactively. Volume rendering systems have also been used in a number of video games for their ability to render complex terrain. These systems have historically done all the necessary rendering work on the CPU. However, the common consumer-level graphics hardware offerings from NVIDIA and AMD/ATI do not targeting volume rendering. Advances in graphics hardware have ignored volumetric rendering, and the necessary performance to render large voxel-based data sets with a high level of detail interactively has been very difficult or impossible to achieve. Fortunately dedicated consumer-level graphics hardware has been identified for its massively parallel computational abilities. With the introduction of programmable shaders in modern graphics hardware, pixel shaders have been

utilized for volume rendering acceleration [6]. Furthermore, graphics hardware companies are now releasing APIs and frameworks which expose the architecture of these devices for general purpose programming. Currently, the most popular and prevalent API is NVIDIA's CUDA platform, which stands for "Compute Unified Device Architecture." This platform allows for the programming of any CUDA capable device to perform general purpose computations in parallel.

This thesis implements a system capable of rendering and interacting with volumetric data sets by programming graphics hardware to perform the general purpose computations necessary for volume rendering. This is known as General Purpose computation on Graphics Processing Units (GPGPU). The target platform is NVIDIA's CUDA, but the general C/C++ ray casting parallel algorithm should also be adaptable to other dedicated graphics hardware, such as the offerings from AMD/ATI, Intel, or SGI. The volume renderer demonstrates a large improvement in performance and at a high level of detail over rendering using a CPU. The system is capable of interactively rendering relatively large data sets, larger and more complex than those which can be rendered interactively on the CPU. Furthermore, this thesis also demonstrate a number of applications of volume-based rendering; specifically the rendering of hypertextures, multiple volumes, and the visualization of stereoscopic anaglyphs.

2.2 Applications

Several applications exist which would benefit from an interactive volume rendering system, such as medical imaging applications. Common body scans such as CT and MRI scans generate images of several "slices" of a patient's body. These slices may be reassembled into voxel data. Using the system developed in this thesis, this data set may be displayed and explored interactively by doctors or surgeons. Furthermore, the volume rendering system developed here allows for multiple volumes to be positioned in the same space, performing a type of rigid registration. This would allow doctors to gain new insight on patients by exploring multiple medical scans simultaneously.

While the possibility of rendering static volumetric data interactively has many applications, the ability to deform these data sets opens the way to many more possibilities. Current 3D modeling systems operate by placing vertices and exporting meshes based on those vertices. With volumetric data, a 3D model could instead be created by "sculpting" the model. Similar to the way popular image editing tools provide brushes to paint with, and blending tools to smooth edges, a voxel painting system could provide cube or sphere brushes as an extension to constructive solid geometry, and smoothing functions to round or blend together 3D models. In the case of medical images, surgeons could use the volumetric representation of a human body to simulate surgery. In real-time, a surgeon could practice the

cutting, bending, pinching and suturing of a completely virtual body; something that has not yet been efficiently demonstrated.

Voxels have been used in computer games and simulations to define complex terrain geometry in lieu of another popular technique known as "height maps." Height maps are gray scale images where the light and dark spots are converted into peaks and valleys in terrain. The shortcoming is that height maps do not support concave structures. Voxels offer a solution to this problem, providing the additional information.

In computer simulations and gaming, volumetric data could provide much higher levels of detail and realism for certain objects. Much like the advancements ray tracing brings to lighting, volume rendering could provide similar advancements to geometry. Deformable volumes would allow a player to alter the environment without pre-computed demolition models. For example, using a physics system, it would be possible to blow up walls, dent trash cans, or shatter glass, and perform these actions on anything in a scene without the need to pre-program a model's reaction to certain behavior.

It would also be possible to interact with volumes in ways typically reserved for other media. For example, stereoscopic visualization is becoming more prevalent in video games and motion pictures. Volumes can also be visualized using stereoscopy. This thesis provides a system for rendering stereoscopic anaglyphs of volumes.

Chapter 3

CUDA Background

Background information on NVIDIA's CUDA platform is described here. A fundamental understanding of CUDA is necessary to understand how volume rendering algorithms may be exploited by its architecture.

3.1 CUDA Architecture

The CUDA programming guides [9], [10] provided by NVIDIA Corporation document the architecture of the CUDA framework and runtime, the CUDA programming paradigm, and list several performance considerations when programming for CUDA. The following summarizes the CUDA architecture in terms of thread organization, memory hierarchy, processing elements and compute models.

3.1.1 Thread Organization

In the CUDA processing paradigm (as well as other paradigms similar to stream processing) there is a notion of a "kernel." A kernel is essentially a mini-program or subroutine. Kernels are the parallel programs to be run on the device (the NVIDIA graphics card inside the host system). A number

of primitive "threads" simultaneously execute a kernel program. Batches of these primitive threads are organized into "thread blocks." A thread block contains a specific number of primitive threads, chosen based on the amount of available shared memory, as well as the memory access latency hiding characteristics desired. The number of threads in a thread block is also limited by the architecture to a total of 512 threads per block. Each thread within a thread block can communicate efficiently using the shared memory allocated to each thread block. Using this shared memory, all threads can also sync within a thread block. Every thread within a thread block has its own thread ID. Thread blocks are conceptually organized into 1D, 2D or 3D arrays of threads for convenience.

A "grid" is a collection of thread blocks of the same thread dimensionality which all execute the same kernel. Grids are useful for computing a large number of threads in parallel since thread blocks are physically limited to only 512 threads per block. However, thread blocks within a grid may not communicate via shared memory, and consequently may not synchronize with one another.

Figure 3.1 demonstrates the thread hierarchy described. Here, kernel 1 contains a 3x2 grid of thread blocks. Each thread block is a 5x3 block of threads, for a total of 90 threads in kernel 1. Kernel 2 may contain a different organization of thread blocks, which in turn may contain an array of threads different from the arrays in the thread blocks of kernel 1.

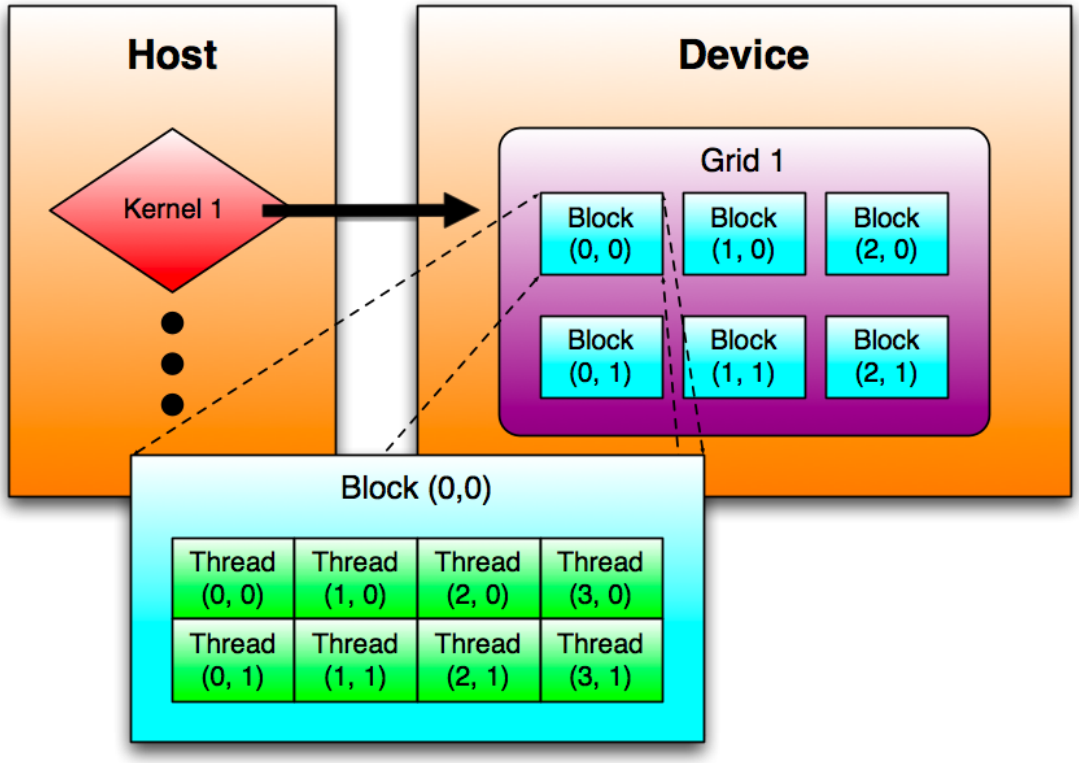


Figure 3.1: CUDA Computational Hierarchy.

3.1.2 Memory Hierarchy

There are several levels of memory on the GPU device, each with distinct read and write characteristics. Every primitive thread has access to private "local memory" as well as registers. This "local memory" is really a misnomer; the memory is private to the thread, but is not stored local to the thread's registers; instead it is located off-chip in the global GDDR memory available on the graphics card. Every thread in a thread block also has access to a unified "shared memory," shared among all threads for the life of that thread block. Finally, all threads have read/write access to "global memory,"

which is located off-chip on the main GDDR memory module, which therefore has the largest capacity but is the most costly to interact with. There also exists read-only "constant memory" and "texture memory", in the same location as the global memory.

The global, constant and texture memory are optimized for different memory usage models. Global memory is not cached, though memory transactions may be coalesced to hide the high memory access latency. These coalescence rules and behaviors are dependent on the particular device used. The read-only constant memory resides in the same location as global memory, but this memory may be cached. On a cache hit, regardless of the number of threads reading, the access time is that of a register access for each address being read. The read-only texture memory also resides in the same location as global memory and is also cached. Texture memory differs from constant memory in that its caching policy specifically exploits 2D spatial locality. This is due to the use of "textures" in 3D graphics: the use of 2D images to texture the surface of 3D polygons. Textures are frequently read and benefit from caching the texture spatially.

Figure 3.2 shows the scope of each of the memory segments in the CUDA memory hierarchy. Registers and local memory are unique to a thread, shared memory is unique to a block, and global, constant, and texture memories exist across all blocks. It's important to note that local memory has the same performance characteristics as global memory.

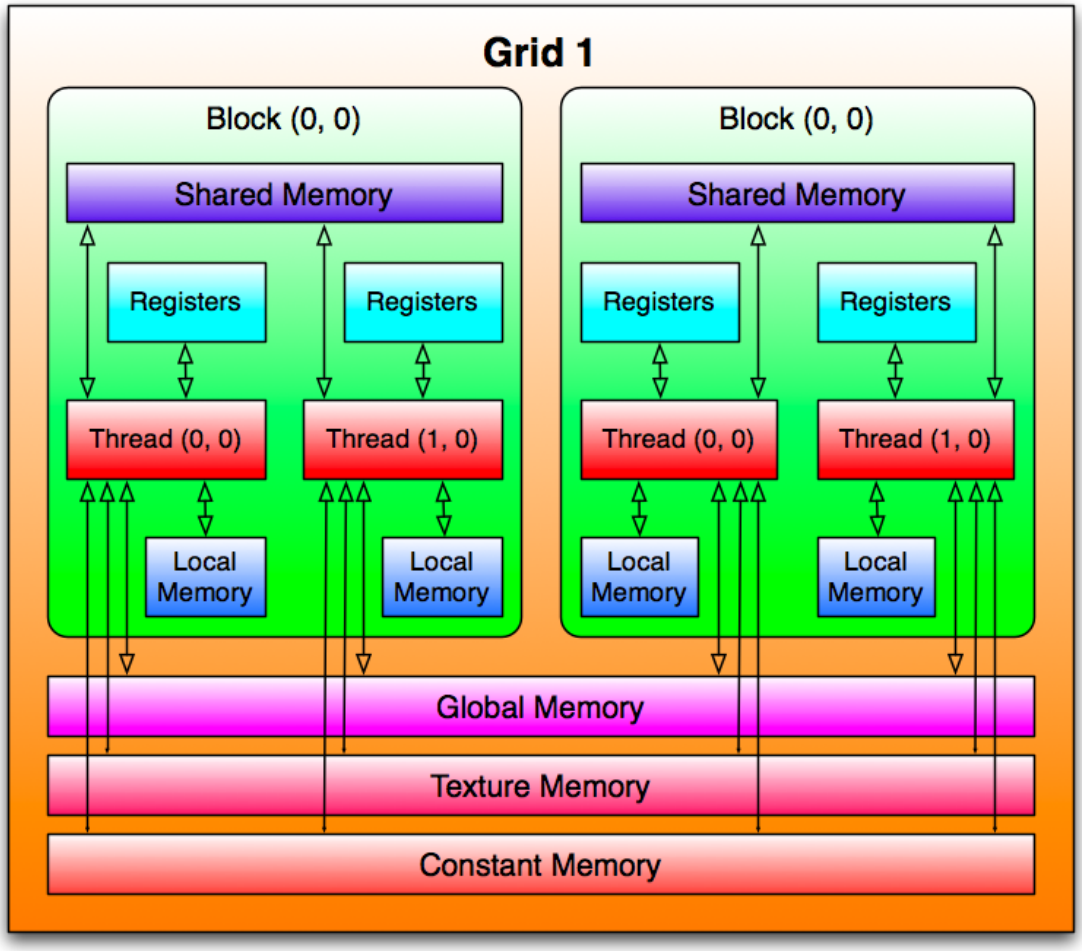


Figure 3.2: CUDA Memory Hierarchy.

3.1.3 Multiprocessors

CUDA capable GPUs are constructed with the Tesla architecture. CUDA applications may be run on any card which supports this architecture, but each GPU device may have different specifications and therefore a slightly different set of supported features and a different number of available computational resources. When a kernel is invoked, each thread block executes

on a "multiprocessor." This multiprocessor contains the resources to support a certain number of threads.

- 8 Scalar Processor cores
- 2 special function units for transcendentals
- 1 multithreaded instruction unit
- On-chip shared memory

One or more thread blocks are assigned to a multiprocessor during the execution of a kernel. The CUDA runtime handles the dynamic scheduling of thread blocks on a group of multiprocessors. The scheduler only assigns a thread block to a multiprocessor when enough resources are available to support the thread block. Each block is split into SIMD (Single-Instruction Multiple-Data) groups of threads called "warps." The SIMD unit creates, manages, schedules and executes 32 threads simultaneously to create a warp. Every warp is synchronous, and therefore care must be taken to ensure that certain threads within a warp do not take substantially longer than other threads in that same warp, because the warp only executes as fast as the slowest thread. There are a number of programming hints provided in the CUDA programming guide to help prevent such warp divergence.

3.1.4 Compute Model

Every CUDA-enabled device has a compute capability number. This number indicates a standard number of registers, memory size, etc. for all devices of that capability number. Compute capability numbers are backwards

compatible. The CUDA Programming Guide [10] available from NVIDIA details the compute capabilities of various graphics devices, an excerpt from which can be found in figure 3.1.

	Num Multiprocessors	**Compute Capability**
Tesla C870	16	1.0
GeForce 9600m GT	2	1.1
GeForce 9800GT	14	1.1
GeForce GTX260	24	1.3

Table 3.1: Selection of graphics devices and their respective compute models [10].

The primary devices used for the development of this thesis are the GeForce 9600m GT, a discrete graphics card found in mid-high range laptops, and the GTX260, a graphics card found in mid-high range gaming desktops. At the time of writing, the GTX200 series conforms to the most recent compute model. This latest compute model, 1.3, has a number of significant improvements over previous compute models, including the following.

- Double precision support
- Higher memory bandwidth
- Double the number of available registers

The double precision floating point support is a feature not often used in computer graphics, but is frequently used in various scientific and engineering calculations, which may performed using CUDA. The higher memory bandwidth decreases transfer times between the host and the device. Doubling the number registers from 8192 to 16384 per multiprocessor allows for

greater multiprocessor occupancy, resulting in higher computational performance.

Chapter 4

Volume Rendering Concepts and Existing Work

This chapter outlines several fundamental volume rendering concepts and references the various academic publications in which they are discussed. A brief analysis of the rendering techniques is made, and their suitability for implementation on the CUDA platform is also considered. Finally this chapter takes survey of existing CUDA volume rendering implementations.

4.1 Introduction

There are two primary categories of algorithms associated with direct volume rendering. The first, ray casting algorithms, operate by casting a ray down the line of sight for every pixel in the image of the scene. Each ray assigns a color value to a pixel by compositing the color and transparency of the voxels in the volumetric model intersected by each respective ray. Ray casting algorithms may use early ray termination [14], as well as octree subdivision [8] and empty space skipping [6] to reduce computational complexity by occluding certain volumetric data. Generally speaking, the

complexity is relative to the image size times the depth of the volume [2].

Plane composing algorithms are the other main category of direct volume rendering algorithms. These algorithms operate by compositing several slices which comprise a volumetric model. These algorithms tend to work best when the view-plane is fixed parallel to the model, so that each of the voxels being composed may be accessed on separate planes. Texture-based rendering is one key technique which composites slices of volumetric data. Shear-Warp factorization can be used to enhance performance of rendering volume slices by applying a shearing transformation to exploit spatial coherency of the model, reducing computational complexity of rendering a volume from an arbitrary viewing angle [7]. The computational complexity is relative to the volume size [2].

The following is a brief overview of the various rendering techniques, and comparisons of their advantages and shortcomings, such as computational complexity, required memory capacity, and visual quality. These techniques will be used to identify the most suitable category of techniques to target for enhancements, as well as to provide a relative comparison of computational efficiency and quality.

4.2 Ray Casting

4.2.1 Description

One primary technique for rendering volumes is known as “ray casting.” Ray casting is an object order rendering technique, where the computations are performed based on the voxels in the volumetric model intersected by each ray. Ray casting has been overlooked as a primary volume rendering algorithm in favor of slicing methods, in part because the advances in graphic acceleration hardware brought optimized texturing units capable of greatly enhancing the performance of 3D texture-based rendering methods. However, more recent trends in hardware graphics acceleration have given way to CUDA (as well as other GPGPU platforms), putting the performance of ray casting methods on par with other slicing methods and reaffirming ray casting as a worthwhile volume rendering effort. Figure 1.3 shows an example rendering of a ray casted volume.

4.2.2 Ray Casting Process

The general ray casting procedure is similar for most volume ray casting systems. For each pixel in a resulting image, a single ray is cast through the scene, intersecting with the volumes to be rendered. These rays may be cast independently, resulting in full pixel-parallelism. Each voxel intersected with the ray has a certain color and opacity. Shading can be applied based on the angle of the volume to a light source. The ray accumulates a value

that is a function of color and opacity for each voxel the ray intersects, and renders a pixel. This approach fits well into a parallel stream processing paradigm, exhibited by fragment processors, as well as the CUDA architecture. Figure 4.1 shows a graphical representation of the basic ray casting procedure. Figure 4.2 details the pseudo code for a fragment-program-based implementation of a volume ray-caster, described by S. Stegmaier *et al.*for a volume rendering framework [15].

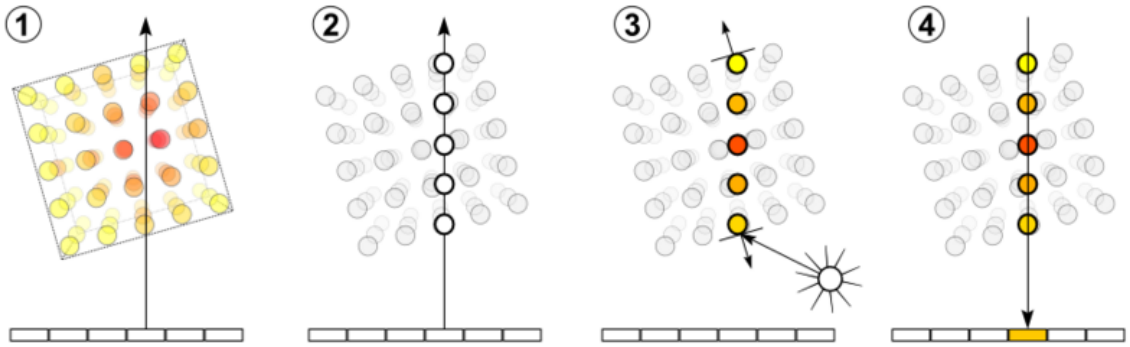


Figure 4.1: The four basic steps of volume ray casting:
(1) Ray Casting: cast a ray from the viewplane.
(2) Sampling: sample the intersected volume along the path of the ray.
(3) Shading: compute a color value for each voxel based on a shading model.
(4) Compositing: accumulate color and assign to a pixel on the viewplane
Courtesy of Wikipedia [17].

```
Compute volume entry position
Compute ray of sight direction
While in volume
     Lookup data value at ray position
     Accumulate color and opacity
     Advance along ray
```

Figure 4.2: Pseudo code of a fragment-program-based volume ray-caster [15].

A great deal of emphasis has been placed on fitting the ray casting process into a stream processing system. Stream processing is essentially a system where chain of microprocessing units perform simple "kernels" (small programs or subroutines), with one microprocessor's output feeding directly to the input of the next. The advantage to stream processing coincide with the advantages of pipelining, by splitting the work into stages and feeding the pipeline in a specific interval, so that the pipeline produces a final result every interval. Stream processing takes this concept further by having a data "stream" run through several kernels, with dedicated hardware (and often scratchpad memory) for each kernel performed, without the overhead of context switching one subroutine to another. This offers very high throughput of a complex series of operations. Modern graphics hardware performs this kind of stream processing in SIMD fashion, where several data-parallel homogeneous streams may be computed with each instruction.

In its relation to ray casting, initial efforts spawned from Purcell's publication of a ray tracing technique that took advantage of the stream processing paradigm of modern graphics hardware [12]. Following this publication, one of the most ground-breaking volume ray casting papers was published by Kruger and Westermann, where a ray casting approach that uses multiple passes over the scene (for early ray termination) to render an image using shader model 2.0 was described [6]. A volume rendering framework was finally described by S. Stegmaier *et al.*, which requires only one pass by using a shader model 3.0 approach that supports dynamic branching [15].

The author of the pseudo code in figure 4.2 envisioned using a 3D texture for the lookup of the raw data used to accumulate color and opacity information. 3D textures are described in section 4.3.1 of this document. In the CUDA architecture, texture memory exhibits the most desirable storage and access properties in a system. Texture memory is located in main memory on the device, and therefore has the largest capacity on the device (exact size dependent on the specific hardware device). Texture memory is constant and is optimized for data exhibiting 2D spatial locality through its caching system [10]. Considering its capacity and caching properties, texture memory is a prime target for volumetric data storage.

4.2.3 Shortcomings and Enhancements

Ray casting algorithms have historically required multiple passes to render a scene [6], resulting in long rendering times when compared with other slicing-based methods. Although a single-pass implementation of the ray casting algorithm has been recently discussed [15], this approach requires support for dynamic flow control within the data streams. While NVIDIA's CUDA platform is capable of performing dynamic flow control, it requires significant effort to implement efficiently, and often results in a performance hit.

Due to the relatively lengthy rendering time involved with ray casting, a number of performance enhancements have been identified to speed up the process. Early ray termination, octree space subdivision and empty space

skipping are all methods employed to reduce the degree of computations that must be performed on each ray, leading to quicker renderings. In addition to the various performance enhancements for volume ray casting, there are also image quality enhancement techniques such as supersampling.

4.2.4 Early Ray Termination

Early ray termination can be performed when a volume is rendered in front-to-back order (as opposed to back-to-front order). This typically requires multiple passes through the volume to extract the necessary information about the front and back faces. If a ray leaves the volume being traced, the ray terminates and no more computations are performed. Furthermore, this technique employs a threshold value, indicating a maximum opacity/density. As a ray progresses and opacity and color is accumulated, the resulting composited value is checked against the threshold. If the integrated opacity reaches approximately 1 (within the threshold), it is decided that the remaining voxels along the ray will not significantly contribute to the color of the pixel being rendered, and the ray will terminate [14]. This early termination enhancement efficiently reduces the degree of computation while preserving the visual integrity of the rendered image.

4.2.5 Octree Space Subdivision

Octrees are a tree structure commonly employed when dealing with 3D graphics. In an octree, each parent node may contain either zero or 8 children. The 8 children achieve a 2x2x2 regular subdivision of the parent node, essentially dividing each node into 8 equal parts, or "leaves" [8]. The subdivision of space by the octree is simply a very convenient hierarchical indexing mechanism, which is commonly used when performing occlusion. In volumetric rendering applications, octrees are used to perform empty space skipping.

4.2.6 Empty Space Skipping

Empty space skipping is a technique that allows a ray, when cast through a volume, to "jump" through areas of empty space by avoiding sampling along the ray as long as the ray is intersecting with empty space. This technique requires a separate data structure to contain the empty space information [6]. Octrees are often used to store the statistical information on the child nodes within the tree, such as the min/max bounds for the region covered by the node, providing the requisite information to perform space skipping.

4.2.7 Supersampling

Supersampling or oversampling in general is the act of sampling the scene at a higher resolution than the resolution displayed, such that each pixel value

is calculated by sampling multiple times per pixel. The goal of supersampling is to combat artifacts and aliasing caused by the coarse granularity of per-pixel sampling. The approach is to sample within a pixel region multiple times and average the color values returned from each of those samples to obtain the best possible color representation for a given pixel. The specific qualities of an image have been identified as causing noticeable artifacts due to aliasing, most notably, abrupt changes in intensity commonly found along the silhouette edge of an object in the scene [1]. Adaptive approaches involving convolution have been used to target these areas of high-contrast in an image and perform supersampling on these key points.

A number of supersampling distribution techniques exist, most with their own positive and negative aspects. Some selected examples of sampling techniques include "Regular" sampling, "Jittered" sampling, "n-Rooks" sampling, "Multi-Jittered" sampling and "Hammersley" sampling. Each of these techniques is detailed in chapter 5 of [16]. This chapter includes code listings and excellent figures detailing the effects of each sampling technique. Furthermore, the chapter identifies three characteristics of a well-distributed 2D sampling pattern: the samples are uniformly distributed on a 2D unit square, projecting the points in the x- and y-directions yields a uniformly distributed 1D projection, and finally there must be a minimum distance between samples. Of the examples mentioned here and in [16], only Hammersley sampling exhibits all these characteristics, but is still less than ideal. Although uniform distribution is desired, regular spacing where the distance

between sample points in the x-and y-direction can result in aliasing. Hammersley samples are regularly spaced in the x- and y-directions.

4.3 Slicing Methods

4.3.1 Description

Direct rendering of a volume can be achieved on modern dedicated graphics cards using a three dimensional texture. A "texture", in the field of computer graphics, is an image typically applied to the surface of some geometry in order to simulate the details and appearance of the surface of that geometry. A texture in this sense is two dimensional. A 3D texture can be achieved using a stack of 2D texture slices, which is capable of texturing the volume of an object. Other 3D texture objects exist in hardware specially designed to deal with 3D textures. Historically, 3D texture mapping systems were used by a number of companies in their graphics workstations, such as Silicon Graphics and their InfiniteReality system. Many modern graphics hardware devices support 3D Texture Objects.

3D Texturing is an image order rendering technique, meaning computations are performed based on image size and not based on the number of objects in the scene. Typically, there are a far greater number of pixels in a scene than the number of geometric primitives, making most object-order techniques quicker than image-order rendering techniques. However, in the case of volume rendering, the complexity of the scene typically exceeds that

of the image to be rendered, making the image-order techniques more efficient in most situations. It is generally accepted that, given that nearly all modern graphics hardware supports 3D texture acceleration, real-time interactive rendering of reasonably sized volumes may be achieved using this technique [6].

4.3.2 3D Texturing Process

The "optical properties" of the given volume, specified by an "optical model" (a model which relates the available volume data, typically density, to optical properties such as the emission and absorption of light) can be defined by either using data values directly, or by a "transfer function." Simple transfer functions may be implemented using a fragment shader, but typically a texture lookup table is used [4].

The sampling and compositing of texture-based volume rendering systems are typically performed in a series of steps identified in "Volume Rendering Techniques" [4]. The GPUGems books are compilations of advanced rendering techniques, and are copyrighted by NVIDIA corporation. First, a series of 2D geometric primitives (proxy polygons perpendicular to the viewing direction, parallel to the viewing plane) is placed within the volume. Each of these primitives is assigned coordinates to sample from within the 3D texture. The proxy geometry is then rasterized, such that each proxy polygon is rasterized with its respective 2D slice of the 3D texture. The interpolated texture coordinates are then used as a dependent lookup into

the transfer function textures to assign the optical properties to the volume. Various illumination techniques such as shading may also be applied before the textures are finally composited.

A texture-based volume rendering implementation typically contains three primary steps. Figure 4.3 demonstrates these steps. These are initialize, update and draw. Initialize is usually performed only once, while the update and draw routines are executed when user input is received, and the scene must be redrawn, for example when the camera changes position or orientation.

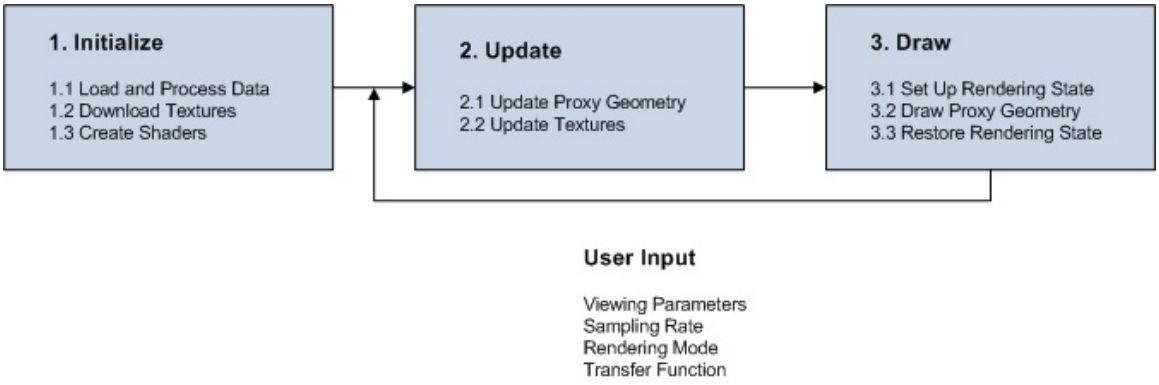


Figure 4.3: The Steps of a Typical Texture-Based Volume Rendering Implementation [4].

The initialization routine prepares the data and shader routines to be used in the volume rendering process. The volume data must be compressed and downloaded to main memory on the graphics device. Before sending the volume data, it may first undergo a variety of processing stages, such as the computation of gradients or downsampling of the data. The initialization stage is also responsible for the construction of the transfer function lookup tables, and any shaders used in the rendering process [4].

The next step is to update the scene. This step occurs every time user input is received which changes any properties of the scene being rendered, or the viewpoint from which the scene is viewed. The proxy geometry is computed in this stage, with a number of slices through the volume being computed, and stored in vertex arrays. Vertices are computed by intersecting planes with the volume bounding box (perpendicular to the viewing angle), and sorting the resulting vertices either clockwise or counter clockwise around their center. For each vertex, the corresponding 3D texture coordinate is calculated. [4].

4.3.3 Shortcomings and Enhancements

Slicing techniques have historically been much faster than ray casting techniques, but most recently the performance gap has closed and these algorithms no longer have such a large performance advantage. Now that the performance gap is closing, the performance versus quality trade off is becoming unbalanced, and images obtained by slicing approaches are of lower quality than those rendered by ray casting. Slice-based techniques lead to artifacts due to aliasing and cannot easily model viewing rays which change direction, as exhibited by refracting volumes [15]. It is also difficult for slice-based methods to implement more complicated optical models, compared to the ease of implementation of the same optical models using a ray casting method. These quality issues have been addressed by the use of over sampling and pre-integration techniques.

A number of performance enhancements have been proposed for the various shortcomings of the 3D texturing approach by integrating specific ray casting techniques with slice-based approaches. It has been observed that in the standard implementation of 3D texturing, a number of unnecessary operations are performed (such as texture fetch operations, numerical operations, and per pixel blending operations) are performed on fragments that do not contribute to the final image. Therefore, it was proposed that acceleration techniques such as early ray termination and empty-space skipping be integrated into 3D texturing approaches [6].

For reference, the following are extensions or enhancements to the slicing approach.

- Shear-Warp Factorization [7]
- Volumetric Clipping [14]
- Pre-Integration [14]
- Integration of Ray Casting Techniques [6]

4.4 Analysis of Techniques for CUDA

The ray casting algorithm is embarrassingly parallel, making it an ideal target for implementation on CUDA. Despite the dynamic branching inherent in the ray casting algorithm, the algorithm is much easier to exploit using CUDA than the slicing method. The ray casting approach also has the advantage of superior image quality when compared to ray casted images due

to aliasing, as well as more potential illumination models. Several performance enhancements have been identified and listed for ray casting such as early ray termination, octree subdivision and empty space skipping. Each of these performance enhancements may be implemented using CUDA. As discussed, the storage of volume data may also be exploited using CUDA by placing it in texture memory. Texture memory exhibits a caching policy that exploits spatial locality; ideal for accessing volume data. The ray casting approach was chosen for implementation for this thesis due to its parallel and easily exploitable nature, as well as the ease of implementation of the standard performance and quality enhancements.

4.5 Existing CUDA Implementations

A small number of efforts to perform volume ray casting using CUDA exist. However, most of these efforts are non-academic in nature, and are closed source projects still in development. The most relevant volume ray casting effort was discovered on NVIDIA's CUDA website. A recent doctoral thesis released in May 2008 by Jusub Kim at the University of Maryland, College Park, focuses on volume ray casting using the CUDA platform, as well as the CELL Broadband Engine [5]. A second relevant volume ray casting on CUDA system which deserves mention is an example volume rendering application provided with the NVIDIA CUDA SDK.

4.5.1 Jusub Kim's Thesis

Kim's doctoral thesis makes a number of advancements in ray casting on the CUDA platform and the CELL Broadband Engine [5]. First, new out-of-core data management techniques were implemented; specifically a new layout scheme of the slice data used to compose the volume and a new multidimensional indexing structure. Second, an efficient stream-based parallel implementation of volume ray casting was implemented on the CELL processor and then extended to the CUDA platform. This implementation specifically optimized two of the primary ray casting performance techniques, early ray termination and empty space skipping.

The results of Kim's CUDA implementation show CUDA as the clear champion of the three architectures assessed (NVIDIA's CUDA on an 8800GTX, IBM's CELL processor and Intel's Xeon processor). The CUDA implementation achieved a 1.5x speedup over the CELL, and a 15x speedup over the Xeon processor, with only a third of the lines of code used for the CELL processor's implementation. Figure 4.4 shows a performance comparison in frames per second (fps) of four different volumes on the three architectures.

The important distinction between Kim's doctoral thesis and this thesis is the implementation and exploration of various unique ray casting techniques to improve performance, visual quality, and investigation into various volume rendering applications. Foremost, although an exact replication

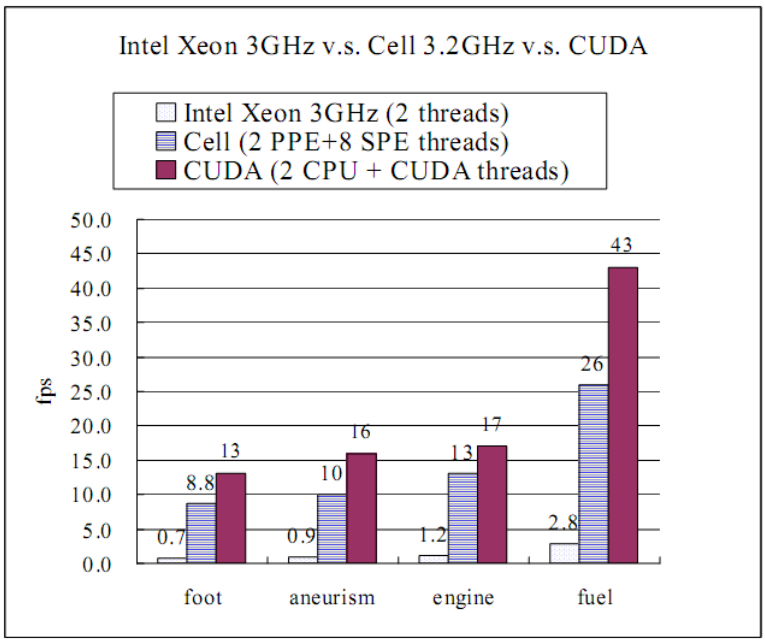


Figure 4.4: Performance Comparison (CPU v.s. CELL v.s. CUDA) [5].

of testing conditions is impossible given the details from Kim's thesis, the volume renderer implemented in this thesis has been shown to achieve significantly higher framerates and far greater speedup, which can be seen by comparing Kim's results with the results collected in chapter 6. The volume renderer implemented in this thesis uses dynamic line segment allocation, while Kim's uses dynamic block allocation for partitioning threads which this thesis declares inferior to line segment partitioning (see section 5.2). Kim's thesis makes no apparent investigations into visual quality enhancements while this work examines the quality and performance aspects of both supersampling and texture filtering. Finally, the work put forth in Kim's thesis performs direct volume rendering on a single volume, while this volume

renderer developed in this thesis is capable of displaying 32 volumes simultaneously capable of rigid registration, can perform hypertexturing, and implements stereoscopic anaglyphs.

4.5.2 CUDA SDK Volume Renderer

The NVIDIA CUDA SDK assists developers by providing them with a number of example CUDA applications. Among these applications exists a volume renderer which utilizes the ray casting algorithm. The source code for this volume rendering system is available as part of the SDK, and copyrighted by NVIDIA. As an important disclaimer, the volume renderer contained in the CUDA SDK was examined as a resource which provided example code for utilizing texture memory and pixel buffer objects. The behavior and implementation of the volume renderer implemented in this thesis was conceived exclusively by the author of this thesis document except where noted and credited, and the volume rendering system developed in this work is fundamentally unique from the volume renderer provided by NVIDIA.

A large number of differences exist in both feature sets and implementation of the volume renderer developed in this work and the volume renderer provided in the CUDA SDK; in fact the only common feature is the ability to perform texture filtering, because 3D textures were used to store the volume data in both implementations. The volume renderer in the CUDA SDK performs no early ray termination or supersampling, and supports strictly

one volume. None of the applications explored in this thesis are included in the SDK's volume renderer. The approach to implementation is also entirely different, with the CUDA SDK placing all host functions in a single ".cpp" file and all device functions in a single ".cu" file, while this thesis makes significant effort to structure source code in a nearly object oriented fashion. The SDK's renderer does not make use of global memory, or contain a world structure to define the scene. Finally, the CUDA SDK is not particularly extensible, making it difficult or clumsy to add support for applications such as multiple volume rendering or anaglyphs. The volume renderer implemented in this work addresses and overcomes these issues by focusing on the thesis goals: completion of an extensible and portable framework capable of interactive rendering while investigating a number of performance and image quality enhancements, as well as exploring potential applications of direct volume rendering.

Chapter 5

Implementation and Features

In order to achieve the stated goals of high quality interactive volume rendering, a number of key techniques and optimizations were implemented. Along with these techniques, a framework was constructed allowing simultaneous rendering of multiple volumes, and a number of unique example application of volume rendering were developed, including the rendering of multiple volumes, hypertexturing, and stereoscopic anaglyphs. This chapter discusses each of these features in detail, and explains how they contribute to the thesis objectives.

5.1 Volume Rendering Framework

Due to the complexity of describing a scene consisting of multiple objects in 3D space (as well as an associated camera and view plane), a volume rendering framework was implemented. This framework simplifies the scene description and makes it easily modifiable and interchangeable by placing the necessary variables in a single build function, essentially a script defining the scene. The initialization phase of the program invokes the build

function, which populates the “world” structure; the primary data structure containing all necessary information about the scene to be rendered. For more information on the world structure, please see section 5.4.

The volume rendering program is separated into nearly object-oriented code. As of CUDA version 2.2, C++ style objects are not supported. However the various structures (camera, volume, world, etc.) and the functions related to these structures are separated into their own source files. This particular feat was difficult to accomplish due to errors in resolving symbols between the gcc/g++ and nvcc compilers. Duplicate symbol errors are common, and CUDA does not have an elegant way to handle multiple source files organized in this fashion; in the example SDK, many of the projects have CUDA source code directly including other CUDA source code. Despite these issues, the compartmentalizing of this code, and the use of type defined ‘objects’ (header files corresponding to the objects define the structure of the object) potentially allow the structures to be replaced with true objects; in the event that objects are supported by the CUDA compiler.

A more general but important aspect of the rendering system is accessibility and compatibility. Special care was taken to write code that was capable of compiling and running correctly under a variety of different operating systems and hardware. The renderer has been programed to allow interactive frame rates on desktop systems as well as laptop systems with CUDA capable graphics chips. The renderer was developed and tuned on a laptop

to ensure that a laptop was capable of producing the bare minimum processing power required, bringing the utilization of this visualization technology further away from a dedicated high performance computing environment and onto a mobil platform. The renderer has also been programmed to compile under a variety of different compilers for different operating systems in an attempt to produce a highly cross-platform system. The renderer has been compiled and tested to function correctly under 32 and 64 bit versions of Windows XP, Windows Vista and Mac OS X. Multiple projects which target the same source code have been maintained to enable building across these various platforms; a makefile exists for Mac OS X and a Visual Studio 2005 project exists for the Windows platform.

5.2 Load Balancing

Whenever an application is parallelized, load balancing must be addressed to ensure the maximum utilization of the available compute nodes. Volume rendering, ray tracing, and other rendering systems which utilize ray casting are parallel on the ray level. Assuming that one primary ray corresponds to one pixel in the rendered scene, then the smallest parallelizable block in the workload is the computation of that pixel. Some pixels take longer to compute than others; in ray tracing, the number of rays spawned from the primary ray increase the amount of work to perform for each pixel, and with volume rendering, the distance a ray must march through a volume before exiting the volume (or becoming opaque and terminating early) determines

the amount of work to perform before accumulating a color for the pixel.

Research was conducted to determine the optimal partitioning of work for a ray tracing application on an MPI system. Because the ray casting algorithm is the algorithm being parallelized, and the ray tracing/marching is performed sequentially for each primary ray, the optimal partitioning for ray tracing is the same as the optimal partitioning for ray marching. Figure 5.1 shows an example of type of ray traced scene rendered to analyze partitioning performance.

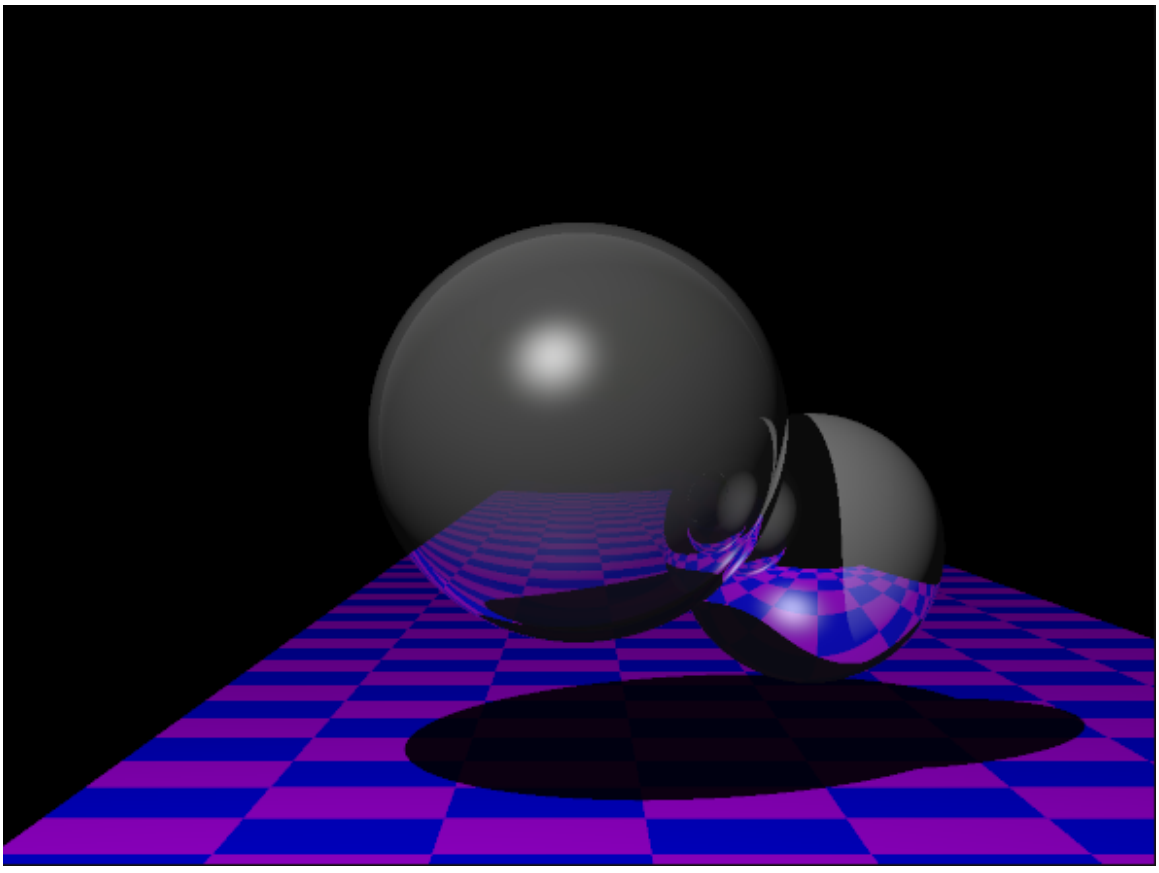


Figure 5.1: Example of a ray traced scene used to collect partitioning performance statistics.

In the CUDA programming paradigm, one ray maps to one thread, and given a scenario with no supersampling, one ray maps to one pixel. If a supersampling of four is chosen, then four rays map to a pixel. These rays can be computed independently and partitioned based on their corresponding pixel in the scene. Three static and two dynamic partitioning methods were implemented and analyzed to determine the optimal partitioning of rays within a scene. Figure 5.2 shows the partitioning methods.

A frame of the scene cannot be rendered until every thread has completed. Block partitioning inherently does a poor job of partitioning the work in a scene because the majority of the complexity in a scene may exist in one of the block partitions, requiring every thread in that partition to execute complex work, while other partitions have little work to do. In a ray casted scene, the threads of greater complexity are most often clustered together, where a particular object to be drawn exists. Row partitioning is the second best option, where the rows are more likely to equally divide the work to be done evenly among the partitions. However, cyclic partitioning achieves the best performance, where the complex work (the object or volume in the scene) is most likely to be evenly divided among the partitions. Figure 5.3 shows that cyclic partitioning performs the best in the static partitioning scenarios.

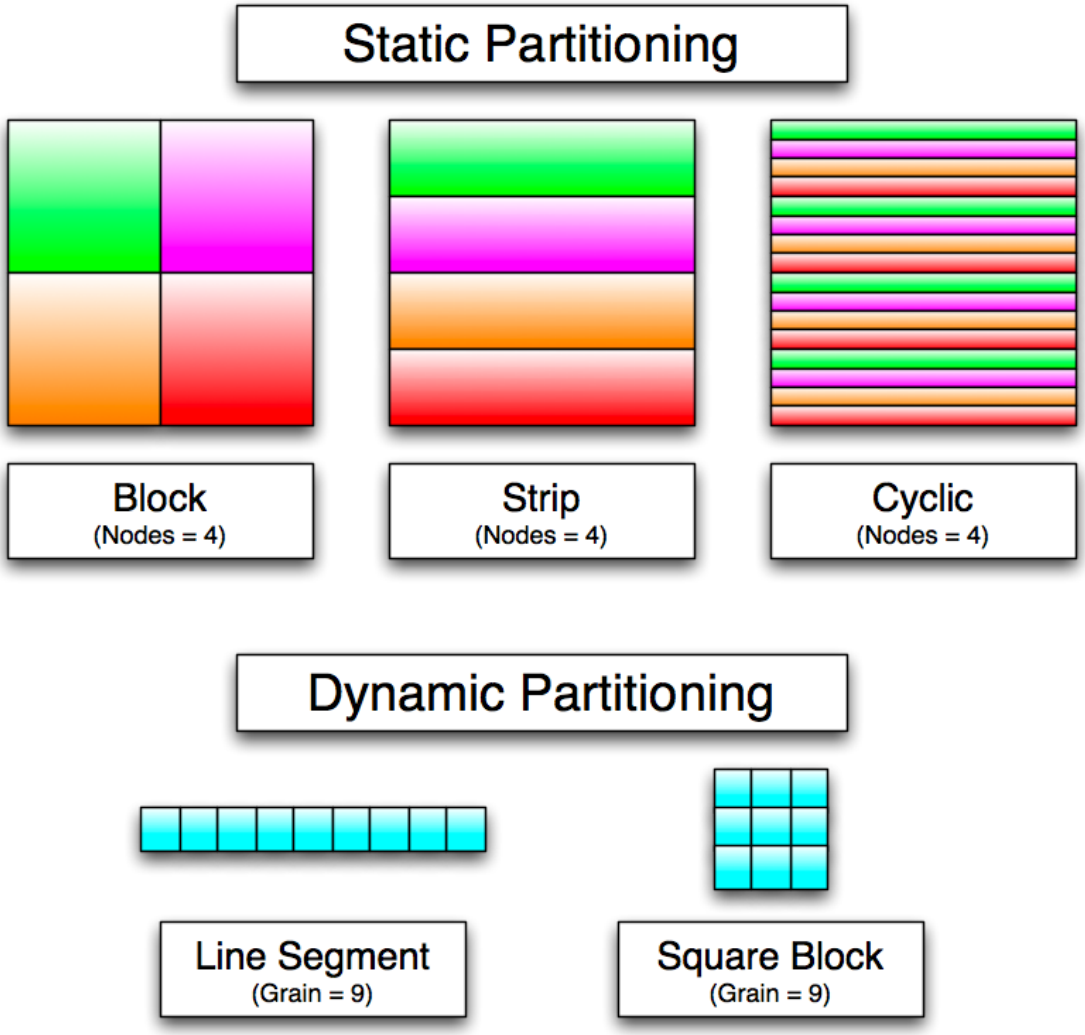
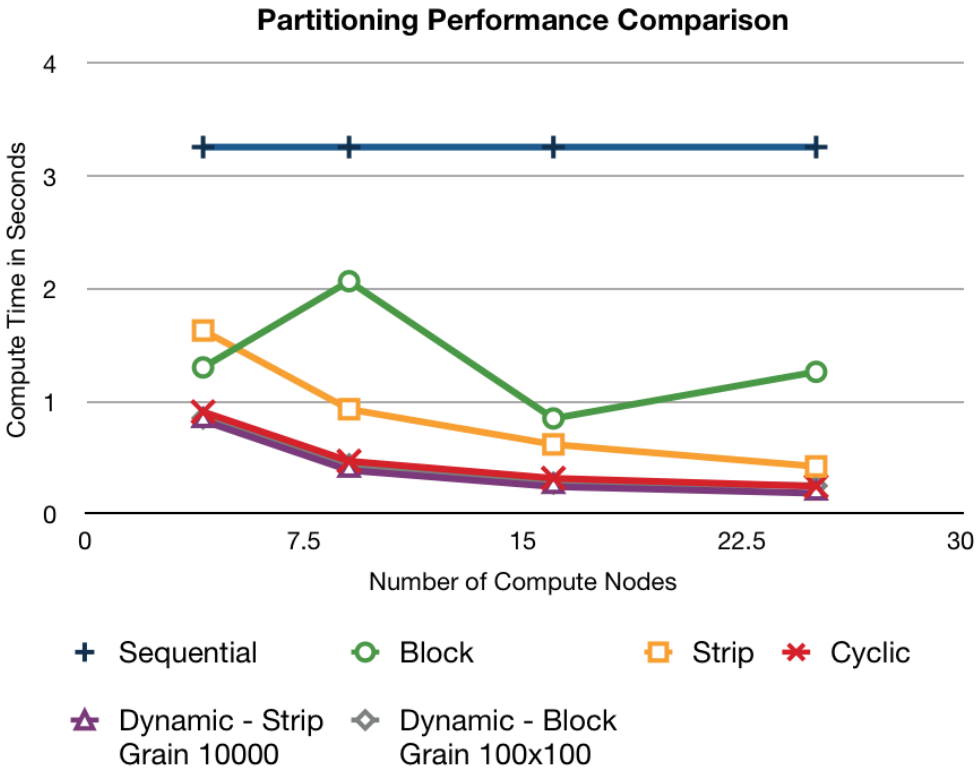


Figure 5.2: Static and Dynamic Thread Partitioning Methods.



# Compute Nodes	Sequential	Block	Strip	Cyclic	Dynamic - Strip Grain 10000	Dynamic - Block Grain 100x100
4	3.257812	1.300781	1.628906	0.90625	0.828125	0.851562
9	3.257812	2.066406	0.929688	0.46875	0.386719	0.4375
16	3.257812	0.847656	0.617188	0.316406	0.242188	0.273438
25	3.257812	1.261719	0.421875	0.246094	0.183594	0.25

Figure 5.3: Performance of Static and Dynamic Partitioning Methods.

There is a notion of static and dynamic partitioning. Static partitioning statically divides the amount of work to be performed among all available processing elements at once and waits for the results. Dynamic partitioning divides the work to be performed into batches of a particular "grain size," and issues the corresponding work to the next available processing

node until all the computations are performed. Dynamic partitioning generally offers greater performance than static partitioning, providing the cost of communication among processing nodes is mitigated. This is further supported by the experimentation using MPI, where both static and dynamic partitioning algorithms were implemented. Figure 5.3 shows that dynamic partitioning performs slightly better than the best static partitioning method, and furthermore that line segment partitioning (essentially cyclic partitioning with a certain grain size) performs the best of all partitioning methods.

CUDA operates by dynamically dispatching thread blocks to available multiprocessors. Assuming there are 64 threads per block, then the grain size is 64 elements. There is no sense of static partitioning in CUDA because the number of available multiprocessors on a device cannot be known at compile time. The volume rendering application partitions threads within a thread block as a line segment to be rendered in a scene, resulting in the best partitioning method investigated. Figure 5.4 illustrates exactly how the threads and thread blocks are allocated to render a frame. The dimensions of the rendered frame are 640x480, and there are 64 threads per block. The pixels are colored from black to blue according to their thread ID within a thread block, and from black to red according to their block ID within a grid. There are ten distinct columns in the figure because 64 threads are evenly divided ten times into the horizontal resolution of 640. This figure shows that a line segment of grain size 64 is used, and dynamically distributed in the form of thread blocks to available multiprocessors.

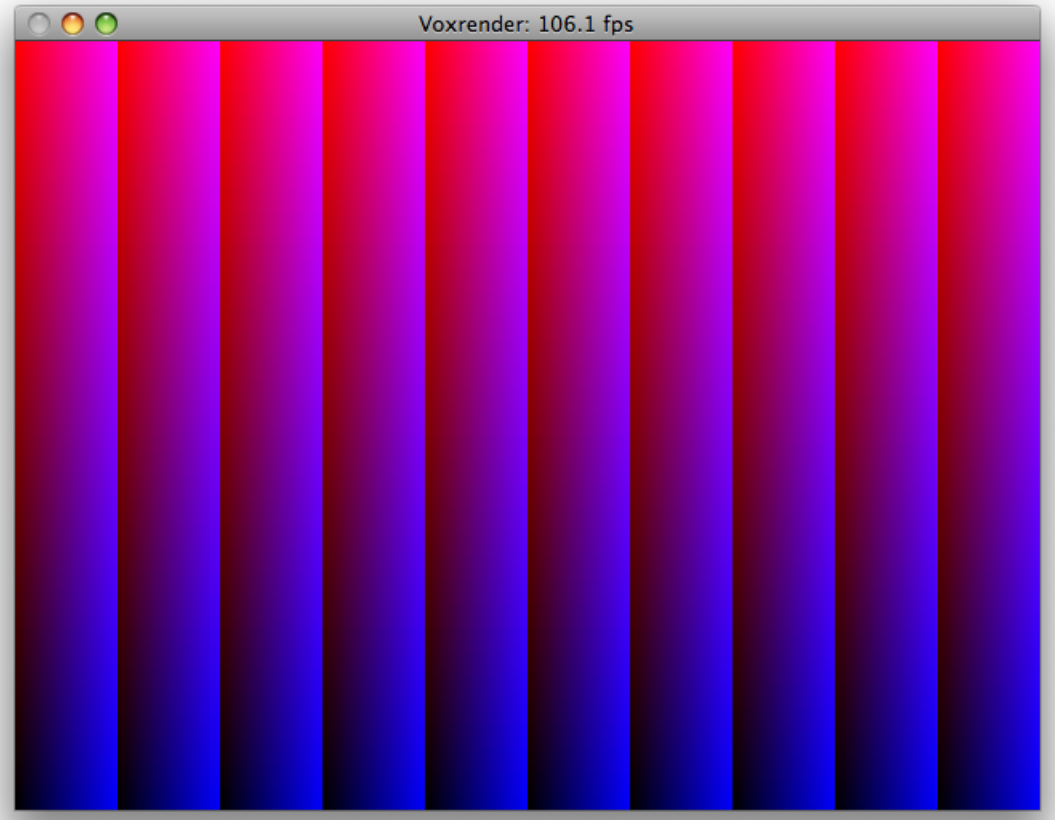


Figure 5.4: Actual partitioning of threads in the volume renderer. The blue channel is increased based on the thread ID within a block, and the red channel is increased based on the block ID within a grid. This shows dynamic partitioning of line segments of a grain size equal to the threads per block; in this case, 64.

The last variable to adjust regarding load balancing is the number of threads per block (or essentially the grain size). However, altering this number may be misleading because it does not simply alter the grain size, but it also alters the number of blocks which can fit on a multiprocessor, and therefore the number of available multiprocessing elements. This is because

there is a certain amount of memory required to fit a thread block on a multiprocessor, and in this particular application, the available multiprocessor memory is the limiting factor in the number of thread blocks which may be simultaneously assigned to a multiprocessor. For this reason, the effects of the adjustment of this parameter must be examined experimentally. The results of adjusting this parameter can be seen in table 6.3. NVIDIA recommends a multiple of 64 threads per block for optimal performance, which is further validated by these performance results. To properly balance the multiprocessor occupancy and the distribution of work among the systems tested, 64 threads per block was chosen as a default value for the volume renderer. Further information can be found in section 5.3.

5.3 CUDA Multiprocessor Occupancy

Multiprocessor occupancy is defined as the ratio of the number of active warps on a multiprocessor to the maximum number of active warps. In other words, occupancy describes the ratio of the actual number of simultaneous computations on a multiprocessor to the maximum number of simultaneous computations. NVIDIA provides a "CUDA Occupancy Calculator" spreadsheet to determine the occupancy of a graphics device based upon compute capability, threads per block, registers per thread and shared memory per block. Significant effort was placed in finding the optimal allocation of resources on a variety of CUDA capable systems. Considering the various CUDA compute capabilities, the volume renderer was designed

with portability as a priority, and tailored to run on compute model 1.0 as a minimum specification. Both register usage and thread partitioning are addressed. however the volume renderer does not make use of shared memory. Although shared memory exhibits excellent performance characteristics, all of the necessary variables and data in the volume renderer exist in memory locations capable of similar performance, sometimes via caching. Therefore, usage of shared memory was not warranted, and hence this variable is not considered in the investigation of multiprocessor occupancy.

CUDA has a notion of register usage within a multiprocessor. The complexity of calculations within a device function contribute to the register usage, however the exact method of calculation of registers used is unknown. The volume renderer uses a very large number of registers, approximately 46 registers for the rendering kernel possibly varying slightly based on settings such as ray marching order, *etc.*The number of registers used per thread directly contributes to the potential occupancy of the graphics device. Figure 6.2 (top left) shows a compute capability 1.1 device using 46 registers and 64 threads per block. The occupancy of the device is only 17%, and limited to 2 blocks per multiprocessor due to the number of registers in use.

It is possible to limit the number of registers used per thread during compile time. The remaining registers are offloaded into “local memory”, which is more accurately described as private memory located in the same space

as global memory. This memory has drastically slower performance characteristics than registers, however the resulting increased occupancy can outweigh the change in memory performance. Figure 6.2 (top right) shows the occupancy of a compute capability 1.1 device using 32 registers and 64 threads per block. The registers can be limited further, but the increased occupancy will no longer overcome the degraded memory performance due to the increased use of local memory. This was determined experimentally. Table 6.2 shows the results of varying the the register usage.

Devices with compute capability 1.3 have twice the number of registers at their disposal. Therefore, different performance trends exist for these devices. However, because devices with compute capability 1.0 were chosen as a minimum system requirement, the register usage was selected using a compute capability 1.1 device (with performance identical performance characteristics to compute model 1.0) and kept consistent across systems. Figure 6.3 shows the effects of varying the register usage on a device with compute model 1.3. The difference in compute models shows little deviation in the optimal register usage, with 32 registers per thread achieving maximum performance.

Another major factor in the device occupancy is the number of threads per block. As discussed in section 5.2, the number of threads per block is also the grain size of the partitioned threads. Because several factors are being varied by changing the number of threads per block, the optimal performance must be determined experimentally. Table 6.3 shows the results

of varying the number of threads per block.

5.4 Constant Memory and the World Structure

The world structure is the primary data structure containing all necessary information about the scene to be rendered. The world structure may be altered between frames but never during a frame. For example, using the mouse to rotate or translate the object will ultimately update the world's camera. However, the world structure is guaranteed not to change within a frame. This behavior can be exploited in CUDA. Because the world structure remains constant during a frame, the render kernel handles one discrete frame at a time and all threads use the same world in their computations, the world can be placed in constant memory.

The benefit of constant memory is that it frees up memory from each multiprocessing element, and after the initial cache miss, the cost of reading from constant cache is the same as the cost of reading from a register, provided that all the threads in the executing half-warp are reading from the same location in constant memory. In the volume renderer, all threads which follow the same branch will access the same address in constant memory.

5.5 Texture Memory, Volumes and Filtering

Although a ray casting approach is used to render volumes, the volumes are still stored in texture memory. This is because the largest available memory

space in CUDA is the memory space containing global memory and texture memory, which is similar to main system memory on traditional computers. Global memory is not cached; however, texture memory exploits 2D spatial locality by caching an element in texture memory along with the elements "around" that memory location. The volume renderer stores volumes in texture memory for both its size (allowing larger volumes to be rendered depending on the specifications of the graphics device), and its caching capabilities.

An additional advantage to using texture memory to store volumes is the ability to filter the data when reading from the texture. It is common in modern video games to filter textures to enhance visual quality. The filtering provides a smoothing effect to the data contained in the texture. CUDA provides two filtering modes: a point filter which returns the closest texel to the coordinate requested in texture memory, and a linear filter which is a linear interpolation of the several nearest texels to the texture coordinate. The volume renderer uses a 3D texture to store the volumes, resulting in the interpolation of 8 texels when reading a texture coordinate. Figure 5.5 illustrates the difference between these two filtering modes. Clearly, the linear filter provides a much higher quality image at little to no performance hit for the examined volumes (32x32x32 voxels).

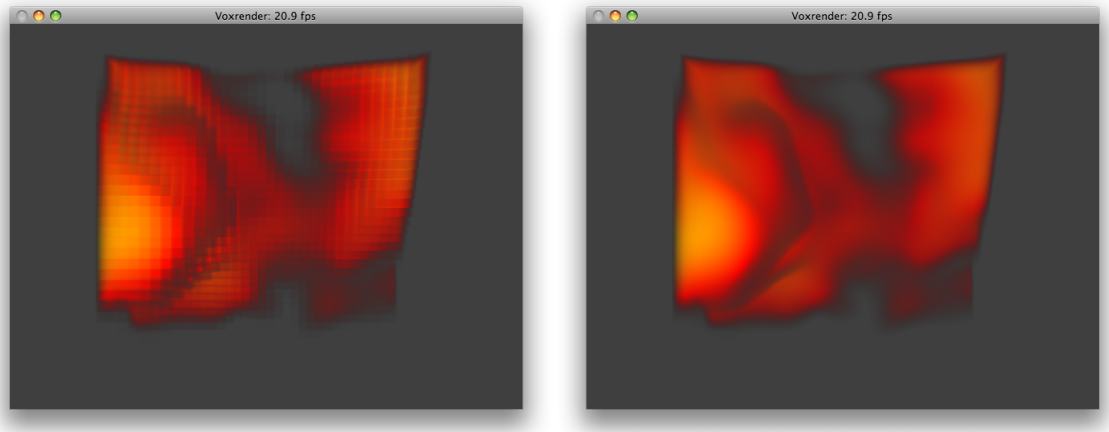


Figure 5.5: An image of a 32x32x32 hypertexture rendered using point filtering (left) and linear filtering (right).

5.6 Early Ray Termination

Early ray termination is a technique used to boost performance by limiting the amount of work to be done for each ray. The idea is that as a ray marches through a volume accumulating density/opacity, eventually the accumulated density becomes so great that the remainder of the volume intersected by the ray will make no significant impact on the resulting color of the pixel. This threshold allows the ray to be terminated early. This technique obviously requires front to back ray marching to be beneficial, as the front voxel data has greater bearing on the resulting image than the back voxel data. Figure 5.7 is pseudocode for accumulating color by marching front to back. Figure 5.6 is pseudocode for the alternative method of marching back to front.

```
For each step through the volume:
    sum = (1 - sum.alpha) * volume.density * color + sum;
return sum;
```

Figure 5.6: Pseudocode used to accumulate color by ray marching through a volume from front to back.

```
For each step through the volume:
    sum = lerp(sum, color, color.alpha * volume.density);
return sum;
```

Figure 5.7: Pseudocode used to accumulate color by ray marching through a volume from back to front.

The volume renderer was implemented using both front to back and back to front rendering techniques. It is possible to select front to back or back to front rendering during compile time. When rendering from front to back, it is possible to specify a density threshold value at compile time, which will determine when to terminate a ray. Figure 5.8 shows the difference between marching back to front (top left), front to back without early ray termination (top right), with early ray termination with a threshold of 0.5 (bottom left) and early ray termination with a threshold of 0.95 (bottom right). It can be seen that marching front to back achieves a different visual representation than marching back to front. The rendering with a ray termination threshold of 0.5 is much darker and has a lower level of detail than without early ray termination, however shows a significant performance gain (approximately a 60% speedup). The rendering with a ray termination threshold of 0.95 is nearly indistinguishable from the rendering without early ray termination,

and manages a small performance increase of approximately 6%.

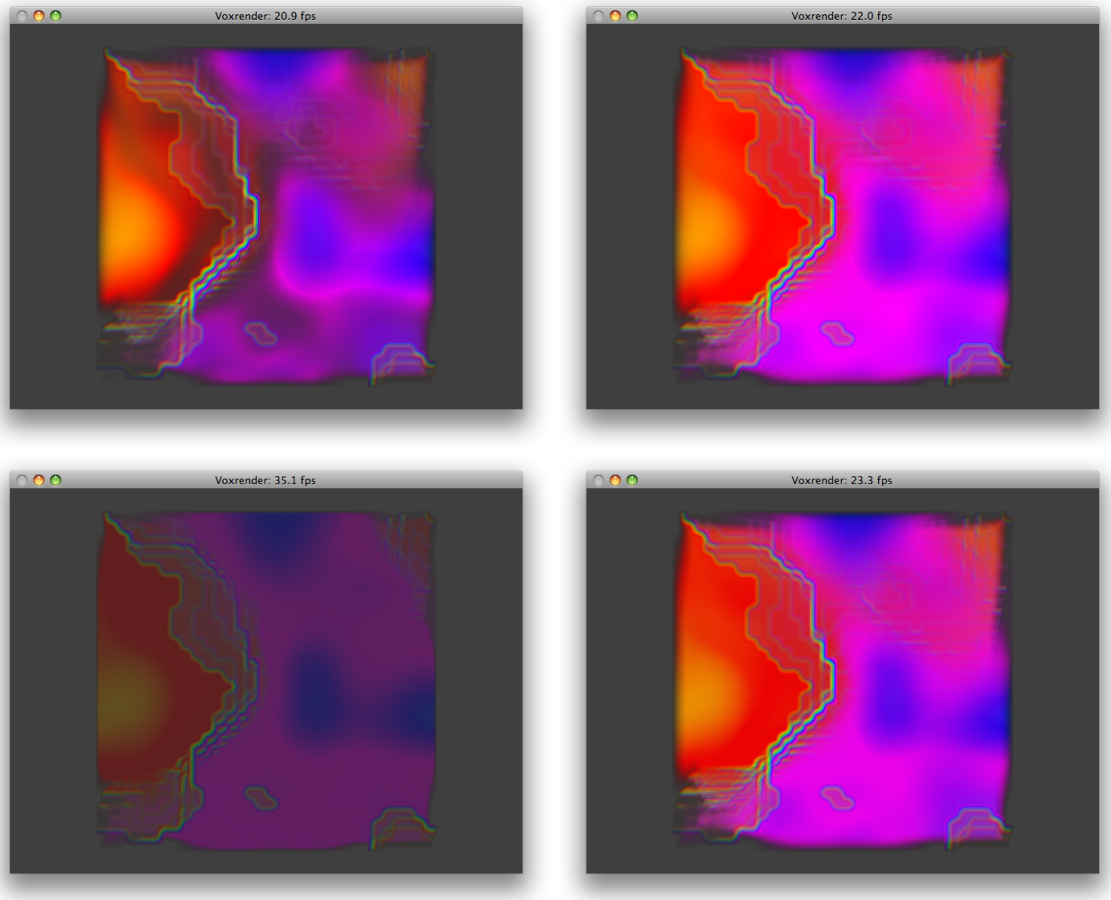


Figure 5.8: Four images showing a hypertexture rendered in back to front order (top left), front to back order (top right), front to back order with an early ray termination threshold of 0.5 (bottom left) and front to back order with an early ray termination threshold of 0.95 (bottom right).

5.7 Supersampling

Supersampling, or oversampling, is a common visual quality enhancement used to reduce aliasing in a rendering. It is often used in ray tracing to

reduce “jaggies” that appear on edge of objects, or on sharp contrast boundaries. If only one ray is fired per pixel, then that ray will either intersect with a particular object in the scene or it will not, resulting in a sharp jagged look along the edge of an object where some pixels strictly represent that object and the rest do not. Supersampling is a technique where multiple rays are fired per pixel, and their resulting color is averaged together. The result is a smoother look on sharp contrast boundaries, where if one pixel lies exactly on a boundary, the rays will average to a color between the two bounded colors. In the volume renderer, supersampling was implemented by simply dividing a pixel into equal parts with rays cast for each sub pixel, and averaging the resulting colors from each ray for the final pixel value. Figure 5.10 shows the pseudocode for supersampling. Figure 5.9 shows the effects of an image without supersampling (left) and with 16x supersampling (right) for both point filter mode (top) and linear filter mode (bottom). Although supersampling is a proven enhancement for other ray casting systems such as ray tracing, volume rendering shows no noticable quality enhancement, likely because supersampling softens edges of sharp contrast, which don’t normally appear in volume renderings.

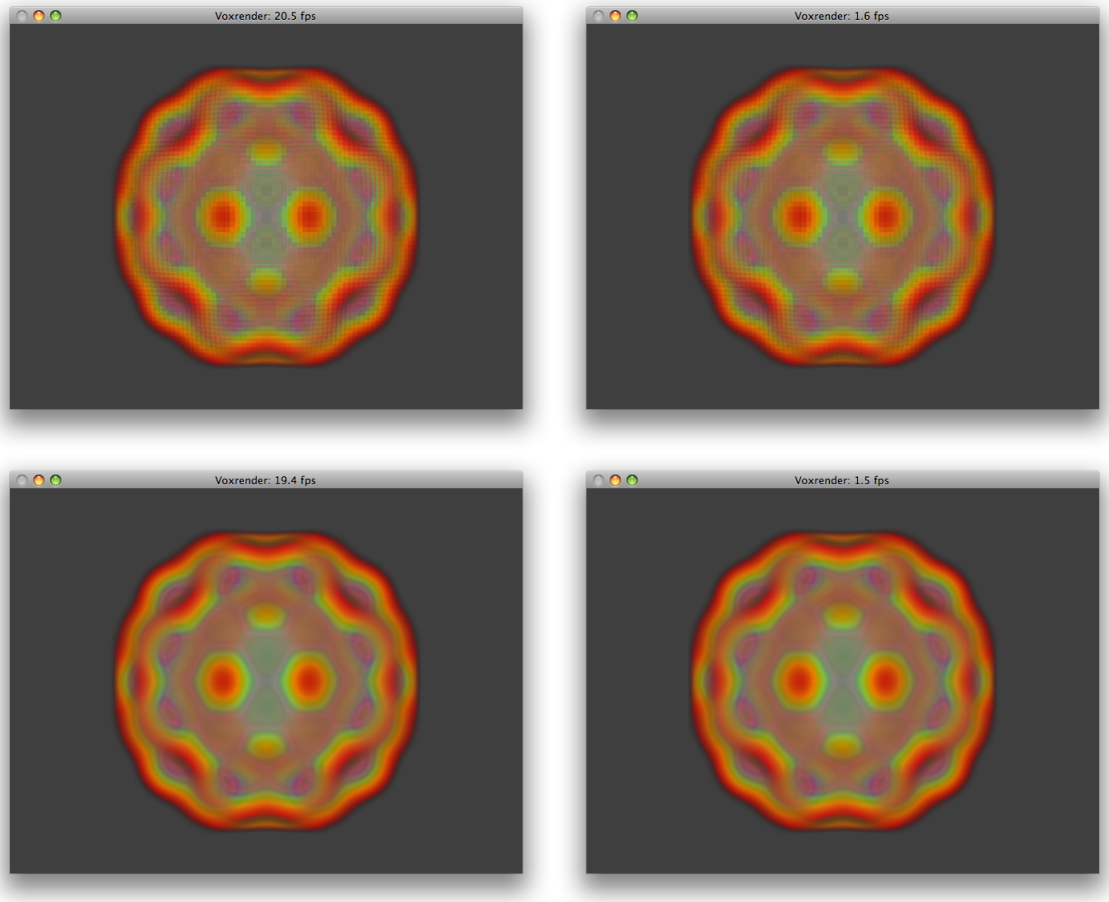


Figure 5.9: Four images showing a simulated Buckyball rendered using point filtering with no supersampling (top left), point filtering with 16x supersampling (top right), linear filtering with no supersampling (bottom left) and linear filtering with 16x supersampling (bottom right).

```
int n = (int)sqrt((float)num_samples);
    for (int p = 0; p < n; p++)        // up pixel
        for (int q = 0; q < n; q++)    // across pixel
        pp.x = pixel_size * (x - 0.5 * hres + (q + 0.5) / n);
        pp.y = pixel_size * (y - 0.5 * vres + (p + 0.5) / n);
        ray.d = get_direction(pp);
        pixel_color += trace_ray(ray);
pixel_color /= num_samples;
return pixel_color;
```

Figure 5.10: Pseudocode for supersampling.

5.8 Multiple Volume Rendering

One of the most novel and unconventional aspects of the volume renderer developed in this thesis is the ability to render multiple volumes simultaneously. The intention is to further research into potential for volume rendering systems to be used in more popular applications such as simulations and video games. Figure 5.11 shows six volumes rendered simultaneously; each with a different voxel resolution, placement, and z extent.

The rendering of multiple volumes simultaneously was taken into careful consideration when designing the volume renderer, and the rendering framework reflects this. As shown in figure 5.11, it is possible to specify a number of different volumes, each of which are maintain their own independent voxel resolution. For example, the protein molecule is 64x64x64 voxels, the engine block is 256x256x256 voxels, the orange is 256x256x62 voxels, and so on. There is also the notion of a dimension's extent, or the "true

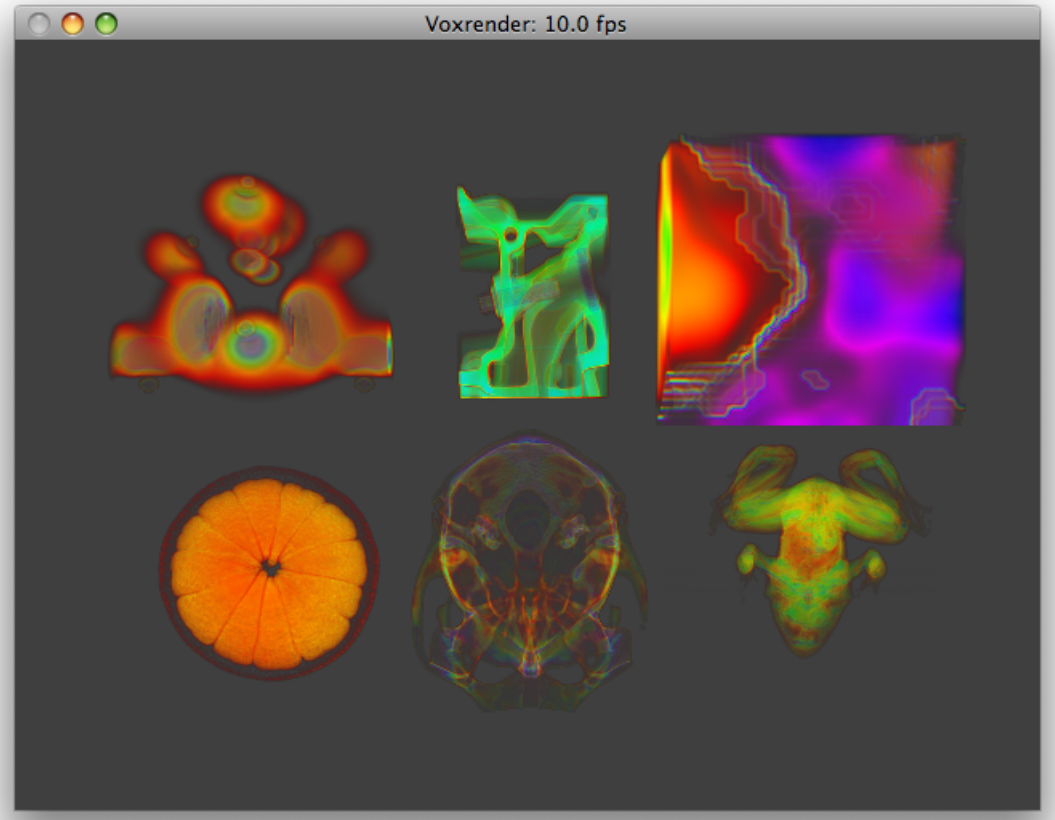


Figure 5.11: An image showing six volumes rendered simultaneously. From left to right, top to bottom they are a protein molecule, an engine block a hypertexture, an orange, a monkey's skill and a frog.

dimensions" of a volume. This is essentially a ratio describing the voxel resolution in a particular dimension, and the actual distance that dimension is intended to represent. Medical imaging systems typically operate by obtaining "slices" of a volume at a time; however the distance represented by the depth of a slize (z axis) is different than the distance represented by a pixel within that slice (x and y axis). The rendering system is capable of dealing with this difference as well: for instance, the CT scan of the monkey's head

is 256x256x62 voxels with an extent ratio of 1:1:3. This indicates that the 62 slices in the z dimension represents the same physical distance as 186 (62x3) pixels in the x or y dimension.

Each volume is described using a volume structure which contains the metadata indicating how to position and render the volume, and the raw volume data stored in texture memory. An unfortunate limitation of CUDA requires the programmer to access volumes by variable name and not as part of an array, requiring the number of textures used to be known at compile time, rather than runtime. This requires a hard-set limit on the number of volumes represented in the scene; similar to how openGL limits the number of lights in a scene to only 8. The volume renderer has a set maximum of 32 specifiable volumes. It is trivial to increase this number at compile time; but 32 was determined to be a sufficient proof of concept. Figure 5.12 shows 32 separate buckyball volumes being displayed simultaneously. The ability to perform multiple volume rendering was one of the greater difficulties encountered while developing the volume renderer. Due to limitations in CUDA, there can be no references or pointers to textures. Therefore, an array of textures may not be used to simply index into a particular texture based on a numerical ID. The solution to this was to generate a several hundred line switch statement in the code to perform the necessary actions for each volume, resulting in very verbose source code. See section 7.4 for more information on the difficulties encountered with texture memory.

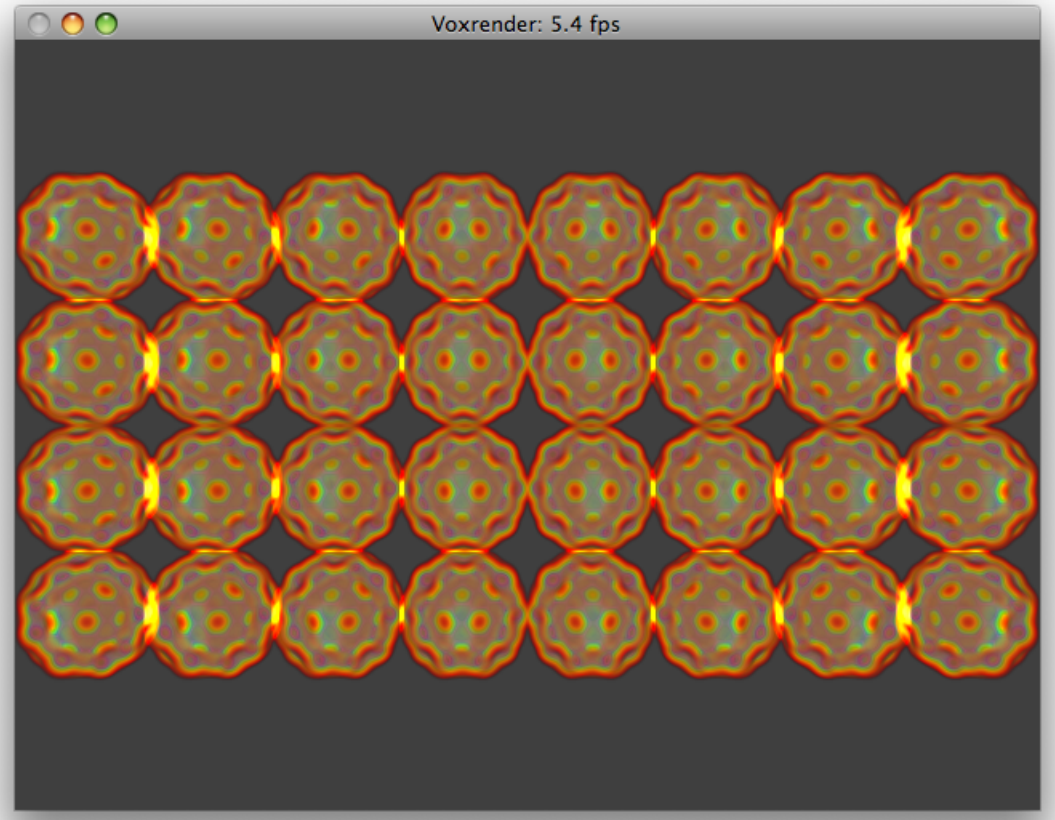


Figure 5.12: An image showing 32 individual volumes, the current maximum number of displayable volumes in the renderer.

Volumes may also also intersect with one another. One of the major benefits of volume rendering is the ability to visualize the interior of an object, as opposed to strictly visualizing exterior which is common for nearly all other rendering systems. In conjunction with the opacity associated with volume rendering, it is possible to to nest two semi-opaque volumes to achieve an entirely new visual representation. Medical imaging for example is greatly

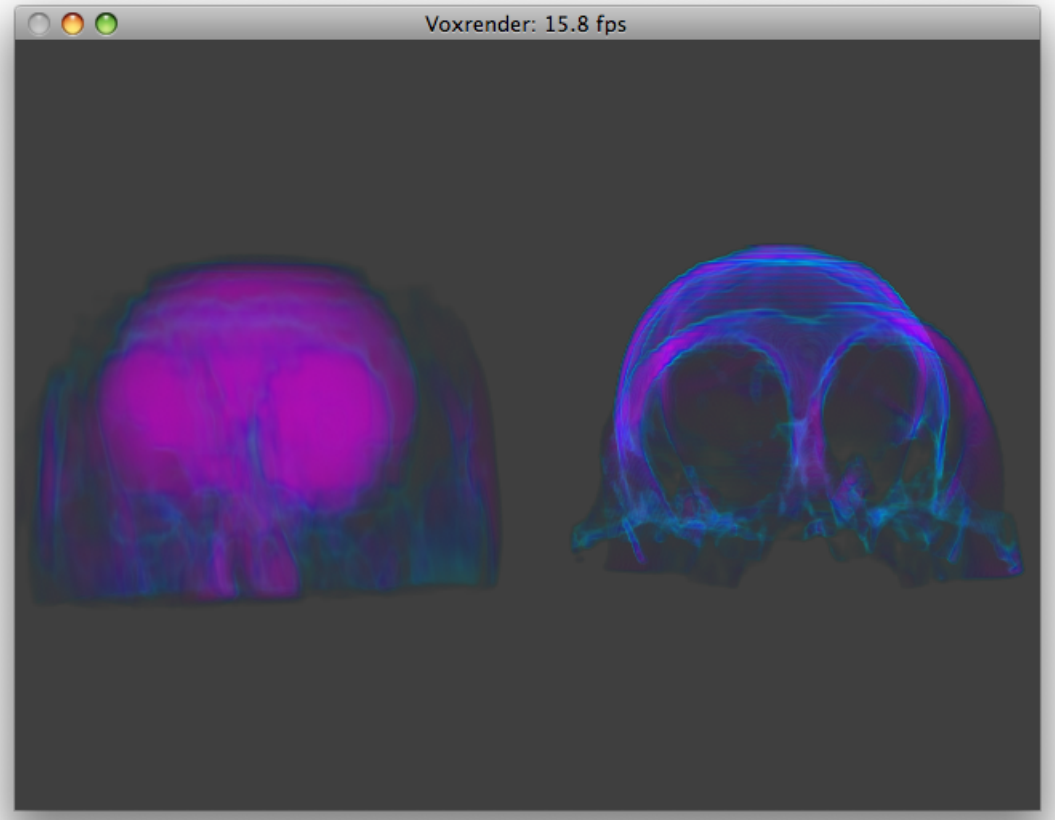


Figure 5.13: An image showing an MRI scan (left) and a CT scan (right) of a monkey's head. The density factor of the MRI scan is one fifth the density factor of the CT scan.

enhanced by the ability to overlay scans of different characteristics to visually achieve a better understanding of the subject. The volume renderer allows this by enabling quick rigid registration of volumes. Figure 5.13 shows an MRI (left) and a CT (right) of a monkey's head. The MRI volume has one fifth the density factor as the CT volume. Figure 5.14 shows the MRI and CT volumes nested. The volume rendering framework provides a density factor for each volume which helps highlight the areas the user wishes to stand out. In this case the bone structure of the monkey is slightly

more distinct than the tissue.

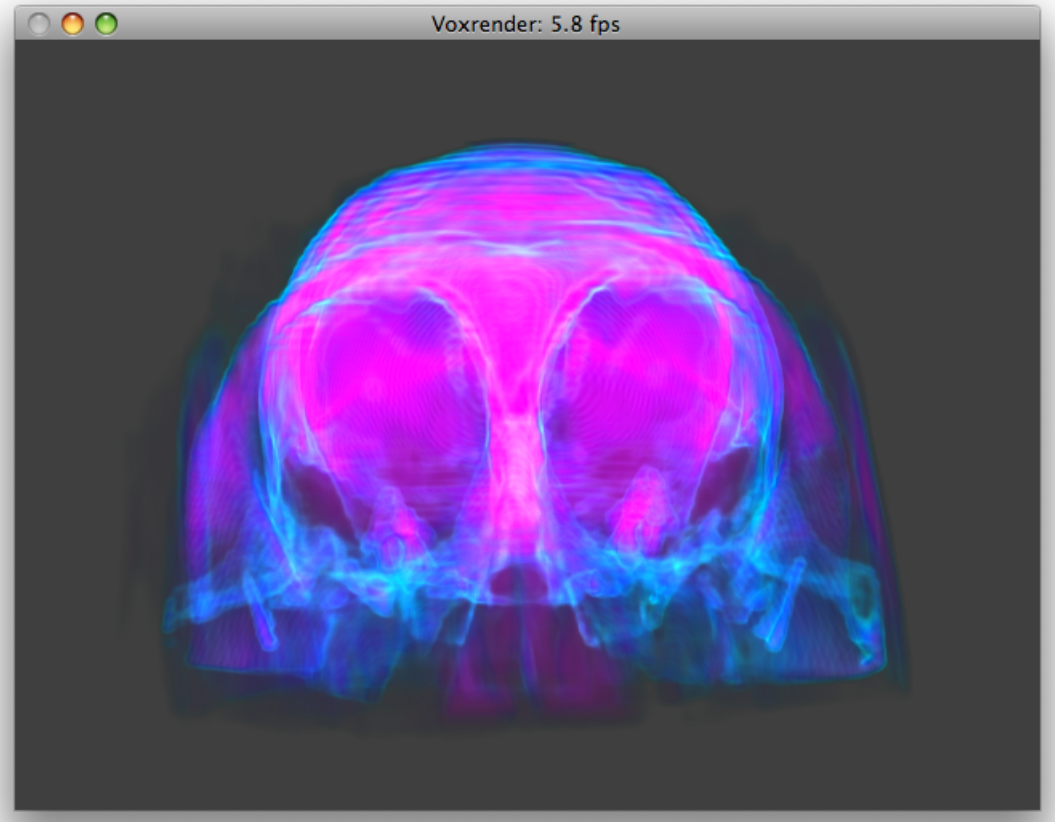


Figure 5.14: An image showing an MRI scan and the CT scan of a monkey's head nested together. The density factor of the MRI scan is one fifth the density factor of the CT scan to highlight the results of the CT scan.

5.9 Hypertextures

Another novel contribution is the implementation of hypertexturing using CUDA. Hypertextures are a concept propsed by Ken Perlin [11], and are a way of producing objects with surfaces particularly difficult to define with conventional polygonal meshes; such as gaseous or liquid objects like fire,

smoke and ice. While volume rendering is the traditional approach to displaying hypertextures, this thesis is very likely the first attempt at rendering these hypertextures using CUDA.

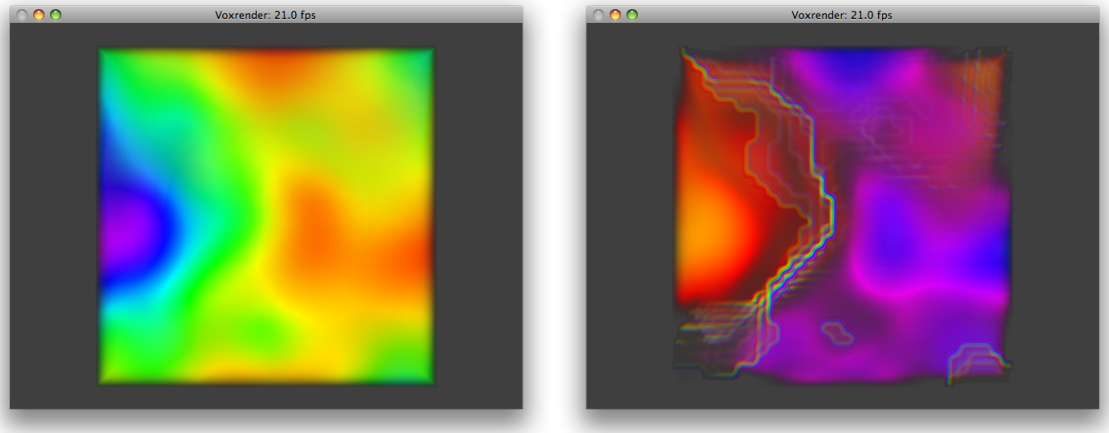


Figure 5.15: Two images showing the density modulation function of a hypertexture. The image to the left was generated with voxel values between [0,1]. The image to the right was generated using voxel values between [-0.5,0.5], and reinterpreted as unsigned values.

The hypertextures rendered are actually the density modulation functions more commonly used to determine how to adjust the density or shape of an existing or implicit object. Perlin noise, another invention of Ken Perlin, is used to generate random but cohesive density values to generate a volume. The Perlin noise is essentially computed on the host at runtime, loaded into a texture, and rendered using the volume renderer running on the graphics device. Figure 5.15 shows a hypertexture with densities between [0, 1] (left), and a hypertexture without bounds checking with densities between [-0.5, 0.5]. However the voxel data is interpreted as unsigned characters, where negative numbers casted as unsigned values become interpreted as

large positive numbers. The hypertexture with bounds checking provides a much smoother image, while the hypertexture without has sharp contrast between dense areas, and the linear texture filtering results in the visually interesting rainbow-colored lines.

5.10 Stereoscopic Anaglyph

An anaglyph is an image rendered from two different camera perspectives with a red and cyan filter for the two perspectives. Wearing glasses with a red lens over one eye and cyan lens in the other reveals a stereoscopic image. The human brain interprets the difference of perspective provided by the stereoscopic image as a three dimensional scene. Anaglyphs are have been a novelty in media for many years, with a recent increase of "3D" motion pictures coming to movie theaters.

As another novel application of volume rendering, this thesis incorporates the ability to view and interact with volumes rendered as an anaglyph. This is performed by placing two cameras in a scene instead of one. Each camera is responsible for rendering the scene from a different perspective, and applying the red or cyan filter appropriately. The resulting color observed by each camera is then accumulated to produce a final image. In the volume renderer, each thread is responsible for casting a single ray from both the red and the cyan cameras and accumulating the results. Figure 5.16 shows the pseudocode for anaglyph rendering. Figure 5.17 shows an engine block rendered as an anaglyph.

```
For each pixel or sub-pixel
        ray.o = camera.eye_center - stereo_separation;
        temp_pixel_color = CYAN * trace_ray(ray);
        ray.o = camera.eye_center + stereo_separation;
        temp_pixel_color = RED * trace_ray(ray);
        pixel_color += temp_pixel_color/2.0f;
return pixel_color;
```

Figure 5.16: Pseudocode for anaglyph rendering.

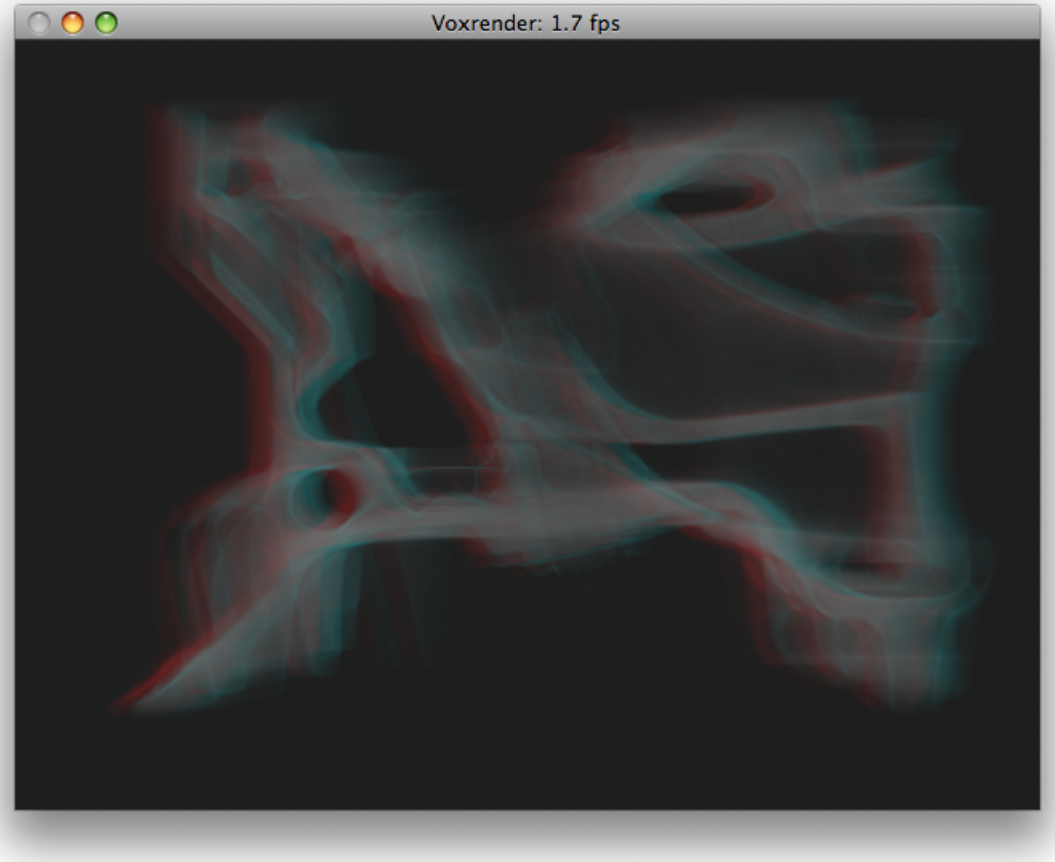


Figure 5.17: An image showing a stereoscopic anaglyph of an engine block. To properly view the image, it must be reproduced in color, and the observer requires “3D glasses” with red and cyan lenses.

Chapter 6

Performance Results

In the previous chapter, several volume rendering techniques implemented in this thesis were examined, and require some metric of performance for proper analysis. A number of factors affect the overall performance of the volume renderer developed in this thesis. Many settings adjust multiple variables simultaneously, requiring experimentation to determine optimal behavior. This chapter visits each of these factors and how they contribute to the performance of the volume renderer. All performance trials were run at a resolution of 640 by 480, using back to front ray marching with no early ray termination, linear texture filtering, 64 threads per block, 32 registers per thread and no supersampling unless otherwise noted.

6.1 Sequential vs. CUDA Implementations

A primitive version of the volume renderer was implemented for Microsoft Windows based systems for the purpose of performance comparison with a sequential implementation. The primitive sequential volume renderer differs from the CUDA volume renderer in many ways. It does not perform

linear interpolation between the data points, allow for interaction with the volume, or utilize a transfer function to assign color based on the sampled density. Most significantly, it does not accumulate opacity while marching through a cube, instead it simply accumulates density until a threshold is reached, and if the threshold is reached, the gradient is computed and returned. However, this is essentially the same behavior as early ray termination, and more significantly with a threshold far below a typical threshold that would be chosen for early ray termination. Therefore, even in optimal cases, the performance of the primitive sequential volume renderer is significantly faster than a sequential implementation of the CUDA volume renderer.

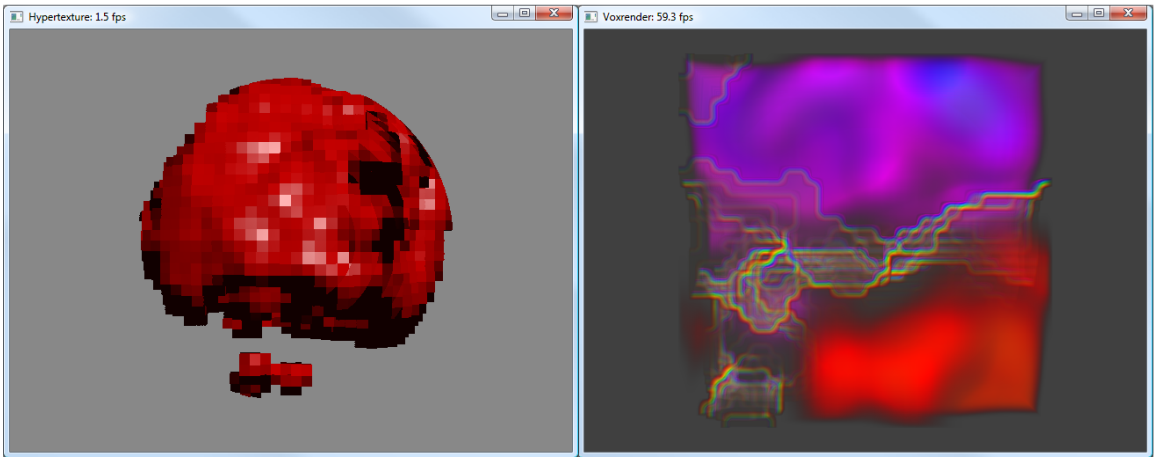


Figure 6.1: Renderings of a 32x32x32 hypertexture from the simple sequential volume renderer (left) and the CUDA volume renderer (right).

	Device	Frames Per Second	Speedup Over Sequential
Sequential	Intel Q9550	1.5	1.0
CUDA	NVIDIA GTX260	59.3	39.53

Table 6.1: A performance comparison in frames per second between the simple sequential volume renderer and the CUDA volume renderer.

Figure 6.1 shows the visual difference between the two rendering systems implemented. The primitive volume renderer to the left renders a hypertexture of the same resolution as the CUDA volume renderer to the right, however the linear texture filtering provided by CUDA produces a significantly more refined image. Table 6.1 shows that the CUDA volume renderer implemented in this thesis is capable of rendering at a rate of approximately 60 frames per second, nearly a 40x increase over the sequential implementation, despite the biased workload in favor of the sequential renderer.

6.2 Register Usage

CUDA has a notion of multiprocessor occupancy, essentially the ratio of work performed per multiprocesser versus the maximum potential work performed per multiprocessor. Register usage is one of the primary factors that determines this occupancy. It is possible to limit the number of registers used per thread by offloading the excess data into local memory, which exhibits significantly worse performance than registers. A balance between the

multiprocessor occupancy and the local memory usage must be experimentally determined to achieve maximum performance. Figure 6.2 and figure 6.3 show the projected occupancy obtained from NVIDIA's CUDA Occupancy Calculator. A trend line of the occupancy is shown given the particular register usage. The actual multiprocessor occupancy is determined by intersecting the number of threads per block (discussed in the next section) along the x axis with the trend line in the graph. These two compute models were chosen because the devices used to collect the performance results seen in this chapter are a 9600m GT (compute model 1.1) and a GTX260 (compute model 1.3). Table 6.2 shows the performance outcome of varying register usage.

Device	**Compute Model**	**Register Usage**	**Frames Per Second**
GeForce 9600m GT	1.1	46	13.2
GeForce 9600m GT	1.1	32	22.0
GeForce 9600m GT	1.1	24	13.2
GeForce GTX260	1.3	44	58.1
GeForce GTX260	1.3	32	59.3
GeForce GTX260	1.3	24	52.8

Table 6.2: A performance comparison in frames per second between devices of two different compute models obtained by varying the register usage of each thread. The number of threads per block remained a constant 64 threads.

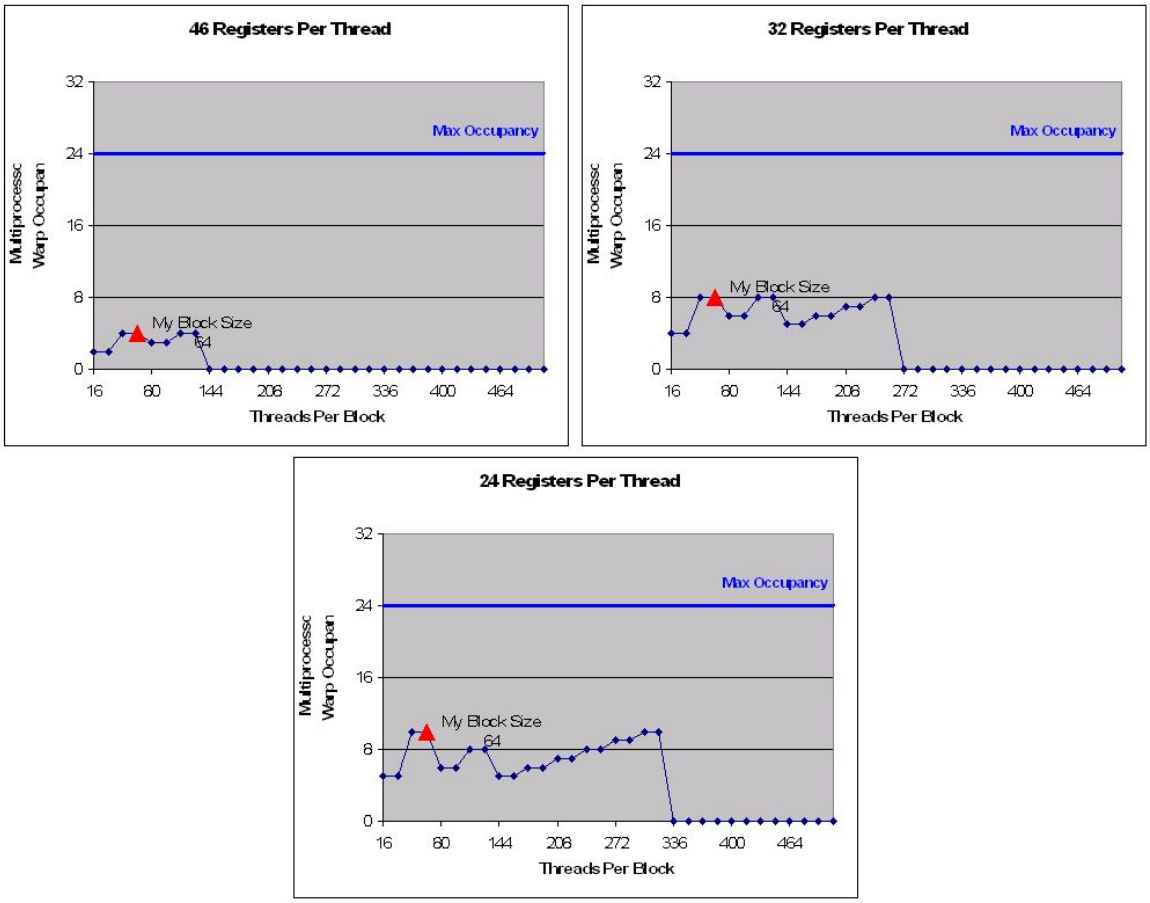


Figure 6.2: Charts produced by the CUDA Occupancy Calculator showing occupancy by varying the register usage for a compute model 1.1 device.

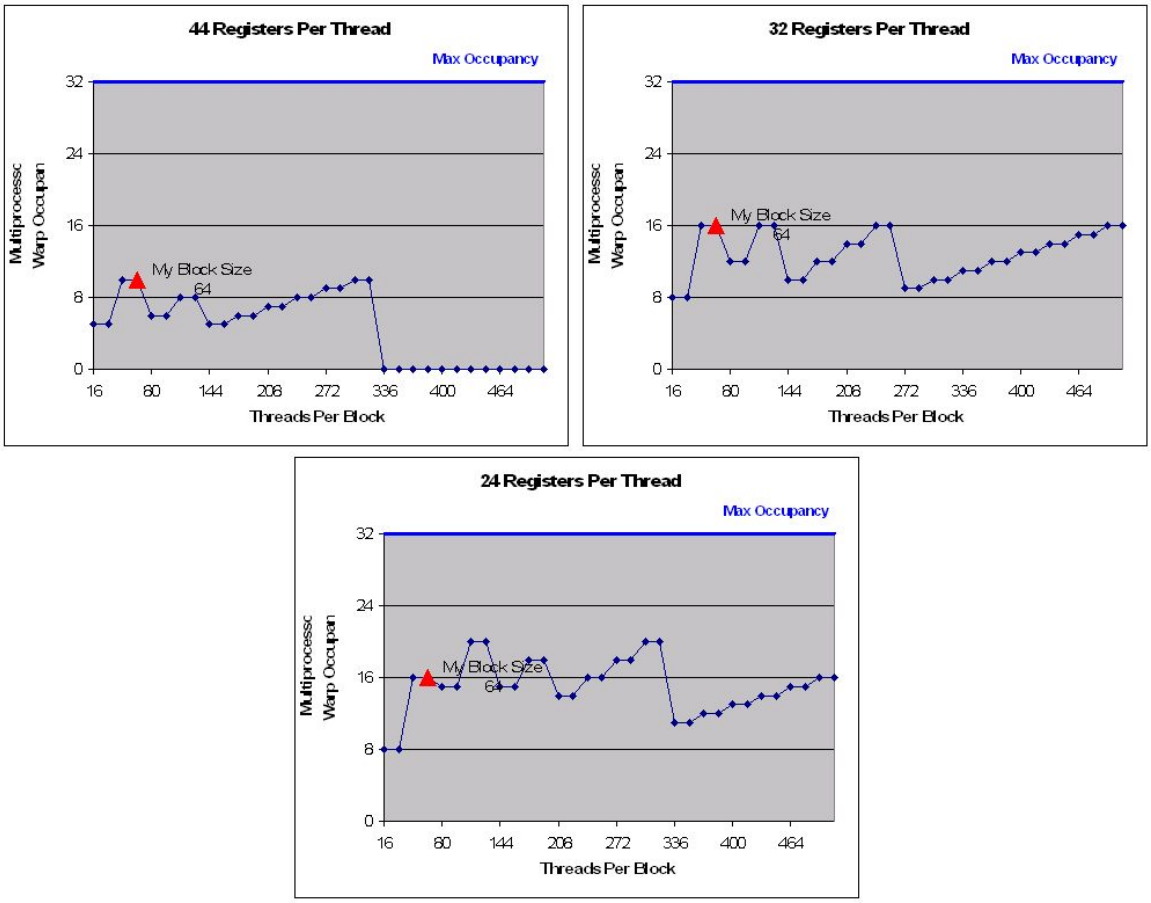


Figure 6.3: Charts produced by the CUDA Occupancy Calculator showing occupancy by varying the register usage for a compute model 1.3 device.

6.3 Threads Per Block

Load balancing is a key performance aspect of parallel systems. Section 5.2 shows a number of methods examined for partitioning work, and concludes that dynamic strip partitioning of threads is the ideal partitioning method. In this case, dynamic partitioning is performed by CUDA, which dynamically allocates a certain number of threads, known as a grain size, to execute on the next available multiprocessor. This grain size is determined by the

number of threads per block. The number of threads per block is also one of the primary factors used to determine multiprocessor occupancy. Because there are a number of factors which affect the performance when varying the number of threads per block, performance results were obtained experimentally.

Device	Compute Model	Threads Per Block	Frames Per Second
GeForce 9600m GT	1.1	32	12.3
GeForce 9600m GT	1.1	64	22.0
GeForce 9600m GT	1.1	96	15.5
GeForce 9600m GT	1.1	128	21.1
GeForce 9600m GT	1.1	160	12.4
GeForce GTX260	1.3	32	51.2
GeForce GTX260	1.3	64	59.8
GeForce GTX260	1.3	96	57.8
GeForce GTX260	1.3	128	59.9
GeForce GTX260	1.3	160	43.7

Table 6.3: A performance comparison in frames per second between devices of two different compute models obtained by varying the threads per block. The register usage remained a constant 32 registers.

Table 6.3 shows the performance trend when varying the number of threads per block. The particular thread counts were chosen as a multiple of warp size (32 threads), however NVIDIA recommends a multiple of 64 threads per block. The table validates this suggestion, showing relatively little difference in performance between 64 and 128 threads per block. Figure 6.2 and figure 6.3 shown in the previous section illustrate a graph of multiprocessor occupancy given 32 registers used per thread. The number

of threads per block shown in table 6.3 may be marked along the x-axis of these figures to reveal the multiprocessor occupancy given a particular number of threads per block.

6.4 Texture Filtering Modes

Texture memory is used to store the data to be visualized in the volume rendering system; for both the capacity of texture memory and its spatial caching policy. In addition to these performance considerations, texture memory also allows filtering of the stored data when indexing into memory. Figure 5.5 shows the visual results of the point and linear filtering methods. The performance characteristics of these two filtering methods have been collected in table 6.4. A visual comparison of these two filtering modes show that linear average filtering provides a significantly greater quality image. The results in this table show that there is virtually no difference in performance between linear and point filtering, making linear filtering an excellent choice for improving visual quality.

Device	**Filter Mode**	**Frames Per Second**
GeForce 9600m GT	Point	21.9
GeForce 9600m GT	Linear	22.0
GeForce GTX260	Point	59.7
GeForce GTX260	Linear	59.8

Table 6.4: A performance comparison in frames per second of point and linear texture filtering modes, across different devices.

6.5 Early Ray Termination

Early ray termination is a common performance enhancement in volume rendering systems, discussed in several academic papers and various resources. Early ray termination reduces the amount of computation necessary by terminating the ray marching algorithm once a particular threshold has been reached. Varying this threshold potentially varies the amount of computations to be performed. Because the threshold is compared to the accumulated density/opacity of a particular volume, it is necessary to examine the affects of early ray termination on a variety of volumes. Table 6.5 and table 6.6 list the volume renderer's performance for two different volumes while varying the termination threshold. It is important to note that early ray termination may only be performed when marching in front to back order. The volume renderer implemented is capable of both front to back and back to front marching; therefore the performance results of back to front marching are also included here. The same set of tests were performed for two different volumes; one exhibiting relatively low density, and another exhibiting high density. The drastic difference in performance confirms that early ray termination is volume dependent. Dense volumes are capable of more than doubling their rendering speed with a reasonably selected early ray termination threshold.

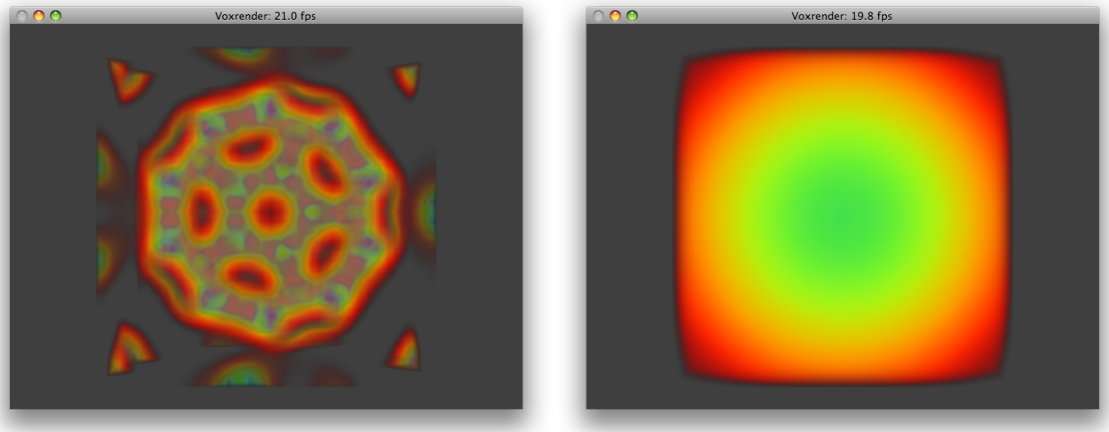


Figure 6.4: Images of the two models used for gathering performance results for early ray termination. The images are of a Buckyball (left) and a Sphere distance field (right).

Device	March Order	Threshold	Frames Per Second
GeForce 9600m GT	B-to-F	NA	22.0
GeForce 9600m GT	F-to-B	NA	22.6
GeForce 9600m GT	F-to-B	0.95	22.4
GeForce 9600m GT	F-to-B	0.9	22.3
GeForce 9600m GT	F-to-B	0.85	22.7
GeForce 9600m GT	F-to-B	0.8	22.7
GeForce GTX260	B-to-F	NA	59.9
GeForce GTX260	F-to-B	NA	59.9
GeForce GTX260	F-to-B	0.95	60.1
GeForce GTX260	F-to-B	0.9	60.4
GeForce GTX260	F-to-B	0.85	61.2
GeForce GTX260	F-to-B	0.8	60.1

Table 6.5: A performance comparison in frames per second of ray march order and early ray termination, across different devices. F-to-B indicates front to back ordering, B-to-F indicates back to front ordering, and a threshold of NA indicates early ray termination was not enabled. The buckyball volume included in the NVIDIA SDK was used when collecting data.

Device	March Order	Threshold	Frames Per Second
GeForce 9600m GT	B-to-F	N/A	19.5
GeForce 9600m GT	F-to-B	N/A	21.2
GeForce 9600m GT	F-to-B	0.95	39.4
GeForce 9600m GT	F-to-B	0.9	44.3
GeForce 9600m GT	F-to-B	0.85	47.7
GeForce 9600m GT	F-to-B	0.8	50.7
GeForce GTX260	B-to-F	N/A	54.6
GeForce GTX260	F-to-B	N/A	59.9
GeForce GTX260	F-to-B	0.95	61.3
GeForce GTX260	F-to-B	0.9	60.8
GeForce GTX260	F-to-B	0.85	61.7
GeForce GTX260	F-to-B	0.8	60.6

Table 6.6: A performance comparison in frames per second of ray march order and early ray termination, across different devices. F-to-B indicates front to back ordering, B-to-F indicates back to front ordering, and a threshold of N/A indicates early ray termination was not enabled. The sphere testing volume [13] was used when collecting data.

6.6 Supersampling

Supersampling is a visual quality technique commonly used in ray casting systems. The implementation of supersampling in this thesis operates by dividing each pixel in a frame into several sub pixels, and casts a ray for each sub pixel, averaging the result of the sub pixels to assign a color. Because the number of rays casted directly represents the amount of work to be performed by the rendering algorithm, it is necessary to take a look at the performance aspects of supersampling. Table 6.7 lists the performance characteristics of adjusting the degree of pixel segmentation, or in other words,

the number of rays casted to produce a complete frame. As to be expected, 16x supersampling generates 16 times the number of rays to be computed, generating on the order of 16 times the amount of work to be performed, and resulting in up to a 16x slowdown.

Device	Supersampling	Frames Per Second
GeForce 9600m GT	1x (none)	21.7
GeForce 9600m GT	4x	6.3
GeForce 9600m GT	16x	1.7
GeForce GTX260	1x (none)	59.9
GeForce GTX260	4x	28.1
GeForce GTX260	16x	9.5

Table 6.7: A performance comparison in frames per second of various degrees of supersampling across different devices.

6.7 Voxel Resolution

The volume renderer implemented in this thesis supports volume data sets with varying voxel resolution. The higher the resolution, the greater the memory requirements to store the volume data. It is also necessary to examine the speed of rendering volumes of varying resolution. Table 6.8 details the framerate achieved when rendering a single volume of varying voxel dimensions. This table shows a linear correlation to the voxel resolution of the rendered volume and the frames per second achieved.

Device	Voxel Resolution	Frames Per Second
GeForce 9600m GT	16x16x16	21.1
GeForce 9600m GT	32x32x32	21.0
GeForce 9600m GT	64x64x64	19.4
GeForce 9600m GT	128x128x128	13.0
GeForce 9600m GT	256x256x256	6.4
GeForce 9600m GT	512x512x512	N/A
GeForce GTX260	16x16x16	59.9
GeForce GTX260	32x32x32	58.1
GeForce GTX260	64x64x64	55.6
GeForce GTX260	128x128x128	48.5
GeForce GTX260	256x256x256	46.5
GeForce GTX260	512x512x512	36.4

Table 6.8: A performance comparison in frames per second of rendering volumes with varying voxel resolutions across different devices. Hypertextures were generated with varying voxel resolutions to collect the data. Note that a 512x512x512 volume could not be rendered using the GeForce 9600m GT because it exceeds the memory capacity of the device.

6.8 Number of Volumes

A unique feature of the volume renderer implemented in this thesis is the ability to render multiple volumes. A varying number of volumes may be arbitrarily placed in a scene, and can be overlapped performing volume registration. There are a number of applications for multiple volume rendering, and it is reasonable to assume some applications would benefit from registered volume rendering, while others would benefit from discrete placement of volumes within a scene. Table 6.9 and table 6.10 show the performance when rendering a varying number of volumes, both in registered,

and discrete placed fashion. Both orientations exhibit a decline in rendering with an increase in the number of volumes, however the frames per second achieved with overlapping volumes is much lower than the frames per second achieved with a side-by-side array of volumes.

Device	Number of Volumes	Frames Per Second
GeForce 9600m GT	1	21.1
GeForce 9600m GT	2	11.6
GeForce 9600m GT	4	5.8
GeForce 9600m GT	8	2.7
GeForce 9600m GT	16	1.4
GeForce 9600m GT	32	0.5
GeForce GTX260	1	59.9
GeForce GTX260	2	43.3
GeForce GTX260	4	29.4
GeForce GTX260	8	16.4
GeForce GTX260	16	7.7
GeForce GTX260	32	3.4

Table 6.9: A performance comparison in frames per second of rendering multiple volumes different devices. The buckyball volume included in the NVIDIA SDK was used when collecting data. Every volume is placed in the center of the scene, overlapping each other.

Device	Number of Volumes	Frames Per Second
GeForce 9600m GT	1	49.8
GeForce 9600m GT	2	40.4
GeForce 9600m GT	4	34.0
GeForce 9600m GT	8	25.4
GeForce 9600m GT	16	13.7
GeForce 9600m GT	32	6.5
GeForce GTX260	1	61.5
GeForce GTX260	2	61.3
GeForce GTX260	4	61.0
GeForce GTX260	8	60.2
GeForce GTX260	16	47.8
GeForce GTX260	32	30.2

Table 6.10: A performance comparison in frames per second of rendering multiple volumes different devices. The buckyball volume included in the NVIDIA SDK was used when collecting data. The volumes were placed from left to right, bottom to top in a 8x4 2D grid configuration which can be seen in figure 5.12.

6.9 Anaglyph Results

Anaglyphs are stereoscopic images which allow a viewer to visualize a 2D image in "3D" by providing each eye a different perspective of the same scene. The volume renderer is capable of producing anaglyphs, rendering the scene once with a red filter, and again with a cyan filter, and accumulating the two. In order to produce these images, two cameras must be used simultaneously to compute two different perspectives of the same scene. The addition of a second camera effectively doubles the number of rays which must be fired, requiring each thread to fire two rays instead of one. Table

6.11 lists the effect the increased workload for each thread has on performance. By rendering an anaglyph, two cameras are placed in a scene with a perspective taken from each camera, doubling the number of rays casted through a scene. The table shows that rendering anaglyphs results in a 25% to 50% reduction in frames per second.

Device	**Render Method**	**Frames Per Second**
GeForce 9600m GT	Normal	6.5
GeForce 9600m GT	Anaglyph	3.4
GeForce GTX260	Normal	41.5
GeForce GTX260	Anaglyph	30.2

Table 6.11: A performance comparison in frames per second of normal vs. anaglyph rendering methods across different devices. The engine volume seen in figure 5.17 was used to collect results.

Chapter 7

Analysis

This chapter individually analyzes the effectiveness and efficiency of various implementation choices, rendering techniques and features, and their performance characteristics discussed in previous chapters.

7.1 CUDA vs. Sequential Performance

CUDA was selected as a platform for implementing the volume renderer because of it's tremendous performance advantages for embarrassingly parallel applications. It is natural to compare the performance of this platform with a baseline sequential system. Although the exact volume rendering system has not been duplicated on a sequential system, a more primitive version of the volume renderer does compile for a Microsoft Windows based system. The primitive sequential volume renderer differs from the CUDA volume renderer in many ways. It does not perform linear interpolation between the data points, allow for interaction with the volume, or utilize a transfer function to assign color based on the sampled density. Most significantly, it does not accumulate opacity while marching through a cube,

instead it simply accumulates density until a threshold is reached, and if the threshold is reached, the gradient is computed and returned. However, this is essentially the same behavior as early ray termination, and more significantly with a threshold far below a typical threshold that would be chosen for early ray termination. Therefore, even in optimal cases, the performance of the primitive sequential volume renderer is significantly faster than a sequential port of the CUDA volume renderer.

Despite the clearly reduced workload of the sequential application, the volume renderer parallelized using CUDA achieves extremely impressive speedup. On a test system using an Intel Core 2 Quad 9550 for the sequential implementation, and an NVIDIA GTX260 (192 core) for the CUDA implementation, a speedup of approximately 40x was achieved. Furthermore, the CUDA implementation was capable of achieving far better graphical quality at real time performance of approximately 60 frames per second. Figure 6.1 shows a sample of the rendered hypertexture that the sequential application produces (left) and the rendered hypertexture using the CUDA implementation (right). These two scenes were used to obtain the performance results seen in table 6.1.

7.2 Price vs. Performance

One major advantage of general-purpose computing on graphics processing units or GPGPU is the price-performance ratio. Figure 1.1 shows a comparison in raw compute power between Intel-based CPUs and NVIDIA GPUs.

This graph, provided by NVIDIA, claims modern NVIDIA GPUs has a GFLOP/s peak approximately 10x greater than Intel CPUs. However, the mid to high range products offered by Intel may be up to twice as expensive as the products offered by NVIDIA. As discussed in section 7.1, the CUDA implementation is capable of achieving a 40x or greater speedup over conventional processors. As of May 2009, the Intel Q9550 used for this performance analysis costs approximately $280, while the NVIDIA GTX260 costs approximately $160. Table 7.1 shows that for this specific volume rendering application, GPUs are capable of delivering 69 times the frames per dollar of CPUs. Although the sequential implementation of the volume renderer is not identical to the CUDA implementation, it is extremely unlikely that an identical implementation would yield increased performance on the sequential system. However, the sequential implementation does not explore other capabilities of modern CPUs, which are trending toward many core solutions capable of parallel processing.

Device	**Approx. Cost**	**Frames Per Second**	**Frames Per Dollar**
Intel Q9550	$280	1.5	0.005357
NVIDIA GTX260	$160	59.3	0.370625

Table 7.1: A price-performance comparison between a primitive sequential volume rendering implementation running on a CPU and the CUDA volume rendering implementation running on a GPU.

7.3 CUDA Object Orientation Difficulties

As of CUDA version 2.2, C++-style object oriented code is not supported. This requires the CUDA source code to be written in the style of standard C. However, for maintenance and future development reasons, the volume renderer was written using typed structures instead of objects, and code relating to that "class" of objects is encapsulated in a file named after that class. However, CUDA does not support the linking of device code from multiple files, and generally requires all the functions used by a kernel to exist in the same file. The source provided with the CUDA SDK sidesteps this issue by directly including other CUDA source code to pull all the necessary device functions into one file. This is of course extremely inelegant and problematic, and duplicate symbol errors are common. Specifically, the CUDA compiler is capable of resolving duplicate device functions, but not duplicate host functions; so host functions should generally be kept in a ".cpp" file (compiled by gcc/g++) while device functions are kept in a ".cu" file (compiled by NVIDIA's nvcc). However, due to the limitations of interacting with texture memory (see section 7.4), host functions are required in the file "Volume.cu" in order to initialize and allocate texture memory. In order to avoid duplicate symbol errors, the code checks if the files which include "Volume.cu" are defined; if they are defined then the host functions are removed by the precompiler. This unfortunately requires knowledge of which files include "Volume.cu", which is not ideal, but unavoidable.

7.4 CUDA Texture Memory Difficulties

Volume data is stored in texture memory, which is initialized at runtime. The volume renderer supports the use of multiple volumes, which can be specified using a script file read during runtime. However, CUDA does not support arrays or pointers to textures, and instead requires distinct variables to be defined for each texture in use. Functions may be invoked using these variables, but pointers or references to these texture variables may not be used. This requires the potential number of volumes and texture variables to be defined at compile time. The lack of support for pointers to textures also means that there cannot be an array of texture memory variables. Therefore, in order to select the correct texture associated with a particular volume in the volume array, each volume structure contains an ID which is used to index into a large switch statement when reading or writing a texture. This is likely one of the most verbose approaches to selecting a variable based on a number, but it is one of the only remaining options given the restrictions surrounding texture memory.

7.5 Occupancy and Partitioning Analysis

Several variables factor into the occupancy of the multiprocessors in CUDA. Threads per block, registers per thread and shared memory per block all determine the multiprocessor occupancy for a particular device, and different compute models result in a different occupancy trend for these variables.

Furthermore, the adjustment of the number of threads per block directly affects the distribution of work among the multiprocessors by changing the grain size. To determine the optimal settings for these values, a series of performance trials were conducted. Shared memory was not used in the volume renderer, so the affect of varying the threads per block and registers per thread were individually examined.

The number of registers used per thread was varied to determine the highest performing setting. The number of registers used is determined by a number of factors regarding the complexity of the computations in the program, however the exact calculation of the necessary registers is unknown. CUDA provides a way to limit the number of registers used per thread by setting the -maxrregcount compiler flag. This reduces the number of registers required by offloading the data from excess registers into "local memory". Local memory has the same performance characteristics as global memory, which are relatively poor. However, the reduction of registers per thread allows greater multiprocessor occupancy, which potentially overcomes the disadvantage of using local memory. The CUDA occupancy calculator was used to graph the potential benifits of altering the register usage, as seen in figure 6.2 and figure 6.3.

A number of renderings were then performed to collect the actual frames per second achieved by varying the register usage. Table 6.2 shows the results of these trials. The table shows the optimal number of registers per thread is 32 registers. Although 32 registers per thread does not result in the

maximum occupancy possible, it is the optimal balance of multiprocessor occupancy and memory performance.

The number of threads per block were then adjusted to determine the optimal settings. Similar to the tradeoffs inherent with register usage, the number of threads per block must balance the benefits of multiprocessor occupancy against a potentially poor grain size for load balancing. The graphs in figure 6.2 and figure 6.3 show the multiprocessor occupancy with the threads per block as the x-axis. These figures help choose the number of threads per block to maximize occupancy. It is important to note that NVIDIA strongly suggests that the number of threads per block is a multiple of 64.

Again, a number of renderings were performed to determine the optimal threads per block. All threads per block values chosen to test with are a multiple of 64. After collecting the data, it was determined that 64 and 128 threads per block both strike an optimal balance between grain size and multiprocessor occupancy. It was determined that both 64 and 128 threads per block achieve nearly identical performance making either value was acceptable, so 64 threads per block was selected for the remainder of the performance characterizations in this document.

7.6 Texture Filtering Analysis

CUDA capable devices have several memory partitions, each with unique performance and capacity characteristics. Texture memory was used to store

a 3D texture of the volume data on the device due to texture memory's large capacity (on the order of hundreds of megabytes on most devices) and caching performance (exploiting 2D spatial locality). Texture memory has the additional benefit of texture filtering, a feature which drastically affects the visual quality of a volume rendering. The filtering modes take affect when indexing into texture memory for reading. Point filtering will return the nearest texel value, resulting in a blocky image. Linear filtering will perform a linear interpolation between the nearest texels (the 8 nearest for a 3D texture), resulting in a much smoother image. Figure 5.5 shows two renderings, with point filtering on the left and linear filtering on the right. It is apparent that linear filtering drastically improves visual quality.

Table 6.4 shows the performance results of the two texture filtering modes across two different test systems. In both cases, the point filtering and linear filtering exhibit nearly identical performance. This makes linear texture filtering an extremely attractive enhancement with virtually no performance drop for significantly higher quality renderings.

7.7 Early Ray Termination Analysis

Early ray termination is a standard performance enhancement for volume rendering systems. It allows the ray marching algorithm to return early if the accumulated density/opacity has reached a set threshold, resulting in fewer computations. Figure 5.8 shows a number of renderings for comparison with early ray termination using various thresholds. When parallelized

using CUDA, early ray termination must overcome the overhead of executing inside a warp. Because the rays are partitioned into threads, and several threads are executed in a batch (known a as a "warp") on a multiprocessor, the multiprocessor will still be consumed until all threads have finished processing. There is no foreseeable way to address this load balancing problem because the execution of threads happens at a hardware level, out of the programmers control.

The the speedup achieved by the early ray termination algorithm is directly dependent on the volume being rendered. If the volume quickly accumulates the threshold value, there will likely be a greater performance benefit. Figure 6.4 shows the two volumes used to collect performance results for early ray termination. The Buckyball (left) is significantly less dense than the sphere distance field (right). Table 6.5 shows the result of rendering the Buckyball with varies threshold densities, while table 6.6 performs the same trials with the spherical distance volume.

While the Buckyball exhibits little to no speedup with the use of early ray termination, the significantly more dense spherical distance volume achieves as great as a 2.6x speedup in rendering time for the examined thresholds. This demonstrates that early ray termination is a significant performance enhancement when visualizing the appropriate volumes.

7.8 Supersampling Analysis

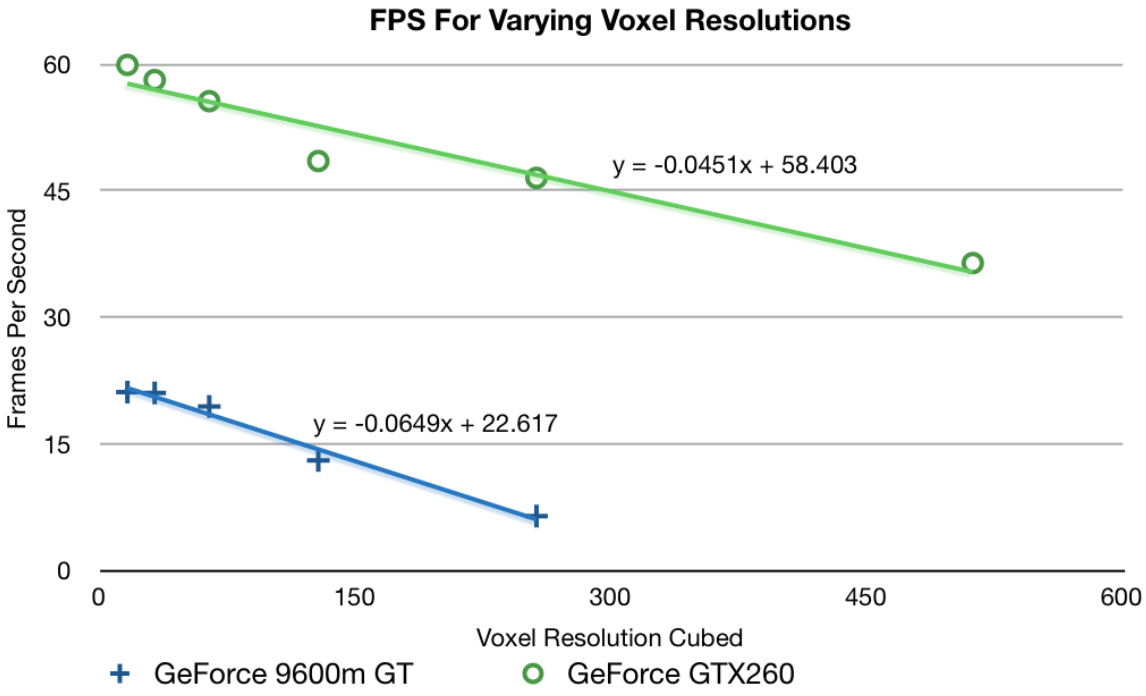
Supersampling was implemented in the volume renderer by simply dividing a pixel into equal parts with rays cast for each sub pixel, and averaging the resulting colors from each ray for the final pixel value. However, this enhancement did not provide as noticeable a quality improvement as it does with other applications such as ray tracing. This is because of the nature of the content in the scene. Ray tracing has a distinctive, crisp, super-real look that has noticeable jaggies at contrast boundaries. However, volume renderings have a significant amount of opacity, and commonly the voxel resolution of the objects rendered is significantly less than the pixel resolution of the screen used to view them, making the jaggies far less noticeable in comparison. Volume renderings which incorporate an illumination model such as Phong illumination, which results in areas of sharp contrast, may achieve a more noticeable improvement to the visual quality of the image. Supersampling also drastically increases the amount of work to be done. It is traditionally reserved for high quality still images, and not for interactive applications. Figure 5.9 shows the effects of an image without supersampling (left) and with 16x supersampling (right) for both point filter mode (top) and linear filter mode (bottom).

As can be seen in table 6.7, supersampling requires a significant amount of additional computation. By performing 16x supersampling, the number of primary rays casted (and therefore number of threads to be computed)

increases by a factor of sixteen. It is to be expected that the rendering of a frame of a 16x supersampled image may take as much as sixteen times as long as the same frame without supersampling. The performance results show a performance drop of between roughly 6x and 13x. It is clear that the performance disadvantage likely outweighs the improvement in visual quality (if any) when attempting interactive frame rates, especially when compared to the performance/quality ratio of linear filtering.

7.9 Analysis of Volume Characteristics

The volumes used by the volume renderer are defined by voxel data, which typically represent density. Volumes have a voxel resolution similar to the pixel resolution of a common television screen or computer monitor, but in three dimensions. Results were collected on the frames per second achieved when rendering volumes with increasingly large voxel resolutions. Hypertextures such as those seen in figure 5.15 were used to collect the performance results. Hypertextures are particularly useful for this application as they can be used to generate the same general volume with increasing resolution. Table 6.8 shows the performance of the volume renderer as the voxel dimensions of the volume increase. Note that GeForce 9600m GT has no data collected for a 512x512x512 volume, because the memory required to store the volume (134,217,728 bytes or 128 megabytes) exceeded the capacity of the texture memory. The results have been graphed in figure 7.1 for analysis.



	16	32	64	128	256	512
GeForce 9600m GT	21.1	21.0	19.4	13.0	6.4	
GeForce GTX260	59.9	58.1	55.6	48.5	46.5	36.4

Figure 7.1: A graph of the results showing how performance scales with the voxel resolution of the rendered volume.

Figure 7.1 shows that the voxel resolution of a volume has significant bearing on the frame rate of the volume renderer. It is shown that the frame rate diminishes linearly with respect to the volume's resolution, though the exact affect the resolution has on performance is unique to the hardware device used.

One of the unique features of the volume renderer implemented in this work is the ability to simultaneously render multiple volumes. These volumes can be arbitrarily positioned within the world, and may potentially

overlap. Overlapping volumes essentially performs a kind of rigid volume registration. This can be particularly useful for medical imaging applications. For example, a doctor may wish to view a combined representation of an MRI and a CT scan of a patient. This is made possible by the volume renderer implemented in this thesis. Figure 5.13 shows an MRI (left) and CT (right) scan of a monkey's head positioned side-by-side. Figure 5.14 shows the result of positioning these two volumes in exactly the same space. By observing the registered image, additional information may be identified from the scans. In contrast to medical imaging applications, simulations and video games desire a number of models arbitrarily positioned throughout a scene. Figure 5.11 shows six different volumes rendered simultaneously side-by-side.

Due to a limitation in CUDA regarding the usage of texture memory, the maximum number of volumes stored in texture memory must be specified at compile time. For the purposes of this application, a limit of 32 maximum volumes was imposed. This is similar to the restriction of 8 maximum light sources in openGL. Figure 5.12 shows the maximum number of volumes displayed simultaneously.

Several scenes were generated to characterize the performance of the volume renderer as the number of volume increase. Table 6.9 shows the performance with a varying number of identical volumes existing in the same space. Table 6.10 shows the performance with a varying number of

volumes placed side-by-side within the scene. These two methods of positioning volumes were chosen to reflect the likely applications of the volume renderer. Both positioning methods show a significant performance drop off as the number of rendered volumes increases. However, an increase in overlapping volumes decreases performance at a far greater rate than non-overlapping volumes. This is primarily due to the number of threads that must be dedicated to each volume. In the case of the overlapping volumes, each volume consumes approximately 60% of the frame. However, each non-overlapping volume consumes only approximately 2% of the frame. This requires far fewer threads dedicated to ray marching. Furthermore, the positioning of the non-overlapping volumes naturally balances the computational load, as opposed to the addition of overlapping volumes which only adds computational load to threads already consumed with work. It can be concluded from these results that applications such as medical imaging may be limited to fewer models to maintain acceptable interactive performance. Simulations and video game applications have a much higher threshold in regards to the number of displayed volumes before the performance becomes unacceptable.

7.10 Anaglyph Analysis

Anaglyphs are a common way to view 2D images as “3D”, typically requiring 3D glasses with a red and cyan lenses to properly view the image. The

images are constructed by combining images from two different perspectives with a red shift on the image from one perspective and a cyan shift on the image from the other. Ideally, when someone looks through the 3D glasses, the eye looking through red film can only perceive the perspective taken with the blue shift, and vice-versa for the other eye. The visual cortex interprets this difference in perspective and gives the illusion of depth. Anaglyphs have appeared several times in recent films, and stereoscopy in general has been in development for simulation, virtual reality and video game applications.

Another novel feature in the volume renderer implemented in this work is ability to interactively generate anaglyph renderings of volumes. Figure 5.17 shows an engine block rendered as an anaglyph. The anaglyph renderings produced by the volume renderer are arguably somewhat difficult to view as stereoscopic images. There are multiple forms of depth being conveyed using the volume renderer. Volumes inherently have a sense of depth as an observer can see through the volume. The ability to interact with the volume by animating the volume in 3D space further introduces a sense of depth. These factors coupled with the anaglyph renderings may initially be confusing for an observer.

Table 6.11 shows the performance difference between traditional volume rendering and anaglyph volume rendering. Rendering an anaglyph may result in up to a 50% decrease in performance. This is to be expected because of the nature of the anaglyph rendering algorithm. The scene is essentially

rendered twice from the two camera perspectives, with only the red channel retained from one rendering and the blue and green channels retained from the other, and the resulting renderings are accumulated for a final result.

Despite the initial difficulty in visualizing such images, the anaglyph renderings form a proof of concept for potential future applications, as well as benefits to existing applications such as the 3D visualization of medical images, fluids, CAD drawings and more. Furthermore, although there is a significant decrease in performance when rendering anaglyphs, the GeForce GTX260 is still capable of rendering an anaglyph at a rate of 30 frames per second, an acceptable interactive frame rate.

Chapter 8

Thoughts for Investigation and Future Work

This thesis investigates a number of implementation choices as well as rendering techniques, features, applications. However, there are many areas left for further exploration and optimization. This chapter discusses several aspects of the volume rendering system developed in this thesis, and volume rendering concepts in general which many benefit from further research.

8.1 Performance

Careful attention has been payed to the register usage of the volume renderer. It has been shown that the registers used per thread may be altered to balance multiprocessor occupancy and local memory usage. However, despite numerous attempts, no consistent method of decreasing the number of required registers was found. Decreasing the required number of registers would allow for increased occupancy, while decreasing local memory usage.

Branching is a concern when designing CUDA applications. If threads

within a warp branch, the warp will be "split," and the threads in the warp will follow different execution paths. Ideally, all the threads in a warp should have the same execution path. Branching is inherent in the ray casting algorithm, introduced by the bounding box hit test. If a bounding box is hit the ray marching algorithm is performed, if the bounding box is not hit the function simply returns. This warrants investigation into techniques to prevent threads within a warp from branching to improve performance.

Early ray termination has been shown to provide performance increases of up to 2.6x. While the implementation of early ray termination works for individual volumes, it does not allow for multiple volumes along the same ray to contribute to the accumulated density/opacity. If an adjustment were made to perform early ray termination across multiple volumes, a further performance benefit would be seen by essentially occluding volumes which do not contribute to a pixels color.

Empty space skipping is another common performance enhancement for volume rendering. Space skipping within a volume requires a volume to be segmented in some way to identify areas of empty space. One of the most common ways to perform this segmentation is octree space subdivision [8]. Octree subdivision may be performed in an initialization phase of the volume renderer, and the results may be stored in texture memory.

8.2 Image Quality

One common visualization feature is an illumination model. This model requires lights and the computation of a gradient of a volume at any given point to obtain a normal vector used when reflecting light. The volume rendering system implemented in this thesis uses a transfer function to assign color and opacity to voxel data of varying density. However, lights are not utilized to perform any sort of shading on the volumes. A significant amount of code exists in the implementation which supports Phong illumination and placing lights within a scene, but the code is not currently utilized. The only missing aspect is the computation of the gradient at a given point within a volume, which necessary to determine the orientation of that point in the volume to a light source.

Supersampling is used to sample multiple rays per pixel, at the expense of sampling multiple times [1]. One possibility is implementing adaptive supersampling for certain areas of "focus," "importance," or possibly just objects closer the the viewer via supersampling, but as models get further away, decreasing the sampling rate to even out the number of samples performed in a scene. This could be implemented by supersampling objects in focus to achieve great detail, and sampling objects out of focus "normally." There could also be a sampling falloff function based on the distance from the viewplane. Particularly, there could be an exponential or linear falloff starting at the viewplane with a sample distance at a unit length, such that the

sampling rate would be measured once the unit length has been traversed.

A step size is utilized in the ray marching algorithm to determine the distance of each step when sampling a volume. Choosing an appropriate step size for a volume is an important measure for reducing aliasing in the rendered image. The step ideal step size is dependent on both the voxel resolution of a volume and the size in world coordinates of that volume. Adaptive selection of step sizes may be implemented to chose the optimal step size for each volume rendered within a scene.

8.3 User Interaction

A camera system was constructed to interactively view the volumes in a scene. The camera controls are styled after video game controls, specifically known as "first person shooter" controls. However, the camera controls are somewhat buggy; it is possible to fully rotate around the x-axis but not the y-axis. Resolving these bugs would be the first step to a better user interface. However, many advancements have been made other forms of user input, such as multi-touch technologies and gesture recognition. The combination of these technologies with volume rendering would provide a unique interactive experience.

The volume renderer makes use of a transfer function stored in texture memory to assign a color to a density. However, this transfer function is globally assigned and cannot be changed on a per-volume basis. There is no inherent limitation in the CUDA platform regarding this capability, it

is simply that the resulting code, much like the handling of multiple textures storing volume data, is very inelegant. A terse and elegant solution to this problem would allow for greater customization of the visualization of multiple volumes. Providing a user with the ability to alter the transfer function of individual volumes in real time may assist in the visualization of the volume data, and allow for more information to be obtained from the data set.

Interaction with volume data sets is an important aspect of volume rendering. The real time interaction of volume data sets developed in this thesis opens the way for other user interface techniques for manipulating volumes. Many human interface technologies are in constant development, such as multi-touch technologies and gesture recognition. Combining the performance and interactive capabilities of the volume renderer implemented in this work with these various human interface technologies would provide a new interactive experience that may benefit the doctors scientists and engineers that wish to explore volume data sets.

8.4 Applications

A highly sought after application in the medical field is the real time deformable registration of volume data. This potentially allows different scans of a patient, taken at different times with the patient in different orientations, to be deformed into a similar orientation allowing the volumes to be overlapped and rendered together. The volume renderer implemented

in this work is capable of multiple volume rendering and rigid registration. However, it is possible to implement additional deformation algorithms such as elasticity, *etc.*in the volume renderer to achieve deformable registration. The implementation of real-time deformable registration is another excellent thesis topic.

Volume data sets are not typically animated in the same way as polygonal meshes. Volume data is typically static data, while polygonal models such as those found in video games and simulations may have rigging to perform structured motion of portions of the same model. If a volume rendering system were to be used in a simulation or video game engine, animation of the volume models would be an excellent feature. The transformations performed in deformable registration such as elasticity may facilitate this animation.

Constructive solid geometry is a technique often used when modeling solid objects. It allows the creation of new geometry by intersecting objects of a certain geometry and rendering a boolean union, difference, or intersection of the intersecting objects. When multiple volumes intersect or overlap each other in the volume renderer, the resulting color must be determined by a sort of accumulation of the composited densities. However, in the same vein as constructive solid geometry, it is possible to perform boolean union, difference, or intersection operations on a number of intersecting volumes by altering their densities. This has a number of potential applications, such

as removing unnecessary information from a large volume data set by applying a type of mask and rendering the boolean intersection or difference of the rendered volume and the mask.

Mipmapping is a common texturing technique used in computer graphics. Several resolutions of the same texture known as mipmaps may be stored and accessed depending on the required resolution of the texture. Textures that appear further away require less resolution than textures that appear close to the viewer. As an extension of current techniques to deal with bitmap texturing in modern 3D graphics applications, it is possible to mipmap a 3D Texture [19], and by extension, the volumes data sets used in the volume renderer. Mipmapping a 3D Texture should allow for the same types of performance enhancements exhibited by the technique in two dimensions. When using octrees, mipmapped 3D texturing techniques can be applied in conjunction with more common empty space skipping techniques to enhance performance with very minimal overhead [8]. Furthermore, it may be possible to compress the representations of these volumes [3].

Hypertextures, discussed by Ken Perlin are one way of applying various volumetric effects to a ray casted surface [11]. Hypertextures are often implemented using volume ray casting, and are capable of representing objects extremely difficult or impossible to represent using only surfaces, such as fire, gas, or furry objects. Similar to bump mapping techniques used in more conventional computer graphics with polygon meshes it is possible to apply hypertextures on the surface of an object to form a highly detailed surface

effect. This effect could be applied to volumetric models, as well as ray casted polygonal meshes commonly used in computer graphics. Performing volume rendering as well as polygonal rasterization in the same scene would require some sort of hybrid rendering engine. Both the hybrid rendering system and Perlin noise function used to generate hypertextures may be candidates for implementation using CUDA. Some volumetric effects are discussed in [4].

Chapter 9

Conclusion

This thesis has explored the benefits of general-purpose computing on graphics processing units by implementing and analyzing a volume rendering system. The renderer has been made accessible to both Mac OS X and Microsoft Windows operating systems, with the ability to achieve interactive frame rates on both desktop and laptop machines. The volume renderer has exhibited a tremendous performance increase over alternative sequential implementations, allowing for the real time interaction of volume data. In comparison to a primitive sequential volume renderer, the CUDA volume renderer achieved approximately a 40x speedup. When comparing the price-performance ratio of a mid to high range traditional CPU to a mid to high range GPU performing general purpose computations for the purpose of volume rendering, the GPU was capable of delivering up to 70x the frames per second per dollar of the CPU.

The ray casting algorithm has been successfully exploited using CUDA, and a number of performance and quality optimizations have been implemented including early ray termination, texture filtering and supersampling. Early ray termination has shown a performance benefit of up to 2.6x speedup

on certain systems. Despite the image quality to performance ratio achieved by supersampling, linear texture filtering has been shown to tremendously enhance the quality of volume renderings at no discernible reduction in frame rate.

A number of applications of volume rendering have been implemented and accelerated to allow the real time interaction of volume data sets. A framework was constructed to allow the interaction of multiple arbitrarily placed volumes within a scene, providing a proof of concept for volume based simulations and video games. The volumes may be overlapped to perform a type of rigid registration with interactive frame rates useful in medical imaging and engineering fields. A unique way of viewing volumes has been introduced using stereoscopic anaglyphs to provide a depth effect also while maintaining interactive frame rates.

This thesis has demonstrated the performance benefits that general-purpose computing on graphics processing units can bring to parallelizable applications such as volume rendering. The framework developed using this thesis is easily extensible to allow future efforts the ability to investigate new performance and quality enhancements, as well as facilitating more radical approaches to volume rendering applications and interaction. With the concepts and applications discussed in this thesis, and as GPGPU technology matures, hopefully new and exciting approaches to volume rendering will emerge.

Bibliography

[1] Franklin C. Crow. The aliasing problem in computer-generated shaded images. *Commun. ACM*, 20(11):799–805, 1977.

[2] Zvi Devir. Introduction to volume rendering. Course Presentation at Technion - Israel Institute of Technology.

[3] N. Fout and Kwan-Liu Ma. Transform coding for hardware-accelerated volume rendering. *Visualization and Computer Graphics, IEEE Transactions on*, 13(6):1600–1607, Nov.-Dec. 2007.

[4] Milan Ikits, Joe Kniss, Aaron Lefohn, and Charles Hansen. Volume rendering techniques. In *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, chapter 39, pages 667–692. Pearson Higher Education, 2004.

[5] Jusub Kim. *Efficient rendering of large 3-D and 4-D scalar fields*. PhD thesis, University of Maryland, College Park, May 2008.

[6] J. Kruger and R. Westermann. Acceleration techniques for gpu-based volume rendering. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 38, Washington, DC, USA, 2003. IEEE Computer Society.

[7] Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *SIGGRAPH*

'94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques, pages 451–458, New York, NY, USA, 1994. ACM.

[8] Sylvain Lefebvre, Samuel Hornus, and Fabrice Neyret. Octree textures on the gpu. In *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*, chapter 37, pages 595–613. Addison-Wesley Professional, 2005.

[9] NVIDIA. *NVIDIA CUDA Programming Guide*, 1.0 edition, June 2007. Available: `http://www.nvidia.com/cuda`.

[10] NVIDIA. *NVIDIA CUDA Programming Guide*, 2.0 edition, June 2008. Available: `http://www.nvidia.com/cuda`.

[11] K. Perlin and E. M. Hoffert. Hypertexture. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 253–262, New York, NY, USA, 1989. ACM.

[12] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 268, New York, NY, USA, 2005. ACM.

[13] Stefan Roettger. The volume library. `http://www9.informatik.uni-erlangen.de/External/vollib/`.

[14] Stefan Roettger, Stefan Guthe, Daniel Weiskopf, Thomas Ertl, and Wolfgang Strasser. Smart hardware-accelerated volume rendering. In *VISSYM '03: Proceedings of the symposium on Data visualisation*

2003, pages 231–238, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[15] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. A simple and flexible volume rendering framework for graphics-hardware-based raycasting. *Volume Graphics, 2005. Fourth International Workshop on*, pages 187–241, June 2005.

[16] Kevin Suffern. *Ray Tracing from the Ground Up*. A. K. Peters, Ltd., Natick, MA, USA, 2007.

[17] Wikipedia. Volume ray casting — wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Volume_ray_casting`, 2008.

[18] Wikipedia. Volume rendering — wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Volume_rendering`, 2009.

[19] Lance Williams. Pyramidal parametrics. In *SIGGRAPH '83: Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, pages 1–11, New York, NY, USA, 1983. ACM.

Appendix A

Compiling the Volume Renderer

The volume renderer has been developed under Mac OS X and Microsoft Windows Vista. A makefile has been maintained for compilation under OS X, and a Visual Studio 2005 project has been maintained and verified to work under Windows XP 32 bit and Windows Vista 64 bit. The various projects reference necessary libraries and other common data within the main volume renderer directory, so the directory may be located anywhere on the desired system providing the subdirectories are not altered. OpenGL, GLUT and CUDA are required for compilation. The necessary libraries should be included in the CUDA toolkit and SDK, available from the following source:

`http://www.nvidia.com/object/cuda_get.html`

Appendix B

Using the Volume Renderer

The volume renderer has a number of settings that may be set before compilation. The "Scene.h" file in the Scene directory contains several definitions used throughout the project. This file contains settings for the image resolution, texture filtering mode, early ray termination threshold, degree of supersampling hypertexture random seed and bounds checking, stereoscopic anaglyph enable, and various debug options. When changing these settings, make sure to rebuild the entire project when compiling.

Part of the volume rendering framework goals were to create a kind of script that defines the scene to be rendered. "World.cpp" in the World directory contains a single function called "build_world."

This function is responsible for setting the background color of the scene, the camera position and orientation within the scene, the position and color of lighting (not currently used), and several aspects of the volumes to be rendered. The function specifies the number of volumes to be displayed, which volume data to load for each volume, their coordinates in world space, the step size of each volume and a density modifier for each volume.

Once compiled and running, a user can interact with the volume using

keyboard and mouse controls. These controls and their intended actions are described below. Note that the camera exhibits some unexpected behavior under special cases; the following controls describe the intended operation of the camera.

Keyboard Controls:

'w'	Move forward along Z axis
's'	Move backwards along Z axis
'a'	Move left along X axis
'd'	Move right along X axis
' '	Move up along Y axis
'c'	Move down along Y axis
'+' or '='	Move forward along Z axis
'-' or '_'	Move backwards along Z axis
up	Tilt the camera up
down	Tilt the camera down
left	Tilt the camera left
right	Tilt the camera right

Mouse Controls:

left click	Click and drag to rotate the camera
right click	Click and drag to translate the camera
middle click	Click and drag up and down to zoom

Appendix C

Structure of Included CD

A data CD has been included with this thesis document containing the source code and related files for compilation and execution of the volume renderer developed in this thesis, as well as digital copies of various documents pertaining to the thesis and other various resources. The following lists each directory on the CD, with a description of the directory's contents.

bin – Contains the executable binaries for Mac OS X (darwin), Windows 32 bit (win32) and Windows 64 bit (win64) operating systems.

common – Contains the common files used to compile the volume renderer. The various projects contain relative path references to this folder.

data – Contains a number of volume data sets compatable with the volume renderer, as well as a readme listing the necessary metadata for each volume and a citation of where the volume was obtained. Appendix D contains the information from this readme.

doc – Contains digital copies of this thesis document and the related images used in it's creation, a digital copy of the approved thesis proposal, and a digital copy and video of the defense presntation.

lib – Contains libraries necessary for the compilation of the volume renderer on Mac OS X.

misc – Contains miscelaneous scripts and utilities used to generate segments of code and convert volume data sets for use in the volume renderer.

obj – Contains the intermediate object files produced in the compilation of the volume renderer on Mac OS X and Windows systems.

projects – Contains the project files for compiling the volume renderer under Mac OS X and Windows. The Make directory is used for compilation under Mac OS X using the make utillity, and the VS2005 is used for compilation under Windows using Visual Studio 2005.

src – Contains the source code for the volume renderer. This directory is further divided into self-explanatory sub-directories containing the relative source code for that particular component of the volume renderer.

Appendix D

Listing of Sample Volumes

The volumes used in this thesis have been collected by a variety of sources. The sources, as well as the necessary metadata about each volume are documented here. All edge ratios are 1:1:1 unless otherwise noted.

```
Bonsai2-lO
x-512 y-512 z-189    0.402344 : 0.402344 : 1
Linear Quantized CT Scan with contrast dye of a bonsai tree
http://www9.informatik.uni-erlangen.de/External/vollib/

Box
x-64 y-64 z-64
A solid box
http://www9.informatik.uni-erlangen.de/External/vollib/

Bucky
x-32 y-32 z-32
A Buckyball provided in the NVIDIA CUDA SDK
http://nvidia.com/cuda

C60
x-64 y-64 z-64
A simulated Buckyball
http://idav.ucdavis.edu/~okreylos/PhDStudies/Spring2000/ECS277/index.html

C60Large
x-128 y-128 z-128
A simulated Buckyball
```

http://idav.ucdavis.edu/~okreylos/PhDStudies/Spring2000/ECS277/index.html

DTI
x-128 y-128 z-58
DTI scan of a brain
http://www9.informatik.uni-erlangen.de/External/vollib/

Engine
x-256 y-256 z-256
Two cylinders of an engine block
http://www9.informatik.uni-erlangen.de/External/vollib/

Frog
x-256 y-256 z-44 0.5 : 0.5 : 1
This is the second frog used in the Whole Frog Project
http://www9.informatik.uni-erlangen.de/External/vollib/

Fuel
x-64 y-64 z-64
Simulation of fuel injection into a combustion chamber
http://www9.informatik.uni-erlangen.de/External/vollib/

Monkey-CT
x-256 y-256 z-62 1 : 1 : 3
CT scan of a monkey head
http://www9.informatik.uni-erlangen.de/External/vollib/

Monkey-MRI-T2
x-256 y-256 z-62 1 : 1 : 3
MRI scan of a monkey head
http://www9.informatik.uni-erlangen.de/External/vollib/

Neghip
x-64 y-64 z-64
Probability distribution of electrons in a high potential protein molecule
http://www9.informatik.uni-erlangen.de/External/vollib/

Orange
x-256 y-256 z-64 0.390625 : 0.390625 : 1
MRI scan of an orange
http://www9.informatik.uni-erlangen.de/External/vollib/

```
Sphere
x-64 y-64 z-64
Spherical distance volume
http://www9.informatik.uni-erlangen.de/External/vollib/

Spheres
x-128 y-128 z-128
Simulated testing spheres
http://www9.informatik.uni-erlangen.de/External/vollib/

Tomato
x-256 y-256 z-64    0.390625 : 0.390625 : 1
MRI scan of a tomato
http://www9.informatik.uni-erlangen.de/External/vollib/
```