ACCELERATION TECHNIQUES ANALYSIS FOR GPU RAY TRACING

by

Arif Yetkin Sarı

B.S., Computer Science Engineering, Yeditepe University, 2008

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering
Boğaziçi University
2011

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor, Prof. Lale Akarun for her advice, support, guide, optimisim, friendliness, understanding and extreme patience not only during my thesis study but my whole master degree. I am also heartily thankful to my previous supervisor, Dr. Ali Vahit Şahiner, whose encouragement, guidance and support enabled me to develop an understanding of the computer graphics, and shaped my future path. He will be always remembered.

I also want to thank all my friends and colleagues for their support and advice during my studies. Finally I am grateful for the love of my family and their support for my education no matter the conditions are.

ABSTRACT

ACCELERATION TECHNIQUES ANALYSIS FOR GPU RAY TRACING

With the increasing availability of massively parallel processing capable GPUs in recent years, research on the acceleration of ray tracers has become popular again. Since the rays shot into the scene are independent of each other, ray tracing mechanism itself is naturally suited to parallelization. Until the last decade, ray tracing in the industry has been used for mostly offline rendering and parallelization-based speed enhancements were accomplished by either shader programming on GPUs or distributed computer systems. Shaders are specificly designed for rasterization and impose limitations for general programming. On the other hand, distributed computer clusters allow multiple computers to work concurrently over networks to increase speed; but maintaining such systems is not affordable for the end user. Hopefully, modern GPUs and GPU general programming frameworks like Cuda and OpenCL will bring ray tracing to our personal computers and make it more feasible, faster and interactive in the near future. One of the major acceleration enhancement techniques in this context is the parallelization of the ray tracing process. This thesis focuses on efficiently parallelizing ray tracing and testing the raw computational power GPUs offer in terms of ray tracing. As our test bed, we have implemented two ray tracers with secondary ray capability that work on CPU and Nvidia Cuda supported GPU respectively. We then examined the dramatic performance gain of GPU parallelization along with various GPU specific optimizations and their benefits on different Cuda GPU architectures, compared to a CPU based ray tracer.

ÖZET

GPU ÜZERINDE IŞIN İZLEME HIZLANDIRMA TEKNİKLERİ ANALİZİ

Son yıllarda paralel programlama kapasitesi yüksek GPU'ların ortaya çıkması ile araştırmacılar arasında ışın izleme hızlandırmalarına dair çalışmalar tekrar yaygınlaştı. Atılan her ışın birbirinden bağımsız olduğu için ışın izleme doğal olarak paralel programlamaya çok uygun bir mekanizmadır. Son on yılda ışın izleme endüstri içinde çoğunlukla çevirim dışı görüntü üretmek için kullanılıyordu ve paralelleştirmeye bağlı hızlandırmalar shader programlama veya dağınık bilgisayar kümeleri ile gerçekleniyordu. Shader dilleri scanline pikselleştirme yöntemi için özel dizayn edilmişlerdir ve genel programlama için limitleyici yönleri vardır. Öte yandan, dağınık bilgisayar kümeleri birden fazla bilgisayarın ağ üzerinden birlikte çalışması ile hızlandırma sağlarlar, ancak bu sistemler bir ev kullanıcısı için aşırı pahalıdır. Modern GPU'lar ve Cuda ve OpenCL gibi GPU üzerinde genel programlama platformları ışın izlemeyi kişisel bilgisayarlarımıza kadar taşıyıp onu daha ucuz, hızlı ve interaktif kılabilecektir. Bu bağlamda en çok kullanılan hızlandırma yöntemlerinden biri ışın atmanın paralelleştirilmesidir. Bu tezin konusu modern GPU'lar ve Cuda platformu üzerinde ışın izlemenin verimli bir şekilde paralleştirilmesi ve modern GPU'ların sunduğu hızlandırma gücüne dair çalışma ve deneyler yapılmasıdır. Tez kapsamında CPU ve Nvidia Cuda GPU üzerinde çalışan iki adet ikincil ışın kabiliyetli ışın izleyici gerçekledik. Daha sonra GPU paralelizasyonunun sunduğu performans artışını ve değişik Cuda mimarileri üzerindeki farklı optimizasyon yöntemlerini CPU ışın izleyicimiz ile karşılaştırdık.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii
ABSTRACT iv
ÖZET v
LIST OF FIGURES viii
LIST OF TABLES x
LIST OF ACRONYMS/ABBREVIATIONS xi
1. INTRODUCTION 1
1.1. Introduction to Ray Tracing 2
1.2. Research Question 5
1.3. Research Scope 5
1.4. Summary 6
2. LITERATURE SURVEY 7
2.1. Parallelization in Ray Tracing 9
2.2. Motivation 13
3. RAY TRACER IMPLEMENTATION 14
3.1. Ray Tracing Algorithm 14
3.1.1. Phong Illumination 17
3.1.2. Camera Model 19
3.1.3. Ray Triangle Intersection 19
3.2. Limitations and Delimitations 20
3.3. CPU Ray Tracer 21
3.3.1. CPU Ray Tracer Implementation 21
3.4. GPU Ray Tracer 23
3.4.1. Cuda Overview 23
3.4.2. GPU Ray Tracer Implementation 25
3.4.3. Optimizations 30
4. EXPERIMENTS 37
4.1. Benchmark Framework 37

4.2. Benchmark Results 39
5. CONCLUSION AND FUTURE WORK 42
REFERENCES 44

LIST OF FIGURES

Figure 1.1. Ray Tracing (blue) vs Scanline Rasterization (green) [1]. 2

Figure 1.2. Ray Tracing in a 3D scene [2]. 3

Figure 1.3. Basic Ray Tracing Pseudocode. 4

Figure 3.1. Our Ray Tracing Implementation Pseudocode. 14

Figure 3.2. Function: CalculateRadiance(Ray,Scene). 15

Figure 3.3. Function: CalculateDiffuseSpecularColor(Ray,Scene). 16

Figure 3.4. Diffuse Lighting Overview [3]. 17

Figure 3.5. Specular Lighting Overview [3]. 18

Figure 3.6. Ray Triangle Intersection [4]. 20

Figure 3.7. CPU Ray Tracer Class Diagram. 21

Figure 3.8. CPU Ray Tracer screenshot with reflection, refraction, textures and bump mapping. 23

Figure 3.9. An overview of the architectures of a Cuda GPU and a CPU. . . . 24

Figure 3.10. Overview of GPU ray tracer structs, programs and routines. . . . 26

Figure 3.11. Kernel and device function flow of the GPU ray tracer on a single frame. 35

Figure 3.12. Stanford bunny with 69499 triangles, 1 light, recursion depth 5, 1024x768 resolution, rendered in 25967 ms on GPU. 36

Figure 3.13. Stanford dragon with 871414 triangles, 1 light, recursion depth 5, 1600x900 resolution, rendered in 594224 ms on GPU. 36

Figure 4.1. Benchmark Scenes. 38

Figure 4.2. Test cases and resulting render times of GPU and CPU ray tracers in miliseconds. 40

Figure 4.3. Effects of the changing benchmark variables in derived tests and CPU/GPU render time ratios. 41

Figure 4.4. Test Set 4 scenarios in which Cornell box is made of mirrors and resulting render times of GPU and CPU ray tracers in miliseconds. 41

LIST OF TABLES

Table 3.1. A 1000 triangle render in different configurations and architectures. 33

Table 4.1. Scenes and Triangle Counts. 38

Table 4.2. Benchmark Variables. 39

Table 4.3. GPU Ray Tracer speed enhancement compared to Cpu Ray Tracer. 39

LIST OF ACRONYMS/ABBREVIATIONS

2D	Two Dimensional
3D	Three Dimensional
GPU	Graphics processing unit
CPU	Central processing unit
RAM	Random access memory
PC	Personal computer
GPGPU	General programming on GPU
Cuda	Compute Unified Device Architecture
SIMD	Single instruction multiple data
ALU	Arithmetic Logic Unit
Nvcc	Nvidia cuda compiler
BVH	Bounding volume hierarchy
OpenCL	Open Computing Language
LOD	Level of detail

1. INTRODUCTION

In the world of visualisation, there are two commonly used methods to draw a 3D scene on a 2D image: scanline rasterization and ray tracing.

Scanline rasterization is the technique which is supported by modern graphics hardwares and used for almost all interactive 3D applications. It generates rapid images by exploiting the regular structure of the pixel matrix. It is fast, but does not directly support complex visual effects. The light interactions with the materials of 3D objects are imitated, and higher order illumination (global illumination) effects are either difficult or impossible to achieve. Also running times increase linearly with the triangle count of the scene. OpenGL and Direct3D APIs use scanline rasterization method to generate 3D images using GPU hardware.

On the other hand, ray tracing allows for higher order illumination effects. It easily and naturally simulates all transport paths of light and produces real shadows, reflections, refractions and indirect illuminations. Ray traced 3D animation movies are far better looking than 3D interactive video games. Ray tracing is also exteremely versatile and simpler to implement than scanline rasterization. It is sublinear in the number of primitives if acceleration structures are used, but is slow when implemented in a brute force approach. Ray tracing unifies rendering of visual phenomena, using fewer algorithms with fewer interactions between algorithms; which means it is easier and cleaner to combine advanced visual effects robustly. Because of the lack of hardware support and the overwhelmingly repetitive ray shooting mechanism, ray tracers are slower and often not preferred for real time 3D applications. Ray tracing implementation is simple and effective; provided that computation resources are available. It makes use of physically correct simulation of light, easily producing soft shadows, subsurface scattering effects, caustic effects and ambient occlusion. Fortunately, modern GPU programming environments like Cuda or OpenCL allow general programming on GPU (GPGPU), which made it popular among researchers for accelerating ray trac-

ing. Formerly, either computer clusters or fragment shaders were used to parallelize ray tracers, which was either unaffordable or hard to achieve due to limiting factors of shaders.

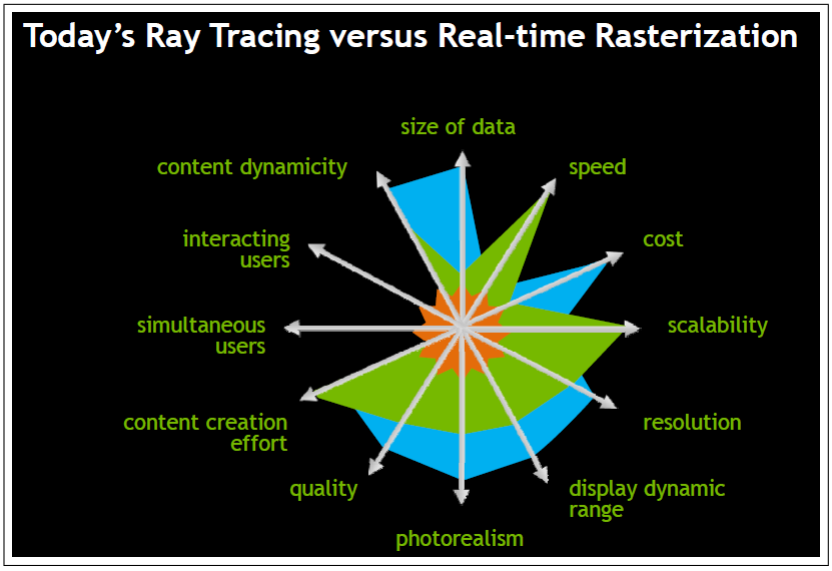

Figure 1.1. Ray Tracing (blue) vs Scanline Rasterization (green) [1].

In scanline rasterization algorithms, for each triangle, we find the pixels it covers, and find the closest triangle visible from that pixel. However, in ray tracing, for each pixel, we traverse through triangles and mark the closest one as intersected. So, without any optimizations like LODs (level of detail), occlusion queries and frustum culling one can say that rasterization is linear in primitives where ray tracing is sublinear, and rasterization is sublinear in pixels where ray tracing is linear. Consequently, both methods have advantages and disadvantages. Comments from industry officials [1, 5] and reviewers indicate that the future of 3D rendering will either depend solely on ray tracing or on hybrid systems that will utilize both ray tracing and scanline rasterization techniques.

1.1. Introduction to Ray Tracing

A Whitted style basic ray tracer using Phong illumination simulates reflection, refraction, hard shadows, diffuse and specular surfaces. In nature, light rays scatter from light sources, bounce through the world and reach our eye; creating the visual perception. In (classic) ray tracing, this calculation is done in reverse. That is, given

an eye point and an image plane, we generate rays from the eye to the sample positions on the image plane. By following the rays beyond the image plane and into the scene, we can determine which object is projected onto the image plane. Once the nearest intersection has been found, we can now calculate the correct color of that point. This is done by measuring the incoming light that is reflected in the viewing direction. This measurement can be accomplished by recursively spawning new rays into the scene, and measuring the light coming from those directions (Figure 1.2). This backward ray tracing mechanism allows us to process only the rays that reach our eye, reducing the complexity of the process immensely. Otherwise, like the global illumination algorithms, we would have to trace all possible rays generated by the light sources (infinite in reality), calculate the lighting of the scene and project it to our image plane [6].

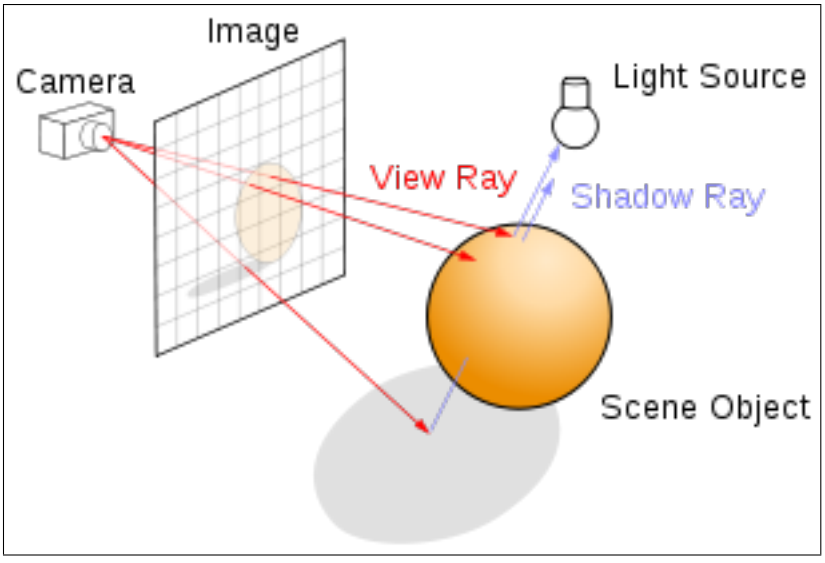

Figure 1.2. Ray Tracing in a 3D scene [2].

3D scenes consist of a list of geometric primitives, which are usually simple geometric shapes such as polygons, spheres, cones and ellipsoids. In fact, any kind of object can be used as a ray tracing primitive as long as it is possible to compute an intersection between a ray and that primitive. The core concept of any kind of ray tracing algorithm is to efficiently find intersections of a ray with a scene consisting of a set of geometric primitives. The generic pseudocode for ray tracing is given in Figure 1.3.

```
for all pixels do
  Generate a ray from the camera position to the pixel
  Perform intersection tests between the ray and the scene
  if scene is intersected then
    for all lights do
      Generate a shadow ray
      if shadow ray reaches the the current light source then
        Calculate and accumulate the color based on illumination equation
      else
        The object resides in shadow regarding the current light
      end if
    end for
    if object is reflective and/or refractive then
      Generate reflective and/or refractive rays
      Recursively calculate color on secondary ray until max recursion depth
    end if
    Save the accumulated colors as pixel color
  end if
end for
```

Figure 1.3. Basic Ray Tracing Pseudocode.

The approximate cost of this basic ray tracing algorithm is shown in Equation 1.1.

$$C = n_p \times i_{cost} \times n_s \times r \tag{1.1}$$

where C is the total cost, n_p is the number of primitives, i_{cost} is the intersection cost of a triangle and ray, n_s is the number of shadow rays generated per ray and r is the max recursion depth.

Without any optimizations, this brute force algorithm may take days to render a complex scene. To reduce the number of pixels and primitives in the equation, parallelization and acceleration structures are used, respectively. Commonly, pixel based parallelizations are used rather than primitive based and other types of parallelizations since tracing of each ray is an independant task and intuitively appropriate for concurrency. To reduce the number of intersection tests on primitives, acceleration structures (usually trees) are used to hold the scene, culling large proportions of primitives during ray traversal.

1.2. Research Question

In order to determine the applicibility and performance benefits of parallelizing a basic brute force CPU based ray tracer on a modern Cuda GPU, we aim to implement and compare two different ray tracers respectively.

1.3. Research Scope

We have implemented two ray tracers that work on CPU and GPU, respectively. CPU renderer is implemented with C++, GPU renderer is implemented with C and the Cuda framework. There are no additional libraries or software packages used in the ray tracing process. To draw the output images on the screen, the standard OpenGL library is used. The CPU renderer uses object oriented capabilities of C++ like inheritence

and recursion. The GPU renderer uses the same algorithms (slightly altered to work within Cuda) that are used in the CPU ray tracer, replacing the language from C++ to C, adapting the code to the Cuda framework and converting some of the for loops to mass thread launches.

Scope of this study and benchmarks are limited to Cuda architecture, and the renderers run under Windows only, although it is possible to port them to Linux. More advanced Cuda hardware configurations that will be introduced by Nvidia in the future may offer and support additional techniques for speed enhancements or require different optimizations. The CPU renderer can run on any computer, where the GPU renderer runs only on machines that have a Cuda supported GPU hardware (8800 series and later). Cuda 4.0 runtime library is used to implement our GPU ray tracer.

Support for multiple graphics cards, support for other GPGPU environments like OpenCL, support for ray tracing middlewares like Optix, support for other hardware vendors like ATI, support for dynamic scenes where primitives may be added, removed or moved in the scene during runtime, implementation of advanced rendering effects like caustic effects, photon mapping, ambient occlusion are outside the scope of this research. The speed benchmarks of the renderers are performed using pre-defined 3D scenes containing Stanford University 3D repository models.

1.4. Summary

In this chapter, we briefly discussed and compared ray tracing and scanline rasterization rendering methods, explained the classical ray tracing shortly along with its computational complexity, and noted our research question and scope. Our main objective in this study is to measure the performance gain of a brute force Whitted style basic ray tracer in a parallel GPU environment by implementing two different renderers running on a CPU and a Cuda GPU. Chapter 2 surveys the previous related study. Detailed algorithms of our ray tracers and illumination model is given in Chapter 3. Our results are presented in Chapter 4 and Chapter 5 concludes the thesis.

2. LITERATURE SURVEY

In this literature survey, parallelization techniques for ray tracing have been examined along with their relevance, importance, comparison and usage in GPU ray tracing. Finding ray-object intersection is computationally expensive. In a brute force approach without acceleration, each ray has to be tested with all objects and then the intersection with the smallest distance from ray origin has to be obtained as the closest intersection. For a scene containing O number of objects and a resulting image composed of I pixels, the complexity will be I x O. Ignoring super-sampling, shadows, reflection and refraction calculation, one million primitives with 1024^2 pixels will result in one trillion ray-object intersection calculations.

As mentioned above, there are numerous acceleration techniques for ray tracing. Some of them are:

- Reducing the average cost of intersecting a ray with a scene:
  - (i) Faster intersection calculations
  - (ii) Fewer intersection calculations
- Reducing the total number of rays that are traced
  - (i) Adaptive recursion depth control
- Using generalized rays
  - (i) Bean tracing
  - (ii) Cone tracing
- Parallelization
  - (i) Specialized hardware
  - (ii) Multicore CPUs
  - (iii) Multiple CPUs
  - (iv) Computer clusters

Total cost of ray tracing one frame (image) of a 3D scene can be crudely expressed as in Equation 2.1. Parallelization techniques help to reduce h and w factors, where acceleration structures reduce the number of primitives (n_p) to be tested for intersection. Total cost C can be expressed as:

$$C = res_v \times res_h \times n_p \times i_{cost} \times n_{sh} \times n_{ss} \times n_g \times n_t \times r \times o \tag{2.1}$$

where res_v is the vertical resolution, res_h is the horizontal resolution, n_p is the number of primitives, i_{cost} is the intersection cost of a triangle and ray, n_{sh} is the number of shadow rays cast, n_{ss} is the number of super sampling rays cast, n_g is the number of glossy rays cast, n_t is the number of temporal samples, r is the max recursion depth and o is the all other costs.

Each ray shot from the eye point travels and intersects the scene independently of other rays. Thus, it may seem like a trivial task to parallelize ray tracing at first glance, but parallelizing using a GPU is actually a challenging task. Until a few years ago, GPUs did not support GPGPU, so researchers had to use fragment shaders and rasterization pipelines to utilize GPUs in their parallelization tasks. The scene data for ray tracing was loaded into textures while the intersection, ray generation and traversal codes were being loaded as fragment shaders to the GPUs. Fragment shaders have lots of limitations including maximum number of operations in a kernel, very limited memory, very poor branching capabilities, inability of recursion and inability of dynamic memory allocation. Thanks to hardware manifacturers and researchers, in recent years, new frameworks and development environments like OpenCL and Cuda have been released which allow easier utilization of GPUs for GPGPU. These advancements did not eliminate all of the drawbacks mentioned above, but official support for general programming on GPU allowed for easier programming at the very least.

A specific framework for general programming on Nvidia GPUs, Cuda toolkit offers developers to speed up their programs by utilizing the parallel processing power

of the GPU cores. It supports C language as well as assembly, gives far more control on the GPU than shaders and has good documentation. However, altough it is not hard to code on Cuda framework, it requires experience and a good understanding of the GPU architecture to efficiently optimize the program.

2.1. Parallelization in Ray Tracing

There are numerous methods for parallelizing ray tracing. Utilizing computer clusters (render farms), multiple CPUs, CPUs with multicores, shader programming on GPUs (GLSL, HLSL) and general programming on GPUs (OpenCL, Cuda) are some of them. Until the advent of GPGPU frameworks, OpenRT was very popular among researchers. This open source project allowed its users to perform efficient ray tracing, utilizing multiple computers over a network. An interactive game engine using ray tracing has been implemented by Daniel Pohl *et al.* in Saarland University in 2004 [5], and they have reprogrammed the famous PC game Quake 3 with this engine, achieving real time frame rates with high quality 3D effects. The engine builds upon the OpenRT-API and thus supports computer clusters.

Intel has its own ray tracing research going on, competing with GPGPU ray tracing technologies. They have developed the Quake Wars game, which uses a specialised ray tracing engine optimized to run on Intel CPUs. The engine works on multiple multi-core x86 arhitecture Intel processors. Their long term target is to offer real time ray tracing for games as a possible rendering path in addition to scanline rasterization on future multi-core CPUs. They support advanced effects like realistic glass shaders, accurate shadows, camera portal effects, megatextures and ray tracing based collision detection. Their demos run at approximately 15-35 fps at 1280x720 resolution with four quad-core Intel CPUs [5].

GPURT is another middleware library for interactive rendering on Cuda GPUs developed in Saarland University. It provides the basic building blocks needed by ray tracing algorithms. It basically abstracts the GPGPU Cuda framework and provides

higher level functionality specific to ray tracing.

Optix is the official ray tracing framework of Nvidia built upon Cuda architecture. It utilizes bounding volume hierarchies, octrees and special hardware optimizations to offer a generic and fast solution for ray casting. Optix aims to help in optical and acoustical design, radiation research and collision analysis. Since its internal structure and implementation details are not available for public research, academic studies regarding GPU ray tracing instead focus on building ray tracing engines from scratch based on Cuda.

OpenCL (Open Computing Language) is a framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, and other processors. OpenCL includes a language for writing kernels (functions that execute on OpenCL devices), plus APIs that are used to define and control the platform. OpenCL provides parallel computing using task-based and data-based parallelism. It has been adopted into graphics card drivers by both AMD/ATI and Nvidia, which renders OpenCL as equal choice to its cousin Compute Unified Device Architecture (Cuda) of Nvidia. OpenCL's architecture shares a range of computational interfaces with both Cuda and Microsoft's competing DirectCompute. It is more generic, supporting many platforms including both Nvidia and ATI GPUs, thus it has less hardware optimizations then Cuda, making it somewhat slower but more portable. There are numerous projects that utilize OpenCL for GPU ray tracing, which are out of our scope.

GPU ray tracing studies have accelerated along with new graphics hardware after 2000's with the new shader languages and capabilities provided at the time, even before GPGPU frameworks appeared. Elhassan [7] has implemented both CPU and shader based GPU ray tracer along with kd-tree spatial acceleration structure in 2006. In his results it is concluded that GPU ray tracer was even slower than its CPU counterpart by a factor of two on the average. It is noted that the primary reason for the poor performance is the high cost of per pixel branching. SIMD nature of GPUs impose performance drops when the threads have divergence. We will discuss our own

endeavors to reduce divergence in Chapter 3.

Foley and Sugerman [8] have conducted another study, benchmarking a CPU and GPU based ray tracer while implementing uniform grid and kd-tree scene structure on GPU in 2005. Using shader programming for ray tracing, multiple kd-tree traversal algorithms were implemented on GPU. However, it is stated that their renderer was still slower than optimized CPU ray tracers, due to inefficient load balancing on threads and recirculation of data, which is moving datasets between CPU and GPU during shader pass switches.

Filip Karlsson and Carl Johan Ljungstedt have implemented a shader based GPU renderer in 2004, discarding effects like shadows, reflections and refractions to avoid secondary rays and increase thread coherency. However, it is stated that their ray tracer performance is far below 3D Studio's Mental Ray, one of the best CPU based ray tracer products in the market. It is also noted that their aim was to realize such a system on a GPU, rather than performance optimization. They advise to create a hybrid ray tracer utilizing CPU for scene structure traversals to achieve better results [9].

Christen's ray tracer (2005) on GPU using shader programming had almost same performance with the CPU based test ray tracer. It is stated that shader programming has stability issues with different hardware and drivers [10].

Britton [11] has conducted a recent Cuda based study in 2010, implemented both a CPU and a GPU based ray tracer with recursion capability, and compared the results with Mental Ray program. No spatial partitioning structures were used in both renderers of this study. GPU based ray tracer achieved better results than the other ray tracers for up to 30000 primitives. But it is stated that market product Mental Ray will outperform the Cuda ray tracer if primitive count is sufficiently increased due to its spatial partitioning mechanisms. The average GPU render time is 75 times faster than the average CPU render time where the fastest GPU render time is 116 times faster than the fastest CPU render time.

Tsiodras [12] has also ported a formerly coded CPU ray tracer to Cuda. Interactive frame rates at around 20 fps with 67000 primitives is achieved by the new renderer. It is claimed that the ported GPU version of the ray tracer is 10-20 times faster than the CPU version. Various spatial scene structures including BVH and a bin-structure have been implemented on the renderers. BVH performed better then the customized bin structure where it is noted that the bins were not ideal for secondary rays.

Colak [13] has implemented a non-recursive Cuda based ray tracer using kd-tree spatial structure in 2010. It is claimed that there is 34-85 times speed increase compared to the CPU based ray tracer. The GPU based ray tracer is also compared with the production quality CPU based renderer Rhinoceros, and resulted in 3-6 times speed increase in scenes with 14000-21000 triangles.

One of the first GPU based ray tracers was implemented by Carr *et al.* [14] who used the GPU as a brute force ray/triangle intersection tester without any ray generation or shading implemented on CPU. The second major approach was of Purcell [15]. Purcell has implemented a GPU based ray tracer with shader programming in 2004. In overall this GPU ray tracer was slower than the CPU version. However global illumination algorithms on GPU were implemented and tests showed that GPU performance is higher than CPU on these advanced effects. It is stated that, without changing graphics hardware, GPUs have the potential to outperform CPUs in ray tracing with more general programming support on GPU.

Carr, Purcell and those following them focused on utilizing the GPU for the first time in ray tracing, so they did not conduct a complete performance evaluation of the speedups achieved by utilizing GPU. They rather focused on finding out the most suitable acceleration structure for the GPU environment which lacks the stack structure and recursion capability.

One of the recent publications in GPU ray tracing other then Britton's [11] work is the study of Havran *et al.* [16]. They implemented a basic recursive Whitted-style

ray tracer with Phong illumination model, running in parallel on a cuda GPU. Their study is restricted to a static setting irrespective of the construction time as the data structures are built offline for the tests. The ray tracing algorithm entirely ran on the GPU as Britton's. Their motivation was to perform an experimental comparison of popular ray tracing acceleration structures on a modern programmable GPU environment. Thus, results regarding the raw computational power of GPU in ray tracing were not produced.

Another study about parallel ray tracing on GPU was performed in 2005 by Thrane and Simonsen [6]. They are the first ones to fully perform a comparison of numerous accelerating structures on GPU. They used fragment shaders, implemented KD-tree, BVH and Uniform grids along with full ray tracing on GPU. However although implemented, comparisons regarding the CPU ray tracers were not given, since it is noted that the CPU ray tracers were not optimized.

2.2. Motivation

There have been lots of parallelization projects of ray tracers on GPUs, but most of them are either old studies, had different focuses than comparing CPU and GPU raw parallelization power, are implemented using shader programming instead of GPGPU frameworks or are personal/commercial projects that do not involve academic research and comparison except Britton's work [11]. Therefore, we find it interesting to conduct our own study and measure the raw parallel processing power in ray tracing context that today's modern GPU hardwares offer.

3. RAY TRACER IMPLEMENTATION

3.1. Ray Tracing Algorithm

We have implemented two ray tracers that work on CPU and Cuda environment respectively. CPU and GPU based ray tracers both utilize Phong Illumination model and the Whitted style ray tracing algorithm to generate synthetic images. We specificly took caution to code the ray tracers in a standard and basic way to be able to measure their effectiveness robustly, keeping apart optimizations or enhancements specific to some types of scenes and illumination algorithms. The pseudocode of the ray tracing process has been given in Figures 3.1, 3.2 and 3.3. Actual ray tracing loop and Phong illumination calculations take place in Figures 3.1 and 3.3, respectively. Our model supports ambient, diffuse and specular lighting, point lights, hard shadows, reflection, refraction, antialiasing, texture mapping and bump mapping. Note that this outlined method is a brute force approach that tests all primitives against all rays. Acceleration structures help at this point to reduce the number of performed tests where parallelization helps to reduce the complexity of the main loop (Figure 3.1), converting it to massive thread launches.

```
for all pixels do
    Generate a ray from the eye point to the current pixel on the image plane
    CalculateRadiance(Ray,Scene) // algorithm in Figure 3.2
end for
```

Figure 3.1. Our Ray Tracing Implementation Pseudocode.

Although features like sphere primitives, textures and bump maps have been implemented, we have discarded them in our tests for the sake of GPU code clarity and complexity in our study. Texture fetches on GPU require additional memory traffic and adding spheres in Cuda imposed additional thread divergence, which will distort our reasoning and accurate testing of GPU raw computational power in parallel ray tracing. We hope to add or enable more advanced features and compare those versions of GPU and CPU ray tracers in a latter phase of our research. Thus we will not talk

```
for all primitives in the scene do
  find the closest intersection between the primitives and the ray
end for
if current ray intersects an object then
  if intersected object has a texture assigned then
    Call the UV mapper of the geometry and get texel location
    Set the texel value at UV coordinates as object color
    if bump mapping is used then
      Get the surface normal distortion vectors from bump map texture
      Alter the surface normal to be used in lighting calculations
    end if
  else if object has no texture then
    Set the current objects material color as the object color
  end if
  CalculateDiffuseSpecularColor() // algorithm in Figure 3.3
  if current material is reflective then
    Form a reflection ray and calculate its radiance (recursive)
  end if
  if current material is refractive then
    Form a refraction ray and calculate its radiance (recursive)
  end if
  Sum up and save diffuse, specular, reflective and refractive intensities
else
  Save background color as the pixel color
end if
```

Figure 3.2. Function: CalculateRadiance(Ray,Scene).

```
for all lights do
  Generate a shadow ray from the intersection point to the current light
  for all primitives in the scene do
    Perform intersection test between the primitive and the shadow ray
  end for
  if the shadow ray reaches the light without interference of other objects then
    if diffuse coefficient of the current material > 0 then
      Calculate the diffuse color and accumulate
    end if
    if specular coefficient of the current material > 0 then
      Calculate the specular color and accumulate
    end if
  else
    The object resides in shadow regarding the current light, no illumination
  end if
end for
```

Figure 3.3. Function: CalculateDiffuseSpecularColor(Ray,Scene).

about the implementation details (sphere-ray intersection, UV texture mapping, bump mapping, antialiasing etc.) of these additional features at the moment.

3.1.1. Phong Illumination

In our illumination model, diffuse, specular, reflective and refractive properties of the surface are taken into account. The energy received in one direction is absorbed and partially reemitted in all directions on a diffuse material. A perfect diffuse surface will emit the same amount of energy in all directions. That is why the perceived lighting of a perfectly diffuse surface will only depend on the angle of the incident light with that surface, making it totally independent of the viewing angle.

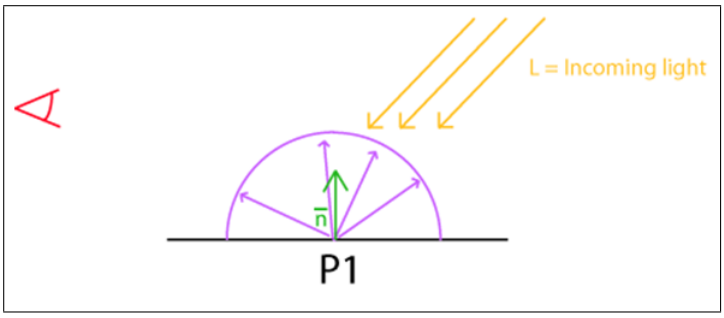

Figure 3.4. Diffuse Lighting Overview [3].

When a ray hits such a diffuse surface, the cosine of the angle theta that the incoming ray makes with the surface normal is computed, indicating percieved energy amount. Then, the Lambertian coeficient is multiplied with the diffuse color property of the surface material, giving the perceived lighting for the current viewing ray.

$$\begin{aligned} e &= L \cdot N \\ l_{coef} &= e \times k_d \end{aligned} \tag{3.1}$$

where e is the percieved energy, L is the light direction vector, N is the surface normal, l_{coef} is the Lambertian coefficient and k_d is the diffuse coefficient of the material.

Specular lighting, on the other hand, depends on the viewer's direction. Materials

reflect the light's color in all directions but mostly towards the perfect reflection (priviliged) direction. Viewers see this reflection relative to the angle between the priviliged direction and the viewer's direction. So, in specular lighting, one more dot product is performed between these vectors, calculating the phong direction. The phong term is exponentially magnified with the specular power property of the surface material before adding specular color to the total color intensity.

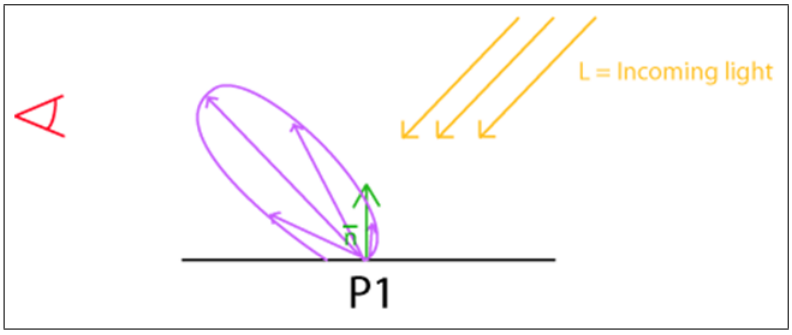

Figure 3.5. Specular Lighting Overview [3].

$$
\begin{aligned}
R &= 2 \times N \times e - L \\
p_{term} &= (P \cdot V)^{\alpha}
\end{aligned}
\tag{3.2}
$$

where R is the phong reflection direction, p_{term} is the phong term, V is the view direction and α is the specular power of the material.

Diffuse and specular colors for the intersection point are accumulated for all the light sources if the light in question is not occluded. Then the cumulative diffuse and specular lighting is calculated for the current pixel, performing a weighted sum along with the global ambient lighting. In this way, surfaces that do not see any lights directly stay in the shadow, benefitting only from ambient lighting. Finally, reflection and refraction rays are generated and recursively ray traced, adding to the final color of the current pixel. Full Phong illumination model of our ray tracer is given in Equation 3.3 [17]. Further details regarding the algorithms and their implementation may be found in [3].

The Phong illumination model may be expressed as:

$$I_p = k_a \times i_a + \sum_{m=1}^{n_{lights}} (k_d \times (L_m \cdot N) \times i_{m,d} + k_s \times (R_m \cdot V)^{\alpha} \times i_{m,s}) \tag{3.3}$$

where $i_{m,s}$ is the mth light's specular component, $i_{m,d}$ is the mth light's diffuse component, i_a is the ambient light component, k_s is the material specular reflection coefficient, k_d is the material diffuse reflection coefficient, k_a is the material ambient reflection coefficient, α is the specular power of the material, n_{lights} is the total number of lights, L_m is the m_{th} light's direction, N is the surface normal and I_p is the total Phong illumination of the surface point.

The direction vector R_m is calculated as the reflection of L_m on the surface characterized by the surface normal N using:

$$R_m = 2 \times (L_m \cdot N) \times N - L_m \tag{3.4}$$

3.1.2. Camera Model

There are two types of cameras typically used in ray tracing, perspective or orthogonal. Perspective camera represents the basic pinhole virtual camera system, where rays originate at the eye point and shoot through flat image plane pixels into the scene, resembling the human eye's vision mechanism. On the other hand, orthographic camera works like a scanner, sending parallel rays through each pixel of the flat image plane into the scene. In our study, we have used the pinhole camera model.

3.1.3. Ray Triangle Intersection

We have used Moller's standard ray triangle intersection method. The intersection point on the triangle is calculated as in Equation (3.5). If the barycentric coordinates α and β are both between 0 and 1 and their sum does not exceed 1, then

the ray passes through the triangle.

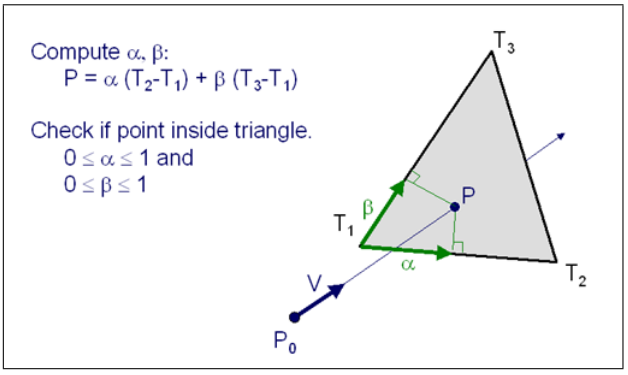

Figure 3.6. Ray Triangle Intersection [4].

$$P = \alpha(T_2 - T_1) + \beta(T_3 - T_1) \tag{3.5}$$

where P is the vector from T_1 to the intersection point, T_n is the nth vertice's 3D position and α and β are the barycentric coordinates.

3.2. Limitations and Delimitations

- In our implementations and experiments, we restricted ourselves to Whitted style ray tracing with Phong illumination and static scenes.
- Our benchmarks produce quantitative results measuring rendering times only. This research does not represent qualitative data of the rendered synthetic images.
- Our CPU benchmarks included a dual core Intel i3 540 3.06 GHz processor and 4 GB 1600 MHZ ram.
- The use of the Cuda programming interface requires the use of Nvidia graphics cards 8800 series or higher. Two GPUs were used for our Cuda benchmarks, Geforce GTX460 192 bit and Geforce 9800GT.
- GPU renderer is implemented with the Cuda 4.0 toolkit.
- Nvidia Parallel Nsight debug tool along with the Nvidia Visual Profiler is used in the development process as auxiliary tools.

- Programming with Cuda applies certain limitations: no support for polymorphism, recursion, double data types, using only a recursion-free, function-pointer-free subset of the C language, plus some simple extensions and other minor limitations. Although some features mentioned above and more are introduced with Cuda 4.0, we did not use all of them since much of the programming phase of the project took part mainly during Cuda 3.0 release.

3.3. CPU Ray Tracer

CPU ray tracer implements the ray tracing algorithm explained at the beginning of this chapter (Figure 3.1). It is implemented in C++ using object oriented programming and runs as a single thread with no parallelization or spatial acceleration structures.

3.3.1. CPU Ray Tracer Implementation

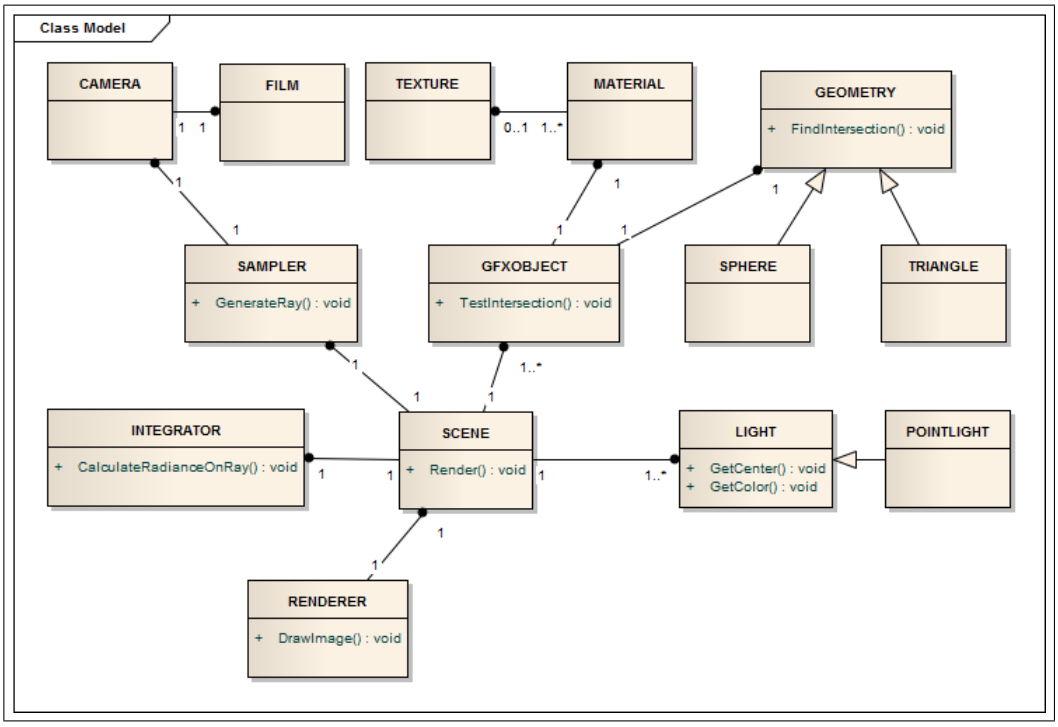

Figure 3.7. CPU Ray Tracer Class Diagram.

Important classes and their relations are shown as a class diagram in Figure 3.7. The program flow first starts in the renderer class. It includes necessary openGL

code to initialize the pixel buffer and the OpenGL window. Then it creates the scene object, and signals it to load and render the scene. The renderer fills the pixel buffer of OpenGL with the output image of the scene object and draws the final image on the screen.

The scene class first creates three lists for objects, lights and textures, respectively. It loads these lists with the scene loader function. Scene loader function loads the specified mesh file, Cornell box, lights, objects, textures. Then the scene class creates the camera, the integrator and the sampler. Sampler generates a ray based on the camera and image plane orientation, where integrator (the actual ray tracing engine code) gets this ray as an input and returns back a single color value for the current pixel being processed. The scene's render function loops for all pixels, generates a ray with the help of the sampler and calculates the radiance on this ray with the help of the integrator.

The sampler class provides a ray generating routine to the scene class. Its main functionality is to abstract the ray generation process. The scene gives the pixel coordinate to the sampler, and the sampler generates a ray from the camera center towards the appropriate point on the image plane. Also, it has an extra routine to provide antialiasing. This extra routine generates any number of sthocastic rays per pixel.

Camera is a simple class, holding the position of the camera center and a pointer to the image plane (Film). It has routines to help ray generation of the sampler. Film is another auxiliary class that holds the 3D coordinates of the image plane. It can return the 3D position of the $(i, j)_{th}$ pixel.

The Integrator is the engine of the ray tracer. It calculates the color intensity on the given ray, scene and the lights. It implements the algorithm in Figure 3.2.

Gfxobject is a class for holding information about the objects in the scene. It keeps the material and the location of the object and abstracts the details of the underlying geometry. It provides intersection routines to integrator based on the geometry

type along with UV mappers and normal map distorters. The actual intersection functions are implemented in the sphere or the triangle classes deriving from Geometry.

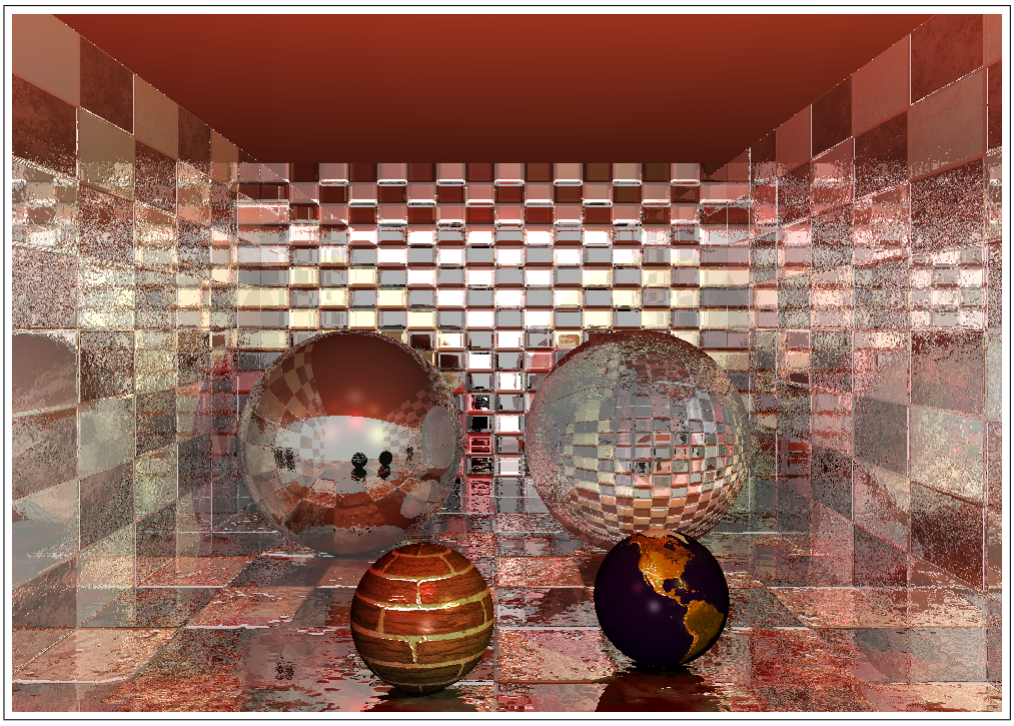

Figure 3.8. CPU Ray Tracer screenshot with reflection, refraction, textures and bump mapping.

3.4. GPU Ray Tracer

In this section, the GPU implementation of the ray tracer will be described. Using the Cuda toolkit imposed us to use a subset of C language and prohibited us from using object oriented programming and recursion. Porting, adapting and optimizing the ray tracing code on Cuda was more challenging than coding the ray tracer itself. At this point it is important to mention that we have parallelized the ray tracing on pixel basis. Other approaches exist like per object based parallelization, though not commonly used.

3.4.1. Cuda Overview

Coding in a GPU environment is a challenging task. One needs to keep the data transfer between the GPU and the CPU minimal, perform minimal and sequential

memory accesses as much as possible while avoiding if statements and other branching codes to decrease thread divergence. Also unlike CPUs, GPU multi threading mechanism is not abstracted from the programmer. The programmer must be fully aware of the parallel working cores while fully controlling threads, thread launches, synchronizations, data transfers, different memory types and thread memory access patterns.

Cuda basically offers a SIMD environment. Each Cuda GPU has several multiprocessors and each multiprocessor has multiple cores (Gtx460 has 7 multiprocessors and total of 336 cores, 9800GT has 14 multiprocessors and total of 114 cores). Threads are created with blocks, and the programmer can launch grids with multiple blocks. Every block gets executed on a single multiprocessor. Cuda cores have their own ALU (Arithmetic Logic Unit), but there is only a single Control Unit per multiprocessor. This means that, Cuda has dramatic raw arithmetic processing power over large data sets, but code branching cripples the performance immensely. If the threads in a block choose different paths in the code to execute, they are divided and executed sequentially. Note that modern CPUs have branch prediction capabilities and execute if statements and other branching codes quite efficiently when compared to GPUs.

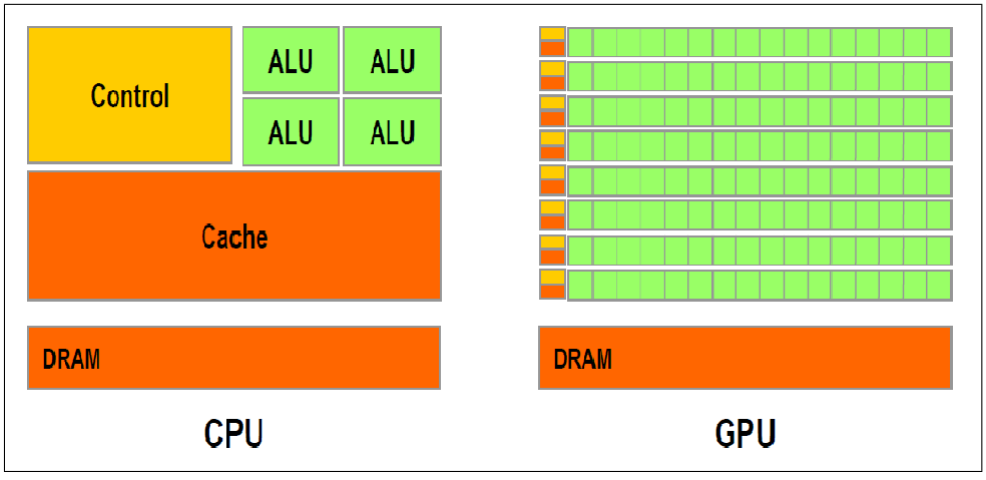

Figure 3.9. An overview of the architectures of a Cuda GPU and a CPU.

Cuda also has various memory types, offering different features and benefits. Global memory is the slowest (off-chip) memory, but is very large and has a reasonable speed if accessed sequentially by the threads in a block. All threads in any block or

grid have access to it. Constant memory is limited in amount and it is off-chip like global memory. But it has a very powerful cache, providing very fast access times in case of cache hits. It can be accessed from all blocks and threads. Shared memory is a per multiprocessor (per block) reserved memory. It is much faster than global memory and should be replaced with it, if viable. Threads in a particular block have access to their own shared memory and they should be synchronized during accesses to avoid race conditions. Texture memory is another type of read-only memory like constant memory, with spatial caching mechanism. It resides on a special section of the global memory. If threads in a block tend to access the same spatial areas of the texture, caching will speed up the access times. We have implemented a simple Cuda program that blurs a given image with a gaussian kernel to test the effectiveness of different memory types and the importance of their usage. Using different levels of optimizations on memory usage, the same bluring process on an image caused up to 80-100 times speedups between a non optimized and a fully optimized version, indicating the dramatic importance of memory types and access patterns.

We have chosen Cuda framework to implement our ray tacing test bed, because it allows C programmers to quickly adapt to the framework, has good documentation, and is a promising Nvidia technology that will be increasingly more dominant in the future. We think that the other alternative, OpenCL framework, is more generalized and thus trades portability with efficiency. Nvidia also offers hardware-specific optimized Optix and Scenix toolkit products, that provide a generalized ray tracing framework and scene structures for ray tracing on GPU, but their source codes are not available publicly. Our aim is to measure the raw computational power benefits of modern GPUs, so we implemented our ray tracers from scratch.

3.4.2. GPU Ray Tracer Implementation

Our Cuda ray tracer basically uses the same illumination algorithms described in Section 3.3 with additional optimizations and adaptations specific to the Cuda environment. GPU implementation replaces the main for loop of the CPU ray tracer

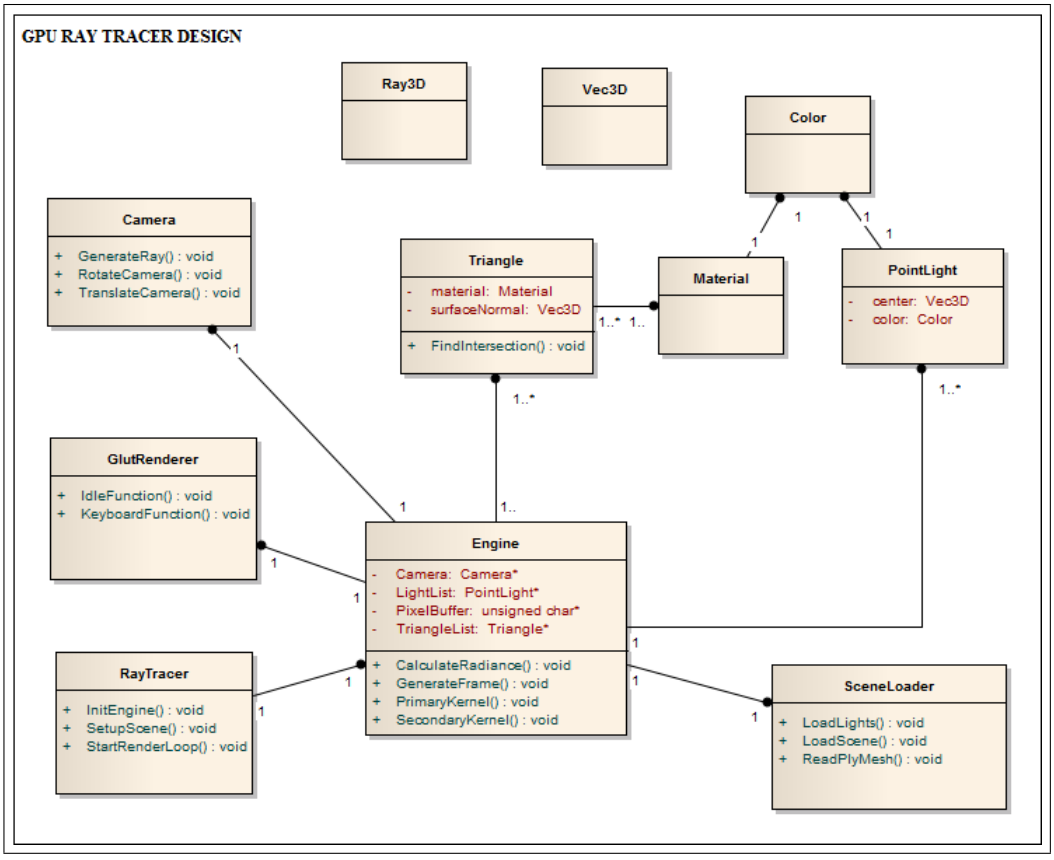

Figure 3.10. Overview of GPU ray tracer structs, programs and routines.

with a mass thread launch. One cuda thread is responsible for calculating the color intensity on a single pixel (which is referred as pixel based parallelization). Because of this method, each thread must have access to the whole scene during ray tracing. The program flow starts in the RayTracer, a C++ wrapper class of our C based GPU ray tracer. It first initializes the GPU ray tracer by determining block and grid size for the thread launch, initializing our OpenGL based renderer program with the specified resolution and registering the frame generation and camera update functions to OpenGL. Since we can achieve real time frame rates on low triangle scenes, we have implemented a camera navigation system controlled via keyboard inputs. RayTracer class also initializes the pixel buffer and the secondary ray data buffer on the Cuda global memory that will be used to hold the output color intensities and store secondary rays during ray tracing, respectively. We have implemented 1D, 2D and 3D memory layouts for the pixel buffer, trying to improve sequential access of the threads to the global memory. We will discuss the results of such tests in Section 3.4.3.

After the initialization stage, the scene setup function is called, which loads the 3D mesh file, Cornell box and other predefined primitive objects along with the lights and the initial camera. All the scene is first loaded to the RAM, and then transferred to GPU. The camera properties do not change during ray tracing of one frame, so it is loaded to the read-only constant memory and updated at every frame according to the user input. We have implemented several different memory schemes for holding the scene including global memory, constant memory (small scenes) and read-only texture memory. Lights, materials, camera and some function parameters are loaded to constant memory to reduce register pressure and low speed memory usage. All of the memories mentioned above are allocated as appropriate C structures, except the texture memory layout that is loaded to hold the scene triangles. Because in texture memory, we can only store primitive variable types like $float$ and int. Thus while using texture memory, there is an overhead of encoding and decoding the data as primitive types.

The ray tracer starts running by calling the ray trace loop function. This function starts the OpenGL based renderer, and OpenGL based renderer calls its registered idle function to generate the next frame at each render loop. Camera updates are handled through registered camera update function and new camera information is uploaded to GPU each time the keyboard funtion is called by OpenGL (that is, when user presses a key). The generate frame function registered to OpenGL idle function executes multiple thread launches and waits for the Cuda ray tracer to process, then it retrieves the results and copies them to OpenGL pixel buffer to draw them on the screen. The actual ray tracing takes place in generate frame function with multiple kernel launches (Kernel is the generic name of the functions that are executed on all threads in Cuda).

Generate frame function first launches the primary kernel once and then launches the secondary kernel until the recursion depth is reached. The primary kernel calculates the color intensities on the primary rays, and saves the generated secondary rays on the secondary ray data buffer. The secondary kernel processes the previously saved secondary rays by reading the buffer and calculates new color intensities. It then

merges these new values on the pixel buffer as a weighted sum based on material coefficients. The secondary kernel also stores more secondary rays, if spawned by hitting more reflective or refractive surfaces. Until the recursion depth is reached, secondary kernel is called and new calculated color intensities are accumulated on the pixel buffer. Buffering of secondary rays and tracing them in multiple passes was a mandatory workaround since recursion is not available.

After all the kernel launches finish by reaching recursion depth, the generate frame function copies the output pixel buffer back to CPU, and sends it to the OpenGL renderer to draw it on the screen. You may notice that Cuda ray tracer calculates a pixel buffer, copies it back to CPU, and then loads it to GPU OpenGL pixel buffer. Actually, Cuda supports OpenGL interoperability and pixel buffers can be directly sent to OpenGL for rendering without intervention of the CPU. However, keeping the pixel buffer in GPU has imposed certain debug limitations which made it hard to develop the code. Thus, although we have implemented interoperability, we discarded it from our tests. Notice that between kernel launches, no data is copied to or from GPU. All the buffers and the scene stays on the GPU during a single frame generation involving multiple kernel passes. Kernels are launched with pointers to these GPU memory allocations and work on them directly.

Threads executing the primary kernel find the pixel they are responsible for first. Then, they generate a ray using the pixel indices and the camera. This ray is sent to color intensity calculation function to generate the final color and save the secondary ray if spawned. The primary kernel then saves the output information to pixel buffer and secondary ray data buffer, respectively. The primary and secondary kernels work in a similar fashion. One major difference is that secondary kernel does not generate the ray using camera; instead, it reads it from the buffer saved beforehand. Also, it accumulates color values based on material properties without erasing previous color data on the pixel buffer.

Color intensity calculation function is a device function that can be called from

kernels. Device functions are special Cuda functions that can be called from kernels and other device functions. Also they are always implicitly inline in Cuda, which is the reason for inability to support recursion (in Cuda 3.0). This function basically implements the ray tracing algorithm given in Figure 3.1. However, instead of casting secondary rays, it saves these new rays on a buffer.

At this point, we want to state that, we have discarded some of the effects from our GPU ray tracer that are implemented in CPU. Sphere primitives are not supported, because the lack of virtual functions (Cuda 3.0) imposes us to write additional if statements during scene traversal. Since thread coherency is a very important performance parameter in GPU, we will not be testing scenes with spheres. Also, we discarded textured objects from our tests in order to not to distort and complicate our performance experiments by extra memory fetches from textures. Note that support and/or exclusion of extra features or different memory layouts do not create performance problems as they are disabled with compiler conditions and other non-performance related methods.

There are two other important device functions to generate reflective and refractive rays. These functions are called after the color intensity calculations, if the intersected surface materials are reflective or refractive. In our GPU ray tracer design, a material can be either reflective or refractive. The reason that we limited ourselves is to avoid dealing with extra buffer allocations and focus on efficiently parallelizing the ray tracing. Thus, our scenes and materials used in benchmarks obey this imposed rule. Although we have also tried stochastic ray generation when a both reflective and refractive material is used to work around this limitation, lack of $rand()$ function in GPU environment forced us to use a psuedo random buffer to determine the type of ray to be shot (reflective or refractive) on that particular intersection, which yielded visually noisy images on materials that have both reflective and refractive properties. So we omitted this feature from our benchmarks.

All the auxiliary structs, classes and functions are coded by ourselves such as

vectors and linear algebra related functions. Some other important structs we have implemented in GPU ray tracer are Color, Vec3D, Ray3D, Camera, Material, Triangle and PointLight. They all have related host (CPU) and device (GPU) functions used by the GPU ray tracing kernels and device functions.

3.4.3. Optimizations

In this section, we will discuss the optimizations endeavors we have tried during the implementation of our GPU ray tracer. We have used two different Cuda GPUs in our benchmarks, Gtx460 and 9800GT. Gtx460 has Cuda architecture 2.1 and 9800Gt has Cuda architecture 1.1. Cuda architecture 2.1 (latest as of 2011 August) offers additional enhancements and features including caching on global memory fetches. Thus, some of the optimizations improved the speed on both architectures, but some of them yielded better results only on 1.1 architecture. The following discussion is not related to our Experiment Chapter, rather it is intended to give the reader and other Cuda programmers hints and insight about our exploration of the Cuda GPGPU world. In Chapter 4, all our experiments are conducted on the Gtx460 GPU hardware.

We have tried holding the scene primitives in numerous ways and memory types. As rays are shot into the 3D scene, all the threads access every scene primitive to find the closest intersection due to lack of an acceleration structure. Our scenes are static and thus we gladly used read-only memory types, including texture memory and constant memory apart from the standard storage method on global memory. Of course, as one of the fastest memory types, constant memory allowed us to ray trace at least four or five times faster. But with the lights and the camera also in constant memory, only 750-800 triangles could be stored in the 64KB limited space.

To reduce the number of accesses to the memory, we have tried moving the materials of the triangles to the constant memory. In this way, only a material identifier value was kept for each triangle where the actual materials resided in constant memory. This reduced the amount of sequential accesses to fetch a triangle. Experiments showed

that this helped to improve the performance by 10 to 25%.

Scenes consist of triangles, which are structs with vertices, surface normal and a material id. However, in texture memory, the programmer is allowed to store only primitive types or built in vector types of Cuda. So, we encoded each triangle as a series of $float4$'s (an array of 4 $floats$). We have used 1D texture arrays, since we cannot create a spatial resemblence of the actual 3D scene on a 2D or 3D texture. Because of the encoded storage, each time a thread requested the next triangle in the scene, we have fetched several sequential elements from the texture and decoded them, recreating the triangle in runtime to process.

Architecture 1.1 tests showed that using texture memory instead of global memory improved the rendering speed by 10-12 times on average. However architecture 2.1 has caching on global memory fetches and it proved to be more efficient than texture caching in our scenario where there is no spatial coherency of memory accesses. Also, both the decode times and access of multiple memory cells overhead for a single triangle included, this new method decreased the rendering speeds by 50% in two scenes with 1000 and 3850 triangles compared to the standard global memory scene storage method on architecture 2.1.

Any automatic variable created within kernels and device functions reside in each thread's local memory. If there are enough availabe registers, these variables are stored in GPU registers which are the fastest memory type. The Nvcc compiler decides which variable can fit into the registers during compile time. If there are not enough free registers, register spilling occurs and some of the local variables are stored in reserved global memory locations for each thread. This so-called local memory is identical to global memory and has very high access times. Local memory usage and register spilling can be checked during compilation with Nvcc compiler parameters to allow the programmer to make further optimizations. Also, large structures and arrays that are accessed with dynamically determined indexes spill to local memory. We have reduced register pressure in our ray tracer by reducing thread divergence, moving some kernel

and function parameters to constant memory and tuning block sizes. We also tuned the compiler to use more registers for automatic variables by using additional compiler parameters and macros (these macros give hints to Nvcc heuristics to more efficiently allocate registers by pre-informing maximum number of threads on a block ever to be launched).

Triangles are referenced many times during calculations on threads. Formerly, we accessed the intersected triangle from its global memory pointer multiple times in the kernels and device functions. Fetching the triangle only once, saving it on a local variable, and performing calculations only on this copy offered us a 10 to 11% speedup on the average. By maximizing register usage for storing these local variables, we gained even 25% more speedup.

During intersection tests, each thread traverses the whole scene independently, requesting the same triangles over and over. To reduce this repetitive memory access, we have implemented a manual cache on a per-block basis, using shared memory. However, shared memory is indeed constructed of registers, so we had a maximum limit for shared memory cache in our scenario, depending on overall register usage and block size. In our caching algorithm, each thread retrieves a single triangle from the memory based on its id (the scene is initially stored in texture or global memory) and save it in shared memory. Then, they perform intersection tests on these local copies of triangles. After the first intersection tests are done, the threads fetch a new bucket of triangles from off-chip memory in an aligned and sequential manner to fill the shared memory again. This way, they consume the whole scene in sufficent number of iterations and find the closest intersected object. This caching mechanism improved the performance dramatically in architecture 1.1 by 10-12 times. However in architecture 2.1, the rendering times increased by 15 to 16%. We believe that in architecture 2.1 global memory caching provides an already optimized memory access for threads; since each thread accesses the same global memory location once, and at the same time during traversal. After the first fetch, the cached triangle is retrieved much faster by other threads. Consequently, shared memory acts as an additional

Table 3.1. A 1000 triangle render in different configurations and architectures.

Architecture	Scene storage type	Caching on shared memory	Render time
1.1	Global memory	No	10826ms
1.1	Texture memory	No	966ms
1.1	Global memory	Yes	891ms
1.1	Texture memory	Yes	845ms
2.1	Global memory	No	305ms
2.1	Texture memory	No	578ms
2.1	Global memory	Yes	348ms
2.1	Texture memory	Yes	342ms

unnecessary buffer, where triangles are written and read, decreasing performance on architecture 2.1.

Using shared memory cache and storing scene on texture memory together resulted in even better performance in both architectures. Although useful in architecture 1.1, speed increase in architecture 2.2 did not compensate the initial loss for using texture memory alone. A default scene with 1000 triangles and one point light along with shared memory caching and texture memory usage is given in Table 3.1. Results show that architecture 2.1 is fastest when it uses global memory and no shared memory caching. On the other hand architecture 1.1 benefits from both options, reducing render times from 10826 ms to 845 ms.

The number of lights are less than the number of triangles in a scene, thus they are referenced with a much higher frequency in every shading calculation. Moving the lights into the constant memory helped us with minumum 10-15% speedup. Constant memory amount allows us to support up to 2500 lights in a scene which is sufficient for a typical ray tracer.

Each thread launched calculates a color intensity and eventually stores it in the pixel buffer. Since we need to write the results to the buffer, we have to use global

memory. We have tried using 1D, 2D and 3D global memory arrays to improve sequential writes on the pixel buffer. In 1D storage, we stored all the pixels (each pixel is 3 floats with red, green and blue colors) in a row major approach. 2D storage directly resembled the output image, storing $float3$'s (an array of 3 $floats$) to hold each pixel's RGB (red green blue) value. In 3D storage using cuda extent, we have stored pixels as 2D, except that each color channel resides on a different 3D slice (2D image x 3 slices = 3D allocation). The 2D storage proved to be the fastest, but only with 2-3 miliseconds compared to 3D allocation and 1D allocation of pixel buffers. Since writing color intensities to the pixel buffer is only performed at the end of rendering and is not a repetitive action like triangle fetching, using different memory layouts did not affect the overall performance much.

Although some information was already precalculated and cached for each triangle like surface normals, additionally caching some internal variables that are used for intersection tests like edge vectors almost did not affect the perforamnce of the GPU ray tracer.

One of the best performance optimizations we have been able to perform has been eliminating branching codes. For example, while calculating diffuse and specular lighting, we loop over all lights, and for every light, if it is not occluded by another object (if we are not in the shadow), we calculate diffuse and specular lighting. If it is occluded, we do not. We altered this flow, calculating the diffuse and specular lighting on all conditions. But in order not to distort the program logic, we have converted the 'occluded' bool to a float and filled it with 1.0 if we are not in shadow, with 0.0 if we are in shadow. In this way, all threads calculated the diffuse and specular lighting no matter what. However, some of them multiplied their results with 0.0 (the ones that are in the shadow), and others with 1.0. This kind of branching code eliminations or uniting several conditions into a single if statement lead to 70% speed increase on average.

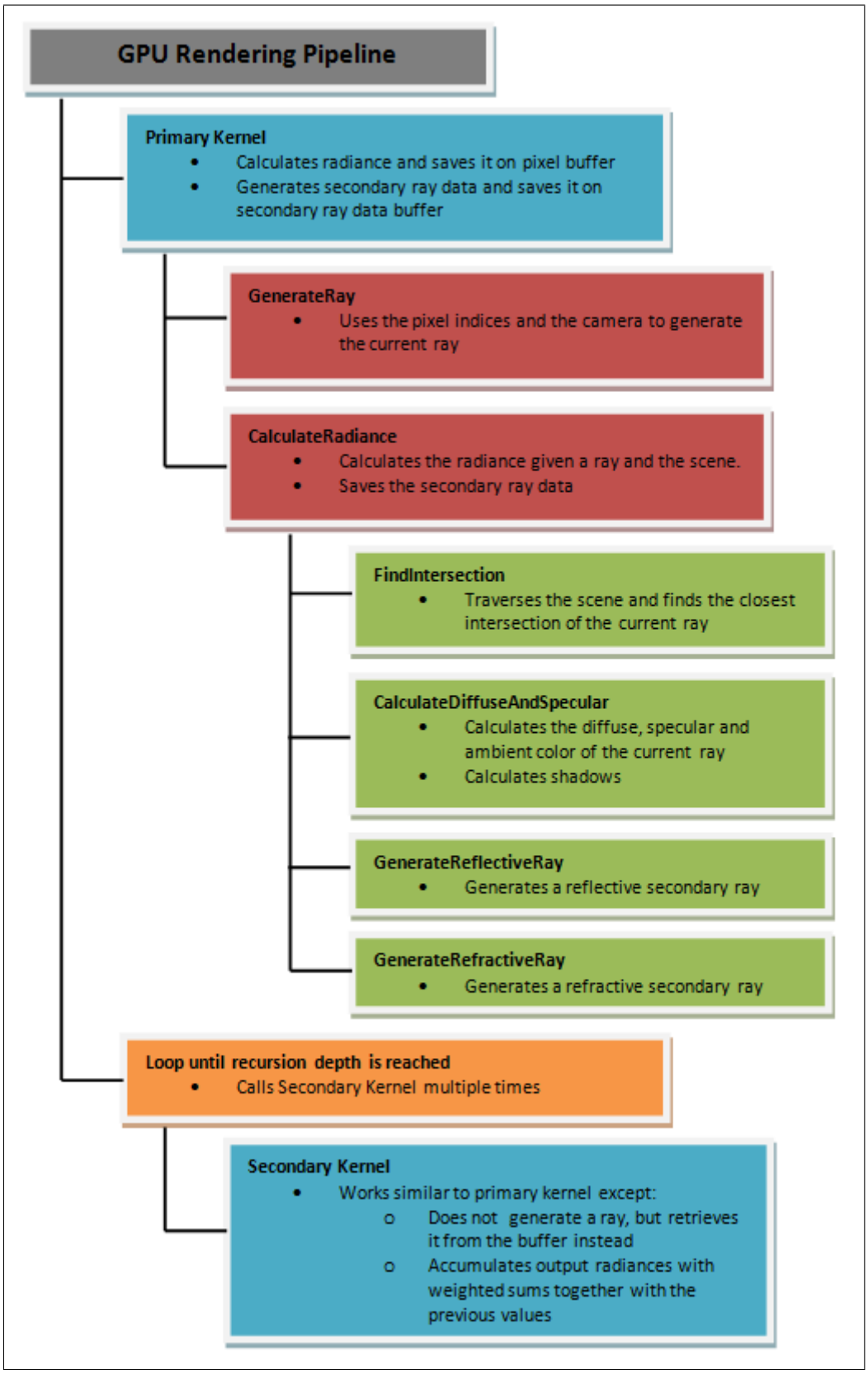

Figure 3.11. Kernel and device function flow of the GPU ray tracer on a single frame.

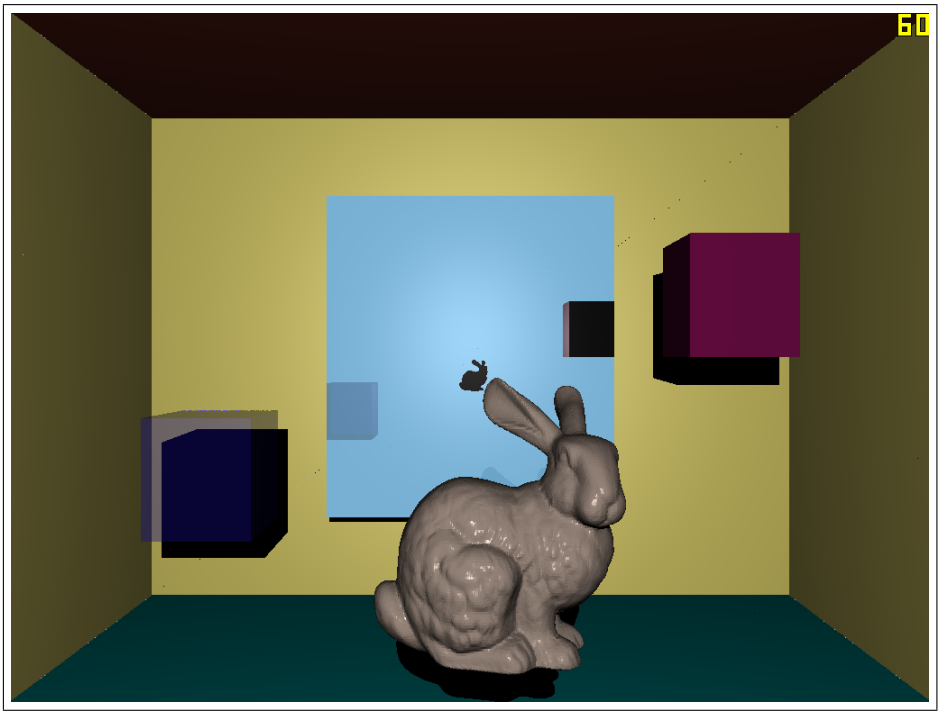

Figure 3.12. Stanford bunny with 69499 triangles, 1 light, recursion depth 5, 1024x768 resolution, rendered in 25967 ms on GPU.

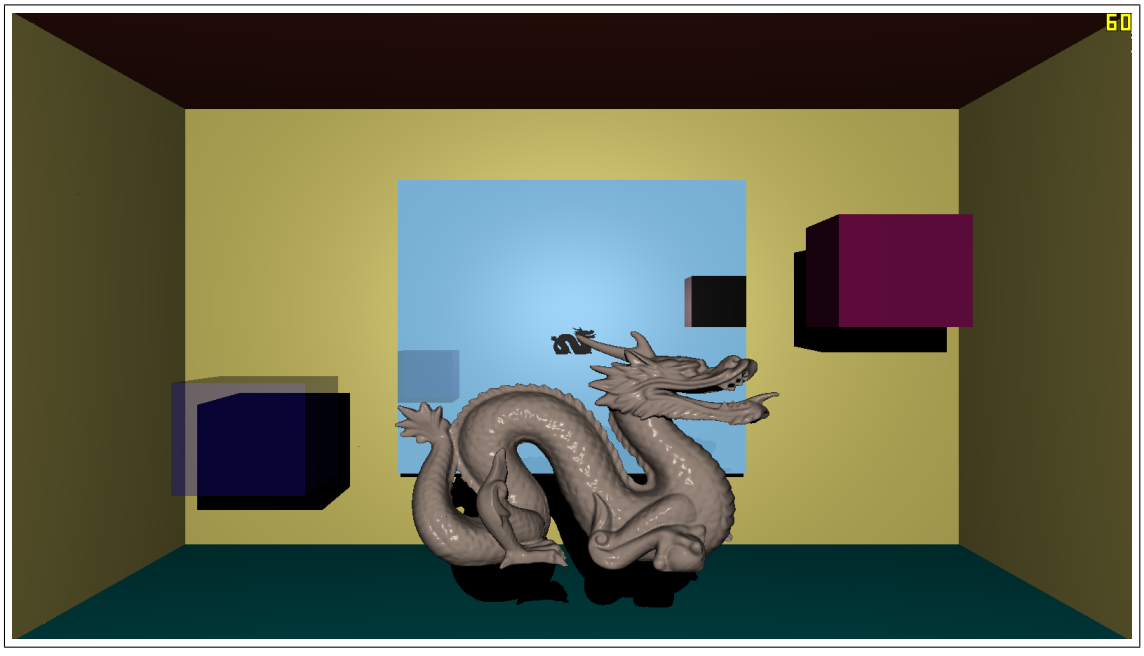

Figure 3.13. Stanford dragon with 871414 triangles, 1 light, recursion depth 5, 1600x900 resolution, rendered in 594224 ms on GPU.

4. EXPERIMENTS

4.1. Benchmark Framework

Our experiments include the comparison of render times of our GPU and CPU based renderers on different configurations and scenes. Note that any other types of tests such as qualitative comparisons of the generated images are beyond our intention and scope. Although we have shared our optimization endeavors on GPU ray tracer in Chapter 3 and gave some experimentation results obtained from different Cuda architectures, the results in this chapter are solely based on our more advanced graphics card, Gtx460 192 bit architecture 2.1. In the following context, by render time, we mean the total execution time of ray tracing a single image of a particular 3D scene. Initialization times like 3D mesh file reading, creation of necessary classes and initialization of Cuda framework and OpenGL are not included in the timings. Although the CPU ray tracer has additional capabilities that we have not enabled on the GPU renderer due to the variety of reasons explained in Chapter 3, we only test the minimum subset of features that are both enabled on GPU and CPU ray tracer. This subset includes triangles as primitives, point lights, diffuse, ambient and specular lighting, hard shadows, reflective and refractive surfaces. Given this subset of effects enabled, our both ray tracers generate the exact same output image given the same input scene.

Our benchmarking hardware includes Intel i3 540 3.06 GHz dual core CPU, Corsair 4 GB 1600 MHZ ram and Zotac GTX 460 192 bit 768 MB Nvidia graphics card with Cuda 4.0 runtime library, Visual Studio 2010 and Windows 7 64bit installed. Our CPU is the primary CPU on our test pc which means it is also used by the operating system routines. Thus, our test results varied by small amounts of time. To compensate this, we have performed all the tests several times and only the average output timings are presented below. Note that unlike CPU, our Nvidia GPUs are plugged in as secondary GPUs on our system, dedicated to our ray tracing application.

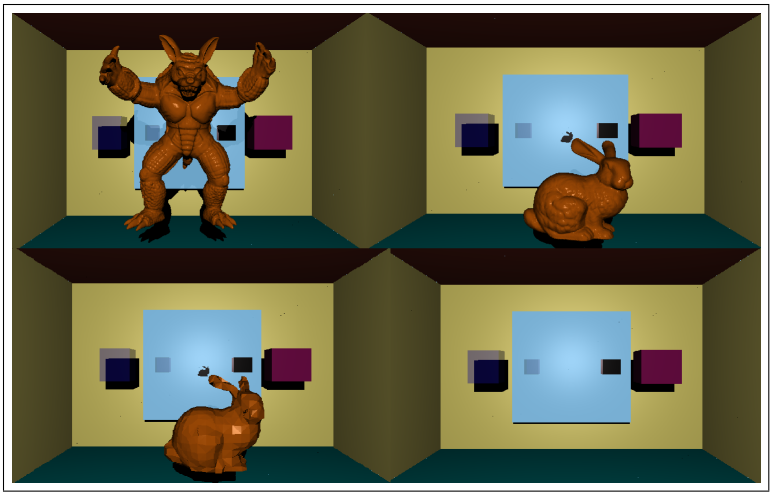

Figure 4.1. Benchmark Scenes.

Table 4.1. Scenes and Triangle Counts.

Scene 1	Cornell box	48 triangles
Scene 2	Low resolution bunny and Cornell box	996 triangles
Scene 3	High resolution bunny and Cornell box	69499 triangles
Scene 4	Armadillo and Cornell box	345992 triangles

In our benchmarks, we have used four scenes with 48, 996, 69499 and 345992 triangles (Table 4.1). Except the Cornell box and the cubes, all geometry is downloaded from Stanford University 3D repository. Cornell box includes three cubes, two of them having reflective and refractive materials. We render the scenes with different resolutions, recursion levels and number of light sources as specified in Table 4.2. Using these parameters, we have setup three sets of tests and total of 16 test cases as shown in Figure 4.2.

Apart from the test sets mentioned above, we have a special test set (Test set 4) with a modified version of the Cornell box having reflective surfaces on walls, floor and ceiling to test the secondary ray performances of the ray tracers. Test set 4 cases and their results are given in Figure 4.4.

Table 4.2. Benchmark Variables.

Resolutions	Recursion Depths	Number of Lights	Number of Triangles
800x600	0	1	48
1024x768	2	2	996
1600x1200	3	3	69499
	4	4	345992
	5		

Table 4.3. GPU Ray Tracer speed enhancement compared to Cpu Ray Tracer.

Best speed up	Worst speed up	Average speed up
628.24 times faster	549.92 times faster	591.38 times faster

4.2. Benchmark Results

The first three test sets and the raw results of CPU and GPU ray tracer are shown in Figure 4.2. Yellow colored lines are base tests for each test set. Other colors indicate a new test derived from the base test of its set by altering only a single parameter. More specificly, orange lines indicate an increase in number of lights, turquoise lines indicate an increase in maximum recursion depth and purple lines indicate an increase in number of triangles. The results showed that Cuda based parallelization offered an extreme performance boost. On the heaviest scenario (Test 3.2), GPU ray tracer (GPURT) is faster than the CPU ray tracer (CPURT) by 563.13 times. On the lightest scenario (Test 1.1), GPURT is 609.55 times faster than the CPURT.

GPURT has been 549.92 times faster than the CPURT at the worst on Test 1.5, and 628.24 times faster at the best on Test 1.3. Average GPURT speedup compared to CPURT over all benchmarks is 591.38 times.

When we examine results in Figure 4.2 in more detail, we see that increasing number of lights have a huge impact on render times. Ray tracing algorithm first traverses the scene to find the closest intersection. Then, for each light it performs an

	Test Set 1	GPU	CPU
Test 1.1	800x600 resolution, 1 light, 0 recursions, 48 triangles	9.84	5998.00
Test 1.2	800x600 resolution, 2 lights, 0 recursions, 48 triangles	14.25	8824.00
Test 1.3	800x600 resolution, 3 lights, 0 recursions, 48 triangles	18.66	11723.00
Test 1.4	800x600 resolution, 1 lights, 2 recursions, 48 triangles	11.81	6909.00
Test 1.5	800x600 resolution, 1 lights, 3 recursions, 48 triangles	12.58	6918.00
Test 1.6	800x600 resolution, 1 lights, 0 recursions, 996 triangles	187.97	114848.00
Test 1.7	800x600 resolution, 1 lights, 0 recursions, 69499 triangles	14167.57	8121723.00
	Test Set 2		
Test 2.1	1024x768 resolution, 2 lights, 3 recursions, 996 triangles	489.16	295537.00
Test 2.2	1024x768 resolution, 3 lights, 3 recursions, 996 triangles	634.91	384419.00
Test 2.3	1024x768 resolution, 4 lights, 3 recursions, 996 triangles	782.85	469545.00
Test 2.4	1024x768 resolution, 2 lights, 4 recursions, 996 triangles	488.04	294312.00
Test 2.5	1024x768 resolution, 2 lights, 5 recursions, 996 triangles	489.23	294545.00
Test 2.6	1024x768 resolution, 2 lights, 3 recursions, 69499 triangles	36840.27	20808012.00
Test 2.7	1024x768 resolution, 2 lights, 3 recursions, 345992 triangles	177702.87	101426954.00
	Test Set 3		
Test 3.1	1600x1200 resolution, 4 lights, 5 recursions, 69499 triangles	142489.92	81609984.00
Test 3.2	1600x1200 resolution, 4 lights, 5 recursions, 345992 triangles	696067.39	391982592.00

Figure 4.2. Test cases and resulting render times of GPU and CPU ray tracers in miliseconds.

occlusion check with the scene primitives. In the worst case, this means traversing the whole scene again for each light, which diminishes the speed gain. On the four scenes tested, there is a moderate number of reflective and refractive surfaces, which renders recursion depth variable less significant than other benchmark variables.

CPU/GPU render time ratios and effects of the changing benchmark variables in derived tests are given in Figure 4.3. Increase of render times in the same scenarios and CPU/GPU render time ratios accross multiple scenarios are steady. GPU parallelization gives a solid 600 times speed up with small variances. However there is a trend that can be concluded by examining purple colored boxes. Although by small amounts, the render time of GPU increases more aggressively than CPU ray tracer when number of triangles is increased. This is probably due to the increased divergence of concurrent threads while looping on primitives. Since we do not use any spatial partitioning structures, at sufficiently large number of primitives all the threads of GPURT will be executed almost sequentially on each multiprocessor which hides most of the performance gain from parallelization.

Test set 4 results in which the Cornell box surfaces are covered with reflective

Test Set 1	GPU render time increases by	CPU render time increases by	CPU time/GPU time
Test 1.1	1	1	609.5528455
Test 1.2	1.448170732	1.471157052	619.2280702
Test 1.3	1.896341463	1.954484828	628.2422294
Test 1.4	1.200203252	1.151883961	585.0127011
Test 1.5	1.278455285	1.153384461	549.9205087
Test 1.6	19.10264228	19.14771591	610.9911156
Test 1.7	1439.793699	1354.071857	573.2615403
Test Set 2			
Test 2.1	1	1	604.1724589
Test 2.2	1.297959768	1.300747453	605.4700666
Test 2.3	1.600396598	1.588785837	599.7892317
Test 2.4	0.997710361	0.995855003	603.0489304
Test 2.5	1.000143102	0.996643398	602.0583366
Test 2.6	75.31333306	70.40746844	564.8170331
Test 2.7	363.281687	343.195451	570.7671125
Test Set 3			
Test 3.1	1	1	572.7421561
Test 3.2	4.88502899	4.803120559	563.1388536

Figure 4.3. Effects of the changing benchmark variables in derived tests and CPU/GPU render time ratios.

materials are shown in Figure 4.4. CPU and GPU render time ratios indicate that average speedup of parallelization remains the same even if the number of secondary rays is very high. Comparing test 4.1 and 4.3, we can conclude that generating more secondary rays has a huge impact on performance.

	Test Set 4 - Scenes with many reflective surfaces	GPU	CPU	CPU/GPU
Test 4.1	800x600 resolution, 2 lights, 5 recursion, 996 triangles	659.45	397700	603.0783228
Test 4.2	800x600 resolution, 4 lights, 5 recursion, 996 triangles	1041.11	637500	612.3272277
Test 4.3	800x600 resolution, 2 lights, 0 recursion, 996 triangles	271.64	168800	621.4106906
Test 4.4	800x600 resolution, 2 lights, 5 recursion, 69499 triangles	49675.78	28241100	568.5084361

Figure 4.4. Test Set 4 scenarios in which Cornell box is made of mirrors and resulting render times of GPU and CPU ray tracers in miliseconds.

We have expected a large increase in performance due to Cuda parallelization, but drastic results exceeded our expectations. Even a brute force GPU ray tracer running on a modern GPU could beat the performance of a brute force CPU ray tracer running on a modern CPU by a factor of 600 and more. We believe that, although there may still be many undiscovered Cuda optimizations for our renderer, implementing a spatial acceleration structure on GPU will boost the performance immensely. Apart from acceleration structures, CPU ray tracer can be enhanced by supporting multithreading and efficiently utilizing multicore CPUs.

5. CONCLUSION AND FUTURE WORK

We have succesfully implemented two brute force ray tracers working on a Cuda GPU and CPU, respectively. Using the same algorithms and producing the same output images, the pixel based parallelized GPU ray tracer increased the performance of ray tracing around 600 times on scenes with 48, 996, 69499 and 345992 triangles including 1 to 4 point lights, with support of diffuse, ambient and specular lighting, hard shadows, reflective and refractive materials and recursions depths up to 5. By minimizing data traffic between GPU and CPU RAM, eliminating branching codes and regulating memory access patterns in threads; we think that we have modestly optimized our Cuda based ray tracer. Using multi passes for secondary rays, we do not copy any data to or from the GPU between kernel launches on the same frame. Only camera updates have been sent and output pixel buffer is retrieved from GPU at each frame render. Branching codes have been eliminated or united under single conditions wherever possible, different memory layouts and types are implemented, tested and chosen for holding scene primitives, lights, materials and output color intensities and finally register spilling have been avoided in several ways.

We think that there are further optimizations possible in terms of increasing the thread coherency by possibly moving shadow rays to a distinct pass, refactoring intersection algorithms, by further eliminating branching codes on kernels and implementing scene acceleration structures on GPU. However, spatial structures impose lots of branching code, and must be treated with caution in the realm of the GPU world. Also GPU limitations disrupt the traversal mechanisms of some scene structures, requiring different solutions which impose additional performance and implementation challenges.

As future work, we want to enhance our GPU ray tracer with more features such as texture support, ambient occlusion, depth of field, bump mapping and other parametric primitives as well as extend our benchmarks by implementing and testing

the applicibility and performance gain of spatial acceleration structures in the GPU environment.

REFERENCES

1. Herken, R., *NVISION08: The Future of Rendering*, 2008, http://origin-developer.nvidia.com/object/nvision08-herken.html, accessed at February 2011.

2. Wikipedia, *Ray Tracing*, 2010, http://en.wikipedia.org/wiki/Ray_tracing, accessed at February 2011.

3. Massal, G., *Ray Tracer in C++*, 2008, http://www.codermind.com/articles/Raytracer-in-C++-Part-I-First-rays.html, accessed at February 2011.

4. Funkhouser, T., *Moller's Ray Triangle Intersection Algorithm*, 2009, http://www.cs.princeton.edu/funk/, accessed at March 2011.

5. Pohl, D., *Ray Tracing Engines in Games*, 2006, http://www.q3rt.de/, accessed at February 2011.

6. Thrane, N., and L. Simonsen, *A Comparison of Acceleration Structures for GPU Assisted Ray Tracing*, M.S. Thesis, University of Aarhus, 2005.

7. Elhassan, I., *An Analysis Of GPU-based Interactive Raytracing*, 2006, ftp://ftp.cs.utexas.edu/pub/, accessed at February 2011.

8. Foley, T., and J. Sugerman, "KD-Tree Acceleration Structures for a GPU Raytracer", *Proceedings of the 2005 ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 2005.

9. Karlsson, F., and C. Ljungstedt, *Ray Tracing Fully Implemented on Programmable Graphics Hardware*, M.S. Thesis, Chalmers University of Technology, 2004.

10. Christen, M., *Ray Tracing on GPU*, B.S. Thesis, University of Applied Sciences Basel, 2005.

11. Britton, A. D., *Full CUDA Implementation of GPGPU Recursive Ray-Tracing*, M.S. Thesis, Purdue University, 2010.

12. Tsiodras, T., *Cuda Ray Tracer*, 2010, http://users.softlab.ntua.gr/ttsiod/, accessed at February 2011.

13. Colak, M., and N. Erdogan, *Accelerated Parallel Ray Tracing on Graphics Hardware*, B.S. Thesis, Istanbul Technical University, 2010.

14. Carr, N., J. Hall, and J. Hart, "The Ray Engine", *Proceedings of the 2002 ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 2002.

15. Purcell T., *Ray Tracing on a Stream Processor*, Ph.D. Thesis, Stanford University, 2004.

16. Zlatuska, M., and V. Havran, "Ray Tracing on a GPU with CUDA - Comparative Study of Three Algorithms", *Proceedings of the 2010 WSCG Conference*, 2010.

17. Wikipedia, *Phong Reflection Model*, 2009, http://en.wikipedia.org/wiki/Phong, accessed at Februrary 2011.

18. Havran, V., *Statistical Comparison of Ray-Shooting Efficiency*, 2001, http://dcgi.felk.cvut.cz/home/havran, accessed at March 2011.

19. Gunther, N., "Realtime Ray Tracing on GPU with BVH-based Packet Traversal", *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, 2007.

20. Whitted, T., "An Improved Illumination Model for Shaded Display", *Communications of the ACM*, 1980.

21. Stanford University, *3D Repository*, 2011, http://www.graphics.stanford.edu, accessed at May 2011.