4-30-2010

Full CUDA Implementation Of GPGPU Recursive Ray-Tracing

Andrew D. Britton
Purdue University - Main Campus, andrew@andrewbritton.com

Britton, Andrew D., "Full CUDA Implementation Of GPGPU Recursive Ray-Tracing" (2010). *College of Technology Masters Theses.* Paper 24.
http://docs.lib.purdue.edu/techmasters/24

Graduate School ETD Form 9
(Revised 12/07)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Andrew Duncan Britton

Entitled FULL CUDA IMPLEMENTATION OF GPGPU RECURSIVE RAY-TRACING

For the degree of Master of Science

Is approved by the final examining committee:

Dr. Bedrich Benes
Chair

Dr. James Mohler

Eliot Mack

To the best of my knowledge and as understood by the student in the *Research Integrity and Copyright Disclaimer (Graduate School Form 20)*, this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): Dr. Bedrich Benes

Approved by: Dr. James Mohler April 21, 2010
Head of the Graduate Program Date

Graduate School Form 20
(Revised 1/10)

PURDUE UNIVERSITY
GRADUATE SCHOOL

Research Integrity and Copyright Disclaimer

Title of Thesis/Dissertation:
FULL CUDA IMPLEMENTATION OF GPGPU RECURSIVE RAY-TRACING

For the degree of Master of Science

I certify that in the preparation of this thesis, I have observed the provisions of *Purdue University Teaching, Research, and Outreach Policy on Research Misconduct (VIII.3.1)*, October 1, 2008.*

Further, I certify that this work is free of plagiarism and all materials appearing in this thesis/dissertation have been properly quoted and attributed.

I certify that all copyrighted material incorporated into this thesis/dissertation is in compliance with the United States' copyright law and that I have received written permission from the copyright owners for my use of their work, which is beyond the scope of the law. I agree to indemnify and save harmless Purdue University from any and all claims that may be asserted or that may arise from any copyright violation.

Andrew Duncan Britton
Printed Name and Signature of Candidate

April 29, 2010
Date (month/day/year)

*Located at http://www.purdue.edu/policies/pages/teach_res_outreach/viii_3_1.html

FULL CUDA IMPLEMENTATION OF GPGPU RECURSIVE RAY-TRACING

A Thesis

Submitted to the Faculty

of

Purdue University

by

Andrew D. Britton

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

May 2010

Purdue University

West Lafayette, Indiana

This work is dedicated to all hearts and minds that point in multiple directions concurrently, to everyone who, when they look at the blank canvas of future moments, sees possibility over danger, and to anyone whose eyes are bigger than their stomachs when planning the future.

Our mettle is quickened in trials by fire and in leaps of faith beyond our own comprehension. When fatigue has removed our last defenses, who we choose to be in that moment defines who we are ever after.

This, as it has always been, is just the beginning.

ACKNOWLEDGMENTS

This research would not be possible with the specific aid and support of my committee members: Dr. Bedrich Benes (chair), Eliot Mack and Dr. James Mohler. For their pioneering work in the field of the computer graphics I am grateful for the efforts of: Jim Blinn, Pierre Bezier, ACM, Ed Catmull, OpenGL ARB, Mark J. Kilgard, Bui Tuong Phong, Ken Perlin, Peter Shirley, Arthur Appel, Turner Whitted. For their dedication to art I am grateful to the efforts of: Leon Battista Alberti, John Singer Sargent, Krome Barratt, Edgar Degas, Burne Hogarth, Piet Mondrian, Vincent Van Gogh, Alphonse Mucha, and Albrecht Dürer. As I am a pebble standing on the backs of giants in this amazing field, I wish to thank the educators who opened roads for me that would otherwise be all but obscured without their presence: Donna Cox and George Francis. I am very thankful to my friends who gave me some of their time and computer clock cycles in the middle of their own thesis crunch: Jason Lambert and Ondrej Stava. Without them I wouldn't have so much rendering timing data. My double thanks to them for also being my unofficial programming tutors.

My special thanks goes to Norman Brzycki, Aaron Lake, Rebecca Lank and Colleen Maguire.

PREFACE

“To make clear my exposition in writing this brief commentary on painting, I will take first from the mathematicians those things with which my subject is concerned.

In all this discussion, I beg you to consider me not as a mathematician but as a painter writing of these things. Mathematicians measure with their minds alone the forms of things separated from all matter. Since we wish the object to be seen, we will use a more sensate wisdom. We will consider our aim accomplished if the reader can understand in any way this admittedly difficult subject… Therefore, I beg that my words be interpreted solely as those of a painter” (Alberti, L. B. 1435).

TABLE OF CONTENTS

Page

APPENDICES

LIST OF TABLES

Appendix Tables

LIST OF FIGURES

Appendix Figures Page

LIST OF ALGORITHMS

GLOSSARY

Rendering – “The main function of the [graphics rendering] pipeline is to generate, or render, a two-dimensional image, given a virtual camera, three-dimensional objects, light sources, shading equations, textures, and more” (Akenine-Möller & Haines, 2008, pg 11).

Virtual Pinhole Projection Camera – is a construct through which scene data is recorded to a two-dimensional pixel plane. *Virtual* describes the camera as existing and operating in computer code. *Pinhole Projection* defines that the “incoming light” passes through a single point; this is the camera’s position. (Shirley & Morley, 2003, pg 63-67).

Ray – a geometric construct defined by a vector and a point. The vector originates from the point. Both vector and point are defined using three or four-dimensional coordinate space.

Ray Casting - The first *ray* originates from the camera center and is cast in the direction from the camera through the pixel plane. *Rays* are calculated for intersection with geometric objects. This is how the renderer calculates if and where and object is “seen” in the final image.

Ray Tracing – To calculate reflections and refractions a ray-trace renderer creates new rays that bounce into the scene or through objects to calculate new colors per pixel.

BRDF – the *Bi-directional Reflectance Distribution Function*. The BRDF describes the quality of reflected energy, given a set of input energy given over a set of various incident angles, off a measured surface. Every surface has its own BRDF. The BRDF describes the sizes and qualities of reflected light over a surface. This is seen as the soft (diffuse) and crisp (specular) highlights on surfaces (Dutré, Bala & Bekaert, 2006).

Ambient Lighting Component – is a description and approximation of the light that fills a scene. It describes the amount of background light in a scene. It is a color additive component to shaders. This component is described by color.

Diffuse Lighting Component – describes light reflected off of rough surfaces. The presence of the diffuse lighting component on a shader allows for the presence of self-shading where there is a light and dark side of an object based on the object’s surface direction. Where the surface points more toward the light the surface will receive more light; the converse is true as well. This component is described by color.

Specular Lighting Component – is a shading approximation used to describe the highlights on the surface of an object. The specular component is useful in showing where a surface fits on the continuum of rough to smooth. For very smooth objects the highlight generated is very sharp and usually very small. For more course surfaces the highlight generated is usually softer and larger. This component is described by color and shape of highlight.

GPU – is the graphics processing unit located on the video card. Current GPUs contain multiple cores per processing unit. In current gaming and development video cards, the processing units are unified architecture that supports vertex, geometry and pixel processing.

Hardware Profiles – Each graphics card has a set profile describing what it can do. The profiles are defined by two factors: graphics API and GPU architecture. All GPGPU programs must be written to the standard set by the local hardware profile. Newer graphics cards have modern hardware profiles and therefore support greater functionality.

NVidia CUDA™ – is a “general purpose parallel computing architecture – with a new parallel programming model and instruction set architecture…” (pg. 3). Advantages to CUDA are its *scalable programming model*, parallelism, ability to be programmed with C-style native language, and direct connection to GPU.

CUDA Kernel – Kernels are c-style functions which run on all threads of the GPU during calculation

NVCC – is the complier that separates host code from device code and compiles device code to PTX format.

PTX – CUDA assembly style device code.

ABSTRACT

Britton, Andrew D. M.S., Purdue University, May 2010. Full CUDA Implementation of GPGPU Recursive Ray-Tracing. Major Professor: Dr. Bedrich Benes.

Pioneered by the works of Whitted and Appel, ray tracing has become a standard format for image rendering. Ray tracing is a very accurate mathematical calculation of light and color, but is a very slow process. The question becomes how can researchers combine the speed of GPU calculations with the rendering quality of ray-tracing? The focus of this research is to solve this question. Our research will test the effectiveness of decreasing render times by implementing a full GPGPU ray trace renderer with recursive ray casting.

The purpose of this study is to test the speed of brute force ray tracing calculation on the GPU versus the optimized ray tracing capabilities of a production quality renderer. Specifically, how much faster, if at all, can the GPU speed up rendering.

For this study the author created two renderers, a CPU renderer and a GPU renderer, written in C++ and CUDA respectively. The author written renderers are implemented without spatial partitioning or ray-object prediction algorithms. The rendering speed of the CPU, GPU and Mental Ray renderers were tested in two scene groups with the first group containing one scene and the second group containing three scenes. The first test scene contains a 5 sided box of 10 triangles and 48 spheres. The second group of scenes contains the same box of 10 triangles with an expanding set of objects. The first, second and third scenes contain 900, 10000 and 30000 objects, respectively. All renderers

generated 25 frames per scene. The average time for renders was compared for each test. Each renderer was tested on multiple hardware devices.

The GPU renderer outperformed both the author written CPU renderer and the Mental Ray renderer in both tests. In the first test scene, the average render times for the GPU, CPU and Mental Ray renderers were 988.94, 75246.3, and 6007.067 milliseconds, respectively. For the second group of test scenes of 900, 10000 and 30000 objects the author written GPU renderer outperformed Mental Ray in speed of rendering. Due to the spatial partitioning algorithm in Mental Ray, the GPU renderer out performed by smaller amounts as the number of rendered objects increased. It is believed that at a large enough number of rendered objects the parallel nature of the GPU will fail in comparison to the spatial partitioning algorithms in Mental Ray.

CHAPTER 1. INTRODUCTION

Computer graphics requires clear, detailed, visually rich images to successfully communicate intended topics with a desired audience. Of the existing rendering schemes, ray-tracing is at the heart of most rendering models of generating high visual fidelity imagery. The core of this research stems from an artist seeking to understand photorealistic rendering algorithms and how to increase rendering speed. Once the material is understood artists will be able to bring their artistic visions and visual vocabularies to computer graphics. It is the aim of this education of an artist in computer graphics programming that artists will learn another tool to create more artistry. The arena of ray-tracing is chosen because the technologies sit at the nexus of multiple computer graphics disciplines. Implementing a renderer requires the artist learn about linear algebra, shader descriptions, the science of electromagnetic transport over multiple surfaces, programming, geometric descriptions, and rational, versus intuitive, logic. The lessons learned herein should also provide the technology artist with the concept that the computer, with its 0’s and 1’s, is a tool, just as paintbrushes, chalk-sticks or pencils are tools. This research should help the reader better understand ray-tracing applications.

A solid understanding of ray-tracing and its additional functionality is no small task for the artist converting to technology. Whereas the artist in classical training contexts will learn about light, shape, shadow, form and line, these lessons are taught in an experiential environment. Artists are taught to see light, color, form, line and shape and then trained to recreate their vision in a variety of materials: paper and pencil, pen and ink, etching, printmaking, sculpture, or watercolor. The artist must understand, as Alberti mentioned, the mathematics of line of sight.

Ray tracing is a well understood process in computer graphics. This understanding starts with the work of Appel (1968) and Whitted (1980) through their introductions of ray-casting and ray-tracing, respectively. With the advent of the latest GPUs that out-perform CPUs in graphical data calculation, a rising question becomes how can the GPU be used to increase the efficiency of a ray-trace renderer.

The field of mathematics Alberti references in his text is the study of linear algebra. Linear algebra covers matrices, vectors and points, or directions, orientations and locations. A mathematical object that contains both a point and a vector is a *ray.*

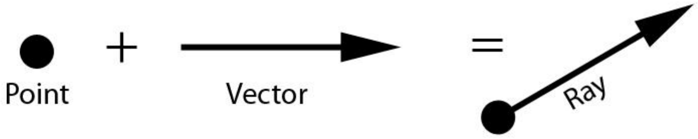


Figure 1.1 - The construction of a ray

Rays are the backbone of ray-tracing. It is these rays that are cast from the ‘eye’ into the world. These rays are then calculated to check intersections with objects. Where an intersection exists, the ray returns color information. This color information is based on two inputs: lighting and materials. Materials are defined through a set of specular, ambient and diffuse parameters. Lighting can be defined through two methodologies: direct lighting and indirect lighting. Direct lighting calculations use linear algebra to define light positions and lighting directions. These calculations define lighting on an object where light is received directly from the light source, and not from multiple bounces. Indirect Lighting simulations calculate the light that bounces over multiple diffuse surfaces. These lighting calculations are more complex and require Monte Carlo integration to approximate light paths over multiple diffuse reflections. The work of Dunn and Parberry (2002), *3D Math Primer for Graphics and Game Development*, outlines all necessary concepts including: vectors, points, matrices, transformations, orientations, coordinate systems and three dimensional math. In addition to the study of linear algebra, the new technology artist must learn to implement the formulas and logic required to construct a ray-tracer. This construction requires a computer and a programming language to support all the ray-tracing functionality required to generate hi-quality images. The programming language of choice is C++. The texts used to study C++ are the works of Gaddis, Walters and Muganda (2007) and two texts by Meyers (2001 & 2005).

Programming on the GPU will be performed with CUDA 2.3. The use of the CUDA programming interface requires the use of NVidia graphics cards. Programming with CUDA applies certain limitations: no support for polymorphism, recursion, or `double` data types (when using CUDA compute architecture 1.1 or less). In addition to these limitations exists a best practices limitation of reducing the amount of data transferred between the GPU and the CPU. Modern prosumer CPUs have anywhere from two through eight logical cores, whereas modern, prosumer graphics cards contain anywhere from 96 to 192 cores. In addition to the greater amount of processing cores, the GPU architecture makes the graphics card operate as a stream processor. The GPU cores are programmable and can process large amounts of data. Because of the highly paralleled nature and stream processing of GPU architecture, rendering on the GPU represents the possibility of a large gain in rendering speed.

1.1. Research Question

What is the speed increase of performing ray tracing on the GPU versus on the CPU?

1.2. Scope

To execute the rendering research the author has created two different renderer implementations: one on the CPU and the other on the GPU. The CPU software will be written using C++, object-oriented coding practices, and recursion. The GPU renderer is written in CUDA replacing recursion with loops. The scope of this project is limited to the creation of a functional software renderer and one functional hardware renderer. Hardware implementation will be written and tested on NVidia graphics cards in the 8800 series or later. Testing on the NVidia cards will ensure that the GPU supports the CUDA language and architecture. Creating cross-platform ports, support for multiple graphics card vendors, and extending the code to multiple languages is outside the scope of this research. The rendering tests will be performed using multiple GPUs and CPUs. Each test will render pre-defined 3D scenes containing a series of implicit spheres and triangles, multiple lights, reflections, shadows and material descriptions.

1.3. Significance

The significance of this research fills a professional void for the author. With an established background in 3D art, 2D art, and 3D animation for production and education, the author is conversant in applying 3D practices and systems to create imagery. This level of understanding reaches a limit as the knowledge base approaches the mathematical and technical realms of 3D computer graphics. Extending the researcher’s understanding of 3D technical and mathematical concepts will enable to researcher to accomplish and understand a wider array of tasks and goals. These goals are achieved through the study of ray-trace rendering because the methods required to create a ray-tracer include: shaders, lights, shadows, rays, geometry. In addition to the base rendering concepts enhancing rendering performance via the GPU is a crucial step because of the architecture of modern GPUs and their abilities to simultaneously calculate large data-sets. The power in performing functions on the GPU is the massively parallel nature of the GPU's architecture. Modern CPUs contain two to four cores. Modern graphics card for the high-end consumer contain as many as 480 cores (figure 1). Each core works in parallel. This massive parallelism is designed to calculate large data sets in real time (60 frames per second).

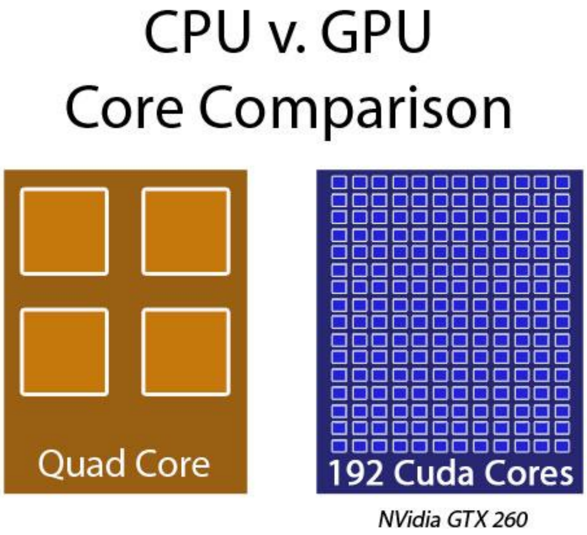


Figure 1.2 - Comparison of amount of computational cores between high-end, consumer CPU and high-end, consumer GPU

1.4. Assumptions

This study assumes the following to be true or in place in for this study:

- All tests will be performed on computers with NVidia graphics cards with 8800 GTX technology, or later.
- The GPU renderer code will be complied using CUDA 2.3.
- The tested video cards will support at least the lowest level CUDA compute architecture with some capable of running the latest compute capability of 1.3.
- All rendering code is precompiled and loaded for the testing machine.

1.5. Limitations

This research draws on the following limitations:

- In order to prove effectiveness this study is only recording timed data of rendering tests.
- All testing GPUs must be CUDA™ enabled, loaded with Windows XP or higher and have software to compile C++ and CUDA code.

1.6. Delimitations

This research draws the following delimitations to reinforce scope of study:

- GPU functionality will not be tested on video cards made by vendors other than NVidia.
- Research will not study qualitative data from a sampling of users and their impressions of quality or effectiveness of tested renderers

1.7. Chapter Summary

This research focuses on applying the calculation efficiency of the GPU to the rendering processes of ray tracing. The purpose in marrying these two technologies together is gaining the speed increases of the GPU.

CHAPTER 2. LITERATURE REVIEW

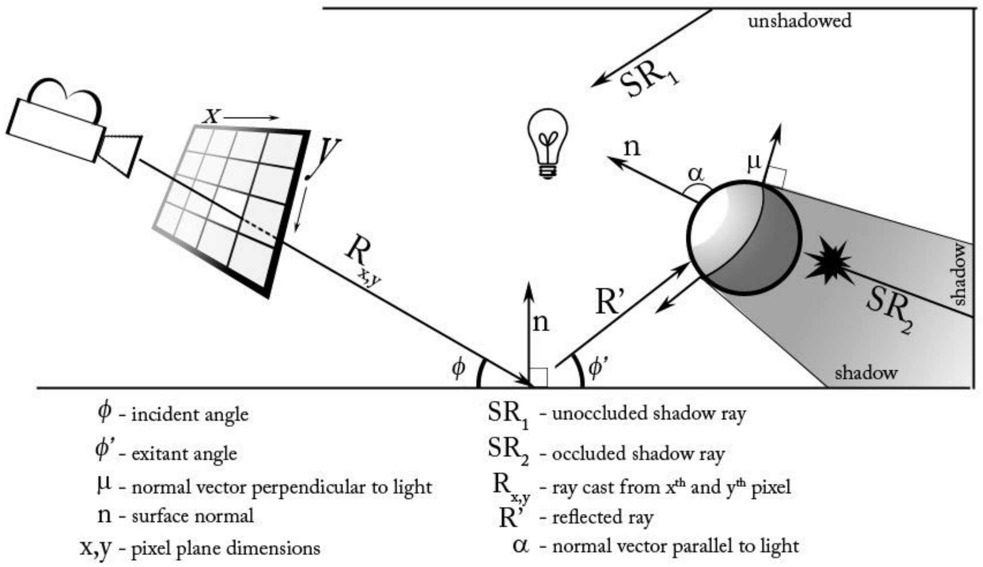

unshadowed
SR1
x
y
n
α
μ
Rx,y
n
R'
shadow
SR2
shadow
ϕ
ϕ'
ϕ - incident angle
ϕ' - exitant angle
μ - normal vector perpendicular to light
n - surface normal
x,y - pixel plane dimensions
SR1 - unoccluded shadow ray
SR2 - occluded shadow ray
Rx,y - ray cast from xth and yth pixel
R' - reflected ray
α - normal vector parallel to light

Figure 2.1 - Illustration of rays, vectors and angles required in ray tracing

2.1. Ray Tracing

Ray-tracing finds its roots in the works of artists and mathematicians from centuries earlier. In Alberti's desire to understand the mathematics behind art and vision, he described sight and vision in a matter aligned perfectly with computer graphics, and especially ray-tracing. Alberti states: "Let us imagine the rays, like extended very fine threads gathered… going back together inside the eye where lies the sense of sight. They are like a trunk of rays from which, like straight shoots, the rays are released and go out towards the surface in front of them" (Alberti, 1435, p. 40).

According to Shirley and Morley (2003), ray-trace renderers are built upon a series of simple algorithms that are used, in turn, to generate digital images. In contrast to a rendering method like scan-line rendering, ray-trace rendering is becoming more-popular because of increased computing power and the renderers ability to cleanly solve problematic topics such as realistic material transparencies and object shadows.

```
for ( current pixel width: x)
    for (current pixel height: y)
        for (every shape)
            find all shapes visible to this pixel
        if (pixel x,y sees shape)
            draw closest shape at pixel x,y
        if (pixel x,y does not see shape)
            draw background color at pixel x,y
Draw all pixels to image
```

Algorithm 2.1 – Ray tracing pseudo-code

The algorithmic process of ray-tracing is simple to understand. There exist objects to create and methods for describing their connections. The base list of required objects to create are: camera, ray (a position and direction), two-dimensional array of pixels (an empty image), light(s), and shape(s). All objects in the scene are connected via independently calculated rays. In order for the renderer to ‘see’ objects and render them, those objects must be in the line of sight of the camera. The line of sight is calculated as a ray, whose originating position is the camera and whose direction is determined by the location of the empty image. Rays are cast into the scene and wherever these line of sight rays intersect with various shapes, the renderer calculates which object is hit first and what color that object is. The color is calculated by casting rays, from a point on the surface, into the scene to “see how the world ‘looks’ to that point” (Shirley &

Morley 2003). Each surface point is checked for facing direction toward or away from the light sources. Any surface section facing toward a light whose view of the light is unobstructed will receive a lighting contribution. This contribution is based also on the amount to which the surface section points toward the light.

2.1.1. Ray – Object Intersections

All ray-trace rendering is computed through collision detection of objects and rays. Rays are cast from the camera, through each pixel of the image, and tested for ray-object intersection. In order for the viewer to see an object in space, it must be in the viewer's direct line of sight, or in the reflected and refracted lines of sight. The computer must test line of site properties by shooting a ray from the camera into the scene and checking to see if the ray intersects with objects. This functions used for checking differ for various objects, though the principles are the same. Without optimization, default ray-casting techniques require a high degree of calculations dependent on the resolution x & y and number of objects in the scene description. This generates ray-intersection calculations in the amount of per pixel width, height and per object in scene description (Shirley & Morley, 2003). It can be described by the following equation:

$$n_{calculations} = res_x * res_y * n_{objects} \quad (1)$$

2.1.2. Spatial Partitioning

In order to speed up the rendering of a scene, an optimized algorithm of ray-casting is required. By grouping multiple objects together into Bounding Volume Hierarchies (BVH), ray-to-object intersection tests can be avoided on a strict per-pixel-per-object basis. With collected hierarchies, if a cast ray does not intersect a BVH, then it does not intersect any members of the BVH (Akenine-Möller & Haines, 2002). The search complexity now becomes a $\log_n$. Both

Akenine-Möller & Haines (2002) and Shirley & Morley (2003) describe variations on well tested implementations of BVHs: axis-aligned bounding-boxes (AABBs) and oriented bounding-boxes (OBBs).

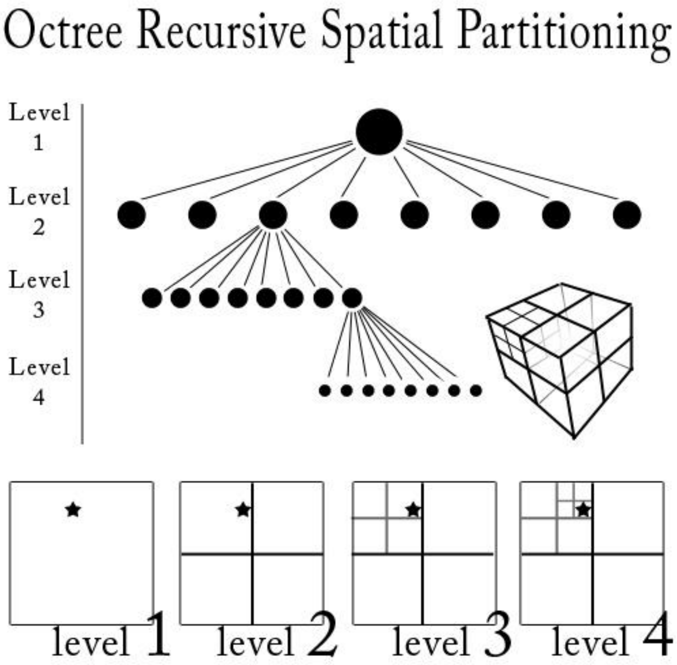


Figure 2.2 – Illustration of octree as spatial partitioning

An octree is a spatial subdivision method for dividing the scene objects into smaller partitions. Referring back to equation 1, that algorithm is of type O(n) per pixel. This is a clumsy searching method to check for ray-object intersections. Most rays will intersect a handful of objects.

Figure 2.2 illustrates the nature of spatial subdividing in octrees. Both Ericson (2005) and Akenine-Möller, Haines and Hoffman (2008) describe the symmetrical subdividing process of the octree. Each node, in order to make child nodes, is subdivided in half along each axis yielding 8 smaller nodes. Ericson presents a linear octree array solution for hierarchy traversal (pg. 314). His research further explains the implementation of a binary key for the hierarchy traversal called the *Morton* key. This binary key positioning simplifies the hierarchy traversal to O(1) access. This is preferred over a pointer-based hierarchy because this is O(log *n*) complexity.

Consider the following scene (Figure 2.3). This render contains 48 objects. The largest amount of objects a single ray will intersect is either three or four. The left-side image will calculate if all rays intersect all objects. However, if the scene is partitioned two levels deep then twelve sub-cubes will only be tested once for intersections. This leaves 75% of the cast pixels to calculate a single intersection each. The remaining 25% of the rays will only perform intersection checks on at most 12 objects.

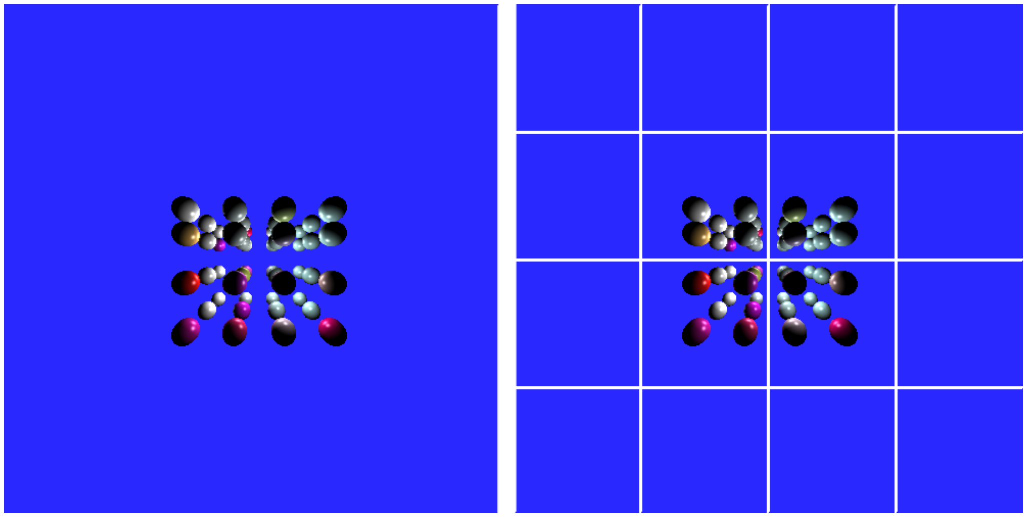


Figure 2.3 – No spatial partitioning (left); octree partitioning (right)

2.1.3. Illumination

Shaders are material descriptions on objects. Shaders describe if an object's look is reflective, refractive, soft, spongy, etc. Most shaders require the presence of lights in order to properly calculate their effects. In order to increase rendering time and render efficiency, shaders have broken natural reflectance functions into two main categories: diffuse reflections and specular reflections. The *Blinn* (Blinn, 1977) and *Phong* (Phong, 1975) shaders are examples of this. Diffuse reflections are soft reflections as seen on matte surfaces, while specular reflections are the shiny highlights found on smooth surfaces like mirrors and chrome. Beyond this description there are other factors of materials that must be described: transparency, bump-mapping, normal-mapping, ambient-occlusion, surface color, specular high-light color, etc. The real world presents a large array of reflection and surface types, and there are many shader technologies to encompass the description of each one.

As ray-tracing algorithms were being “popularized and developed” in the late 70’s and early 80’s (Shirley & Morley, 2003), material descriptions were being generated to meet surface description demands of ray-tracers. The work of material description pioneers such as Blinn (1977), Phong (1975), Cook and Torrance (1981) appears in publications around this time frame as well. Akenine-Möller & Haines (2002) outline some of the canonical material functions to come of this early work, specifically dealing with the specular, diffuse and ambient components of material descriptions.

The work of He, Torrance, Sillion and Greenburg (1991) defines the need for a comprehensive model for reflected light that encompasses multiple reflectance situations: specular, directional diffuse and uniform diffuse. Because the model uses wavelength calculations for light, and incidence angle and surface roughness calculations for the surface descriptions it can describe a smooth transition between the three different reflectance types. Their reflectance model adds further features of describing the role polarized light on material appearance, including polarized light into the shader algorithm library, and provides for listed above. Their model is also analytic. This shader is based on the *bi-directional reflectance distribution function* (BRDF) algorithm. Their results map very closely to experimentally tested reflectance on physical surfaces.

2.1.4. Direct Illumination & Shadows

“There is also a third condition in which surfaces present themselves to the observer as different or of diverse form. This is the reception of light” (Alberti, 1970, p. 44). Pharr and Humphreys (2004, pg 35) describe direct lighting as “light that arrives at the surface directly from emissive objects.” They continue to define this type of light by differentiating it from indirect lighting, saying that direct lighting does not consider the light that bounces through a scene before contributing to the light at a given point. Further reading of Akenine-Möller & Haines (2002) lists multiple iterations of the rendering equation for the local lighting model. The simplest form of the local lighting model is defined as the total intensity for a surface point being equal to the summation of ambient, diffuse and specular components.

As described by Shirley, Ashikhmin, Gleicher, Marschner, Reinhard, Sung, et al. (2005), one of ray-tracing’s strengths is its straightforward approach to calculating shadows and reflections. Their work goes on to say that once a basic ray-tracer program is generated, the addition of shadows is an easy matter.

2.1.5. Cameras

The work of Edward Angel (2008) outlines a variety of camera types and camera declaration scenarios. The cameras used in computer graphics can be divided into two groups: parallel projection and perspective projection cameras. Parallel projection cameras treat all lines of sight originating from the camera as being parallel. This generates images that yield no natural distance-based foreshortening. The objects in these images are consistently sized in relation to other objects regardless of distances from each other. Perspective projection cameras are physically accurate. Cameras of this nature automatically support foreshortening effects. In these cameras, line-of-sight rays, cast from the cameras, are not parallel thereby necessarily causing foreshortening. Angel specifically mentions (see page 241) that the major use of perspective cameras is in applications where it is important to generate realistic imagery, as in animation. Shirley and Morley (2003) discuss a specific version of the perspective projection camera: the thin-lens camera. The thin-lens camera supports automatic depth-of-field focusing or blurring. It achieves this by incorporating the physical camera attribute known as *focal length*.

2.1.6. Objects

Shirley and Morley (2003) give detailed descriptions on basic primitive object implementation. Specifically their work in this chapter relates to the drawing triangles and spheres by defining the intersection of a ray and the triangle, or sphere, primitive object. Using the position and direction of the viewing ray, those parameters can be input in the mathematical descriptions of spheres and triangles with the resultant solved values defining if ray-object intersection has occurred, twice, once or not at all.

Beyond the tasks of drawing a simple triangle primitive, a large section of computer graphics rendering requires the use of triangle meshes. Hill (2000) outlines the process of reading and drawing a complex list of polygons as a series of individually defined faces. These faces are then rendered as triangles (as mentioned above).

2.1.7. Texture Mapping

In his section on texture mapping, Edward Angel (2008) describes multiple uses and generation processes for creating texture maps. Texture mapping is the process of applying color patterns to geometry or fragments. The texture can be generated by procedural means or image digitization. Once the texture is generated it can now be applied to multiple uses as a color map, a bump map, a specular map, normal map, environment map or transparency map (Birn, 2000, pg 204-213). Textures can be defined in a variety of dimensional spaces: 1D, 2D, 3D or 4D textures. Procedural textures can be of type 2D or 3D, by defining texels in a two-dimensional array or a three-dimensional array. Pages 79 – 84 of Shirley and Morley (2003) describe pseudo-code that outlines the processes of generating various procedural textures: stripes, noise and turbulence. As Shirley and Morley outline, these procedural texturing functions are implemented via periodic functions of sine and cosine.

Once textures are created, there exists the problem of assigning textures to the surface of the geometry. This requires multiple coordinate systems: one for the geometric surface, and one for the texture space. Polar coordinates are required for generating surface UV coordinates. Generating the same UV (texture) coordinates for triangles requires separate functions.

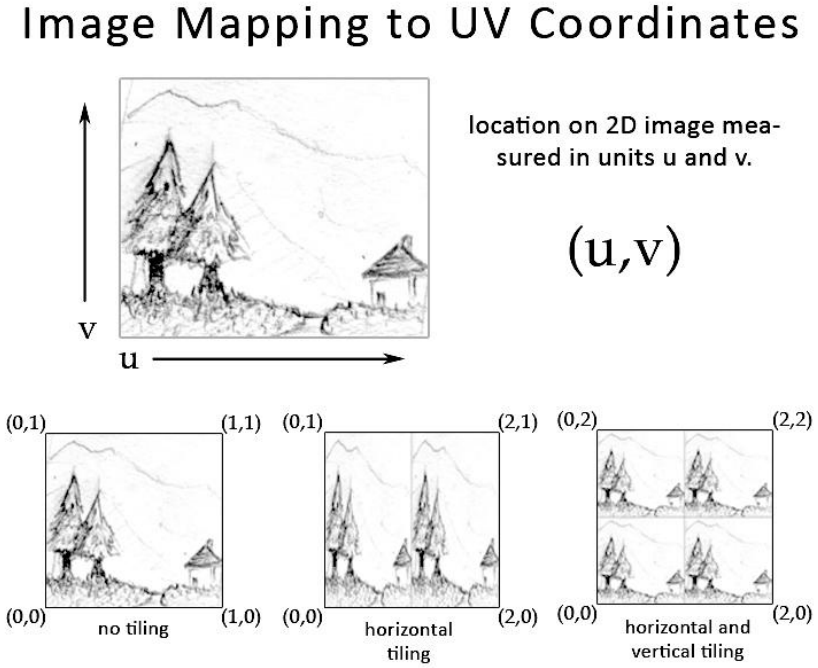


Figure 2.4 - UV coordinates define the placement of a texture on geometry.

2.2. Speed Improvements

Central to the core of this research is decreasing render times. Specifically, this research will decrease render times through the implementation of GPGPU programming.

2.2.1. GPGPU with CUDA™

GPUs are massively parallel processors. As illustrated in Figure 1.1, modern GPUs contain more processing cores than the latest consumer CPUs. The NVidia GTX 260 contains a power of 3.792 more cores than a quad-core CPU. With so many processors ready to be used the GPU needs an effective method of communicating with and managing the system resources to work in sync and not perform redundant work. As defined by NVidia Corporation (2009), CUDA is a “general purpose parallel computing architecture – with a new parallel programming model and instruction set architecture…” (pg. 3). Advantages to CUDA are its *scalable programming model*, parallelism, ability to be programmed with C-style native language, and direct connection to GPU.

Using CUDA requires writing code for the host and the device. Herein, the .cu file contains commands that perform operations on the CPU and that launch operations on the GPU. These device operations are CUDA kernels. Code for each compute hardware must be compiled separately. C for CUDA code is written with C/C++ style implementations for host code and C style implementation for device code. Compiling a .cu file has two stages. The first stage is interpreted by the *NVCC*. A .cu file can contain a mix of host and device code. The *NVCC* separates the host code and the device code into: C code to run on the host, *PTX* code to run on the device.

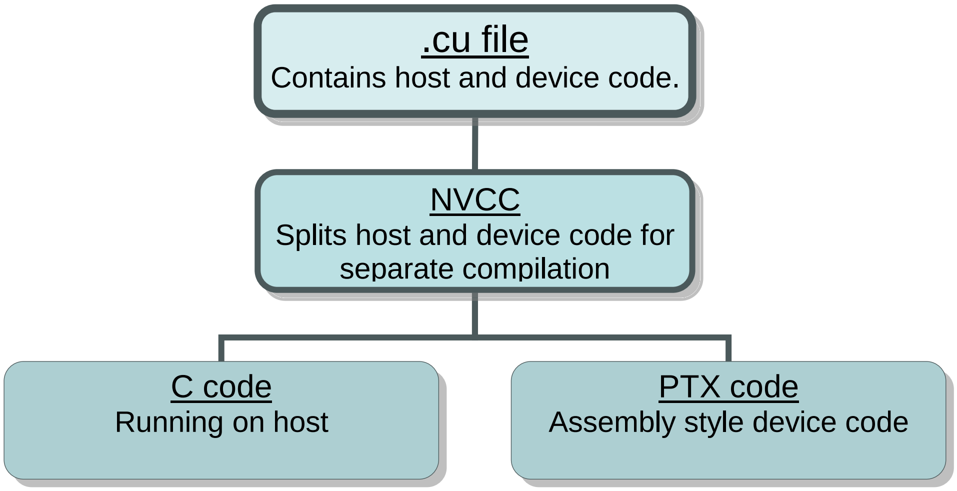

.cu file
Contains host and device code.
NVCC
Splits host and device code for separate compilation
C code
Running on host
PTX code
Assembly style device code

Figure 2.5 – Workflow of .cu file compilation

2.3. Existing Real-Time Ray Trace Rendering Techniques

2.3.1. Hardware

Recent developments in real time ray tracing include new hardware devices. One device is IBM's CAS Cell BE processor. This processor is found in the PlayStation 3. From the work of Cox, Máximo, Bentes and Farias (2009), the processor is not specifically a ray tracing processing unit. This processor can be advantageous for ray trace rendering however. What makes this processor appealing is the parallel nature of the independent cores. The Cell BE processor is similar to modern GPUs because both hardware devices support SIMD architecture.

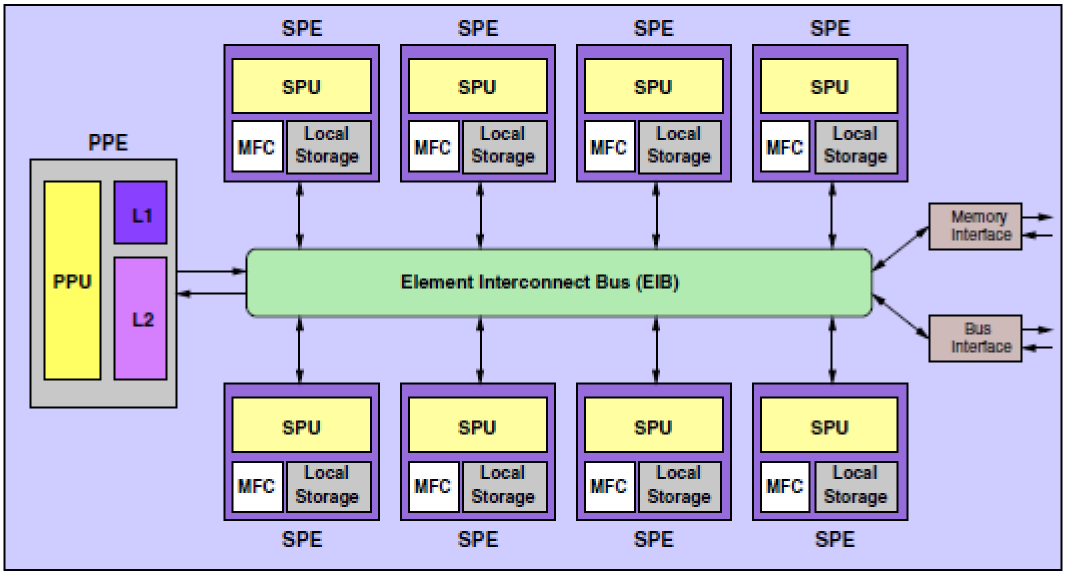


Figure 2.6 – Diagram of Cell BE architecture (Cox, Máximo, Bentes and Farias, 2009, pg. 9)

An altogether different device is the RPU, *ray processing unit*. The RPU was proposed by Woop, Schmittler and Slusallek (2005). Their research introduced a "prototype implementation of a single chip, fully programmable Ray Processing Unit." Most fascinating about the RPU are the ways it is architecturally different from modern SIMD GPUs. Their prototype RPU supports recursion and function branching. The RPU runs at low clock speeds of 66MHz. Despite the low clock rates, the RPU can render at interactive frame rates, thereby competing with powerful GPUs.

2.3.2. Software

Leveraging the power of modern GPUs is NVidia’s real time ray tracer, OptiX. The OptiX renderer is an abstracted perspective on standard graphics ray tracing. While being fully capable of rendering scenes with ray tracing for graphics applications: games, marketing, or design, OptiX was designed to be highly flexible and useful for any ray shooting computation (NVidia, 2010). Additional industrial applications of OptiX are: acoustic design, volume rendering, collision checks and radiation research.

The flexibility of OptiX is found in its generality. The data that rays carry and collect, the intersection algorithms, camera construction, and the shading algorithms are all programmable. This allows for rendering in different environments with different types of radiation. The OptiX engine contains spatial partitioning structures. The included structures are KD-Trees and BVHs. To increase the quickness of calculation, NVidia included a smart load balancing system for thread execution. The OptiX engine also supports recursion and easy OpenGL interoperability. It requires CUDA 2.3.

RTfact is a real-time renderer in development. The authors of RTfact are Georgiev and Slusallek (2008). RTfact is renderer based on C++ templates to create abstract definitions of rendering phenomena: `Primitive`, `Intersectors`, and `Packet <size, type>`. The *primitive* context contains not just mesh objects but can also contain photons. The *primitive* context does not perform intersection calculations; this is what the *Intersectors* are for. *Intersectors* provide the intersection functionality. The *Packet* context is easily scalable to contain one ray, or multiple rays. This ease of ray scalability is due to the template nature of *Packet*. Rays are stored as a bundle of rays in one *Packet* context. The templated functionality allows the multiple rays per bundle to be calculated in parallel. “We advocate generic design as a key to flexibility and efficiency, especially for computationally intensive applications, such as real time ray tracing” (Georgiev & Slusallek, 2008, page 8). The authors show some time improvements with RTfact over other renderers: OpenRT, and Manta.

2.4. Summary

This chapter reviewed existing literature in the area of computer graphics rendering technologies. This review highlighted previous work done in the area of rendering technologies and software features. These features focused on ray tracing technologies such as: direct illumination models, shaders, cameras and shadows. The improvements to rendering speed are implemented using CUDA.

CHAPTER 3. METHODOLOGY

This research is testing to what quantifiable degree is render time decreased when performing full recursive ray tracing on the GPU using CUDA. The testing methodology includes two renderers written by the author and one production-ready renderer available for professionals and consumers. The two renderers written by the author are used as the control (CPU renderer) and the testing group (GPU Renderer) to prove efficiency of GPU Rendering. The GPU renderer is being tested against a production as a more stringent testing benchmark on the efficiency of GPU Rendering. The production quality renderer (Mental Ray) is a robust renderer with many data, mathematical and logic functions to shorten or reduce redundant processes. The GPU-Assisted Renderer will not have this same robustness of rendering efficiency. These tests will show how much time is saved rendering on the GPU, if indeed time is saved.

The measured variable is the average time it takes to complete 25 frames of rendering, for both GPU and CPU. This time to complete will be measured on the computer performing the rendering tests. Variable data will be accurate to milliseconds.

3.1. Algorithms

This section describes the algorithms used in a ray tracer.

3.1.1. The Rendering Equation

The rendering equation is the central algorithm describing the entire rendering process. It describes geometry, light, objects and the BRDF. The rendering equation can be formatted for different types of lighting methods. To calculate direct lighting Dutré, Bala and Bekaert (2006, pg. 44) format the rendering equation in the following manner:

$$L_{direct} = \int_A f_r(x, \overrightarrow{xy} \longrightarrow \Theta) L_e(y \longrightarrow \overrightarrow{yx}) V(x,y) G(x,y) dA_y \qquad (2)$$

The integral sub-A defines the algorithm to take place over all objects. The next two terms describe the BRDF formulation for direct lighting. Specifically, “the direct term is the emitted term from the surface *y* visible to the point *x* along the direction *xy*; *y* = *r(x, xy)*.” The next two terms are to describe the visibility of a surface point from the camera and the local surface information at A_y, *V* and *G* respectively.

3.1.2. Ray Casting

```
for ( current pixel width: x)
    for (current pixel height: y)
        define current view ray
        for (every shape)
            check Intersection-of-Ray-To-Shape
            if (ray sees shape)
                record shape: color, ID, distance from camera
                sort shapes by distance from camera starting with closest
        if (intersection happens for any shape in pixel x,y)
            calculate reflections
            blend reflection colors with main color
            calculate Fog
            assign new color to pixelPlane at x,y
        if (no intersection happens for any shape in pixel x,y)
            backgroundColor = FogColor
            assign new color to pixelPlane at x,y
Draw Image
```

Algorithm 3.1 – Detailed ray tracing algorithm

Figure 3.2 represents the main functional kernel of the rendering engine. For every pixel x and y, for every shape k cast a line of sight ray and check for object collisions. If a collision is detected record the object color, object ID and distance from camera of intersection. As ray-object intersections are calculated the STL::map container sorts them based on distance from camera. Once all shapes are calculated for intersection for one ray the closest object’s color is assigned to the pixel plane. If no intersection is recorded for a ray, the background color is assigned.

3.1.3. Cameras

Most cameras used in CG are of two types: perspective or parallel projection. This does not include specialized or abstracted camera models like multiple centers of projection cameras. For the purpose of this research the perspective projection camera will be our camera model. It matches human vision better because it visually enlarges objects in the foreground and diminishes objects as they are further from the camera. Both parallel and perspective projection cameras cast rays through each pixel in the pixel plane. In order for a projection camera to work, each ray is cast from the center of the camera through each pixel.

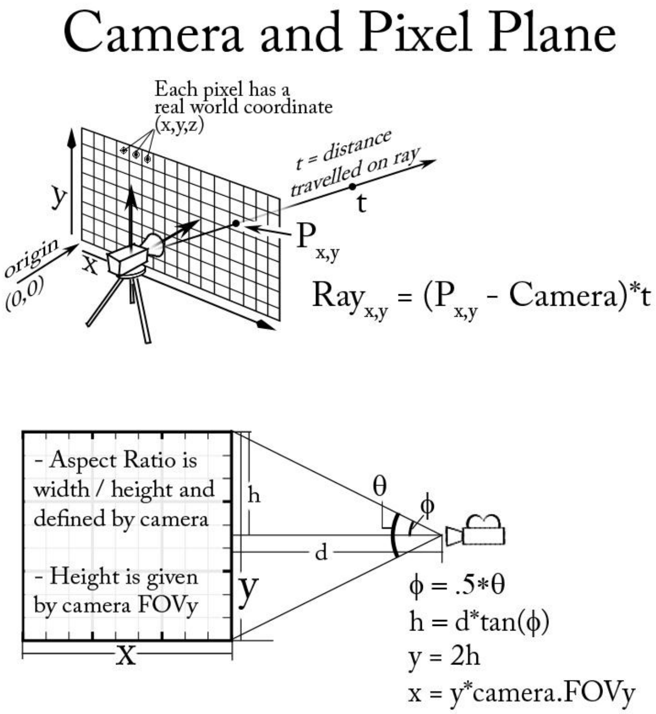

Camera and Pixel Plane
Each pixel has a real world coordinate (x,y,z)
t = distance travelled on ray
t
y
x
origin (0,0)
$P_{x,y}$
$Ray_{x,y} = (P_{x,y} - Camera)*t$
- Aspect Ratio is width / height and defined by camera
- Height is given by camera FOVy
h
θ
ϕ
d
y
x
ϕ = .5*θ
h = d*tan(ϕ)
y = 2h
x = y*camera.FOVy

Figure 3.1 - Casting a ray through a pixel (top); Camera and Pixel Plane models (bottom)

3.1.4. BRDFs

Dutré, Bala and Bekaert (2006) discuss shading models are algorithms that define various BRDFs. There are different classifications of BRDFs: approximations, physically based, and empirically based. The Lambertian, Phong and Blinn models are all approximations of BRDFs because they cannot accurately represent realistic BRDFs. The physically based BRDFs are the Cook-Torrence and He models. These models account for energy conservation and BRDF reciprocity. The empirical models formulate their BRDFs from empirically gathered light-reflectance measurements. These BRDFs were designed to recreate the recorded reflectance phenomena. These are the Ward and Lafortune models. Understanding the Phong model provides the background on which to understand most shading models.

As initially defined by Phong (1975) in his shader model, the reflectance functions are as follows, with equation 5 being the summation of all components:

$$i_{diffuse} = max((n \cdot l), 0) m_{diffuse} \otimes s_{diffuse} \quad (2)$$

$$i_{specular} = max((n \cdot h), 0)^{m_{shininess}} m_{specular} \otimes s_{specular} \quad (3)$$

$$i_{ambient} = m_{ambient} \otimes s_{ambient} \quad (4)$$

$$i_{total} = i_{ambient} + i_{diffuse} + i_{specular} \quad (5)$$

3.1.5. Lighting

Akenine-Möller and Haines (2002) define the local lighting model with a light distance-attenuation component, *d*. Light attenuation is affected by three coefficients s_c (constant), s_l (linear), s_q (quadratic).

$$d = \frac{1}{s_c + s_l \|s_{pos} - p\| + s_q \|s_{pos} - p\|^2} \quad (6)$$

$$i_{total} = i_{ambient} + d(i_{diffuse} + i_{specular}) \quad (7)$$

Each light has its own ambient, specular and diffuse components which can map to any shading model, or modified versions of them.

3.1.6. Ray-Object Intersections

Ray-Object intersections satisfy the *G* component of the rendering equation. As rays are cast from the camera into the scene they are tested for intersections with each of the objects in the scene. The definition of a ray comes from linear algebra and is a vector that originates from a point. It is often given in the form:

$$r(t) = o + td \tag{8}$$

Now that a line of sight can be defined mathematically, they must be integrated into the mathematical definitions of other objects such as spheres and triangles. For spheres the solution to the intersection equation is of the quadratic equation form:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{8}$$

Except values a, b, and c are redefined using ray position and direction, and x is defined in terms of t, where t is distance between camera and intersection location. Thus the equation becomes:

$$t = \frac{-d \cdot (o - c) \pm \sqrt{\left(d \cdot (o - c)\right)^2 - (d \cdot d)\left((o - c) \cdot (o - c) - R^2\right)}}{(d \cdot d)} \tag{9}$$

Ray-triangle intersections are solved through the use of barycentric coordinates: α, β, γ. The point p is on the triangle, if and only if:

$$0 < \alpha < 1 \wedge 0 < \beta < 1 \wedge 0 < \gamma < 1 \tag{10}$$

This can be restated using only two coordinate variables and the third being a combination of the first two. Restating the requirements of a point on a triangle becomes:

$$\beta + \gamma < 1 \wedge 0 < \beta \wedge 0 < \gamma \tag{11}$$

Thus, any cast ray, as either line of sight ray or surface reflection ray, hits the plane where:

$$\mathrm{o} + t\mathrm{d} = \mathrm{a} + \beta(\mathrm{b}\text{-}\mathrm{a}) + \gamma(\mathrm{c}\text{-}\mathrm{a}); \tag{12}$$

3.1.6.1. Line of Sight Rays and Shadow Rays

These formulae are used in line of sight rays (un-occluded visibility, reflected rays, refracted rays) and also in calculating shadow rays. Revisiting figure 2.1 shows the path of two shadow rays, SR_1 (shadow ray one) has a direct path to the light source and therefore that *p1* receives light. Conversely, SR_2 (shadow ray two) is obstructed by the sphere and therefore *p2* receives shadow.

3.2. CPU Implementation

The author written renderer for the CPU follows the work of Shirley and Morley (2003). It is a robust renderer with full C++ implementation.

3.2.1. Classes

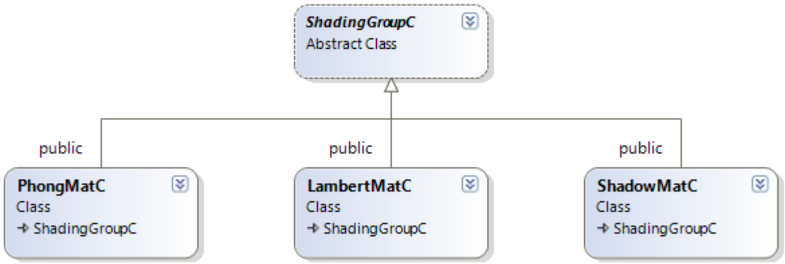


Figure 3.2 – An example of shader construction using classes and inheritance

All logical constructs of a ray tracer are created in separate classes using C++. The use of classes allows for cleaner code and easier editing. It allows for inheritance, polymorphism and makes it easier to add function overriding as needed. A table of classes and organization is included below.

Table 3.1 – *Organization of class types and their classes*

Cameras	**Color**	**I/O**	**Lights**	**Materials**
• CamC • PixelPlane • StaereoCam	• rgb	• targa	• LightC	• LambertMatC • PhongMatC • ShadingGroupC • ShadowMatC
Math	**Scene**	**Shapes**	**Textures**	
• RayC • RNGC • TransformC • Vect2D • Vect3D	• WorldC	• readDataC • Shape • Sphere • Triangle	• MarbleTextureC • NoiseTextureC • SolidNoiseC • TextureC	

3.2.2. Polymorphism

An additional feature of class inheritance is polymorphism. This relationship is used most often in when checking ray-object intersections using the `hit` function.

Both `Sphere` and `Triangle` extend, or inherit from, the `Shape` class. The base class, `Shape`, is defined as an abstract class from which both `Sphere` and `Triangle` inherit their hit functions. `Shape` instantiates hit check functions for use in determining the color of the object at a point and also determining the shadows on the objects through `shadowHit`. `Sphere` and `Triangle` need to define their own hit functions though because each requires very different mathematical processes to define if a given ray does intersect with that object. This is the use of polymorphism. `Sphere` and `Triangle` define the substance of their own `hit` and `shadowHit` checks; they contain the algorithmic processes to determine if a hit occurs with a given ray. The `Sphere` class just declares that each shape type needs these functions. But why use polymorphism at all?

In the `WorldC.h` class, all objects are added to one list. This list contains all the shapes that the renderer will see. To maintain this single list, the `WorldC` contains a member `vector <Shapes>` where both types of shapes are stored. The shapes and shape types can be stored in any arbitrary order. This means that it would be easier if all shapes were seen as the same type of object and treated in the same manner. Because `ShapeC.h` is a base class that contains abstract virtual functions, all shapes – regardless of type – are interpreted by the renderer as the same type of object. Hence, no sorting or segmenting algorithm need be applied. All shapes, because of abstract virtual functions, are treated the same. Algorithm 3.1 shows that the hit check is performed for all shapes regardless of type.

3.2.3. Recursion

Recursion is how reflections are calculated on the CPU. The `RayCast` function is designed to call itself as many times as needed. Recursion is a delicate matter. If a recursive call has no escape the program falls into an infinite loop, memory leaks and the program eventually crashes. To avoid the disastrous eventuality the `RayCast` function has three escape checks: a `rayCount` variable, a no reflection check and a no intersection check. Only if the `rayCount` is less than `MaxTraceDepth`, the ray intersects an object, and the intersected object is reflected, then `RayCast` call itself again. Each time it calls itself, it iterates the `rayCount` variable by one, bringing this escape catch closer to finality with each recursive call.

```
RayCast(inherit previous viewing ray and updated rayCount variable)
    if (rayCount < MaxTraceDepth)
        for (every shape)
            check Intersection-of-Ray-To-Shape
        if (intersection happens for any shape in pixel x,y)
            is new surface reflective?
            if (new surface is reflective)
                rayCount = rayCount + 1
                RayCast (reflected ray, new rayCount)
            return reflected color
        if (no intersection happens for any shape)
            return background color
```

Algorithm 3.2 – The `RayCast` function checks for reflectivity and calls itself

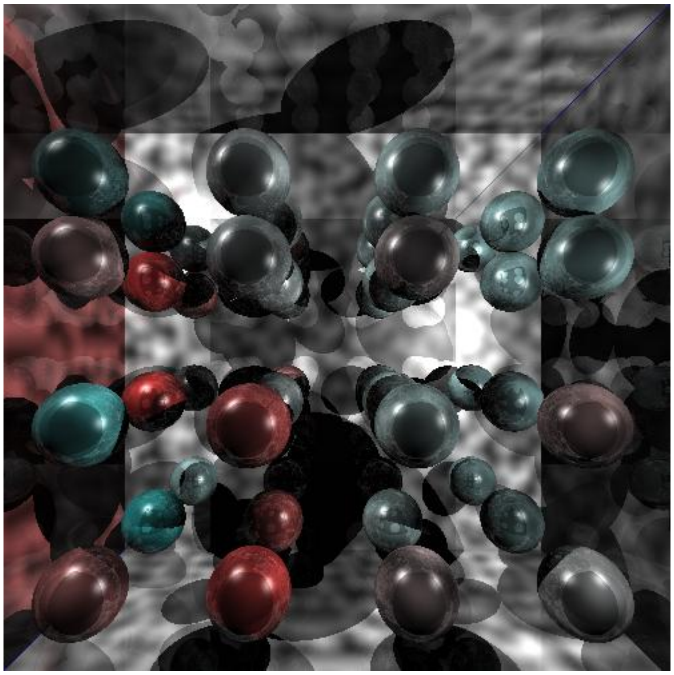


Figure 3.3 – A CPU rendered image with reflectivity

3.3. CUDA Implementation

This section describes the implementation of ray tracing code on the GPU using CUDA and discusses key GPU programming issues: minimizing data transfer between host and device, no support for virtual functions, limited support for classes, and writing recursive functionality in a programming environment which does not support recursive functions.

3.3.1. CUDA Code Overview

Writing code in CUDA requires a different workflow as compared to coding in C++ on the CPU. Because CUDA leverages the parallelism of the GPU, thread counts, block and grid sizes must be established before any calculation can start. Coding for CUDA requires coding with the awareness of multiple cores and parallel processing over cores. Writing code in C++ for the CPU however can be coded without thought as to the existence of multiple CPU cores. This disparity in inherent parallelism coding standards requires an initial coding process to be determined: how to parallelize the rendering process of a single image. Once this is established the porting process from an unparallel, C++ based, CPU implementation to a massively parallel, CUDA based, GPU implementation.

The flow of the CUDA ray trace renderer code is divided into four logical segments: variable and data type declaration, memory initialization, kernel definition and rendering.

3.3.1.1. Parallelization and Thread Assignment

When rays are cast into a scene their final goal is to return a single color value for each pixel and the color value of one pixel is not determined by the color value of a neighboring pixel (excepting anti-aliasing but even here the pixel is still the smallest, independent component). Given that the nature of ray tracing is that of casting rays through each pixel and calculating ray-object intersections over all objects in a scene against all cast rays, the smallest, independent element in the algorithm is the pixel. Because of this property the parallelization of the rendering process runs on a per pixel basis. Therefore each pixel of the final output image has one GPU thread assigned to it. This thread performs all the lighting, shading and shadow calculations and also checks ray-object intersections for all objects. In effect, this means each thread has access to the entire scene description. This is important because when calculating multiple reflections, each ray can as a possibility bounce into any other section of the scene.

3.3.1.2. Variable and Data Type Declaration

In order to begin any calculations on the GPU the entire scene data, already initialized for the CPU renderer, must be passed to CUDA in a format it can support. CUDA 2.3 has limited support for classes and no support for virtual functions. The scene data for the CPU renderer is contained in a series of classes. Each class contains a series of member variables and member functions. Because of the limited support that CUDA 2.3 has for classes, it was decided that the data should be rewritten in a sparse form as a series of structs. A list of the complete set of `typedef structs` are presented in Appendix A.

Almost all of the newly created data types contain only member variables. Only the `curayTri` (struct dataype describing triangles) contains member functions. These member functions are called at time of object initialization and define internal member variables. These are called just on initialization because they define variables that need be only defined once. Redefining the triangle normal is unnecessary regardless of where on the triangle a ray intersects. This is not true for spheres so this member function was not defined.

To pass the data from CPU to GPU, a series of global pointer variables were declared in pairs: one for host memory, one for device memory. Appendix B has a list of the global variable memory pairs and their uses.

3.3.1.3. Memory Initialization

Once variable pointers are declared, their sizes and contents must be filled. This process is performed before the GPU kernels are run. This process is a collection of functions that connect the main CPU renderer to the CUDA rendering file. There are four types of functions used in this process.

Table 3.2 - *Processes Used to Initialize GPU and Pass Data from CPU*

Sample Process Name	**Location**	**Purpose**
`void LightsToCUDA(WorldC &world)`	rayTracer.cpp	Parses all member data needed to describe the object type into one linear array that holds all objects and all object data members
`extern "C" void defineLights(float*, int);`	rayTracer.cpp	Passes linear array data from .cpp to .cu int = number of objects in array
`extern "C" void defineLights(float*, int);`	GPU.cu	Allocates host and device memory in preparation of filling in data. Copies host memory to device memory once arrays are filled.
`void curayLightInit(curayLight* lightData, float* data, int numLights)`	GPU.cu	Copies linear array data to array of struct data types. See Appendices A & B for description of *curay* structs and uses

Allocation of host memory is performed with `malloc` and allocation of GPU memory is performed with `cudaMalloc`. Appendix C gives code examples of these functions.

3.3.1.4. Kernel Definition

Once memory is allocated, defined and moved to the GPU it is time for the GPU to perform the work of ray tracing. For each section of calculation a CUDA kernel was written; these sections include: initial object intersection and depth sorting, object shading, ray reflection, color blending. See Appendices D-G for examples of all the CUDA kernels.

The first kernel creates the base image. It renders all shapes, finds closest intersections and assigns appropriate color the `float3* C` (the color buffer). This kernel sets up the data stored in the `curayFrameBuffer` which will be used in the next three kernels. This data structure stores closest intersection distance, object id, object type, object normal, and point of intersection. This storage is vital because CUDA does not allow for virtual functions. The CPU renderer uses virtual functions numerously to calculate hit checks against multiple object types for shading values and shadow rays. The `curayFrameBuffer` object stores object id and type for use in later calculations. It stores these values on a per pixel basis.

The second kernel creates the shaded and shadowed image. It first checks for shadows by casting rays from each intersection point in the previous image to each light and calculates areas of shadow. If a shadow is present for the position and light then no material shading component is rendered. Conversely, if there are no shadows then that point has Phong shading applied to it. The result is rendered to the color buffer (Appendix D)

Point of intersection and object description are maintained in the FrameBuffer struct: `curayFrameBuffer`. These values (object type, object id, object normal, distance from camera) are generated in the previous kernel and called here (Appendix E).

The third kernel is only called if the max trace depth is a value greater than 1. This kernel reads the normal vector of the object defined in the `curayFrameBuffer` and the current viewing vector and creates a new reflected vector using the CUDA math function, `reflect`. It only calculates reflection if an object was intersected with a viewing ray in the previous pass. If no object-ray intersection occurs then the curayFrameBuffer object records `ObjType` to be `0` (Appendix F).

This final kernel blends the colors of current color and previous color, `C2[i]` and `C[i]` respectively. The blending value will be defined by a material coefficient of reflectivity (Appendix G).

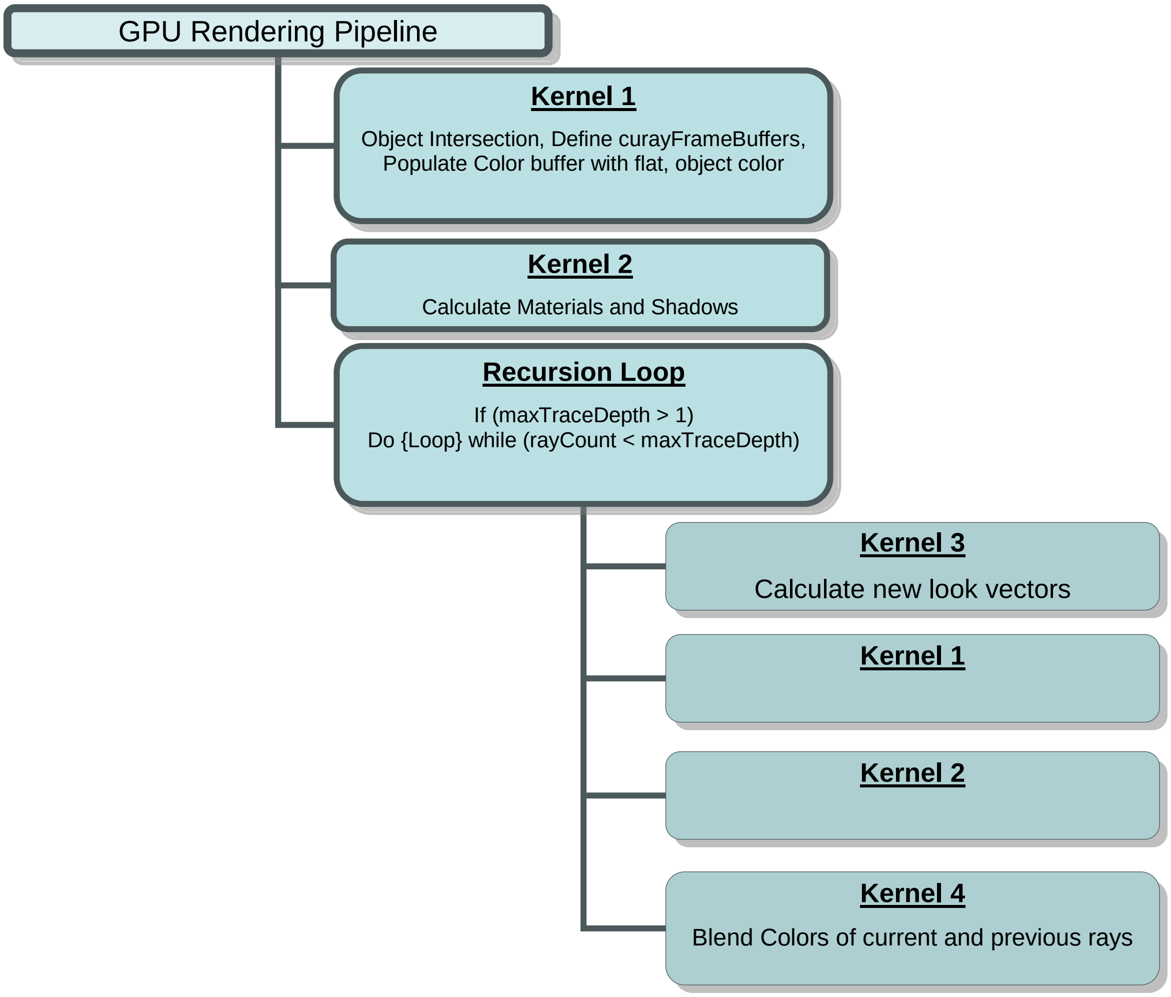

GPU Rendering Pipeline
Kernel 1
Object Intersection, Define curayFrameBuffers, Populate Color buffer with flat, object color
Kernel 2
Calculate Materials and Shadows
Recursion Loop
If (maxTraceDepth > 1)
Do {Loop} while (rayCount < maxTraceDepth)
Kernel 3
Calculate new look vectors
Kernel 1
Kernel 2
Kernel 4
Blend Colors of current and previous rays

Figure 3.4 – Kernel flow of GPU renderer

3.3.1.5. Rendering

The final step in the process is to put each of the steps together. This is where the series of kernels are called to action for as many times as defined by the max trace depth. Figure 3.5 expresses, in pseudo-code, the layout of the GPU rendering process.

There are four main sections to the rendering function: color buffer initialization, rendering first ray cast, rendering multiple ray casts, and writing the image.

The ray cast is kept separate from the reflection ray casts because it is safe to assume there will always be at least ray cast. All subsequent ray casts are decided by the `maxDepth` variable in the C++ side of code. Before the ray casting process is begun for multiple iterations the reflected vectors need to be calculated. `ReflectedViewingRays` is first performed kernel in the recursive loop for this reason. The last step in this process is to blend the reflected color together with the previous color. `BlendColors` performs this operation.

```
Begin GPURenderCycle
        Kernel1 – IntersectShapes
        Kernel2 – CalculateShadingShadows
        If (maxDepth is greater than 1)
                do
                {
                        Kernel3 – ReflectRays
                        Kernel1
                        Kernel2
                        Kernel4 – BlendColors
                        Add one to rayCounter
                }
                while (rayCounter is less than maxDepth)
```

Algorithm 3.3 – Pseudo-code outlining flow of GPU.cu

3.3.2. GPGPU Programming Issues

CUDA 2.3, and earlier versions, do not support certain features found in C++. Two of these features required a large rewrite of the rendering code when porting from CPU to GPU: no support for virtual functions and no support for recursive functionality. In addition to these limitations is the goal of reducing the number of data transfers between host and device.

3.3.2.1. Minimizing Data Transfer Between Host and Device

The original goal of this research was to use the GPU to assist in the ray trace rendering process. At first, the design was to use the data processing efficiency of the GPU to calculate all intersections, this would include all ray-object intersection tests: viewing rays, shadow rays and reflected rays. This would leave the CPU to calculate all material shading and declare the necessity of bounce rays on a per object per pixel basis. Because the CPU renderer is designed to make use of only one CPU thread this would mean that any use of the CPU to calculate on a per object or per pixel basis the CPU becomes a bottleneck in the rendering process. More to the point, calculating ray-object intersections on a per ray cast basis would mean transferring data back from the GPU to the CPU a number of times as shown in the equation below (where SD represents the amount of data to describe the scene, VI represents the accumulated data for ray-object intersections of viewing rays, SI represents the accumulated shadow buffer of ray-object intersections for shadow rays):

$$N_{maxTraceDepth} * \left(SD_{host\rightarrow device} + VI_{device\rightarrow host} + \left(SI_{device\rightarrow host} * M_{numLights}\right)\right) \qquad (13)$$

You can see that the amount of data transfers increase drastically with each increase in `maxTraceDepth`. However, with fully implemented ray tracing on the GPU the number of data transfers, per frame, is limited to two. Even if we are to put the speed lag due to large numbers of data transfers momentarily aside, there would be an increased cost of development time for unraveling the complex data coming from the GPU and interpreting the data for CPU rendering.

Given these considerations it was decided the greatest improvements in speed and implementation would be gained in performing the full rendering process on the GPU.

3.3.2.2. No Support for Virtual Functions

CUDA 2.3 has limited C++ features. Specifically it lacks support for virtual functions. This required a major rewrite from the CPU rendering code. This required two main changes to code. The CPU implementation uses virtual functions in the `Shape.h` to define the hit and `shadowHit` functions. Both the `Sphere.h` and `TriangleC.h` are extended from the `Shape.h` class. Because the hit functions are virtual functions, the sphere and triangle intersection tests are performed in their mathematically appropriate manner while still being called using the same root function name as initialized in `Shape.h`. The benefit of virtual functions in this case is that a list of arbitrary, non-negative, size can be generated containing any amount of spheres and or triangles in an arbitrary order. This implementation is borrowed from Shirley and Morley (2003).

The solution to this lay in creating two separate processes where sphere and triangle intersections are calculated separately and then tested for which is closest. This series of intersection tests is performed in the `IntersectShapes` kernel. For the complete CUDA kernel see Appendix D.

```
//List of Object IDs that are spheres
IDSphere = new int[world.shapes.size()];
//List of Object IDs that are triangles
IDTri = new int[world.shapes.size()];

for(unsigned int i=0; i<world.shapes.size(); i++)
      {
            if(world.shapes[i]->m_objType == 1)//Spheres
            {
                  IDSphere[countSph] = i;
                  countSph++;
            }
            if(world.shapes[i]->m_objType == 2)//Triangles
            {
                  IDTri[countTri] = i;
                  countTri++;
            }
      }
```

Algorithm 3.4 – The intermediate arrays store the placement ID from the vector of <Shape>

The first step in circumventing the lack of virtual functions is in splitting up the spheres and triangles into their own arrays of size defined in scene file. Once the shapes are split into individual arrays and their respective numbers counted, they are then passed to the GPU where a device array is generated and data is copied. Below is the main loop that separates the shapes into their respective arrays.

Appendix H contains the full code for shape separation.

3.3.2.3. Recursive Functionality and GPGPU Programming

3.3.2.3.1. Color Mixing

Another feature not supported by CUDA 2.3 is recursion. This change in coding standards creates a new challenge in writing a ray tracer. The current CPU renderer, coded with recursive functionality, traces reflected light paths to a terminating condition and then starts mixing the color from the back moving forward, always linear interpolating by the reflectivity coefficient. This means that the last two colors to mix are the first two ray casts. The CUDA ray trace algorithm is coded in a forward collective approach. This means that the first two colors to mix are the results of the first two ray casts.

The change in color mixing directions causes a problem. In a forward color mixing approach the first ray cast has the least effect in the final image where ideally and naturally it should have an effect as defined by:

$$Color_{contribution\ of\ first\ ray\ cast} = 1 - R_{coefficient\ of\ reflectivity} \tag{14}$$

There are two easily identifiable methods to solve this approach. The first would be to create a large data storage structure that can hold as many color buffers as there are numbers of ray casts. This would be an unwise usage of memory and generally difficult to code for. The second method is to consider a summation series based on the number of expected ray casts (also equal to the `maxTraceDepth` value). The second method takes no more memory and only a few more divisions or multiplications.

Consider the algorithm in figure 3.4, notice how in the original color mixing paradigm, via a series of linear interpolations where `maxTraceDepth` equals six, the first color to mix at 50% in the first linear interpolation ends up only contributing 3.125% to the overall image.

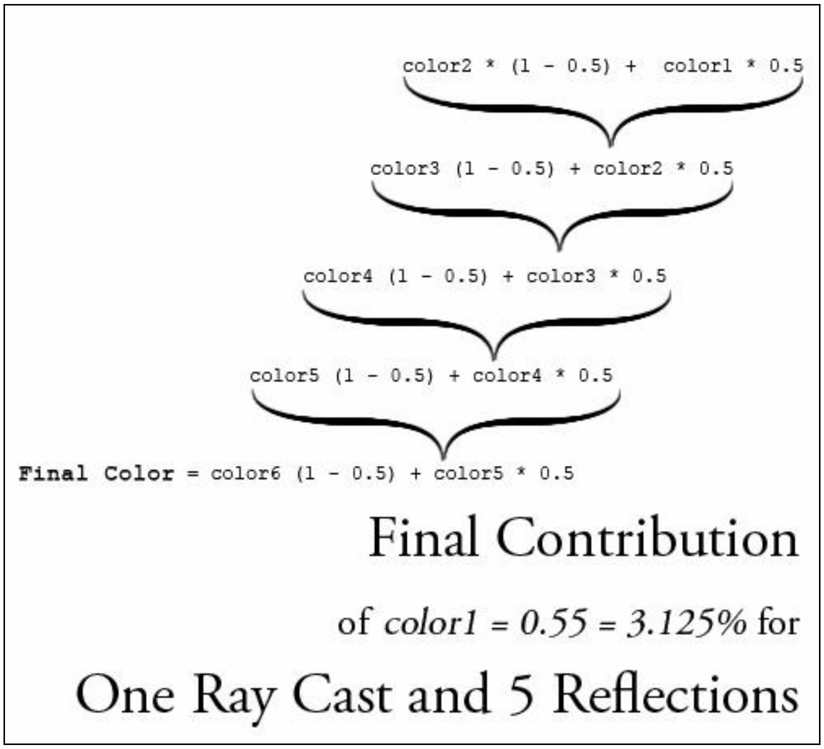

color2 * (1 - 0.5) + color1 * 0.5
color3 (1 - 0.5) + color2 * 0.5
color4 (1 - 0.5) + color3 * 0.5
color5 (1 - 0.5) + color4 * 0.5
Final Color = color6 (1 - 0.5) + color5 * 0.5
Final Contribution
of color1 = 0.55 = 3.125% for
One Ray Cast and 5 Reflections

Figure 3.5 – A series of linear interpolations decreases a color's final effect; note `color1`.

In the CUDA implementation of forward ray color mixing, this means that the result of the first ray cast, where `maxTraceDepth` equals six, will have an overall influence of 3.125% on the final image. This is backwards. We need a predictive summation series that will properly calculate the interpolation values for all colors without having to change ray color mixing directions. Through the equation below a solid formula can be pieced together that describes how much a color should mix into the final image, where x is the coefficient of reflection.

$$color_{final} = c1(1-x)x^0 + c2(1-x)x^1 + c3(1-x)x^2 + c4(1-x)x^3 + c5(1-x)x^4 + c6x^5 \quad (15)$$

Notice particularly the power coefficients in each color mixing value. These values are equal to `1 – current reflection depth`; there is a value already defined as such in the main rendering function. Notice the `rayCount` value defined on the first page of Appendix I and iterated in the second page of Appendix I. The final check will be when `rayCount = 1 – maxDepth`, then power coefficient is equal to `1 – rayCount`.

3.3.2.3.2. Do While Loop

As used in the work of Allgyer (2008), one method of recreating the effect of recursive functionality, without using recursive functions, is to use the `DO WHILE` loop. Appendix I gives the code usage of the DO WHILE loop. Each cycle of the loop iterates a counter value and goes through the render loop again, until the counter value reaches the max limit as defined by the `maxTraceDepth` value.

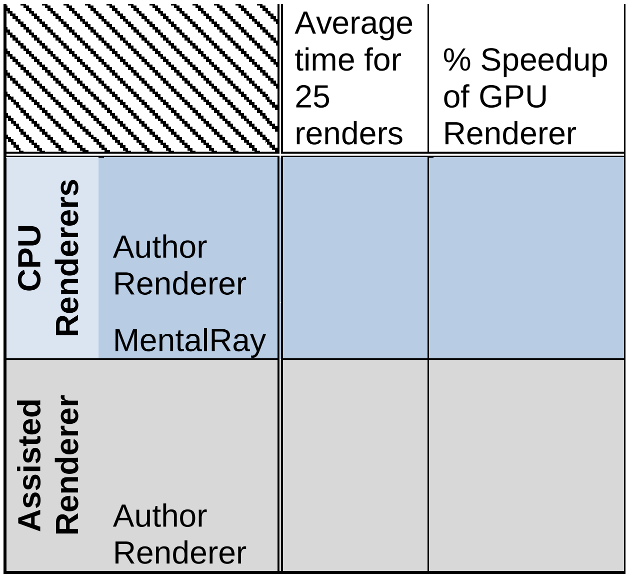
3.4. Research Framework

This research on ray-tracing will present three different renderers and test their respective render times in four main categories. The purpose of these tests is to determine which renderer completes different tests faster. It is the hypothesis of this research that the GPU assisted renderer will out-perform the two CPU renderers. The three renderers that will be tested are:

- Mental Ray
- CPU Ray-Tracer written by Author
- GPU Ray-Tracer written by Author

Each Renderer will be tested for time to complete a render. Each renderer generated 25 frames and the average time was computed. The averaged times for each renderer were compared.

The variable in this study that will be tested for is time to complete each task. All three renderers will be tested. The following figure represents the nature of the study:



		Average time for 25 renders	% Speedup of GPU Renderer
CPU Renderers	Author Renderer		
	MentalRay		
Assisted Renderer	Author Renderer		

Figure 3.6 – Diagram of Testable Rendering Tests

All tests will take place on one computer with all software installed on it. The purpose of this is to limit the introduction of confounding variables through different hardware configurations.

3.4.1. Hypotheses

The testing of GPU rendering speed timing was tested using two hypotheses. For each hypothesis there exist a default and an alternate hypothesis. The purpose of this testing is to prove both alternate hypotheses.

The first hypothesis tests if the author written GPU renderer out-performs the author written CPU renderer. This is designed as a litmus test. This is the test that should be passed very early in the implementation stages of the GPU renderer. If this test cannot be passed, there is no reasonable expectation that the second hypothesis would be passed.

That makes the second hypothesis a strict test of increases in rendering speed. Proving the alternate for hypothesis two is the main goal of this research and will provide statistically significant results for the validity of rendering with massively parallel systems.

Table 3.3 – *Table of testing hypotheses*

	Hypothesis 1	**Hypothesis 2**
Default Hypothesis	H_{o1} – There is no noticeable increased speed of rendering with GPU rendering versus CPU rendering	H_{o2} – There is no noticeable increase in rendering speed when rendering with GPU versus rendering on CPU with a production quality renderer
Alternate Hypothesis	$H_{\alpha 1}$ – There exists a statistically significant increase in rendering speed when rendering with GPU versus rendering on the CPU	$H_{\alpha 2}$ – There exists a statistically significant increase in rendering speed when rendering with GPU versus rendering on the CPU with a production quality renderer

The testing method used to analyze the data will be a comparison of comparison of render times.

3.4.2. Pre-Testing Expectation of Hypotheses

To gauge a level of success or failure, criteria for success will be established for each hypothesis.

3.4.2.1. Proving the Alternative Hypothesis 1

With respect to the cost of rendering as measured in time, any increase in speed is of great benefit, even when the increase in speed is measured 10’s of percentage points. So a renderer that measures 50% faster than previous renderers is considered a marked improvement and worthy of financial investment. A renderer that can improve render times by whole multiples would be welcome in the computer graphics industry.

A success would be an increase of rendering speed where the new renderer is two to four times faster. Therefore, to establish the alternative hypothesis 1 as accurate, the author written GPU will have to measure 4 times faster than the author written CPU renderer.

3.4.2.2. Proving the Alternative Hypothesis 2

Gaining rendering speed against a highly respected, professionally developed and professionally used, renderer is the more stringent test of speed improvements for the author written GPU renderer. The Mental Ray renderer supports many features that both author written renderers do not support. The largest advantage the Mental Ray renderer has over the author written renderers is an implemented spatial partitioning system. Mental Ray uses BSPs to divide the space to decrease render times. The BSP settings are as follows:

Table 3.4 - *Mental Ray BSP settings to increase render speeds for scene1*

BSP Type	Regular BSP
BSP Size	10
BSP Depth	60

Changing the *BSP Depth* value, from default, reduced the final render to 3/4th the original render time. The fastest computer to render the Mental Ray scene dropped the render time from four seconds to three seconds. In addition to the BSP algorithms, Mental Ray also supports multi-threaded rendering on the CPU.

When weighed against these advanced spatial partitioning features and any number of subtle tricks of logic, math or algorithm, I was unsure how much improvement the GPU renderer would have. In order to prove the speed effectiveness of the GPU renderer, a success in the second hypothesis will be measured to be at least 20% faster than the Mental Ray renderer. This percent increase was chosen because any amount less would not be enough of an increase to warrant a financial investment in new software. From the author's experience in industry, this is believed to be a minimal threshold.

3.5. Test Conditions

Four 3D scenes were created using to test the two hypotheses. These two scenes are divided into two groups: (A) high reflection with low object count, (B) no reflection with hi object count.

The scene in group A was rendered with these settings:

- 48 spheres
- 10 triangles for scene extent
- Phong shader
- Two point lights
- Ray trace shadows for each light
- Max Trace Depth of 10

There are three scenes in group B. These scenes were generated through the creation of a set number of shapes with random locations in the scene. Group B was rendered with these settings:

- 900, 10000, or 30000 objects
- 10 triangles for scene extent
- Phong Shader
- Two point lights

- Ray trace shadows for each light
- Max Trace Depth of 1

The number of objects for scene in the group B category was decided on by a Windows operating system feature. This feature times out any GPU process that takes longer than 2 – 3 seconds to complete.

Table 3.5 – *Attributes of test scenes from Group A and Group B*

	Group A	Group B		
Object Count	58	910	10,010	30,010
Max Trace Depth	10	1	1	1

Each renderer will be time tested using the same scene data.

3.6. Chapter Summary

The testing methodologies described in this chapter are four tests of time. Each timed test will test four renderers. The four renderers are divided into two categories: CPU and GPU Assisted. The CPU category of renderers is further subdivided into two more categories: Production and Author Written. The MentalRay renderer is of the production category. The second main category is the GPU Assisted category. This is also an author written renderer. The GPU Assisted renderer will be tested, for time efficiency, against the two CPU renderers. The tests are three in count and test each renderers speed in rendering the similar data.

CHAPTER 4. RESULTS

This chapter will discuss the timed results of the three renderers and their outcomes with respect to the two hypotheses: GPU rendering times compared against author written CPU renderer and production quality Mental Ray renderer.

4.1. Author Written CPU Renderer Results

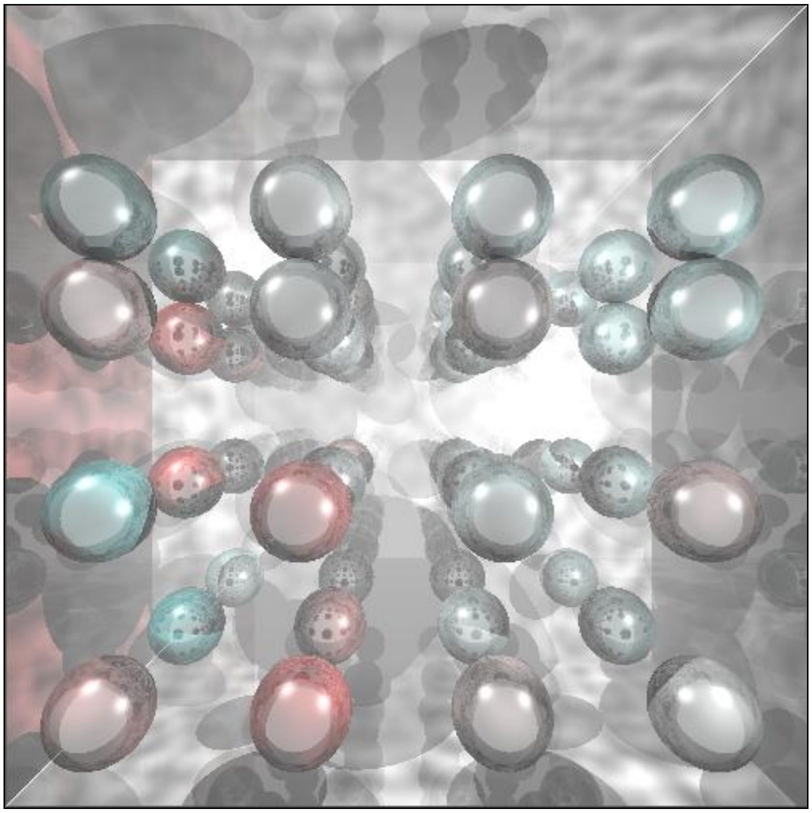


Figure 4.1 - Rendered Image from author written CPU renderer

The CPU renderer written by the author creates images with high visual quality. As is expected, the render times for the author written CPU renderer are slower than the Mental ray renderer. The fastest CPU render times came from a Core2 Duo 3GHz, 4GB RAM with the slowest render times coming from a Dual Xeon 3GHz processors.

4.2. Mental Ray Renderer Results

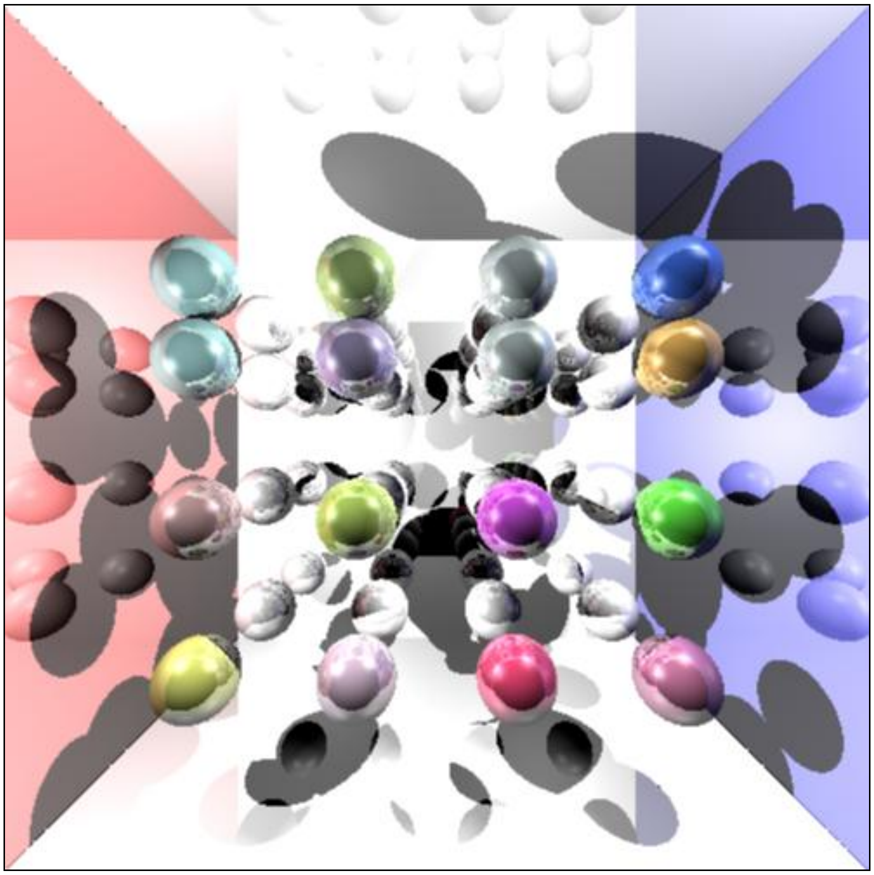


Figure 4.2 – Render results from Mental Ray

The rendering results from Mental Ray proved to be much faster than the author written CPU renderer.

4.3. Results: Scene 1

The results of this study have surpassed the expectations greatly. Both alternative hypothesis were proven true by larger percentages than originally expected. These results apply to the scene defined in Group A: high reflection with low object count. This table breaks down the comparison:

Table 4.1 – *Comparison of rendering times for GPU, CPU and Mental Ray renderings. Units of time are given in milliseconds.*

Scene 1 Render Times					
	GPU	CPU	**CPU/GPU**	Mental Ray	**Mental Ray /GPU**
Average	988.94	75246.33	**76.08786**	6007.067	**6.074248**
Fastest Times	244.84	28773.92	**117.5213**	2423.2	**9.897076**
Slowest Times	1900.76	158151.8	**83.20453**	11256.8	**5.922263**

The results of the data show significant increases in rendering speed with respect to the GPU over both the author written CPU renderer and even the Mental Ray renderer. The complete timing data is presented in Appendix J.

4.3.1. Strict and Favorable Timing Comparisons

Tables 4.2 and 4.3 demonstrate the clear advantage in rendering speed the GPU has over both the author written CPU renderer and the production ready Mental Ray renderer. In the strictest comparison of times, Table 4.2, the GPU renderer is 27% faster than Mental Ray and 1400% faster than the author written CPU renderer.

Table 4.2 – *Strict timing comparisons to GPU*

Slowest GPU to Fastest CPU and Fastest Mental Ray *time measured in milliseconds				
GPU	CPU	**CPU/GPU**	MR	**MR / GPU**
1900.76	28773.92	**15.13811**	2423.2	**1.274858**

In a more favorable comparison of times, the fastest GPU times are compared to the slowest CPU and Mental Ray renders. Here, the advantages in speed of GPU rendering are more prevalent. The GPU is 64,493% faster than the CPU and 4,497% faster than the slowest Mental Ray renderer. Table 4.3 shows the data.

Table 4.3 – *Favorable timing comparisons to GPU*

Fastest GPU to Slowest CPU and Slowest Mental Ray *time measured in milliseconds				
GPU	CPU	**CPU/GPU**	MR	**MR / GPU**
244.84	158151.8	**645.9394**	11256.8	**45.97615**

4.4. Results: Scenes 2 – 4

Scenes 2 through 4 create a series of scenes with ever increasing object counts. These 3D scenes are members of Group B: no reflection with high object count. Figure 4.3 illustrates the render times for the three comparable scenes. The chart shows the times for three CPUs and two GPUs. The CPUs are two I7 processors and one Core2Quad. The two GPUs timed are a GTX 275 and a GTS 250. Each hardware device rendered the 900, 10,000 and 30,000 object scenes.

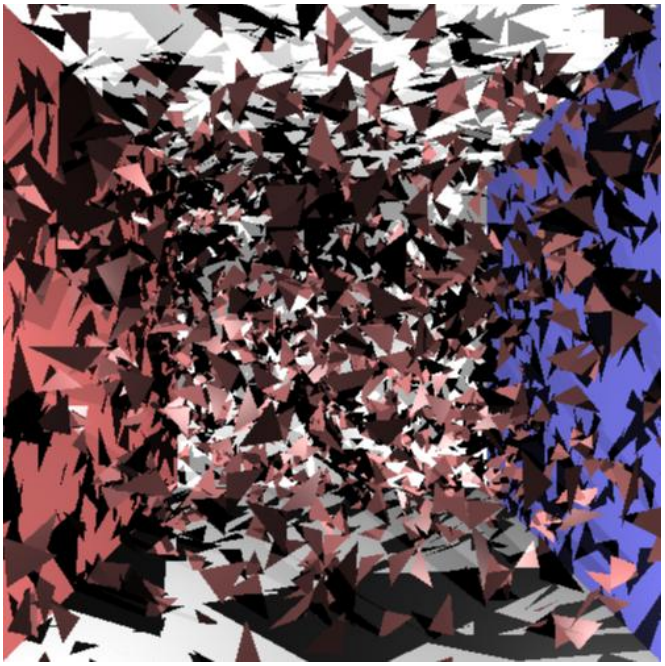


Figure 4.3 – Mental Ray render 900 Triangles, each as a separate pbject

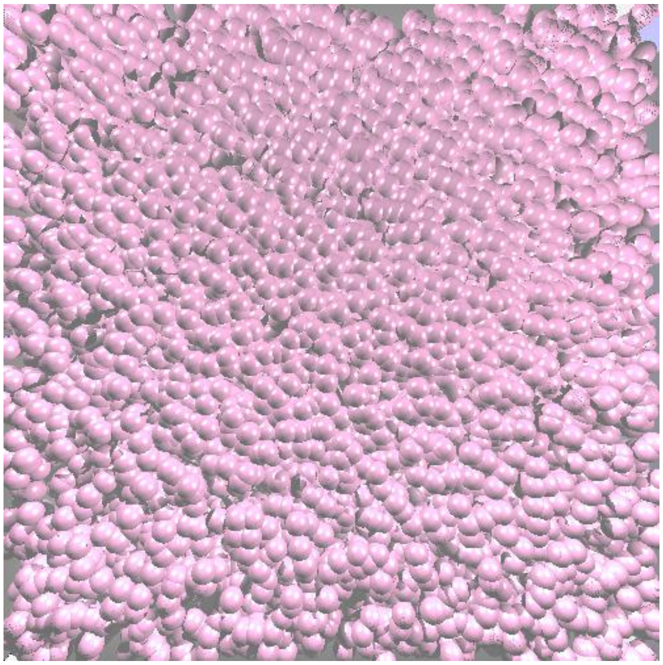
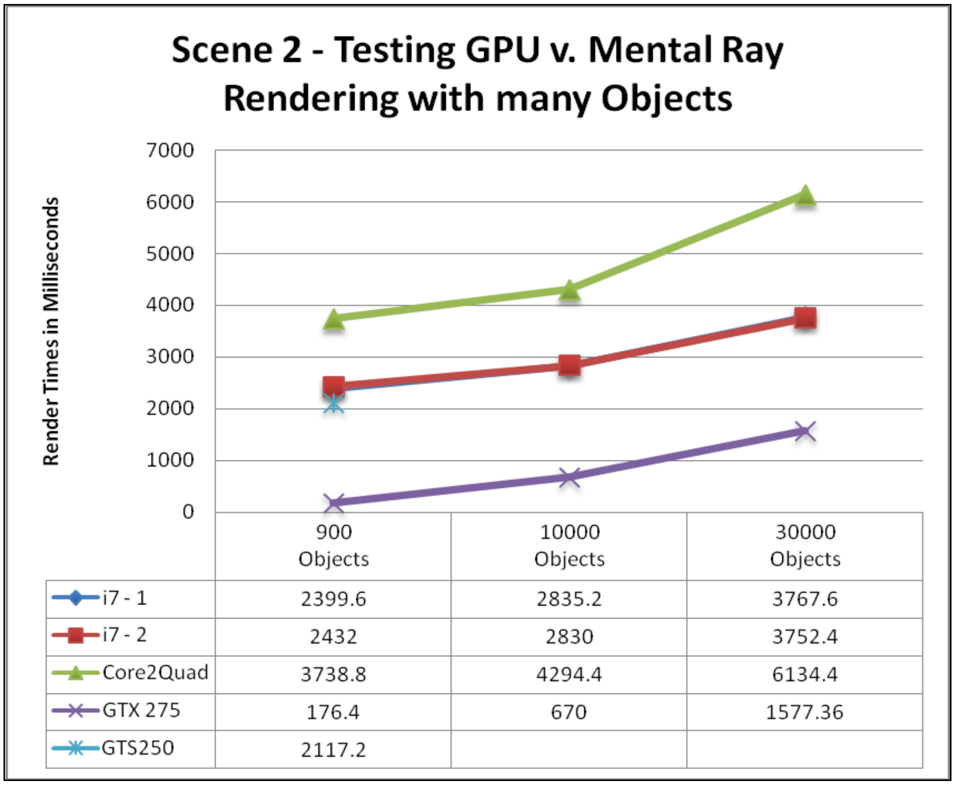


Figure 4.4 – The GPU renders 30,000 randomly placed spheres

The comparison of GPU speed to Mental Ray spatial partitioning is presented in figures 4.5 and 4.6. The first figure shows the render times of the GTX 275 out-performing all Mental Ray renders of similar scenes with the same object counts.

Upon inspection of figure 4.6, a new phenomenon is illustrated. For the same rendered scenes and the same render timing, as seen in figure 4.5, the speed efficiency of the GPU decreases as the object count increases. The data is calculated by dividing the Mental Ray render time by the GPU render time. This shows how many times the GPU can render the same frame by the time Mental Ray can render one complete frame. In the first scene, of 900 objects, the GPU can render one frame almost 14 times before Mental Ray can render one frame. As the number of objects in the scene increases, the comparative GPU rendering performance decreases.



	900 Objects	10000 Objects	30000 Objects
i7 - 1	2399.6	2835.2	3767.6
i7 - 2	2432	2830	3752.4
Core2Quad	3738.8	4294.4	6134.4
GTX 275	176.4	670	1577.36
GTS250	2117.2		

Figure 4.5 – GPU render times versus Mental Ray (CPU) render times for scenes with increasing object counts

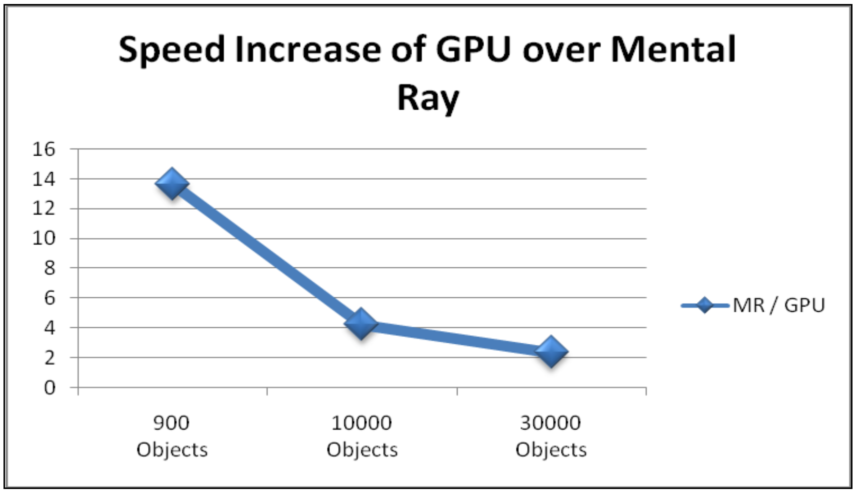


Figure 4.6 – Chart of declining GPU renderer performance with increasing object counts

4.5. Hypothesis 1 Results

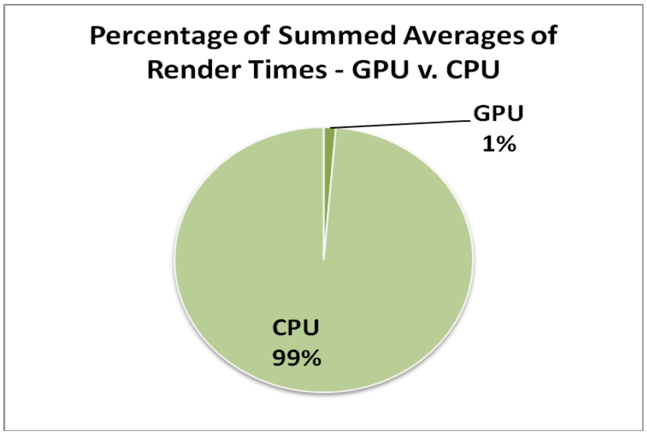


Figure 4.7 - Averaged timing comparison of GPU v. CPU

It is not a surprise that the GPU renderer would be faster than the CPU renderer. What was surprising was the level of speed increase. Looking back at Table 4.1, when comparing the slowest and fastest rendering times for both renderers, the GPU outperforms drastically. The average GPU render time is 75 times faster than the average CPU render time. The fastest GPU rendering time is 116 time faster than the fastest CPU render time.

4.6. Hypothesis 2 Results

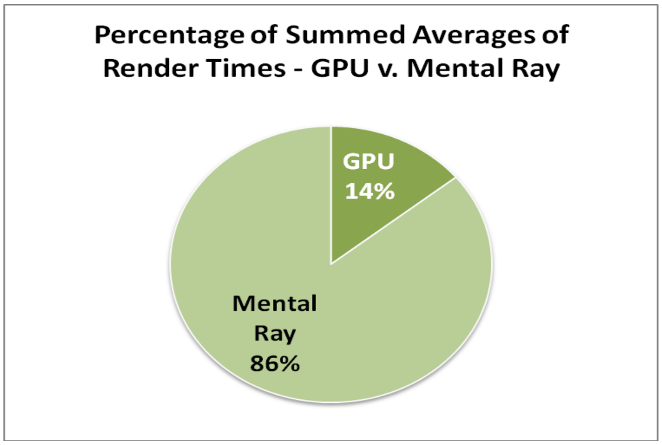


Figure 4.8 - Averaged timing Comparison of GPU v. Mental Ray

The alternative hypothesis two is proven based on the data shown Tables 4.1 and 4.2. The slowest GPU average render time is 27% faster than the fastest Mental Ray rendering speed. Of the time it takes to render both the average render times of GPU and Mental Ray, the GPU takes 14% of the overall render time; see Figure 4.8. Figure 4.6 shows that the GPU renderer, even at 30,000 objects, is more than 2x faster than Mental Ray.

CHAPTER 5. CONCLUSIONS AND FUTURE WORK

5.1. GPU Renderer Design

The original research plan was to leverage the power of GPU paralleled architecture to calculate the most mathematically intense functions. These functions are the intersection check for both viewing rays and shadow rays. They are calculated in a brute force method with no spatial partitioning or scene hierarchy, nor a predictive intersection algorithm for shadow checking.

There are two conditions that slow down GPU efficiency: conditional statements and host to device memory transfers. The first condition is not being dealt with at this time. With respect to solving the second condition, finding a stream-lined method to limit the number of data transfers between the host and device created a major shift in the development plan for the GPU renderer. The easiest way to limit the number of data transfers is to send data only once, to the device, to describe the scene, and then to send the image back to the host when rendering is ended. On a per frame basis, this generates only two mass data transfers. Were the GPU used only for ray – object intersection checks, there would exist at least two intersection tests (one viewing and one shadow intersection) therefore requiring 4 data transfers. However, a scene with multiple lights and setting a max trace depth above 4 or more would create numerous data transfers per frame. This would slow the rendering process down significantly.

The best method, to limit unnecessary data transfer, was to perform all shading and rendering calculations on the GPU. Now data transfer is a constant amount per frame, regardless of the number of lights or reflection bounces.

5.2. Results

The data presented in the timed results of GPU, CPU and Mental Ray renderers shows, without a doubt, that rendering on the GPU increases rendering speeds dramatically. The highly paralleled nature of current GPUs allows for extraordinary increases in rendering speed. This is true even when compared to the added algorithms for rendering efficiency apparent in Mental Ray.

5.2.1. Massive Parallelism versus Spatial Partitioning

Figure 4.4 shows an interesting trend in GPU vs. Mental Ray rendering performance. While the GPU outperforms Mental Ray in all three scenes (900 – 10,000 – 30,000), the degree by which the GPU outperforms decreases as the number of objects increases. This trend shows that, at some point of increased object count, the Mental Ray rendering speed will converge with the GPU render speeds. At some point further in the graph, Mental Ray may even outperform the GPU renderer. These results show that while a massively parallel renderer has definite timing advantages, at some point the efficiency of spatial partitioning approaches similar timing results. This finding illustrates the need to implement spatial partitioning on the GPU renderer.

5.2.2. General Discussion of Results

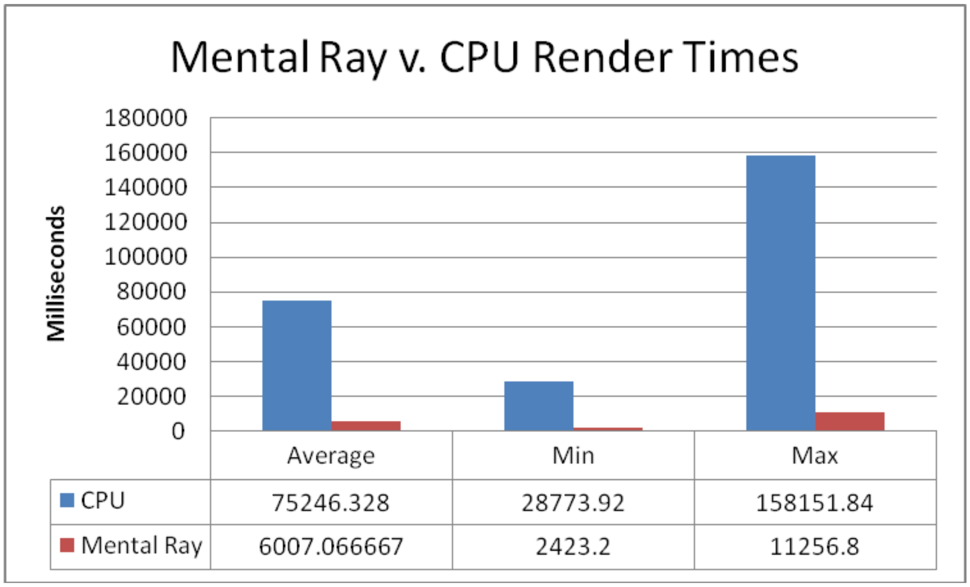
The Mental Ray renderer has added functionality for spatial partitioning through the use of BSP trees, multi-threading on the CPU, and any number of subtle or hidden checks for rendering efficiency. In addition to these efficiencies, the Mental Ray renderer also leverages the strong logic capabilities of CPUs

With respect to GPUs, the architecture favors brute force, stream calculation over logic operations. Conditional statements and operations slow down GPU performance significantly. Another obstacle GPUs face is transmitting scene and image data from the CPU to the GPU and data flow in reverse.

Despite the time to pass data over the bus, to and from host and device, and the lack of spatial partitioning and intersection prediction, the amount of processors and the streaming nature of current GPUs cause them to out-perform a production ready and highly modified renderer.

5.3. Future Work

Future work on the GPU renderer should first be aimed at increasing rendering speed with large data sets. This would require the addition of a scene partitioning system and would therefore limit the practice of brute force rendering via ray – object intersections. Using BVHs will greatly increase rendering speed. To get a sense of the gain in rendering speeds, compare the speed of the Mental Ray renderer versus the author written CPU renderer.



	Average	Min	Max
🟦 CPU	75246.328	28773.92	158151.84
🟥 Mental Ray	6007.066667	2423.2	11256.8

Figure 5.1 – Comparison of render times between two CPU renderers. One is a brute force renderer and the Mental Ray renderer has added efficiency algorithms

In addition to BVHs, creating a CUDA struct or class that supports a hash table type of object storage, per pixel, of intersection data would allow for easier sorting while maintaining low access speeds – similar to the STL::map container. This would be an important step in calculating objects with various levels of semi-transparency.

LIST OF REFERENCES

LIST OF REFERENCES

Agarwal, P. K., & Sharir, M. (1998). Efficient algorithms for geometric optimization. *ACM Comput. Surv.*, *30*(4), 412-458. doi: 10.1145/299917.299918.

Alberti, L. B. (1991). *On painting*. (M. Kemp, Ed., C. Grayson, Tran.). Penguin Classics. (Original work published in 1435).

Akenine-Moller, T., Haines, E., & Hoffman, N. (2008). *Real-time rendering, third edition* (3rd ed.). AK Peters

Allgyer, M. (2008, April 12). Real-time Ray Tracing using CUDA. Retrieved from HTTP://WWW.HANDSFREEPROGRAMMING.COM/MASTERS/.

Angel, E. (2008). *Interactive computer graphics: A top-down approach using OpenGL* (5th ed.). Addison Wesley.

Apodaca, A. A., & Gritz, L. (1999). *Advanced RenderMan: Creating CGI for motion pictures* (1st ed.). Morgan Kaufmann.

Bailey, M., & Cunningham, S. (2008). Introduction to CG shaders (pp. 1-126). Singapore: ACM. doi: 10.1145/1508044.1508069.

Birn, J. (2000). *Digital lighting & rendering* (1st ed., p. 304). New Riders Press.

Blinn, J. F. (1977). Models of light reflection for computer synthesized pictures. *SIGGRAPH Comput. Graph.*, *11*(2), 192-198. doi: 10.1145/965141.563893.

Blinn, J. F. (1978). Simulation of wrinkled surfaces. *SIGGRAPH Comput. Graph.*, *12*(3), 286-292. doi: 10.1145/965139.507101.

Blinn, J. F., & Newell, M. E. (1976). Texture and reflection in computer generated images. *Commun. ACM*, *19*(10), 542-547. doi: 10.1145/360349.360353.

Board, O. A. R., Shreiner, D., Woo, M., Neider, J., & Davis, T. (2007). *OpenGL(R) programming guide: The official guide to learning OpenGL(R), Version 2.1* (6th ed.). Addison-Wesley Professional.

Carr, N. A., Hoberock, J., Crane, K., & Hart, J. C. (2006). Fast GPU ray tracing of dynamic meshes using geometry images. In *Proceedings of Graphics Interface 2006* (pp. 203-209). Quebec, Canada: Canadian Information Processing Society. Retrieved November 19, 2009, from http://portal.acm.org/citation.cfm?id=1143079.1143113&coll=GUIDE&dl=GUIDE&CFID=63865150&CFTOKEN=17787473.

Chen, C., & Liu, D. S. (2007). Use of hardware Z-buffered rasterization to accelerate ray tracing. In *Proceedings of the 2007 ACM symposium on Applied computing* (pp. 1046-1050). Seoul, Korea: ACM. doi: 10.1145/1244002.1244231.

Cook, R. L. (1984). Shade trees (pp. 223-231). ACM. doi: 10.1145/800031.808602.

Cook, R. L., & Torrance, K. E. (1981). A reflectance model for computer graphics (pp. 307-316). Dallas, Texas, United States: ACM. doi: 10.1145/800224.806819.

Cox, G., Máximo, A., Bentes, C., & Farias, R. (2009). Irregular Grid Raycasting Implementation on the Cell Broadband Engine. *2009 21st International Symposium on Computer Architecture and High Performance Computing*, 93 - 100.

Dempski, K. (2004). *Advanced lighting and materials with shaders*. Wordware Publishing, Inc.

Dunn, F., & Parberry, I. (2002). *3D math primer for graphics and game development* (p. 429). Wordware Publishing.

Dutre, P., Bala, K., & Bekaert, P. (2006). *Advanced global illumination* (2nd ed.). AK Peters.

Ericson, C. (2005). *Real-time collision detection*. Morgan Kaufmann.

Fernando, R., & Kilgard, M. J. (2003). *The CG tutorial: The definitive guide to programmable real-time graphics*. Addison-Wesley Professional.

Fischer, J., Bartz, D., & Straβer, W. (2005). Artistic reality: fast brush stroke stylization for augmented reality (pp. 155-158). Monterey, CA, USA: ACM. doi: 10.1145/1101616.1101649.

Friedrich, H., Günther, J., Dietrich, A., Scherbaum, M., Seidel, H., & Slusallek, P. (2006). Exploring the use of ray tracing for future games (pp. 41-50). Boston, Massachusetts: ACM. doi: 10.1145/1183316.1183323.

Funkhouser, T. (2002, Fall). Monte Carlo Integration for Image Synthesis. Princeton University, COS 526. Retrieved from http://www.cs.princeton.edu/courses/archive/fall02/cs526/lectures/montecarlo.pdf.

Gaddis, T., Walters, J., & Muganda, G. (2007). *Starting out with c++* (6th ed., p. 1122). Boston: Addison-Wesley.

Goral, C. M., Torrance, K. E., Greenberg, D. P., & Battaile, B. (1984). Modeling the interaction of light between diffuse surfaces. *SIGGRAPH Comput. Graph.*, *18*(3), 213-222. doi: 10.1145/964965.808601.

Gottschalk, S., Lin, M. C., & Manocha, D. (1996). OBBTree: a hierarchical structure for rapid interference detection (pp. 171-180). ACM. doi: 10.1145/237170.237244.

Gu, X., Gortler, S. J., & Hoppe, H. (2002). Geometry images. *ACM Trans. Graph.*, *21*(3), 355-361. doi: 10.1145/566654.566589.

Guenter, B., Knoblock, T. B., & Ruf, E. (1995). Specializing shaders (pp. 343-350). ACM. doi: 10.1145/218380.218470.

He, X. D., Torrance, K. E., Sillion, F. X., & Greenberg, D. P. (1991). A comprehensive physical model for light reflection. *SIGGRAPH Comput. Graph.*, *25*(4), 175-186. doi: 10.1145/127719.122738.

Heidrich, W., Slusallek, P., & Seidel, H. (1998). Sampling procedural shaders using affine arithmetic. *ACM Trans. Graph.*, *17*(3), 158-176. doi: 10.1145/285857.285859.

Hill, F. S. (2000). *Computer graphics using OpenGL* (2nd ed.). Prentice Hall.

Hubbard, P. M. (1996). Approximating polyhedra with spheres for time-critical collision detection. *ACM Trans. Graph.*, *15*(3), 179-210. doi: 10.1145/231731.231732.

Jacobs, K., Ward, G., & Loscos, C. (2005). Automatic HDRI generation of dynamic environments (p. 43). Los Angeles, California: ACM. doi: 10.1145/1187112.1187163.

Jarosz, W., Jensen, H. W., & Donner, C. (2008). Advanced global illumination using photon mapping (pp. 1-112). Los Angeles, California: ACM. doi: 10.1145/1401132.1401136.

Jensen, H. W. (1997). Rendering Caustics on Non-Lambertian Surfaces. *Computer Graphics Forum*, *16*(1), 57-64. doi: 10.1111/1467-8659.329000.

Jensen, H. W., & Christensen, P. (2007). High quality rendering using ray tracing and photon mapping (p. 1). San Diego, California: ACM. doi: 10.1145/1281500.1281593.

Kajiya, J. T. (1986). The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques* (pp. 143-150). ACM. doi: 10.1145/15922.15902.

Kass, M., Lefohn, A., & Owens, J. (n.d.). Interactive Depth of Field. Retrieved April 6, 2009, from http://graphics.pixar.com/library/DepthOfField/.

Keller, A. (1997). Instant radiosity (pp. 49-56). ACM Press/Addison-Wesley Publishing Co. doi: 10.1145/258734.258769.

Klein, J., & Zachmann, G. (2003). Time-critical collision detection using an average-case approach (pp. 22-31). Osaka, Japan: ACM. doi: 10.1145/1008653.1008660.

Lafortune, E. P. F., Foo, S., Torrance, K. E., & Greenberg, D. P. (1997). Non-linear approximation of reflectance functions (pp. 117-126). ACM Press/Addison-Wesley Publishing Co. doi: 10.1145/258734.258801.

Lawrence, J., Rusinkiewicz, S., & Ramamoorthi, R. (2004). Efficient BRDF importance sampling using a factored representation (pp. 496-505). Los Angeles, California: ACM. doi: 10.1145/1186562.1015751.

Luong, T., Seth, A., Klein, A., & Lawrence, J. (2005). Isoluminant color picking for non-photorealistic rendering (pp. 233-240). Victoria, British Columbia: Canadian Human-Computer Communications Society. Retrieved March 27, 2009, from http://portal.acm.org/citation.cfm?id=1089508.1089547&coll=ACM&dl=ACM&CFID=28592590&CFTOKEN=72825349.

Marschner, S. R., Westin, S. H., Lafortune, E. P. F., & Torrance, K. E. (2000). Image-Based Bidirectional Reflectance Distribution Function Measurement. *Applied Optics*, *39*(16), 2592-2600. doi: 10.1364/AO.39.002592.

Meyers, S. (2005). *Effective c++: 55 specific ways to improve your programs and designs (3rd Edition)* (3rd ed.). Addison-Wesley Professional.

Meyers, S. (2001). *Effective STL: 50 specific ways to improve your use of the standard template library*. Addison-Wesley Professional.

Nettle, P. (1999, May 20). Radiosity In English. *Radiosity in English*. Retrieved from http://www.paulnettle.com/pub/FluidStudios/Radiosity/Radiosity_in_English.pdf.

Nicodemus, F. E., Richmond, J. C., Hsia, J. J., Ginsburg, I. W., & Limperis, T. (1977). *Geometrical considerations and nomenclature for reflectance*. National Bureau of Standards monograph (p. 67). Washington: Department of Commerce, National Bureau of Standards.

NVidia Corporation. (2009, Aug 26). NVIDIA_CUDA_Programming_Guide_2.3 (application/pdf Object). NVidia Corporation. Retrieved January 6, 2010, from http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf.

NVidia Corporation. (2010, Jan 19.). NVIDIA® OptiX™ ray tracing engine. *NVIDIA OptiX ray tracing engine*. Retrieved April 29, 2010, from http://developer.nvidia.com/object/optix-home.html.

Parker, S., Martin, W., Sloan, P. J., Shirley, P., Smits, B., & Hansen, C. (2005). Interactive ray tracing (p. 12). Los Angeles, California: ACM. doi: 10.1145/1198555.1198751.

Pharr, M., & Fernando, R. (2005). *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional.

Pharr, M., & Humphreys, G. (2004). *Physically based rendering : From theory to implementation (the Morgan Kaufmann series in interactive 3D technology)*. Morgan Kaufmann.

Phong, B. T. (1975). Illumination for computer generated pictures. *Commun. ACM*, *18*(6), 311-317. doi: 10.1145/360825.360839.

Ragan-Kelley, J., & Massachusetts Institute of Technology. (2007). *The lightspeed automatic interactive lighting preview system*.

Rushmeier, H. (2008). Input for participating media. In *ACM SIGGRAPH 2008 classes* (pp. 1-24). Los Angeles, California: ACM. doi: 10.1145/1401132.1401141.

Schwan. (1994). *Raytracing on the macintosh book*. The Great State of Texas: FT Prentice Hall.

Shah, A., Ritter, J., & Gronsky, S. (n.d.). Fast, Soft Reflections Using Radiance Caches. Retrieved April 6, 2009, from http://graphics.pixar.com/library/SoftReflections/index.html.

Shirley, P., Ashikhmin, M., Gleicher, M., Marschner, S., Reinhard, E., Sung, K., et al. (2005). *Fundamentals of Computer Graphics, Second Ed.* (2nd ed.). A K Peters, Ltd.

Shirley, P., & Morley, R. K. (2003). *Realistic ray tracing* (2nd ed., p. 225). A K Peters.

Shirley, P., Wang, C., & Zimmerman, K. (1996). Monte Carlo techniques for direct lighting calculations. *ACM Trans. Graph.*, *15*(1), 1-36. doi: 10.1145/226150.226151.

Shreiner, D., & Group, T. K. O. A. W. (2009). *OpenGL programming guide: The official guide to learning OpenGL, Versions 3.0 and 3.1* (7th ed.). Addison-Wesley Professional.

Speer, L. R. (1992a). An updated cross-indexed guide to the ray-tracing literature. *SIGGRAPH Comput. Graph.*, *26*(1), 41-72. doi: 10.1145/142403.142405.

Speer, L. R. (1992b). An updated cross-indexed guide to the ray-tracing literature. *SIGGRAPH Comput. Graph.*, *26*(1), 41-72. doi: 10.1145/142403.142405.

Tabellion, E., & Lamorlette, A. (2008). An approximate global illumination system for computer generated films. In *ACM SIGGRAPH 2008 classes* (pp. 1-8). Los Angeles, California: ACM. doi: 10.1145/1401132.1401227.

Wald, I., Kollig, T., Benthin, C., Keller, A., & Slusallek, P. (2002). Interactive global illumination using fast ray tracing (pp. 15-24). Pisa, Italy: Eurographics Association. Retrieved March 27, 2009, from http://portal.acm.org/citation.cfm?id=581896.581899&coll=GUIDE&dl=GUIDE&CFID=28594501&CFTOKEN=62038515.

Ward, G., Reinhard, E., & Debevec, P. (2008). High dynamic range imaging \& image-based lighting (pp. 1-137). Los Angeles, California: ACM. doi: 10.1145/1401132.1401170.

Whitted, T. (1980). An improved illumination model for shaded display. *Commun. ACM*, *23*(6), 343-349. doi: 10.1145/358876.358882.

Woop, S., Schmittler, J., & Slusallek, P. (2005). RPU: a programmable ray processing unit for realtime ray tracing. *ACM Trans. Graph.*, *24*(3), 434-444. doi: 10.1145/1073204.1073211.

Xie, F., Tabellion, E., & Pearce, A. (2007). Soft Shadows by Ray Tracing Multilayer Transparent Shadow Maps. Presented at the Eurographics Symposium on Rendering.

Zachmann, G. (2002). Minimal hierarchical collision detection (pp. 121-128). Hong Kong, China: ACM. doi: 10.1145/585740.585761.

APPENDICES

Appendix A. CUDA Struct Declarations and Data Contents

Table A.1 – *Host and Device Global Memory Pairings and Their Use*

Struct Name	Data Contents	Array Size *N = number of pixels* *M = number of objects*
curayCam	**float3 pos** **float3 dir** **float3 up**	**1**
curayLight	**float m_intensity** **float3 m_diffuse** **float3 m_specular** **float3 m_pos**	**1 – n**
caraySphere	**float radius** **float3 m_pos** **float3 m_color** **int matID**	**0 – n**
curayTri	**float3 p0, p1, p2** **float3 m_normal** **float3 m_color** **float3 ABC, DEF** **int matID** **DefineABCDEF()** **DefineNormal()**	**0 – n**
curayFog	**float3 m_color** **float3 m_IntMinMax**	**0 - 1**
curayRay	**float3 m_dir, m_pos**	**1, Defined in Kernel**
curayRec	**float t** **int id**	**1**
curayFrameBuffer	**int id, ObjType** **float t** **float3 normal, point**	**N**
curayMat	**float3 color, spec** **float reflectivity, shininess**	**0 – N** **or** **0 – M**

Appendix B. CUDA Global Memory Variable Pairs

Table B.1 – *Data Contents of New Struct Data Types*

Host Variables	Device Variables	Use
`curaySphere* h_Spheres=NULL;`	`curaySphere* d_Spheres=NULL;`	Stores *n* amount of spheres where *n* is defined by CPU
`curayTri* h_Tris=NULL;`	`curayTri* d_Tris=NULL;`	Stores *n* amount of triangles where *n* is defined by CPU
`curayLight* h_Lights=NULL;`	`curayLight* d_Lights=NULL;`	Stores *n* amount of lights where *n* is defined by CPU
`curayCam* h_Camera=NULL;`	`curayCam* d_Camera=NULL;`	Stores camera data
`curayFog* h_Fog=NULL;`	`curayFog* d_Fog=NULL;`	Stores fog data
`curayFrameBuffer* h_FB=NULL;`	`curayFrameBuffer* d_FB=NULL;`	FrameBuffer stores data for the closest object *id* and *ObjType*, distance to nearest intersection, normal and position of object of nearest intersection
`curayMat* h_Mat=NULL;`	`curayMat* d_Mat=NULL;`	Material description. Note: Because of the thread per pixel nature of the GPU renderer, material definitions can be defined per pixel and/or per object.
`float3* h_Vectors=NULL;`	`float3* d_Vectors=NULL;`	Stores ray cast and reflection vectors

`float3* h_C=NULL;`	`float3* d_C=NULL;`
	`float3* d_C2=NULL;`
		Stores color data
		d_C2 is defined as the intermediate color value during recursive ray casting steps.

Appendix C. Host and Device Memory Allocation and Assignment

```
extern "C" void defineLights(float* data, int numLights)
{
      // data in is defined in strips of number numLights: 1 x
      // float(intensity), 3 x float(colDiff), 3 x float(colSpec),
      // 3 x float(pos)

      cudaError_t error; //Define cudaError to bug check memory
                  //allocation!! Really Important for debugging!!

      //allocate Host memory
      h_Lights = (curayLight*)malloc(sizeof(curayLight)*numLights);
      if(h_Lights==0) Cleanup(false);

      //allocate Device Memory
      error = cudaMalloc((void**)&d_Lights,
sizeof(curayLight)*numLights);
      if (error!= cudaSuccess) Cleanup(false);

      //Init Lights in CUDA
      curayLightInit(h_Lights, data, numLights);

      //copy Lights memory to GPU memory
      error = cudaMemcpy(d_Lights, h_Lights,
sizeof(curayLight)*numLights, cudaMemcpyHostToDevice);
      if (error != cudaSuccess) Cleanup(false);

      NumLight = numLights;
};
```

Figure C.1 – GPU.cu host and device memory allocation code

```
//Defines the lights (h_Lights) with pos, m_diffuse, m_specular,
m_intensity
void curayLightInit(curayLight* lightData, float* data, int numLights)
{
      for (int i=0; i<numLights; i++)
      {
            lightData[i].m_intensity = data[i*10+0];
            lightData[i].m_diffuse.x = data[i*10+1];
            lightData[i].m_diffuse.y = data[i*10+2];
            lightData[i].m_diffuse.z = data[i*10+3];
            lightData[i].m_specular.x = data[i*10+4];
            lightData[i].m_specular.y = data[i*10+5];
            lightData[i].m_specular.z = data[i*10+6];
            lightData[i].m_pos.x = data[i*10+7];
            lightData[i].m_pos.y = data[i*10+8];
            lightData[i].m_pos.z = data[i*10+9];

      }
      // Lights data is defined and assigned to curayLight data type
and array
}
```

Figure C.2 – GPU.cu curayLight array data assignment, per light

```
void LightsToCUDA(WorldC &world)
{
  float *data;
  int floatCount = 10; //number of floats needed to represent one Light
  data = new float[world.phong1.m_numLights*floatCount];
  for(int i=0; i<world.phong1.m_numLights; i++)
    {
      data[i*floatCount+0] = world.phong1.m_lights[i]->m_intensity;
      data[i*floatCount+1] = world.phong1.m_lights[i]->m_diffuse.r();
      data[i*floatCount+2] = world.phong1.m_lights[i]->m_diffuse.g();
      data[i*floatCount+3] = world.phong1.m_lights[i]->m_diffuse.b();
      data[i*floatCount+4] = world.phong1.m_lights[i]->m_specular.r();
      data[i*floatCount+5] = world.phong1.m_lights[i]->m_specular.g();
      data[i*floatCount+6] = world.phong1.m_lights[i]->m_specular.b();
      data[i*floatCount+7] = world.phong1.m_lights[i]->m_translate.x();
      data[i*floatCount+8] = world.phong1.m_lights[i]->m_translate.y();
      data[i*floatCount+9] = world.phong1.m_lights[i]->m_translate.z();
    }
  defineLights(data,world.phong1.m_numLights);
  delete [] data;
  data = NULL;
}
```

Figure C.3 – RayTrace.cpp assignment of light data to linear array and passing to GPU.cu

Appendix D. CUDA Kernels – Shape Intersection Kernel

The first kernel creates the base image. It renders all shapes, finds closest intersections and assigns appropriate color the float3* C (the color buffer).

```
__global__ void IntersectShapes(float3* C, curayFrameBuffer* C_FB, int
N, craySphere* gpuSpheres, int SphereCount, curayTri* gpuTris, int
TriCount, float3* Vectors, curayCam* d_Cam, float _tmin, float _tmax)

{
      __shared__ curayCam gpuCamera;
      gpuCamera = d_Cam[0];
    int i = blockDim.x * blockIdx.x + threadIdx.x;
      float3 colorBG = {0.1f,0.1f,0.5f};
      if (i<N)
    {
            curayRec recordSph, recordTri;
            recordSph.t = _tmax;
            recordTri.t = _tmax;
            int tickSph=0;
            int tickTri=0;
            bool chk1, chk2;
            chk1 = chk2 = false;


      /////////////////////////////////////////////////////////////////
            ////        Calculate Sphere Intersections

      /////////////////////////////////////////////////////////////////
            if(SphereCount>0)
            for (int j=0; j<SphereCount; j++)
            {
                  float3 temp = gpuCamera/*[0]*/.pos-
gpuSpheres[j].m_pos;
                  float3 temp2;
                  temp2 = Vectors[i];

                  float a = dot(temp2,Vectors[i]);
                  float b = 2 * dot(temp2,temp);
                  float c = dot(temp,temp) - gpuSpheres[j].m_radius *
gpuSpheres[j].m_radius;

                  float discriminant = b*b - 4*a*c;
                  if (discriminant > 0)
                  {
                        //cout << "Discriminant Check if > 0" << endl;
                        discriminant=sqrt(discriminant);
                        float t = (-b - discriminant) / (2*a);

                        //now check for valid interval ???
                        if (t < _tmin)
                              t = (-b + discriminant) / (2*a);
                        if (t < _tmin || t > _tmax)
                              break;
                        // we have a valid hit!!!!
                        tickSph++;
                        if (t<recordSph.t)      {     recordSph.t=t;
recordSph.id = j; }
                  }
            }


      ////////////////////////////////////////////////////////////////
            ////        Calculate Triangle Intersections

      ////////////////////////////////////////////////////////////////
            if(TriCount>0)
            for (int j=0; j<TriCount; j++)
            {
                  float3 temp = gpuCamera/*[0]*/.pos;
                  float3 temp2 = Vectors[i];

                  float tval;

                  float A = gpuTris[j].ABC.x;
                  float B = gpuTris[j].ABC.y;
                  float C = gpuTris[j].ABC.z;
                  float D = gpuTris[j].DEF.x;
                  float E = gpuTris[j].DEF.y;
                  float F = gpuTris[j].DEF.z;
                  float G = temp2.x;
                  float H = temp2.y;
                  float I = temp2.z;

                  float J = gpuTris[j].p0.x - temp.x;
                  float K = gpuTris[j].p0.y - temp.y;
                  float L = gpuTris[j].p0.z - temp.z;

                  float EIHF = E*I - H*F;
                  float GFDI = G*F- D*I;
                  float DHEG = D*H - E*G;

                  float denom = (A*EIHF + B*GFDI + C*DHEG);
                  float beta = (J*EIHF + K*GFDI + L*DHEG) / denom;

                  if(beta <= 0.f || beta >= 1.f) { chk1=true;}
                  float AKJB = A*K - J*B;
                  float JCAL = J*C - A*L;
                  float BLKC = B*L - K*C;


                  float gamma = (I*AKJB + H*JCAL + G*BLKC)/denom;
                  if (gamma <= 0.f || beta + gamma >= 1.f) {
chk2=true;}

                  tval = -(F*AKJB + E*JCAL + D*BLKC) / denom;
                  if (tval >= _tmin && tval <= _tmax)
                  {
                        if(chk1==false && chk2==false)
                        {
                              tickTri++;
                              if(tval<recordTri.t)    { recordTri.t =
tval; recordTri.id = j; }
                        }
                  }
                  chk1 = chk2 = false;
            }

            if(tickSph>0 && tickTri>0)
            {
                  //sort by t
                  if(recordTri.t<recordSph.t)
                  {
                        C[i]= gpuTris[recordTri.id].m_color;
                        C_FB[i].id = recordTri.id;
                        C_FB[i].ObjType = 2;
                        C_FB[i].t = recordTri.t;
                        C_FB[i].normal =
gpuTris[recordTri.id].m_normal;
                  }
                  else
                  {
                        C[i] = gpuSpheres[recordSph.id].m_color;
                        C_FB[i].id = recordSph.id;
                        C_FB[i].ObjType = 1;
                        C_FB[i].t = recordSph.t;
                        C_FB[i].normal = normalize((C_FB[i].t *
Vectors[i] + gpuCamera/*[0]*/.pos) - gpuSpheres[recordSph.id].m_pos);
                  }
                  C_FB[i].point = C_FB[i].t * Vectors[i] +
gpuCamera/*[0]*/.pos;
            }
            else if(tickSph>0 && tickTri==0)
            {
                  //draw Sph
                  C[i] = gpuSpheres[recordSph.id].m_color;
                  C_FB[i].id = recordSph.id;
                  C_FB[i].ObjType = 1;
                  C_FB[i].t = recordSph.t;
                  C_FB[i].normal = normalize((C_FB[i].t * Vectors[i] +
gpuCamera/*[0]*/.pos) - gpuSpheres[recordSph.id].m_pos);
                  C_FB[i].point = C_FB[i].t * Vectors[i] +
gpuCamera/*[0]*/.pos;
            }
            else if(tickSph==0 && tickTri>0)
            {
                  //draw Tri
                  C[i] = gpuTris[recordTri.id].m_color;
                  C_FB[i].id = recordTri.id;
                  C_FB[i].ObjType = 2;
                  C_FB[i].t = recordTri.t;
                  C_FB[i].normal = gpuTris[recordTri.id].m_normal;
                  C_FB[i].point = C_FB[i].t * Vectors[i] + 
gpuCamera/*[0]*/.pos;
            }
            else
            {
                  //draw BG
                  C[i] = colorBG;
                  C_FB[i].ObjType = 0; //no object present
                  C_FB[i].normal = make_float3(0.f);
            }
    }
}
```

Appendix E. CUDA Kernels – Shading Calculation

The second kernel creates the shaded and shadowed image. It first checks for shadows by casting rays from each intersection point in the previous image to each light and calculates areas of shadow. If a shadow is present for the position and light then no material shading component is rendered. Conversely, if there are no shadows then that point has Phong shading applied to it. The result is rendered to the color buffer.

Point of intersection and object description are maintained in the FrameBuffer struct: curayFrameBuffer. These values (object type, object id, object normal, distance from camera) are generated in the previous kernel (Appendix D) and called here.

```
__global__ void CalculateShading(float3* C, curayFrameBuffer* FB, int
N, curayLight* d_Lights, int countLight, curayCam* d_Camera, float3*
d_Vectors, curaySphere* gpuSpheres, int SphereCount, curayTri* d_Tris,
int TriCount, float _tmin, float _tmax)

{
      /*__shared__ curayCam gpuCamera;
      gpuCamera = d_Camera[0];*/
      __shared__ curayLight gpuLights[2];
      gpuLights[0] = d_Lights[0];
      gpuLights[1] = d_Lights[1];
      int i = blockDim.x * blockIdx.x + threadIdx.x;
      if (i<N)
      {
            //Diffuse = clamp(dot(lightV,
lookV),0,1)*light[j].m_diffuse*light[j].m_intensity*object.color
            float3 lightV, reflV;
            float3 lookV = normalize(FB[i].point - d_Camera[0].pos);
            float3 color = {0.f, 0.f, 0.f};
            float3 Diff, Spec;
            Diff = Spec = color;
            float3 colorInit = C[i];
            float dotProd;
            for(int h=0; h<countLight; h++)
            {
                  bool isShadow = false;

      /////////////////////////////////////////////////////////////
                  ////  Calculate Shadows of Spheres
       ////

      /////////////////////////////////////////////////////////////
```

```
                    if(SphereCount>0)
                    for (int j=0; j<SphereCount; j++)
                    {
                          float3 temp = FB[i].point -
gpuSpheres[j].m_pos;//the currentPoint - all shapes
                          float3 temp2 = gpuLights[h].m_pos -
FB[i].point; //light minus the point

                          float a = dot(temp2,temp2);
                          float b = 2 * dot(temp2,temp);
                          float c = dot(temp,temp) -
gpuSpheres[j].m_radius * gpuSpheres[j].m_radius;

                          float discriminant = b*b - 4*a*c;
                          if (discriminant > 0)
                          {
                                //cout << "Discriminant Check if > 0" <<
endl;
                                discriminant = sqrt(discriminant);
                                float t = (-b - discriminant) / (2*a);

                                //now check for valid interval ???
                                if (t < _tmin)
                                      t = (-b + discriminant) / (2*a);
                                if (t < _tmin || t > _tmax)
                                      break;
                                isShadow=true;
                          }
                          //if (shadowChk>0)break;
                    }

                    if(!isShadow)
                    {
                          lightV = normalize(gpuLights[h].m_pos -
FB[i].point);
                          reflV = reflect(lightV, FB[i].normal);
                          dotProd = clamp(dot(FB[i].normal,
lightV),0.f,1.f);
                          Diff = dotProd * gpuLights[h].m_diffuse *
gpuLights[h].m_intensity * colorInit + color;
                          dotProd = clamp(dot(reflV, lookV), 0.f, 1.f);
                          Spec = gpuLights[h].m_specular *
gpuLights[h].m_intensity * pow(dotProd,20.f);
                          color = Diff + Spec;
                          color = clamp(color, 0.f, 1.f);
                    }
              }
              C[i] = color;
        }
}
```

Appendix F. CUDA Kernels – Calculating Reflected Viewing Rays

The third kernel is only called if the max trace depth is a value greater than 1. This kernel reads the normal vector of the object defined in the curayFrameBuffer and the current viewing vector and creates a new reflected vector using the CUDA math function, `reflect`. It only calculates reflection if an object was intersected with a viewing ray in the previous pass. If no object-ray intersection exists then the curayFrameBuffer object records `ObjType` to be `0`.

```
__global__ void ReflectViewingRays(curayFrameBuffer* FB, float3*
Vectors, int N)
{
	int i = blockDim.x * blockIdx.x + threadIdx.x;
	float3 tmp;
	if (i<N)
	{
		if(FB[i].ObjType != 0)
		{
			tmp = reflect(Vectors[i],FB[i].normal);
			Vectors[i] = tmp;
		}
	}

}
```

Appendix G. CUDA Kernels – Reflection Color Mixing Kernel

This final kernel blends the colors of current color and previous color, `C2[i]` and `C[i]` respectively. The blending value will be defined by a material coefficient of reflectivity.

```
__global__ void BlendColors(float3* C, float3* C2, int N)
{
	int i = blockDim.x * blockIdx.x + threadIdx.x;
	if (i<N)
	{
		C[i] = lerp(C[i],C2[i],0.5f /*material reflectivity 
coefficient*/);
	}
}
```

Appendix H. GPU Shape Splitting into Independent Arrays

```
void ShapesToCUDA(WorldC &world)
{
      //search list of shapes and count number of spheres and tris
      int countSph, countTri;
      countSph = countTri = 0;
      int *IDSphere, *IDTri;
      IDSphere = new int[world.shapes.size()]; //List of Object IDs
that are spheres
      IDTri = new int[world.shapes.size()]; //List of Object IDs that
are triangles
      for(unsigned int i=0; i<world.shapes.size(); i++)
      {
            if(world.shapes[i]->m_objType == 1)//Spheres
            {
                  IDSphere[countSph] = i;
                  countSph++;
            }
            if(world.shapes[i]->m_objType == 2)//Triangles
            {
                  IDTri[countTri] = i;
                  countTri++;
            }
      }
      float *dataSphere;
      float *dataTri;
      int sphereFloatSize = 7; //Number of floats to represent a sphere
      int triFloatSize = 12; //Number of floats to represent a triangle
      dataSphere = new float[countSph * sphereFloatSize];
      dataTri = new float[countTri * triFloatSize];
      Vect3d p0, p1, p2;
      rgb color;
      for(int i=0; i<countSph; i++)
      {

            p0 = world.shapes[IDSphere[i]]->getSphereCenter();
            color = world.shapes[IDSphere[i]]->m_color;
            color.UINTtoRGBcheck();
            dataSphere[i*sphereFloatSize+0] =
world.shapes[IDSphere[i]]->getSphereRadius();//get sphere radius data
            dataSphere[i*sphereFloatSize+1] = p0.x();
            dataSphere[i*sphereFloatSize+2] = p0.y();
            dataSphere[i*sphereFloatSize+3] = p0.z();
            dataSphere[i*sphereFloatSize+4] = color.r();
            dataSphere[i*sphereFloatSize+5] = color.g();
            dataSphere[i*sphereFloatSize+6] = color.b();
      }
      for(int i=0; i<countTri; i++)
      {
            p0 = world.shapes[IDTri[i]]->getTriangleP0();
            p1 = world.shapes[IDTri[i]]->getTriangleP1();
            p2 = world.shapes[IDTri[i]]->getTriangleP2();
            color = world.shapes[IDTri[i]]->m_color;
            color.UINTtoRGBcheck();//convert color from 0-255 to 0-1
            dataTri[i*triFloatSize+0] = p0.x();
            dataTri[i*triFloatSize+1] = p0.y();
            dataTri[i*triFloatSize+2] = p0.z();
            dataTri[i*triFloatSize+3] = p1.x();
            dataTri[i*triFloatSize+4] = p1.y();
            dataTri[i*triFloatSize+5] = p1.z();
            dataTri[i*triFloatSize+6] = p2.x();
            dataTri[i*triFloatSize+7] = p2.y();
            dataTri[i*triFloatSize+8] = p2.z();
            dataTri[i*triFloatSize+9] = color.r();
            dataTri[i*triFloatSize+10] = color.g();
            dataTri[i*triFloatSize+11] = color.b();
        }

        if(countSph>0) defineSpheres(dataSphere,countSph);
        if(countTri>0) defineTris(dataTri,countTri);

        delete [] dataSphere;
        delete [] IDSphere;
        delete [] dataTri;
        delete [] IDTri;
        dataSphere = NULL;
        IDSphere = NULL;
        dataTri = NULL;
        IDTri = NULL;
        color.~rgb();
        p0.~Vect3d();
        p1.~Vect3d();
        p2.~Vect3d();


    }
```

Appendix I. GPU.cu Rendering Function

```
extern "C" void startKernel(int frame, int threadsPerBlock)
{
      //int Loop;
      int I=500,J=500;
      int N=I*J;
      int rayCount = 1;
      float tMin, tMax;
      tMin = 0.00001f;
      tMax = 100000.f;
      clock_t start_t, end_t;
    printf("Vector addition\n");
    size_t size = N * sizeof(float3);
    cudaError_t error;
    //Generate Materials
      //curayMatInit(h_Mat);

      cudaDeviceProp prop;
      int dev;

    // Allocate input vectors h_A and h_B in host memory
    h_C = (float3*)malloc(size);
    if (h_C == 0) Cleanup(false);
      defineFrameBuffer(N);

    // Initialize input vectors

    // Allocate vectors in device memory
    error = cudaMalloc((void**)&d_C, size);
    if (error != cudaSuccess) Cleanup(false);
      error = cudaMalloc((void**)&d_C2, size);
    if (error != cudaSuccess) Cleanup(false);

      error = cudaGetDevice( &dev );
      error = cudaGetDeviceProperties(&prop, dev );
      printf("Major, minor of GPU is: %i.%i\n", prop.major,
prop.minor);
    // Invoke kernel
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

    int f, g;
      /*VecAdd<<<blocksPerGrid,threadsPerBlock>>>(d_C,N, d_Camera,
d_Lights,

      d_Spheres, d_Tris, MaxDepth,

      d_Fog, d_Vectors

      );*/
      //getchar();
    error = cudaGetLastError();
    if (error != cudaSuccess) Cleanup(false);
```

```
#ifdef _DEBUG
    error = cudaThreadSynchronize();
    if (error != cudaSuccess) Cleanup(false);
#endif


            start_t = clock(); // Start Timer
////////////////////////////////////-----
/////////////////////////////////////
////              First Ray Cast
//./..//./..//./..//./..//./..//./..-----
//./..//./..//./..//./..//./..//./../
/////////////////////-----Intersection Calculation-----
///////////////////////
    IntersectShapes<<<blocksPerGrid,threadsPerBlock>>>(d_C, d_FB, N,
d_Spheres, NumSphere, d_Tris, NumTri, d_Vectors, d_Camera, tMin, tMax);
      error = cudaGetLastError();
/////////////////////-----Material Shading Calculation-----
///////////////////////
      CalculateShading<<<blocksPerGrid,threadsPerBlock>>>(d_C, d_FB, N,
d_Lights, NumLight, d_Camera, d_Vectors, d_Spheres, NumSphere, d_Tris,
NumTri, tMin, tMax);

//./..//./..//./..//./..//./..//./..//
//                Begin Ray Tracing       //
//./..//./..//./..//./..//./..//./..//

      if(MaxDepth>1)
      do
      {
            //recalculate rays
            ReflectViewingRays<<<blocksPerGrid,threadsPerBlock>>>(d_FB,
d_Vectors, N);

            //Intersect Shapes
            IntersectShapes<<<blocksPerGrid,threadsPerBlock>>>(d_C2,
d_FB, N, d_Spheres, NumSphere, d_Tris, NumTri, d_Vectors, d_Camera,
tMin, tMax);

            //Calculate Shading
            CalculateShading<<<blocksPerGrid,threadsPerBlock>>>(d_C2,
d_FB, N, d_Lights, NumLight, d_Camera, d_Vectors, d_Spheres, NumSphere,
d_Tris, NumTri, tMin, tMax);

            //Blend Colors based on reflectivity
            BlendColors<<<blocksPerGrid,threadsPerBlock>>>(d_C, d_C2,
N);

            //increase Ray count by one
            rayCount++;
            printf("do-while loop iter: %i\n",rayCount);
      } while (rayCount<MaxDepth);


      end_t = clock() - start_t; // End GPU Timer
      printf("finished CUDA render in %d milliseconds.\n\n", end_t);
    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    error = cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    if (error != cudaSuccess) Cleanup(false);


    printf("Resorting\n");
    // Convert h_C to fractalOut[f][g][h]
    for (int i=0; i<N; i++)
    {
            //N=k+(j*3)+(i*3*J);
            //h=(int)i%3;
            g=(int)i%I;
            f=(int)((i-g)/J)%I;
            fractalOut[f][g][0] = (unsigned char)(h_C[i].x*(unsigned
char)255);
            fractalOut[f][g][1] = (unsigned char)(h_C[i].y*(unsigned
char)255);
            fractalOut[f][g][2] = (unsigned char)(h_C[i].z*(unsigned
char)255);
            //if(i%150==0)printf("Value in image at index %d, with x%d
y%d ,  is: %f.\n",i,f,g,h_C[i]);
      }


    //Write Image
    sprintf(ImageName, "CUDA_Render.%i.tga",frame);
    SaveTGA(ImageName,(unsigned char*)fractalOut,I,J,24);
    printf("Print after writing image\n");
    //getchar();

    Cleanup(true);
}
```

Appendix J. Scene 1 Testing Data

This first table represents to the timed data for different GPUs. All units of time are measured in milliseconds.

Table J.1 – *List of GPUs and render times*

CUDA compute Architecture	1.3	1.3	1.1	1.0	1.1
Graphics Card	GTX-260	GTX 275	GTS 250	GeForce 8800GTX	9800GTM 512M RAM
	GPU	GPU	GPU	GPU	GPU
Frame 1	303	280	2043	1781	
Frame 2	295	244	1887	1500	
Frame 3	289	243	1887	1468	Out of Memory
Frame 4	284	245	1888	1515	
Frame 5	278	242	1903	1500	
Frame 6	282	242	1903	1468	
Frame 7	285	239	1888	1500	
Frame 8	284	241	1904	1531	
Frame 9	277	240	1903	1484	
Frame 10	288	241	1903	1485	
Frame 11	290	247	1904	1547	Out of Memory
Frame 12	284	245	1919	1484	
Frame 13	289	247	1888	1563	
Frame 14	296	241	1888	1500	
Frame 15	304	240	1903	1515	
Frame 16	287	242	1888	1500	
Frame 17	300	244	1888	1546	
Frame 18	292	243	1888	1500	Out of Memory
Frame 19	291	245	1903	1516	
Frame 20	285	246	1888	1516	
Frame 21	281	246	1903	1531	
Frame 22	278	243	1872	1547	
Frame 23	295	246	1888	1516	
Frame 24	292	245	1903	1516	
Frame 25	295	244	1887	1501	
Average	**288.96**	**244.84**	**1900.76**	**1521.2**	

This second table represents the CPU render times.

Table J.2 – *List CPUs and render times for author written CPU ray tracer*

Processor	Core2 Duo 3GHz, 4GB RAM	I7 (920) @ 2.67GHz 6GB RAM	Core2 Quad Q9400 @ 2.66GHz	Intel Core2 Quad CPU Q7600 @2.66 (4 CPUs)	Dual Xeon 3GHz
	CPU	CPU	CPU	CPU	CPU
Frame 1	30723	66129	92586	123092	248578
Frame 2	30718	44041	78374	62296	155406
Frame 3	30696	45304	77907	62859	158157
Frame 4	29398	46041	77906	63202	154671
Frame 5	28173	45680	77876	62093	153719
Frame 6	28621	45588	78203	62280	154625
Frame 7	28341	44094	78577	62687	156516
Frame 8	28334	45521	77610	62234	154219
Frame 9	28253	44422	78843	62358	152875
Frame 10	28272	45658	77844	62531	156390
Frame 11	28414	45894	78296	62186	154765
Frame 12	28402	45780	79872	62187	151563
Frame 13	28270	45828	78905	59234	152109
Frame 14	29290	45820	77735	62015	152469
Frame 15	28487	45800	74444	61608	153281
Frame 16	28681	45703	78639	62281	150781
Frame 17	28921	44821	79280	62046	152375
Frame 18	28933	45537	78780	62093	152250
Frame 19	28820	44405	79092	62030	153344
Frame 20	28256	40861	78499	62296	151234
Frame 21	28294	42625	79233	62016	155157
Frame 22	28347	42755	79388	62288	153843
Frame 23	28212	42761	79295	62414	155985
Frame 24	28204	41882	79841	62194	160468
Frame 25	28288	48131	79529	62492	159016
Average	**28774**	**45643**	**79062**	**64600.48**	**158152**

This last table represents timing data for rendering scene1 in Mental Ray.

Table J.3 – *List of CPUs and render times for Mental Ray rendering*

Processor	Intel Core2 Quad CPU Q7600 @2.66 (4 CPUs) MR	Dual Xeon 3GHz Mental Ray	2.53GHz Core2 Duo Mental Ray
Frame 1	670	920	580.00
Frame 2	2530	10690	4540.00
Frame 3	2500	10330	4520.00
Frame 4	2410	10620	4500.00
Frame 5	2410	10740	4410.00
Frame 6	2450	11060	4480.00
Frame 7	2390	11140	4430.00
Frame 8	2560	11440	4460.00
Frame 9	2410	11440	4550.00
Frame 10	2480	11430	4480.00
Frame 11	2420	11660	4490.00
Frame 12	2540	11750	4470.00
Frame 13	2450	11660	4430.00
Frame 14	2590	11850	4460.00
Frame 15	2320	11960	4510.00
Frame 16	3150	12110	4720.00
Frame 17	2200	12040	4470.00
Frame 18	2570	12130	4440.00
Frame 19	2420	12110	4730.00
Frame 20	2420	12200	4490.00
Frame 21	2390	12280	4460.00
Frame 22	2490	12440	4450.00
Frame 23	2610	12520	4460.00
Frame 24	2560	12370	4510.00
Frame 25	2640	12530	4490.00
Average	**2423.2**	**11257**	**4341.20**

Appendix K. Classes Chart of CPU Renderer

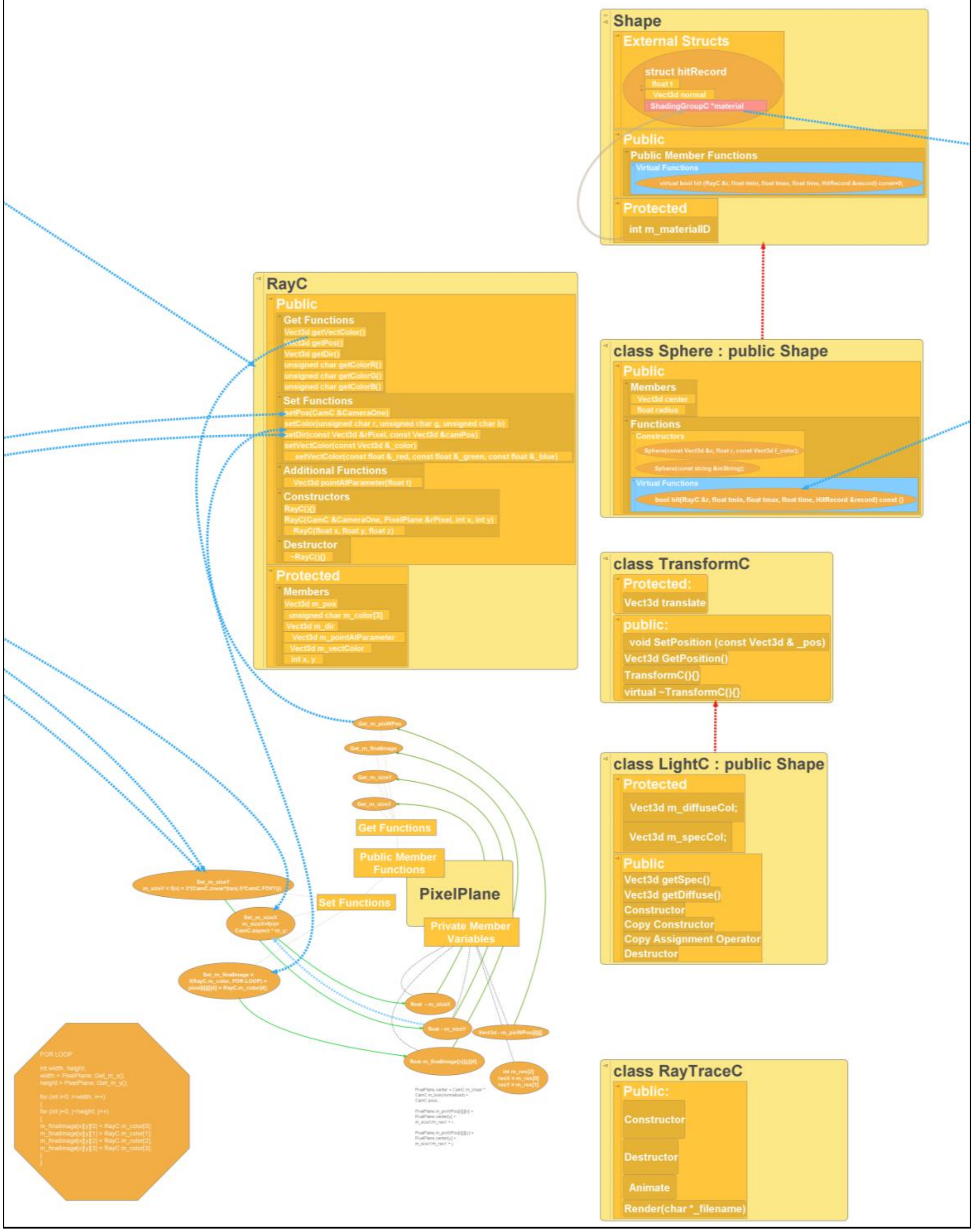

Shape
External Structs
struct hitRecord
float t
Vect3d normal
ShadingGroupC *material
Public
Public Member Functions
Virtual Functions
virtual bool hit (RayC &r, float tmin, float tmax, float time, HitRecord &record) const=0;
Protected
int m_materialID
RayC
Public
Get Functions
Vect3d getVectColor()
Vect3d getPos()
Vect3d getDir()
unsigned char getColorR()
unsigned char getColorG()
unsigned char getColorB()
Set Functions
setPos(CamC &CameraOne)
setColor(unsigned char r, unsigned char g, unsigned char b)
setDir(const Vect3d &rPixel, const Vect3d &camPos)
setVectColor(const Vect3d &_color)
setVectColor(const float &_red, const float &_green, const float &_blue)
Additional Functions
Vect3d pointAtParameter(float t)
Constructors
RayC(){}
RayC(CamC &CameraOne, PixelPlane &rPixel, int x, int y)
RayC(float x, float y, float z)
Destructor
~RayC(){}
Protected
Members
Vect3d m_pos
unsigned char m_color[3]
Vect3d m_dir
Vect3d m_pointAtParameter
Vect3d m_vectColor
int x, y
class Sphere : public Shape
Public
Members
Vect3d center
float radius
Functions
Constructors
Sphere(const Vect3d &c, float r, const Vect3d f_color);
Sphere(const string &inString);
Virtual Functions
bool hit(RayC &r, float tmin, float tmax, float time, HitRecord &record) const {}
class TransformC
Protected:
Vect3d translate
public:
void SetPosition (const Vect3d & _pos)
Vect3d GetPosition()
TransformC(){}
virtual ~TransformC(){}
class LightC : public Shape
Protected
Vect3d m_diffuseCol;
Vect3d m_specCol;
Public
Vect3d getSpec()
Vect3d getDiffuse()
Constructor
Copy Constructor
Copy Assignment Operator
Destructor
Get_m_pixWPos
Get_m_finalImage
Get_m_sizeY
Get_m_sizeX
Get Functions
Public Member Functions
PixelPlane
Set Functions
Private Member Variables
Set_m_sizeY
m_sizeY = f(n) = 2*(CamC.znear*(tan(.5*CamC.FOVY)))
Set_m_sizeX
m_sizeX=f(n)=
CamC.aspect * m_y;
Set_m_finalImage =
f(RayC.m_color, FOR-LOOP) =
pixel[][][][4] = RayC.m_color[4];
float - m_sizeX
float - m_sizeY
Vect3d - m_pixWPos[][][]
float m_finalImage[x][][y][4]
int m_res[2]
resX = m_res[0]
resY = m_res[1]
FOR LOOP
int width, height;
width = PixelPlane::Get_m_x();
height = PixelPlane::Get_m_y();
for (int i=0; i<width; i++)
{
for (int j=0; j<height; j++)
{
m_finalImage[x][y][0] = RayC.m_color[0];
m_finalImage[x][y][1] = RayC.m_color[1];
m_finalImage[x][y][2] = RayC.m_color[2];
m_finalImage[x][y][3] = RayC.m_color[3];
}
}
class RayTraceC
Public:
Constructor
Destructor
Animate
Render(char *_filename)

Figure K.1 – Close-up of flowchart planning for CPU renderer

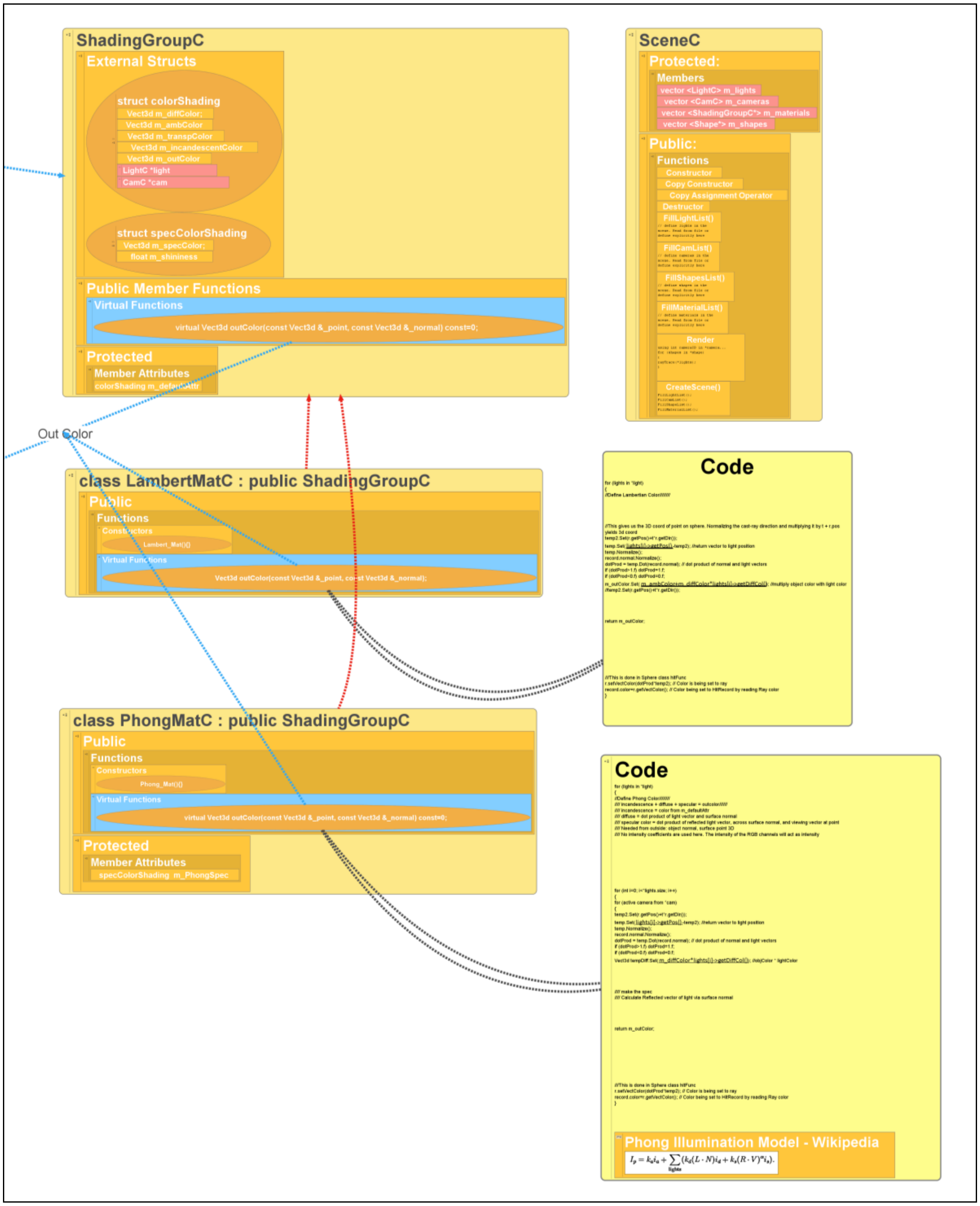


Figure K.2 – Close-up of flowchart planning for CPU renderer