4-25-2011

GPU-Based Global Illumination Using Lightcuts

Tong Zhang
CGT, zhang214@purdue.edu

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 Tong Zhang

Entitled
GPU-BASED GLOBAL ILLUMINATION USING LIGHTCUTS

For the degree of Master of Science

Is approved by the final examining committee:

James Mohler
Chair

Bedrich Benes

Ronald Glotzbach

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): James Mohler

Approved by: James Mohler 04/20/2011
Head of the Graduate Program Date

Graduate School Form 20
(Revised 9/10)

PURDUE UNIVERSITY
GRADUATE SCHOOL

Research Integrity and Copyright Disclaimer

Title of Thesis/Dissertation:
GPU-BASED GLOBAL ILLUMINATION USING LIGHTCUTS

For the degree of Master of Science

I certify that in the preparation of this thesis, I have observed the provisions of *Purdue University Executive Memorandum No. C-22,* September 6, 1991, *Policy on Integrity in Research.**

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.

Tong Zhang
Printed Name and Signature of Candidate

04/21/2011
Date (month/day/year)

*Located at http://www.purdue.edu/policies/pages/teach_res_outreach/c_22.html

GPU-BASED GLOBAL ILLUMINATION USING LIGHTCUTS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Tong Zhang

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

May 2011

Purdue University

West Lafayette, Indiana

For my dear Dad Shaoyi Zhang and my dear Mom Ping Wu.

ACKNOWLEDGMENTS

I would like to show my sincere appreciation to my committee members: Dr. James Mohler (chair), Dr. Bedrich Benes and Professor Ronald Glotzbach. Dr. Mohler's patient guidance on my Master study has kept me on the right track of computer graphics and research methodology; Dr. Benes has inspired me a lot with his rich experiences in graphics research; warm advice and help from Professor Glotzbach also encouraged me in my research work.

My special thanks goes to Rebecca Lank. Thank you very much for your aid and assistance on my study in Purdue for two years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

GLOSSARY

- Binary Space Partitioning: the technique used to speed up ray-object intersection computation in computer graphics by dividing objects in space into two parts recursively, so that objects in one half can be safely avoided if a ray does not intersect the bounding box of that half space. (Pharr, & Humphreys, 2010)

- Global Illumination : a group of computer graphics algorithms those are intended to generate realistic lighting effects. The primary element is directly illumination. However, global illumination is usually about indirectly illumination, such as diffuse surface lighting, caustics, color bleeding, etc. (Moller, & Haines, 2002)

- GPU : A hardware device primarily working on graphics data processing. It is supposed to accelerate graphical computing significantly. With its parallelism essence and its high performance, GPU is becoming pervasive in scientific computing. nVidia and AMD are recognized as the main GPU manufactures at present (Moller, & Haines, 2002)

- Lightcuts: a method that is able to removed insignificant light sources, like ones too far-away, one too dim or ones nearly perpendicular to target points, from illumination evaluation process. It is claimed that Lightcuts is able to reduce Instant Radiosity (Kell, 1997) rendering complexity from linear to sub-linear. (Walter, 2005)

- Ray-Tracing : a computer graphics algorithm to generate direct Illumination by shooting a primary ray from camera and generate subsequent rays (by reflection and\or refraction) recursively. (Shirley et. al., 2005)

- Radiosity : a computer graphics algorithm to generate indirect illumination by applying finite element method. Usually it emphasizes on diffuse lighting. (Moller, & Haines, 2002)

- Rendering Equation: the mathematical representation (usually an integral equation) of physical light transport phenomena in the real world that is used in computer graphics. This equation indicates that the radiance leaving a point should be equal with the irradiance arriving at the same point. (Dutre, Bekaert, & Bala, 2006)

ABSTRACT

Zhang, Tong. M.S., Purdue University, May 2011. GPU-Based Global Illumination using Lightcuts. Major Professor: James Mohler.

Global Illumination aims to generate high quality images. But due to its high requirements, it is usually quite slow. Research documented in this thesis was intended to offer a hardware and software combined acceleration solution to global illumination. The GPU (using CUDA) was the hardware part of the whole method that applied parallelism to increase performance; the “Lightcuts” algorithm proposed by Walter (2005) at SIGGRAPH 2005 acted as the software method. As the results demonstrated in this thesis, this combined method offers a satisfactory performance boost effect for relatively complex scenes.

CHAPTER 1. INTRODUCTION

This chapter provides an overview of the research in this thesis. After introduction of research background, the research question is stated, followed by research scope, significance, assumptions, limitations and delimitations.

1.1. Background

Global illumination is an active research area in computer graphics, and it is widely adopted in various areas of industries such as animation, movie making, gaming and CAD. Its primary goal is to generate as realistic images as possible. However global illumination requires a large amount of computation and it remains difficult to compute in real-time, so faster rendering speed is still one of the main objectives of research in global illumination.

Several software and hardware solutions have been proposed recently. Lightcuts is a newly introduced algorithm for radiosity computation acceleration compared to traditional radiosity called “Instant Radiosity”. At the same time, the Graphics Processing Unit, abbreviated as GPU, is playing a more and more important role in the graphics area. The GPU is the hardware solution for parallel computation efficiency and numerous practical cases have proven that the GPU does a wonderful job in improving graphics computation performance.

However, currently the GPU has not been integrated into Lightcuts, so this thesis was a trial for this solution and the research was intended to reach a higher efficiency of photorealistic image rendering..

1.2. Organization

This research was intended to answer following questions, which consist of one primary question and several secondary questions.

1.2.1. Primary Question

How much can Lightcuts, together with the GPU, speed up global illumination compared to the traditional Instant Radiosity rendering techniques?

1.2.2. Secondary Questions

1) How much can Lightcuts speed up Instant Radiosity?
2) How much can the use of the GPU speed up Lightcuts?

1.3. Scope

The thesis focused on global illumination technology and covered several core concepts and popular modern techniques; however it did not cover every aspect of global illumination.

The concepts and knowledge that were covered include:

a) Ray-tracing

The whole architecture of the proposed project application was built on Ray-tracing because the diffuse lighting was simulated by putting point lights on diffuse surfaces with hit points. Thus, the traditional Ray-tracing framework was used.

b) Instant Radiosity

This was the algorithm that the author was trying to improve. Instant Radiosity enables convincing effects of global illumination, especially diffuse lighting, by putting point lights on diffuse surface hit points. However there could be too many point lights. Its performance enhancement was one of the key parts of this thesis.

c) Lightcuts

Lightcuts (Walter, 2005) addresses the main problem of Instant Radiosity directly. It is able to avoid unnecessary redundant use of point lights. It only chooses significant point lights for rendering one pixel. This algorithm was the key idea of the software method proposed in this research.

d) Another goal of author's thesis was to improve image quality using several modern techniques. The involved algorithms included:

Photon Mapping: the author used it to render caustics from light reflection and refraction.

Ambient Occlusion: it was used to generate object-object occlusion shadows so as to produce more realistic images.

e) Graphics Processing Unit

The hardware aspect of performance optimization cannot be omitted nowadays. The Graphics Processing Unit is playing a more and more important role in graphics area. Ray-tracing is essentially a parallel procedure, which is the perfect situation into which the GPU fits. The GPU technique was the key method of the hardware solution in this thesis.

f) Several Basic Image Processing Skills

Images computed using global illumination algorithms can be made more realistic by using some processing skills. For example, blooming accounts for the fact that the color of each pixel can impact other pixels, and tone mapping adjusts the colors of the image with different display devices.

1.4. Significance

Global illumination suffers from high requirement of computing time so it is really hard for the program to make it into real-time applications. Instant Radiosity simplifies the diffuse lighting quite a lot. It avoids the high Monte Carlo integration methods cost. The introduction of Lightcuts improves the performance of Instant Radiosity significantly.

But no one has integrated GPU acceleration onto Lightcuts yet. Since Lightcuts is still a pixel-parallel algorithm, GPU integration can bring significant improvement to another level.

1.5. Assumptions

The following assumption was inherent to the pursuit of this study:

- GPU accelerated computation would be significantly faster than CPU version implementation. In other words, the use of GPU needed to be at least 20% faster than the CPU implementation. The main argument was that pixels in generated images are independent with each other, in other words, the data is paralleled inherently. The GPU was the most suitable device to process paralleled data so it could compute different pixels at the same time and make image generation scalable.

1.6. Delimitations

The following delimitations were inherent to the pursuit of this study:

- GPU: Graphics card products from other manufacturers such as AMD and Intel were not be considered in this thesis.

- Radiosity algorithm: There are also other kinds of radiosity algorithms using Monte Carlo integration or iterative matrix solutions. But they were not covered in the thesis.
- Other global illumination algorithms: Many other available global illumination algorithms could also contribute to higher quality images, such as sub-surfaces algorithm, participating media algorithms etc. However they were not included.
- Programming Tools: Microsoft Visual Studio was used as the development environment. Other compilers such as GCC were not involved.

1.7. Limitations

The following limitations were inherent to the pursuit of this study:

- GPU: Only nVidia manufactured graphics cards were involved in the thesis. This was because CUDA was adopted as the basic programming tool for GPU and CUDA was only applicable to nVidia products.
- Radiosity Algorithm: Instant Radiosity was the basic algorithm to be implemented. Another optimized algorithm over Instant Radiosity called Lightcuts was implemented also.
- Programming Tools: CPU code was crafted using C++ because C++ was the best choice to implement a relatively large system; GPU code was implemented by CUDA.

1.8. Summary

This chapter offered an overview of the research, including the research question, research scope and significance. Also the limitations and delimitations of the research were introduced.

CHAPTER 2. LITERATURE REVIEW

This chapter gives a general introduction to the recent research and industrial materials related to Global Illumination and C++, OpenGL and GPU. The literature provides the basic foundation and support to the research in this thesis.

2.1. Ray-Tracing

As Witted (1980) mentioned, the development of global illumination started with several local illumination models at which time no global information had been considered yet. Other objects' illumination had not been involved yet, and this is why it was a local illumination model. At a later time, Blinn and Newell (1976) proposed a technique called 'environmental map' to simulate the illumination from other objects. However, it only partially solved the problem and was not a general solution for global illumination. Refractions were simulated by reverse ordering painting, just like the alpha blending in OpenGL. This was not the real simulation of refractions though. Shadows were calculated by checking the pixel visibility to light sources and viewer. Instead of above mocking techniques, Witted (1980) improved the lighting model by simulating the real light travelling way in the real world and formed a recursive tree structure for the light paths. The radiance of one pixel consists of two parts: reflected radiance and transmitted radiance, which is to say that two lights make a resulting light - one light contains a reflected factor and a transmitted factor. This is the typical case in the real world. Then the two child lights can be traced in the same way and a recursive light tree can be obtained. Please note that due to the reflection and transmission of lights, this tree would involve other objects in the scene, and this leads to global illumination. This is the most seminal contribution of this paper.

Another benefit of this model was “effective visible surface checking”. The surfaces that were not visible to the viewer may be visible through the reflections and, what's more, they were discovered as needed, which was more efficient. This model also facilitates anti-aliasing. Several other rays could be shot along the recursive tree to get multiple samplings. Witted (1980) brought a breakthrough for the light tracing model by introducing the real world light traveling style. This work improved the image quality significantly and made many of the following seminal techniques possible.

Ray tracing could be the starting point for all following advanced techniques. Shirley (2005) gives a clear and instructive explanation on Ray-tracing. According to this literature, Ray-tracing simulated the light traveling in the real world and it traces each viewing ray from the viewers’ eyes along its reflection, refraction, transmission directions. Ray-tracing started by computing viewing rays and intersections between a ray and objects. Intersection calculation was always the performance bottleneck. It required intensive geometrical computation. The next effect to compute was shadowing. It could be done by checking the visibility of light sources. Refraction could be done by tracing the ray using Snell's Law. The most critical issue may be the acceleration for intersections. Bounding box and space participation were the most popular ones that are widely used. Bounding box was the simple representative for one unit of geometry that could simplify the intersection checking; space participation splitted the scene into several smaller parts to reduce the computing cost. Other aspects such as anti-aliasing, soft shadows, depth of field, glossy reflection and motion blur were also necessary for generating higher quality images. They all required more than one viewing ray per pixel for shading computing. Figure 2.1 demonstrates a typical Ray tracing scene.

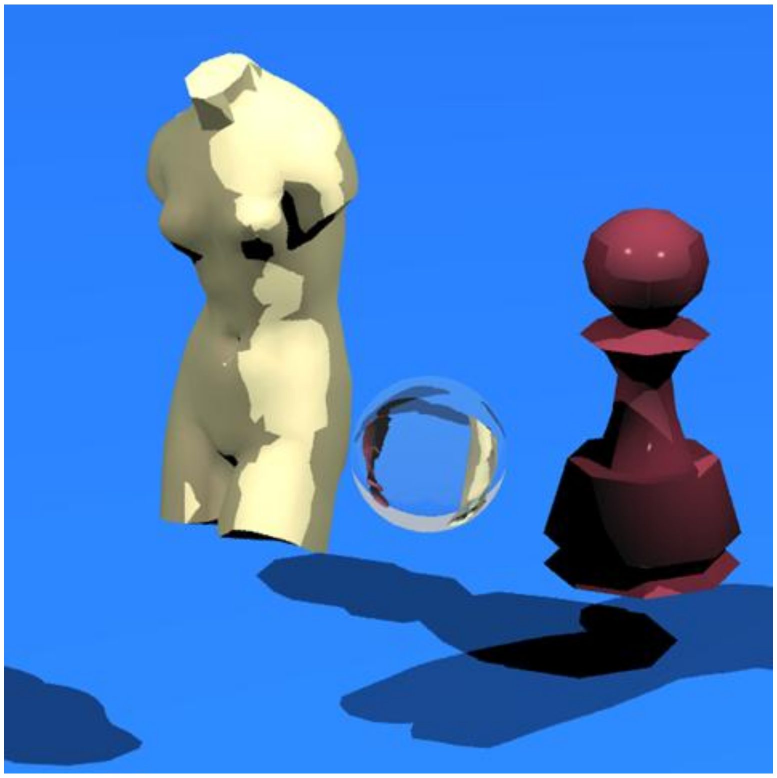


Figure 2.1 A Ray-Tracing Scene

2.2. Radiosity

Radiosity is another major algorithm for global illumination rendering. One introductory book is Cohen and Wallace (1993). This book followed a traditional approach of radiosity computation. Discretizing the radiosity equation was the first step for implementation radiosity. This was the theoretical base for geometry meshing and form factor computation. Form factor was the most important variable that described how much impact each pair of triangles made to each other. It involved the geometrical relationships such as distance and illumination values. There was a form factor matrix, which is the typical linear system. The possible solutions included traditional iterative numerical algorithms and parallel computation. Geometry meshing was the prerequisite to compute form factors. Besides the regular meshing of geometry, adaptive meshing was a more effective algorithm that could generate more meshes on more detailed areas.

Moller and Haines (2002) has a chapter that offers an informative, comprehensive overview of global illumination. This chapter contained both the theoretical and practical aspects and made a perfect bridging and merging between the two. It started with the basic radiometry knowledge with modern optics and photics as the theoretical base. Briefly speaking, Ray-tracing and Radiosity were both derived from concepts and procedures described in physics. Some advanced topics like tone mapping took these as basics. BRDF was a large topic that was also briefed. BRDF depicts the possibility that an influx photon leaves at one specific direction. This was crucial to the realism of the object materials. BRDF was a bidirectional and reversible mathematical abstraction for light reflection, but it doesn't cover the particle characteristics of lights. The traditional BRDF was isotropic and several anisotropic BRDF models have been proposed in recent years too. Two types of BRDF exist: one from physical theories and one from experimental data. The application of BRDF could be categorized into factorization and environment map filtering. Factorization takes the BRDF calculation into the influx angle and outflux angle and the BRDF value was calculated by these two components; environment map filtering took an environment texture and its corresponding filtering map and combines the two to simulate the BRDF effect. Vertex shader, Fragment shader and shader language are the practical aspects of the topic. Vertex shader resides at the GPU as one of the nodes of the graphics pipeline - the vertex processing part. It used to be the fixed procedure in GPU and it was now programmable which allows geometry morphing functions to be customized. Fragment shader also resided at the GPU as a later stage of the graphics pipeline, and it focused on manipulating the pixel shading. Shading language was the tool to program the above two types of shaders. Several interesting effects were covered also. Motion blur was the delayed effect by fast moving objects and it could be implemented using an accumulation buffer. Depth of field was the simulation of the focal effect of a lens. The reflection part included flat reflection, glossy effect and curve reflection. Refraction was based on Snell’s law. Shadow could be obtained by using shadow volume and shadow map. The former one judged whether an object was in the shadow by checking whether it is in the shadow volume or not; the latter one applies the Z-buffer generated by the light source as the shadowing information.

Dutré, Bala, and Bekaert (2006) gives a relatively thorough elaboration on global illumination algorithms in a theoretical view. This book offered sufficient mathematical analysis on the various algorithms and offers a thorough introduction to the current research on global illumination. The most important mathematical tool 'rendering equation' played a key role in global illumination, which was the mathematical tool for almost all relevant algorithms. Almost every algorithm in this book was a variation of this equation. What is noticeable is that this equation made the assumption that light contains only one wave-length and it hit and left the surface at the same point. Instead of an intensive computation on linear systems, Monte Carlo methods had been introduced as an evolutionary solution. Monte Carlo methods were based on probability theories. The initial Monte Carlo methods were used to sample values on continuous signals. This was the typical case of sampling in computer graphics because of the inherent characteristic of discretization for computers. But original sampling method was not always optimal. So importance sampling and stratified sampling were introduced. The former one took more sampling on important data sets and the latter one conducts an adaptive treatment for a non-randomly distributed data sets. Path tracing is introduced first. Original Ray-tracing was presented. Monte Carlo integration into ray-tracing lead to the soft shadowing. A similar technique, called environment map, applied the basic process of ray-tracing to achieve this effect. With probability theory, indirect illumination could be described as the possibility that a light source illuminates the sampled point. In contrast to ray-tracing, light tracing was another possible solution for rendering. It was useful for caustics calculations. Stochastic Radiosity was a great evolution to original radiosity techniques. It replaced the massive form factor calculation with much simpler probability estimations. Now the light energy needed to be transferred iteratively according to the rendering equation. Another way was to randomly walk among the sampled points to estimate the possibilities. Photon density was also one possible solution for radiosity, in which the hit point density was recorded and interpolation was used to render the illumination.

The state of the art research combined several primary algorithms together to form hybrid algorithms. For example, final gathering used another pass to refine the unnecessarily sampled areas again. Multi-pass methods and bidirectional path tracing used more than one tracing style to achieve better effects. Photon-mapping also took two steps: the first one put the photon sampling results and the second phase rendered the illumination by the results. Instant Radiosity adopted point lights to simulate indirect illumination, and Lightcuts was used to make an optimized solution for instant Radiosity.

2.2.1. Instant Radiosity

In the above, many modern radiosity solutions have been explained. Most of them were complex. For example, the original Radiosity required complex object meshing and a form factor matrix solution using iterative matrix solutions. Other Radiosity algorithms adopted Monte-Carlo integration as the algorithm basis. This suffered from relatively slow execution. Instant Radiosity was a very simple but an effective algorithm that simulated diffuse lighting by adding many point lights Kell (1997). This transformed indirect illumination into direct illumination. This is robust, stable and efficient. An important issue was how to sample point lights from traditional area light sources. The Monte Carlo method still served well here. Figure 2.2 shows a scene rendered using Instant Radiosity.

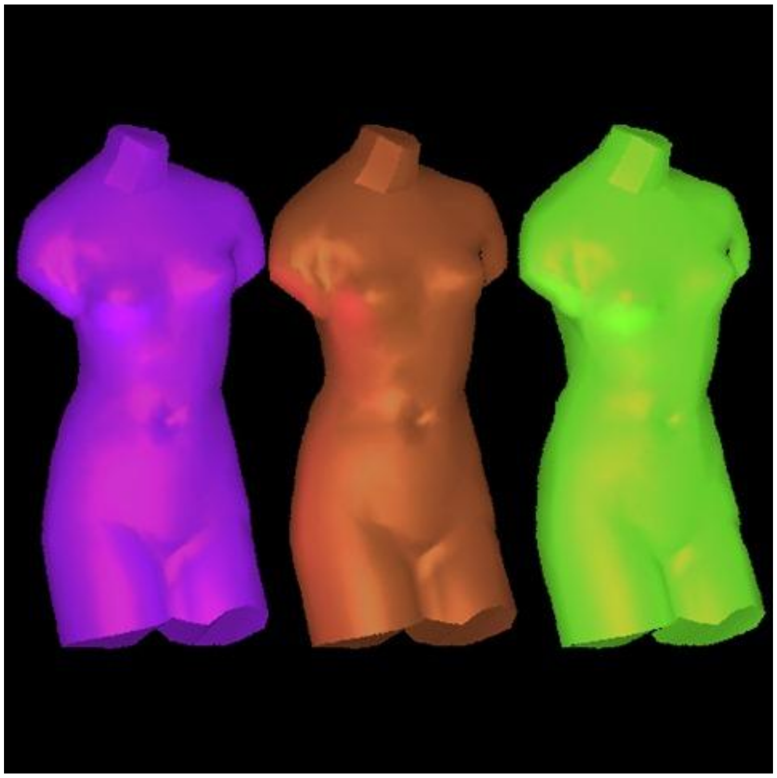


Figure 2.2 A scene rendered using Instant Radiosity

As mentioned above, point lights were mostly formed by the hit points from the rays of the light sources. In other words, it was the result from the light tracing procedure. It suffered from the low efficiency of the light rays projections - some light rays may not be visible at all. To avoid unnecessary light ray projection, this paper proposed a bidirectional tracing procedure. It was a simple and direct process: camera rays were emitted into the scene and record the hit points. These points were visible to the observer and they deserved the light tracing. The next step was to emit light rays to the scene, at the same time whether the hit points were within the area that was visibly evaluated in the first step is check. If yes, put a light point there. The results showed that for the less-occluded scenes, the improvement was quite obvious; however for the scenes that were heavily occluded, the first step would be wasteful.

2.3. Lightcuts

According to Walter, Fernandez, Arbree, Bala, Donikian and Greenberg (2005), Lightcuts was directly derived from Instant Radiosity. From Instant Radiosity, point light sources could be obtained. And later when evaluating indirect lights for each pixel, each point light should be considered. However, not every point light was useful. For some point lights, how much they contributed could even be omitted; another case was that several point lights were too similar so they contributed the same amount. For both the cases, there was space to make further simplification. This paper offered an efficient method for point light simplification. Before conducting simplification, all the light sources needed to be organized in some way and then evaluated. The metrics used in this paper involved the light type, geometry, and materials. With the mathematical tool provided, a final quantity for several lights could be retrieved. With the same metrics, several lights could be represented by a single point light - and it served as the parent node of these lights. Finally, a light tree was formed. When rendering the image, for each pixel it is necessary to first check whether some inter node in the tree is possible so to avoid multiple light evaluation. If so, it can be used. Over all the pixels, the performance improvement was quite significant. One important detail about simplification was that accuracy cannot be lost too much. This was achieved when constructing the tree. For each simplification, only 2% difference was permitted. However this 2% error cannot be accumulated because not all pixels suffered 2% error. Most cases indicated less than 2% error appeared in the final image. So the image quality was almost preserved.

According to Walter, Arbree, Bala and Greenberg (2006), not only point Lightcuts were organized as a tree, but the gather points also had their corresponding tree. When each pixel was evaluated, two trees formed a product graph and all cuts were on this product graph. Also the time instance was involved in this model to support motion blur. With the gather point tree cuts depth of field effect could be provided. The same error analysis as Fernandez, Arbree, Bala, Donikian and Greenberg (2005) was conducted so this method had the same image quality approximation with the original Lightcuts.

2.4. Other Global Illumination Algorithms

2.4.1. Caustics and Photon-Mapping

Caustics was one important lighting effect for transparent objects after direct illumination Jenson (1996). It was very hard to render using traditional ray-tracing, because in caustics each pixel shading value was accumulated from many light sources after refraction so it required a large amount of tracing rays. However it was quite slow and unsatisfactory results had been produced.

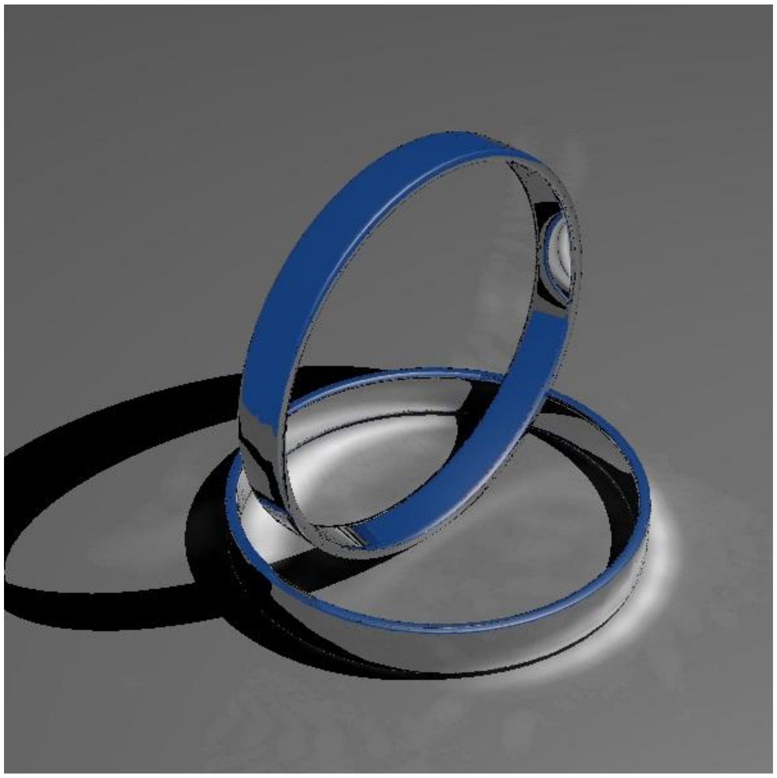


Figure 2.3 Caustics rendering using Photon-mapping

Figure 2.3 is a scene with caustics enabled. Caustics generation was inherently a light tracing process. Another piece of information about how light was accumulated for final rendering was required now. Discretized photons were projected on the plane to trace this information. Next, kernel functions were applied to reconstruct the shading values. Two passes were necessary in this procedure. In the first pass, light sources emitted many light rays that hit on the objects and the planes in the scene - directly or indirectly. For each hit point, a 'photon' was recorded on this point. After this pass, a photon set was formed. In the second pass, the author gave two usages. First was for general use. In this case, not only caustics was considered. With these photon densities the illumination of some area can be calculated. This gave important information about shadowing. Thus the shadow ray number could be reduced. The second case was for caustics computation. In this case, some kernel functions reconstructed the photon illuminations by interpolation so to get continuous caustics.

Photon-mapping is actually another form of traditional Ray-tracing, without any global illumination computation involved. The only difference between Photon-mapping and Ray-tracing is that Photon-mapping shoots rays from light sources.

2.4.2. Ambient Occlusion

Ambient occlusion is one newly proposed image quality refinement method. It emphasized the ambient light aspect: the occlusions among the geometries had an impact on the ambient light illumination (Kontkanen & Laine, 2005). It was darker in the areas that were occluded heavily. This idea can bring more realistic illumination effects. The initial ambient occlusion emphasized self-occlusion effects while no other objects were considered. This literature filled this gap. Its basic idea was as follows: When each pixel was evaluated, a spherical cap was captured with the pixel's normal vector as the axis. This was the normal case for how a point in real world ambient light was absorbed. To sample the occluding factor over this sphere, discretizing the sphere into many sampling squares and sampling an occluding value of this square - 0 for occluded and 1 for not occluded are required. Lastly all these values together were evaluated to get an occluding value. Other objects in the scene were used to check the intersections. Figure 2.4 shows that this method can improve realism significantly. With this algorithm, shadows from ambient light simulated the situations in the real world. However this algorithm required extra preprocessing before rendering, but the performance in run-time was quite promising.

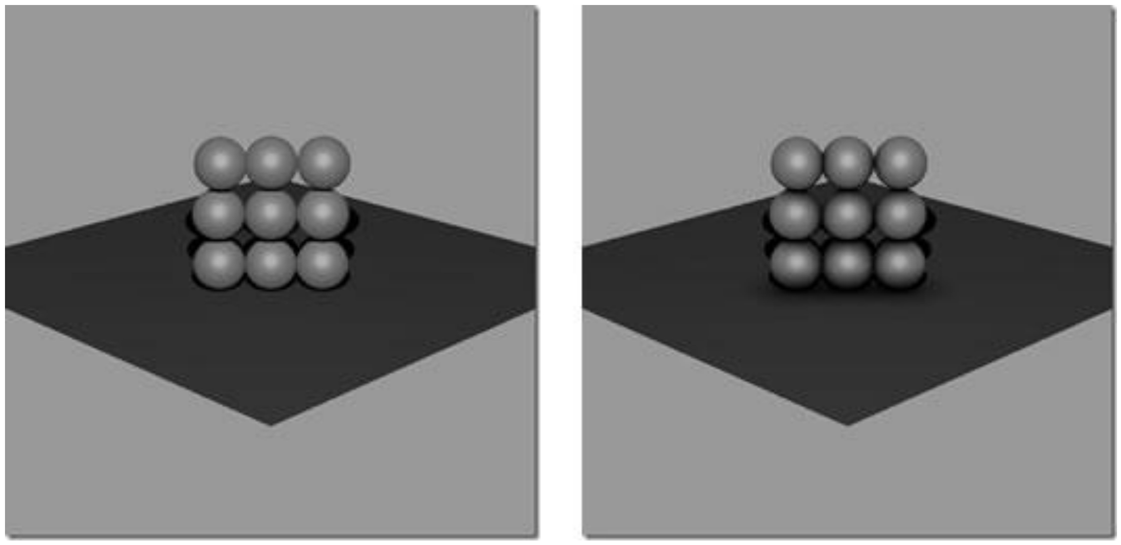


Figure 2.4 An Ambient Occlusion Scene

2.5. Implementation: C++, OpenGL and GPU

The above mentioned techniques mostly focus on the theoretical aspects. Implementation also matters a lot and in this phase many possible effective improvements can be applied.

2.5.1. C++

For instance, a proper application of C++ and a good object-oriented design of the rendering system can improve the extensibility and performance greatly, and can also bring the benefit of easiness of debugging. Indeed, C++ and design patterns took key roles in engineering practices. Nowadays graphics hardware cannot be neglected. Proper manipulation of the graphics processing unit can significantly improve the performance. This was critical because global illumination was acknowledged as an off-line process instead of a real-time interactive process. However, GPU was suitable for parallel algorithms. Luckily the key algorithms were pixel based so it was inherently paralleled.

A rendering engine was a relatively large system, so a lot of engineering aspects needed to be considered, especially for C++ - the API that was adopted as the basic implementation tool. C++ was a powerful but tough language. It was suitable for large systems but special care should be taken in many aspects especially details. Otherwise, the developers would go crazy easily. A very good book for guiding C++ users on how to use C++ properly is Meyers (2005). It provided many useful or even critical suggestions. It covered basic usage such as constructor, destructor, new and delete operators, exceptions, etc., and design views such as inheritance rules, smart pointers, etc. Constructor and destructor was a big topic that brings problems sometimes. The C++ compiler generated default operators in some cases and it had to be realized; for destructors, in base classes, 'virtual' keyword is necessary or the resource may leak in the child classes. What's more important, no exception throwing was allowed in destructors, or the whole system was not exception safe. Resource management included smart pointers, which were a very useful skill to manage resources. Smart pointers were the classes that could handle resource allocation and deallocation automatically and required no extra manual work for the programmers. Several types of these were: scope pointer, shared pointer, reference pointer, etc. Proper manipulations on them can improve the stability and design elegance of the system significantly. Several design issues were also covered. Class design had to comply with one of the basic OOD rules: encapsulation. It meant that no private members could be exposed outside of the class. Another important guideline was: don't return objects directly in functions because it would invoke costly object copying. And maybe the most important consideration is that the less dependence the better. Dependence was probably the first culprit that brings most of the problems. Dependence meant complexity and hardness of debugging. A good system design was to make everything as simple as possible, instead of the contrary. Also, a more advanced topic, 'generic programming', was introduced. In C++, generic programming took templates as the basic tool, which contained the trickiest skills in C++, such as the technique called 'traits'. It used a C++ template specialization to introduce runtime distinctions. Meta-programming mixed macro and template together and it was probably the peak of the C++ usage. With this book, a relatively stable, extensive and efficient C++ rendering system was possible. It was known that most part of the development time was spent in debugging.

2.5.2. OpenGL

The graphics API used is OpenGL. OpenGL is now the industrial standard. It is efficient and versatile. The first book that every OpenGL programmer needs to refer to Shreiner, Woo, Neider, and Davis (2007). This book covered almost every aspect of OpenGL. First the pipeline of OpenGL was introduced. This was the key for the whole of OpenGL. The first stage was geometry related processing nodes such as call lists, evaluators and the vertex shader. They were all about the basic geometrical computations.

The next stage was rasterization. This was the process that parsed the former geometrical objects into pixels to meet the inherent discretization of computers. It cost the most CPU cycles among all the phases in the pipeline. After the pixels were done, textures and other pixel operations such as fragment shaders could be applied.

Lastly, a pixel value was transferred to the frame buffer. The geometry primitives API was used to put the geometrical information into the OpenGL pipeline for rendering; viewing the perspective setup API sets camera parameters. These two parts were basic steps when developing graphics applications, while for global illumination applications an individual mechanism was crafted from scratch. The color API and the lighting API were used for shading functions. They determined the final shading values. Alpha-blending, anti-aliasing, fogging and texturing, were implemented individually. Basically speaking, OpenGL was mainly for interactive applications. Most critical processing steps were inherent in this API and the programmer did not have too much freedom to customize them. For example, in the anti-aliasing part, some new sampling strategies were better, but they cannot be integrated into the existing OpenGL pipelines; another example was viewing (camera) setup, there was no 'depth of field' in OpenGL.

2.5.3. GPU

With careful observation it was clear that pixel-based global illumination algorithm was paralleled. Unlike original patch-based Radiosity, Instant Radiosity enabled individual pixel based calculations. This inherent characteristic enabled us to apply the GPU to rendering algorithms to achieve a faster execution. In other words, the GPU was an unavoidable topic nowadays in graphics, in interactive graphics applications such as games, and in high computation density image synthesis field. The most popular GPU and its programming interface was from nVidia. So CUDA was the tool that was to be used in our project. The key issue about this was how to use the GPU in the right way to maximize the boost by GPU. Ryoo, Rodrigues, Baghsorkhi, Stone, Kirk and Hwu (2008) was a good paper on how to use GPU in the right way. It started with the detailed explanation of GPU architecture because the first step is to understand how the GPU works. The GPU consisted of many paralleled stream processors that represented one thread; eight stream processors form a stream multi-processor and many stream multi-processors formed a processing grid. The numbers varied among the different versions of GPU. There was a global memory on a GPU chip that can be accessed by all threads; one shared memory in each stream multi-processor that was visible to all its eight stream processors and it was much faster than the global memory. Basically speaking, the GPU was a data-paralleled processing device, and special care needed to be taken with the thread-memory accessing model that is critical to the GPU performance.

There were three mentioned methods for CUDA performance tuning. The first one was local memory storage. As mentioned above, global memory was much slower than the shared memory in each stream multi-processor. Shared memory could be used as a cache for eight threads in the same stream multi-processor. According to the paper, this could improve the performance by 4.5 times. The second method was loop unrolling. This didn’t show a significant performance improvement in the paper but it did have some positive effects in some cases. The last method was more strategic. It hid the memory accessing time by allocating another cluster of threads on the processors. For some memory access intensive applications this could improve the performance a lot. Of course there were other ways to improve CUDA efficiency. To make the best of our application performance, how our application works needs to be understood well- is it data parallel or instruction parallel? If it is data parallel, are there many memory accessing actions? Are they reading or writing? After that, with a good understanding of the GPU working model, a proper way to speed up the code can be found.

2.6. Summary

Several important sources relevant to the author’s research have been briefed here. A series of global illumination techniques from Ray-tracing to Radiosity and finally to Lightcuts, together with some other popular algorithms such as Photon Mapping and Ambient Occlusion, are all covered. The corresponding implementation tools such as C++, OpenGL and CUDA are also mentioned. These papers and books offer the basic knowledge and also inspire the author with the research idea in the thesis.

CHAPTER 3. METHODOLOGY

This chapter provides the framework of the research methodology. The process that the research is introduced here.

3.1. Hypothesis

H_0: The GPU cannot accelerate global illumination significantly enough.

H_a: The GPU can accelerate global illumination significantly enough.

3.2. Sampling and Sampling Approach

Since performance is the key concern of this research, sampling was focused on performance, including: CPU rendering time cost, GPU rendering time cost and the breakdown of the rendering time, as initialization time cost and intersection time cost.

The sample approach is to use CPU cycle count as the basic measuring basis to check the execution time of the rendering time cost of both CPU version implementation and GPU version. By obtaining CPU cycle counts of the time when rendering begins and ends, the difference of two values can be adopted as the benchmark for further analysis and judgment of the results of the research.

3.3. Variables and Unit of Measurement

The variables to be measured are of course the measurement of rendering performance - rendering time cost, in seconds, with two or three as the decimal places of precision. However, the initial measure variables - CPU cycle marks - need to be transformed into seconds.

3.4. Assessment Instruments

There are two required assessment instruments:

a) CPU: Intel or AMD CPUs can be adopted in the research, as long as it remains consistent.

b) GPU: Because CUDA is the choice as the implementation tool, only GPUs from nVidia can be considered - CUDA is the product from nVidia and it is not the industrial standard yet.

There are also some required infrastructure elements:

a) A complete computer work station.

b) A data analysis software package. Microsoft Excel is a good choice.

3.5. Implementation

The core algorithm to implement is Instant Radiosity and its enhancement called Lightcuts. Instant Radiosity achieves global illumination by putting point lights on diffuse surfaces followed by another pass of traditional ray-tracing procedure. Lightcuts is a newly proposed improvement on Instant Radiosity that assembles all the point lights and computes one representative point light for a bunch of similar point lights, which reduces the light computation amount in rendering.

The application was implemented using C++ and CUDA. C++ is a very good choice for crafting a relatively large system. CUDA is the language the author chose as the tool to program GPU. CUDA is a C language extension that is easy to start with and enables programmers to control GPUs directly.

3.6. Procedure

The basic procedure of the algorithms is a two-pass procedure. The first pass is basically a traditional ray-tracing that computes specular colors, while the difference is that the point lights that represent diffuse reflections were put on the ray hit points on diffuse surfaces.

The second pass is still a traditional ray-tracing procedure but this pass will only consider the diffuse point lights that have been added in the first pass. The computed colors were directly integrated into the final results. The Lightcuts algorithm (Walter, 2005) goes in this pass. Non-significant virtual point lights to one hit point were skipped.

The procedure on CPU is a sequential style. Each pixel is computed after one pixel and it can only start the computation after the last pixel has finished its calculation. But the GPU can handle many pixels at the same time. So only one pixel’s time cost is required for many pixels. In the author’s implementation, one row of pixels (about 512) was processed at the same time on GPU.

The testing scenes consisted of several versions of the Cornell Box, which is considered as a standard scene for testing global illumination algorithms. The Cornell Box scene provides physically-exact bases for pixel-wise algorithm testing, so they can serve as the ideal testing benchmarks for the GPU version algorithms of author’s work.

3.7. Summary

The methodology of research in the thesis has been introduced in this chapter. Important factors such as sampling, variables to sample, instruments, procedure and implementation are stated in detail.

CHAPTER 4. METHOD OF IMPLEMENTATION

This chapter covers the necessary implementation techniques used for crafting this research application, including data structures, general workflow, algorithms and their parallelization on GPU using CUDA. The algorithm parallelization is the emphasis because it is the key to the performance enhancement. The traditional Ray-Tracing and original Lightcuts algorithm in Walter (2006) is elaborated in detail as well as how they are paralleled using CUDA.

4.1. Workflow

As mentioned in Chapter 3, it is a two-pass procedure of rendering. The first pass is a typical Ray-Tracing, and the only difference is that Virtual Point Lights were put in the target 3D scene on the hit points that are shiny enough, on diffuse surfaces of the objects. However they will not serve as lights sources in the first pass, because they are supposed to contribute to diffuse illumination only. The second pass will take the Virtual Point Lights added in the first pass as light sources, and the same Ray-Tracing process will take place to compute diffuse-diffuse illumination, which is the essence of global illumination.

In second pass, a large amount of light sources (the virtual point light sources) usually cause expensive computations on illumination evaluation of a single hit point - however some of the virtual point light sources do not have significant contribution. For example, the lights that don’t point to the hit point, and the lights those are too far away or too dim. The basic idea of the Lightcuts algorithm proposed in Walter (2006) is the very solution for this situation. Based on its light contribution metric method, the unnecessary virtual point lights can be skipped in color evaluation in the second pass, hence the performance of second pass can be enhanced accordingly. According to Walter (2006), it can reduce the complexity of illumination computation from linear to sub-linear. Figure 4.1 depicts the workflow of the whole system.

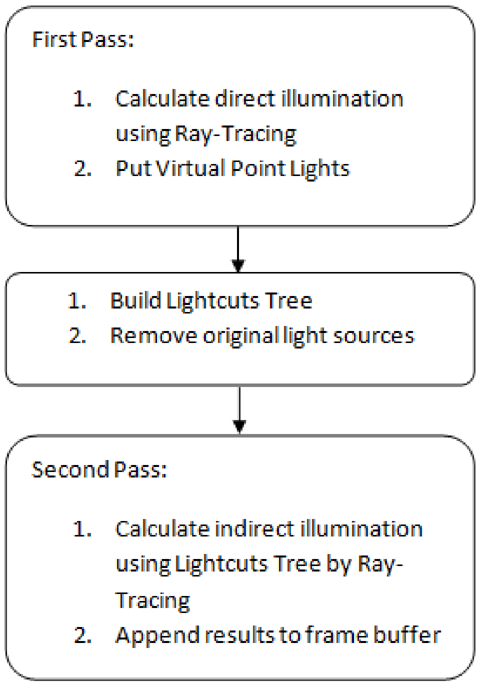

First Pass:
1. Calculate direct illumination using Ray-Tracing
2. Put Virtual Point Lights
1. Build Lightcuts Tree
2. Remove original light sources
Second Pass:
1. Calculate indirect illumination using Lightcuts Tree by Ray-Tracing
2. Append results to frame buffer

Figure 4.1 Workflow Chart

4.2. Data Structures

This section introduces the key data structures used in the research application. The CPU implementation takes Object-Oriented Design as the basic design strategy, however it is not possible for GPU CUDA code, which doesn’t support Object-Oriented language features, so all data members in the class hierarchy should be put in a struct together.

There are many data structures in the system, while only the most important ones are covered in this section, such as Ray, Light and geometry.

4.2.1. Ray

The basic information to take in a Ray is its id, starting position coordinate, direction vector, color, hit point coordinate (if applicable) and some other data that may facilitate the engine to process, such as offset values used in randomized sampler and a boolean mark indicating whether it is inside an object or not.There is no inheritance or polymorphism in this struct, so the GPU CUDA code can share the same struct declaration. Appendix B provides the source code for this data structure.

4.2.2. Light

On the CPU, lights should follow an inherited class hierarchy so as to represent different types of light sources. First, in the parent class of Light, it should contain some common data such as attenuation, ambient color, diffuse color and specular color.

The first concrete light source type is directional point light, from which light rays were emitted from a single point in 3D space along the indicated direction, but they also spread within the hemi-spherical space with a cosine attenuation factor so the extra data needed are only its position and its direction.

The second supported light type is directional light source, which only requires extra information of its direction.

As mentioned earlier, CUDA does not support inheritance and polymorphism, so the only solution is to combine all information in all previous classes together, and put an enum variable to mark its light type.

The initial parameters for rendering were set up through regular CPU C\C++ code, when these parameters are to be transferred from CPU to GPU, the data from different sub-classes, as DirPointLight and DirLight here, were copied to its corresponding fields. Appendix B provides the source code for this data structure.

4.2.3. Geometry primitives

Three types of primitives are supported: sphere, square and triangle. They are represented on the CPU as belows. The parent class contains common data that all geometry primitives require, in which material is a common struct containing the object’s light characteristics: id, reflection factor, refraction factor, refraction constant, emission factor and material reference. In the material section, specular, diffuse and ambient color should be recorded, and also the shininess factor.

A Triangle needs its three vertice’s 3D coordinates, surface normal, and vertice normals if applicable. Two boolean variables mark whether there are vertice normals and whether illumination should be smoothed along the surface.

Only two extra members are needed for a Sphere: center coordinate and its radius, but it is a little more complex for a square. Its center point coordinate, width and height, surface normal, and its two direction vectors along the edges should also be included to indicate square’s direction.

The same strategy could be adopted again for geometry primitives’ representation on GPU, all data members were put in the same GPU struct together, and the same as light class hierarchy, different data members were copied to its corresponding fields in the GPU geometry struct. Appendix B provides the source code for this data structure.

4.3. System Architecture

The whole application consists of several components and each of them addresses one aspect of the rendering process. Different parts as geometry, scene setup (camera, objects coordinates, etc.), rendering engine, anti-aliasing, binary space partitioning, etc. could be organized and understood from input to output, shown in Figure 4.2.

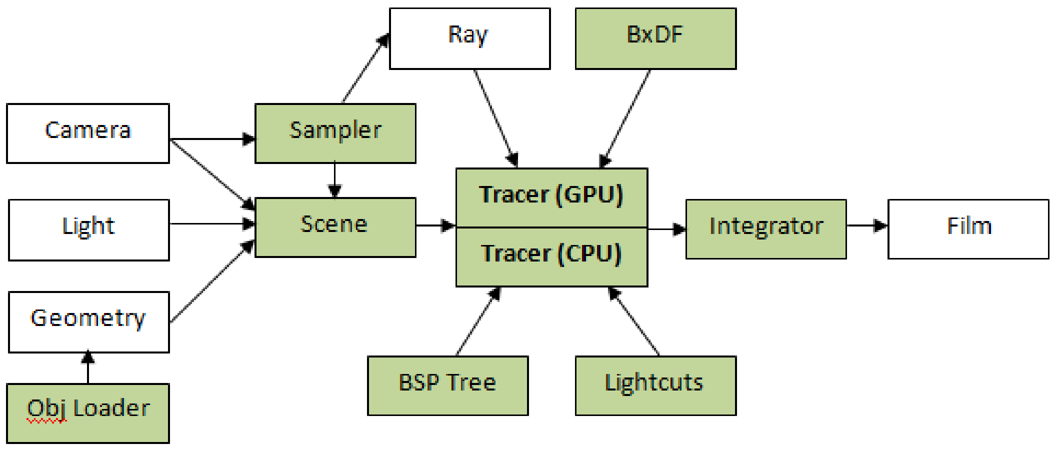


Figure 4.2 System Architecture Chart

As shown in Figure 4.2, several input elements include camera setting, light source setup and geometry data (geometry can be input from external files using Obj Loader). These should be the beginning of the whole engine. After all information mentioned are ready, the Sampler will generate rays according to the camera setting and provide them to scene manager. Then the scene manager will take the geometry, light sources and rays to the Tracer (CPU or GPU version) and drive it to execute the rendering. During rendering, the Tracer applies BxDF (as the color evaluation core), BSP Tree and Lightcuts component to do a two-pass Ray-Tracing procedure, on CPU or GPU, as indicated from user. At last, the Integrator will take the computed colors from the Tracer and compose the final pixel colors. Anti-Aliasing also happens in the Integrator.

4.4. Ray-Tracing and its Parallelization

Ray-Tracing is the basic framework of pixel illumination evaluation. Due to the Independence essence of rays, the parallelization could be applied upon the computation of these rays.

After all data mentioned in section 4.2 has been transferred to GPU, they become read-only global information, visible to all rays shot by the rendering engine. According to Shirley (2005), one ray goes along the shooting direction until it hits a point on an object, after it collects its current evaluated color, another two individual rays for reflection and refraction to go through the same ray process. However one or both of them may not be shot considering the material of the hit object. At last, a recursive binary tree was formed with one ray as its node element Turner (1980). Then a pixel's color can be evaluated by summing up all rays in the tree from bottom to top, weighted by corresponding reflection and refraction ratio in the ray.

Ray-Object hit testing is usually the most time-consuming part of a Ray-tracer. According to Shirley (2005), the most common way to accelerate this computation is using Binary Space Partitioning, which partitions the whole space into two parts recursively so that a ray can search for its hit point faster by avoiding all objects in the half of space which the ray does not intersects. It results in a complexity of lg(n), instead of the original linear complexity.

4.4.1. Ray-Tracing Recursion Iteratization

C\C++ on the CPU supports function recursion intrinsically, so there is no extra difficulty to implement ray-tracing. However, nVidia's GPU programming language extension CUDA doesn't support function recursion, for the sake of performance - because popping and pushing a stack would bring a lot of runtime cost.

The basic strategy is to replace recursion by iteration so it is no longer necessary to use stacks. Each iteration handles one layer of the ray binary tree, which are produced by last iteration as reflection and refraction rays. The depth of a ray binary tree is the maximum ray-tracing depth, and the total ray number is 2^n-1, in which n is the depth number.

As mentioned before, the final pixel color is a weighted illumination sum of all rays in the tree, so the whole tree should be stored, and the pixel color can be accumulated iteratively from rays in the bottom layer and upward to the original single ray (the root ray). This is accomplished on GPU under CPU's control.

Array is the best representation of a ray tree on GPU, in which rays are order from upper layer to bottom layer and from left to right for children in the same layer. Figure 4.3 shows a concrete example:

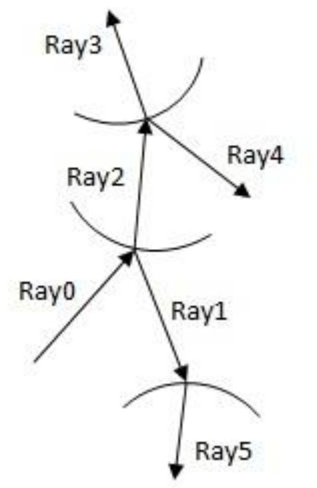


Figure 4.3 A typical ray-tracing scenario

As Figure 4.3 shows, the root ray (ray0) produces two rays, ray1 and ray2 (ray1 for reflection ray and ray2 for refraction ray), then ray2 produces ray3 and ray4, ray2 generates ray5 only, assuming ray2’s hit object cannot refract lights.

Figure 4.4 depicts the procedure of a pixel color computation by iterative recursion along the ray binary tree. In which ray3 and ray4 contributes to ray1, ray5 contributes to ray2, then ray1 and ray2 contributes to the root ray, at last the final illumination can be retrieved from ray0. As shown, this synthesis procedure also goes layer by layer, from bottom to top.

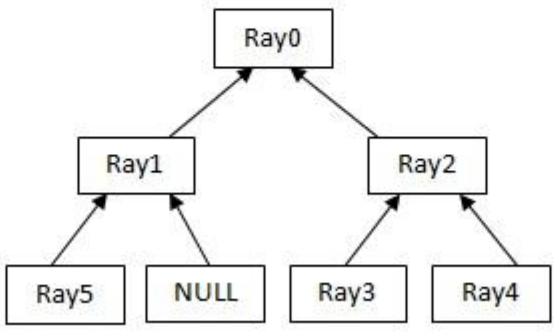


Figure 4.4 A binary ray tree and its synthesis procedure

Figure 4.5 shows the tree representation in GPU memory as an array, and rays are ordered in the way mentioned above.

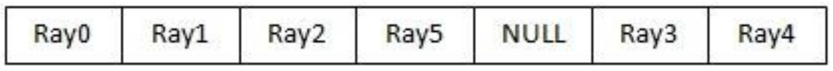


Figure 4.5 Array representation of a ray binary tree

4.4.2. Binary Space Partitioning Recursion Iterization

Consider this scenario: there are 10000 triangles, and a ray hits only one of them, so traversing all of them is not a proper solution. What's more, ray object intersection computation is usually the most time-consuming part in a ray-tracing application. The regular solution is to divide a space into two parts recursively, if a ray hits only one of the two parts, then all the objects in the missing part can be avoided from intersection computation Shirley (2005).

The data structure for recording this partitioning information is called BSP tree, which can be built recursively, similar to the procedure described in sub-section 4.3.1. It takes a bounding box that holds the whole space as the starting point, and then a bounding box was divided into two sub-boxes, with each of them containing nearly same number of geometry primitives. Then the same process was applied on these two sub-boxes, recursively, until there is small enough number of primitives. BSP tree building can be finished on CPU and the resulting tree can be directly copied to GPU. Figure 4.6 shows an example.

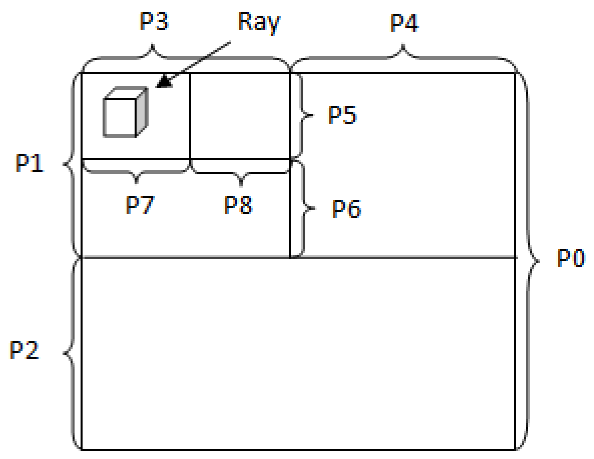


Figure 4.6 Bounding Box and Binary Space Partition

As shown in Figure 4.6, when a ray has been shot, it is checked which box it intersects from outside to inside boxes. It hits box1 in box0 (the root box) first, so all objects in box2 can be safely avoided. Then, box3 in box1 is hit, and box4 doesn’t. So primitives in box4 can be avoided too. As shown, BSP acceleration can avoid unnecessary ray-object intersection computation.

A node of a BSP tree contains the space dimension it covers but only leaf nodes contain geometry primitives. On the CPU, it is not difficult to implement because the CPU supports recursion function calling. And iterative recursion can be applied again to solve this problem on the GPU. All BSP tree nodes can be arranged sequentially in an array on the GPU, and BSP tree traversal follows a layer-by-layer iteration style, as shown in Figure 4.7.

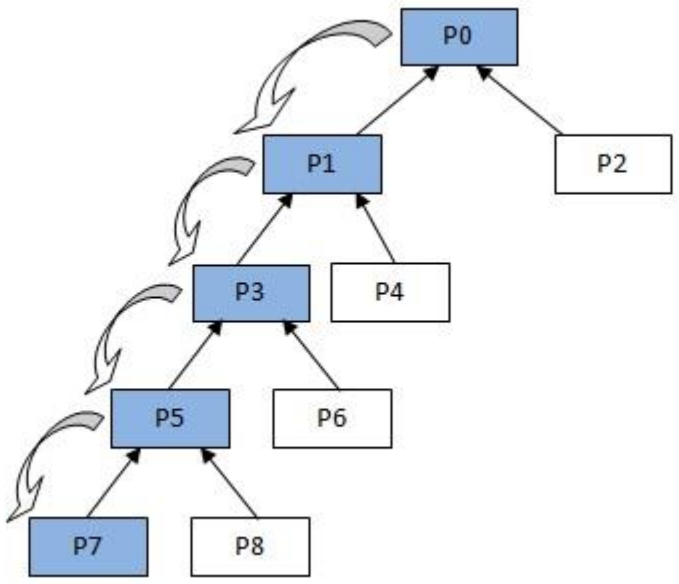


Figure 4.7 BSP Tree and its traversal

In Figure 4.7, node P0 represents the whole space, node P1 and P2 are the two children of node P0. Each represents one half of the space, and the same for node 3, 4, 5, 6, 7 and 8. The workflow for finding the last hit node is conducted from top to bottom – P0, P1, P3, P5 and P7.

On the GPU, the tree node is indexed by its positions in the array, as shown in Figure 4.8.

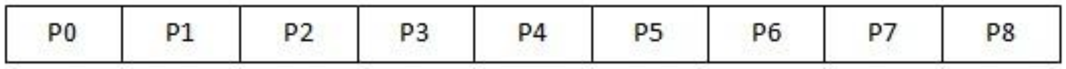


Figure 4.8 Array Representation of a BSP Tree

After the hit node is found, all its primitives were checked. Then the final hit point was found, or not if this ray did not hit any primitive.

4.5. <u>Lightcuts and its Parallelization</u>

Lightcuts was first proposed in Walter (2005), which addresses the problem of over-amount of light sources in a scene. According to the paper, it is able to avoid unnecessary light source illumination evaluations from the ones that have little contribution to the target pixel effectively.

4.5.1. Introduction to Lightcuts

According to Walter (2005), in complex scenes, there are usually tens of thousands of light sources. The traditional way of pixel color evaluation is iterating each of them. It results in a linear complexity for scene illumination calculation that is unacceptable in complex scenes. However, some of light sources are totally non-significant. For example, the lights that are too far away from a hit point, lights too dim or direction lights that don’t point to the target pixel, are not necessary and can be safely avoided.

The solution is actually simple: just cut them off. The problem is: how to tell whether a light is significant enough or not? In Walter (2005), considered factors such as visibility, geometry (distance to a hit point, directions if applicable), intensity and materials are introduced as the metric aspects of significance of a light. And the next question is: what data structure should be used to record significance information? A binary tree is used with each node as one light in a scene Walter (2005). A parent of two nodes is selected from one of its two children, because we don’t need extra virtual lights and what’s more, it is able to save memory space.

After the light tree has been built, it was traversed from the root down to the bottom recursively. Here an error bounding metric method was applied on each node. If the calculated error is below a tolerance value, the visited node was chosen to be present in the final pixel color evaluation; if the bounding error exceeds the tolerance value, its two children were visited then Walter (2005).

4.5.2. Light-Tree Building

The light tree is built from the bottom-up recursively, until there is only one node left. The first step is to select two similar lights from the scene (mainly judged based on their geometric similarity, like distance and light direction), and then one of them was selected as the parent, with intensity as the possibility of selection. Walter (2005) provides a formula to evaluate this “similarity”, and because there are only directional point lights in this research, the formula adopted here is slightly different from the one in Walter (2005):

$$I_C * (D_C^{-2} + (1 - \cos\beta)^2) \qquad \text{(Equation 4.1)}$$

As shown in Equation 4.1, I_C is the total illumination of two virtual point lights; D_C is the distance between two point lights (Walter (2005)) put it as the diagonal length of the bounding box); β is the angle between the direction vectors of two point lights (In Walter (2005) it is the half angle of the bounding cone). Figure 4.9 shows an example.

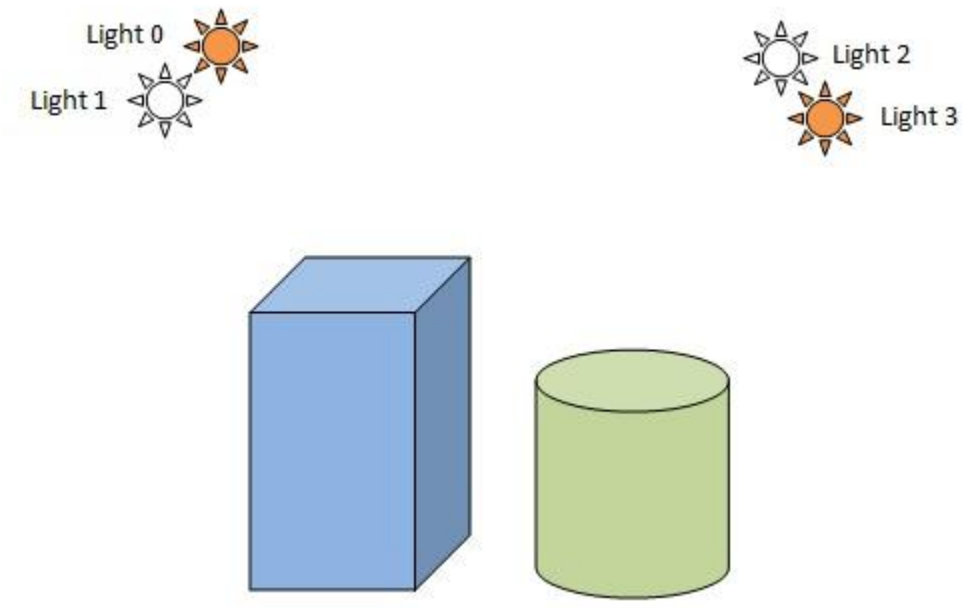


Figure 4.9 A Lightcuts scene

As Figure 4.9 demonstrates, there are four lights in the scene. As shown, light0 and light1 are similar enough because they are close enough, light2 and light3 are considered as similar too by the same reason. Then light0 and light3 are chosen as parents (suppose they are the brighter ones). As last, light 3 is chosen as the tree root (suppose light 3 is brighter than light 1).

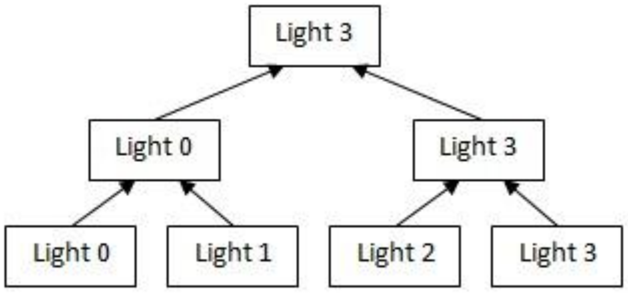


Figure 4.10 A Lightcuts Tree

Figure 4.10 depicts the Lightcuts tree in the sample scene. Since this tree will not change through out the rendering, this building process can be put on CPU, and the tree can be copied to GPU. As mentioned before, this binary tree can also be represented in an array, as shown in Figure 4.11.

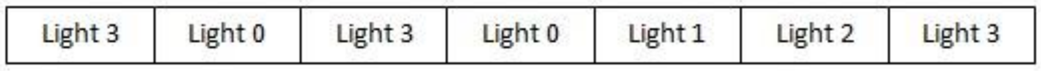


Figure 4.11 Array representation of the Lighcuts Tree

4.5.3. Paralleled Light-tree Evaluation

With a Lightcuts tree is available, it is universally available in the process of a pixel color computation. According to Walter (2005) it is also a recursive process: each traversal starts from the root, if the current node's error bound value is below the preset tolerance value, the light is chosen and its children will not be visited. If the error bound value is above the tolerance value, it was be skipped and its two direct children were visited then, until the lights with error bound values smaller than tolerance are selected. The formula to estimate bounding error of two directional point lights is as below (taken from Walter (2005)):

$$\max(\cos\beta, 0) * (y - x)^{-2} \qquad \text{(Equation 4.2)}$$

In Equation 4.2, β is the angle between the virtual point's direction and the normal vector of the hit point; y is the light's position and x is the hit point position.

It is trivial to implement recursion on CPU because CPU supports function recursive calling; this procedure should be completed iteratively layer by layer, by a top-down manner on GPU.

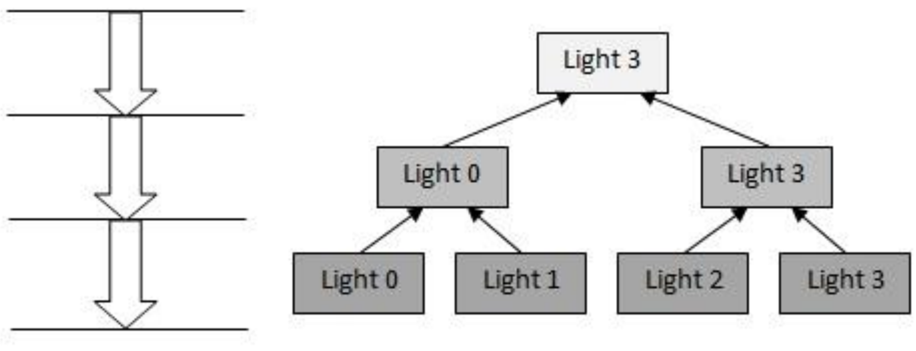


Figure 4.12 Recursion traversal of a Lightcuts tree

By Figure 4.12, a Lightcuts tree is visited from root downward, until the right lights are chosen.

4.6. Summary

As the two key components of this research, Ray-Tracing and Lightcuts, demonstrated, the most important skill here is replacing recursion by iteration on GPU, and the tree data structure on CPU should be stored as a one-dimension array, on which iteration occurs. All these iterations occur simultaneously on GPU cores, with each core taking one task of one pixel's color computation.

Due to the pixel independence essence of Ray-Tracing, parallelism is supposed to be able to enhance performance significantly. What's more, Lightcuts is considered to be able to further improve pixel color computation performance. The detailed performance testing results upon three different scenes with different complexity were elaborated in the following chapter, in which it was clear how much all the mentioned techniques in this chapter could take effect.

CHAPTER 5. RESULTS AND ANALYSIS

In this chapter, the acceleration effects are demonstrated through three different scenes with different complexities. The measured rendering time is laid out first, and its analysis follows. After analysis, it could be clear whether the solution proposed in chapter 4 is effective enough or not, and the validity of this research could be verified.

5.1. Scene Testing Results

This section elaborates testing details including testing platform, perceptual checking tool and final testing results.

5.1.1. Testing Platform

The computer used for testing has the following characteristics:

CPU: Intel Xeon E5520 @ 2.27GHz 2.26GHz

GPU: nVidia Tesla C1060

5.1.2. Perceptual Difference Evaluation Tool

To promise the acceleration solution proposed in this research does not damage image quality by introducing discernible differences or even errors, all accelerated images were compared with the original non-accelerated CPU generated image. A third party open source tool called Perceptual Image Diff was applied to all generated images for perceptual error detection.

Briefly speaking, this tool is based on spatial frequency, luminance, color and observer parameters. With its algorithms, perceptual differences of images can be effectively examined. More details can be found at http://pdiff.sourceforge.net/.

5.1.3. Results

Three scenes with different complexities are designed for the testing work of this research. A very simple scene with only a few geometric primitives, a scene with 1418 triangles and a scene with 4698 triangles are separately rendered and their rendering results including generated images, rendering time cost are recorded.

5.1.3.1. Simple Scene

This scene contains only two spheres in a Cornell Box that consists of five sides (five squares), so there are only seven primitives in the scene.

Four different scene configurations are involved: CPU without Lightcuts, CPU with Lightcuts, GPU without Lightcuts and GPU with Lightcuts. Each of the four configurations has been tested three times and their average values were used for analysis. Table 5.1 presents the testing results (All data values are in seconds).

Table 5.1

Simple Scene Testing Results

	First Pass (In Seconds)	Second Pass (In Seconds)	Total (In Seconds)	Light Ratio (%)	P-Diff Check
CPU - no Lightcuts	0.808	76.945	77.753	X	X
CPU - Lightcuts	0.869	394.284	395.160	63.007%	Pass
GPU - no Lightcuts	7.891	45.457	53.353	X	Pass
GPU - Lightcuts	7.414	65.856	73.287	64.035%	Pass

In Table 5.1, the virtual point light count on CPU is 271 and 253 for GPU. The number difference is caused by randomness of the implemented algorithm. The columns “First Pass” and “Second Pass” contain the rendering time of first pass and second pass, and the total rendering time is in column “Total”. The “Light Ratio” column records how much percent of all virtual point lights are selected for the second pass by Lightcuts. The last column records the perceptual checking among the images, and the reference image is the CPU without Lightcuts, all the other three are checked by comparison with that one. As shown in Table 5.1, all images pass the perceptual checking. As shown in Table 5.1, no perceptual differences have been detected. Figure 5.1 shows the generated images.

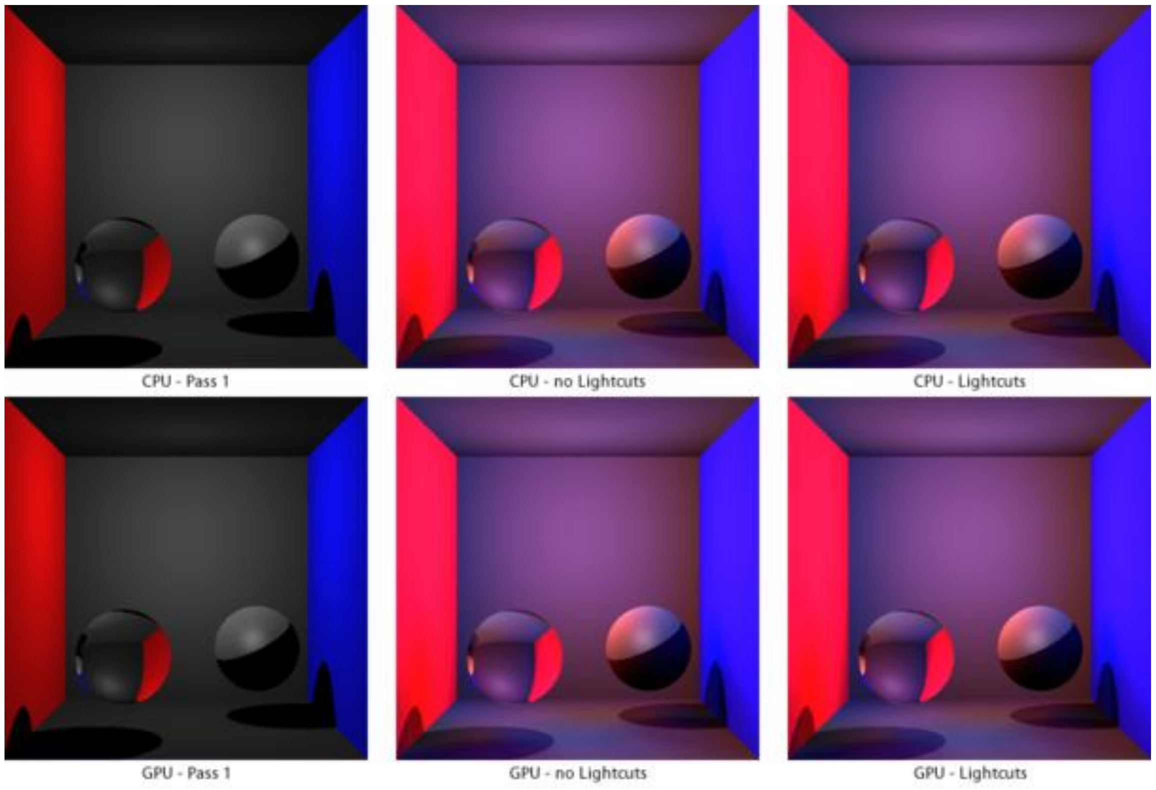


Figure 5.1 Simple Scene images

5.1.3.2. Venus Model Scene

This scene contains a more complicated model, Venus, that possesses 1418 triangles, together with the five squares as in the Cornell Box sides.

Four different scene configurations were involved: CPU without Lightcuts, CPU with Lightcuts, GPU without Lightcuts and GPU with Lightcuts. Each of the four configurations was tested three times and their average values were used for analysis. Table 5.2 shows the testing results (All data values are in seconds):

Table 5.2

Venus Scene Testing Results

	First Pass (In Seconds)	Second Pass (In Seconds)	Total (In Seconds)	Light Ratio (%)	P-Diff Check
CPU - no lightcuts	38.263	1383.921	1422.187	X	X
CPU - lightcuts	38.569	1545.119	1583.69	60.021%	Pass
GPU - no lightcuts	14.461	374.228	388.693	X	Pass
GPU - lightcuts	15.077	354.584	369.667	66.855%	Pass

In Table 5.2, the virtual point light count on CPU is 136 and 131 for GPU. The number difference is caused by randomness of the implemented algorithm. The columns “First Pass” and “Second Pass” contain the rendering time of first pass and second pass, and the total rendering time is in column “Total”. The “Light Ratio” column records how much percent of all virtual point lights are selected for the second pass by Lightcuts. The last column records the perceptual checking among the images, and the reference image is CPU without Lightcuts, all the other three are checked by comparison with that one. As shown in Table 5.2, all images pass the perceptual checking. As shown in Table 5.2, no perceptual differences have been detected. Figure 5.2 shows the generated images.

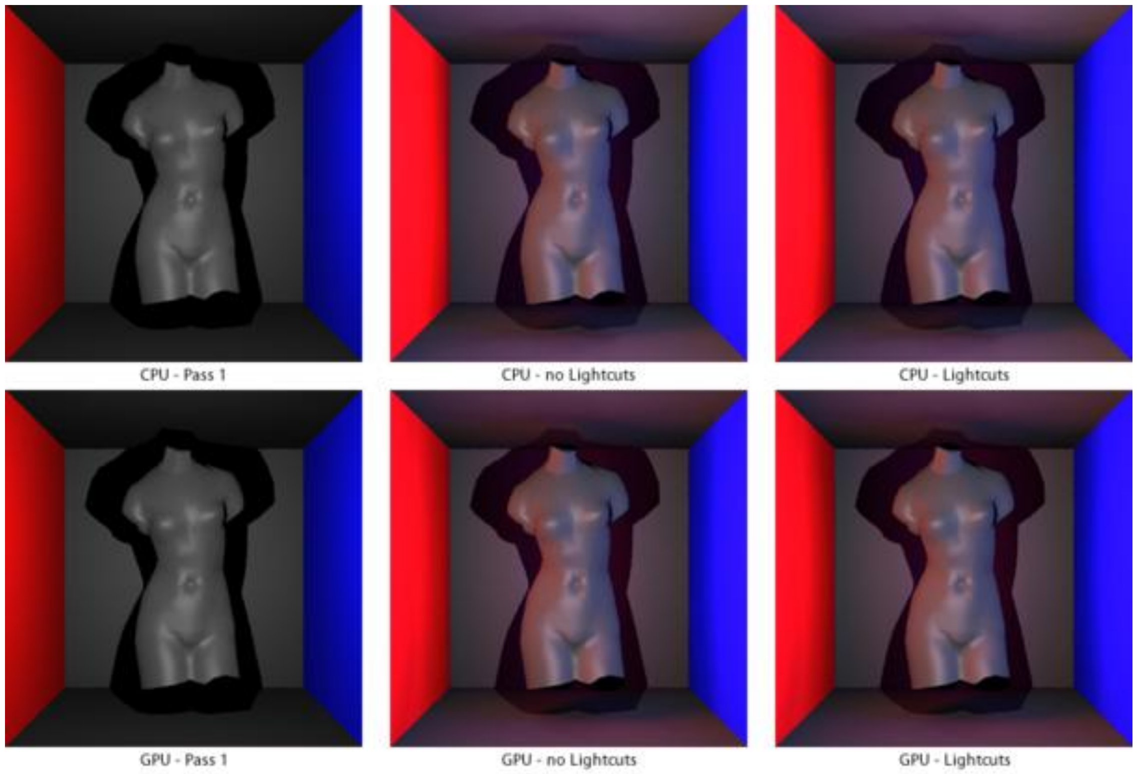


Figure 5.2 Venus Scene images

5.1.3.3. Galleon Model Scene

This scene contains an even more complex model Galleon, containing 4698 triangles, together with the five squares as the Cornell Box sides.

Four different scene configurations are involved: CPU without Lightcuts, CPU with Lightcuts, GPU without Lightcuts and GPU with Lightcuts. Each of the four configurations has been tested three times and their average values were used for analysis. Table 5.3 presents the testing results (All data values are in seconds).

Table 5.3

Galleon Scene Testing Results

	First Pass (In Seconds)	Second Pass (In Seconds)	Total (In Seconds)	Light Ratio (%)	P-Diff Check
CPU - no lightcuts	133.787	4265.908	4399.7	X	X
CPU - lightcuts	133.702	3519.038	3652.74	48.247%	Pass
GPU - no lightcuts	28.345	816.96	845.34	X	Pass
GPU - lightcuts	28.515	741.397	769.93	53.335%	Pass

In Table 5.3, the virtual point light count on CPU is 131 and 146 for GPU. The number difference was caused by randomness of the implemented algorithm. The columns “First Pass” and “Second Pass” contain the rendering time of the first pass and second pass, and the total rendering time is in column “Total”. The “Light Ratio” column records how much percent of all virtual point lights are selected for the second pass by Lightcuts. The last column records the perceptual checking among the images, and the reference image is the CPU without Lightcuts. The other three images are checked by comparison to it. As shown in Table 5.3, all images pass the perceptual checking. As shown in Table 5.3, no perceptual differences have been detected. Figure 5.3 shows the generated images.

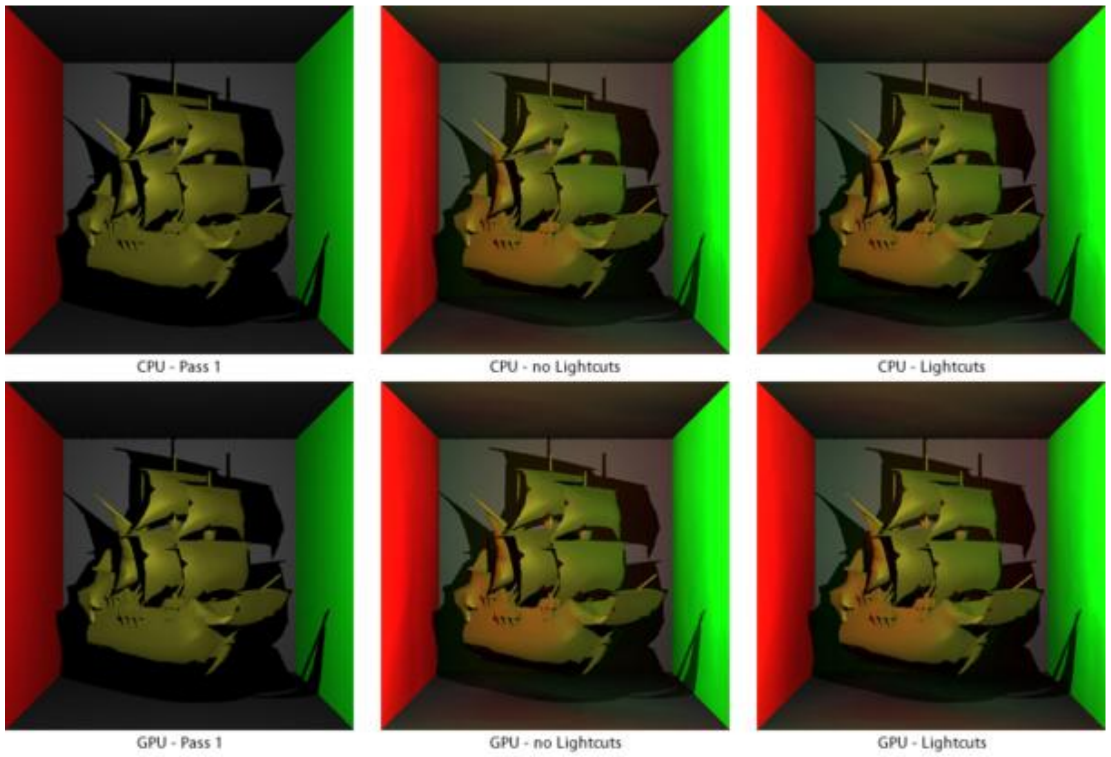


Figure 5.3 Galleon Scene images

5.2. Results Analysis

With experiment results stated in section 5.1, the performance acceleration percentage can be acquired easily, and also an intuitive representation using charts for data is able to be drawn for effective analysis. Three aspects would be emphasized in the follows: GPU acceleration, Lightcuts acceleration, and overall acceleration.

5.2.1. GPU Acceleration

From original testing data, the following compared acceleration percentage of GPU is shown in Table 5.4. Figure 5.4 shows this data as a chart.

Table 5.4

GPU Acceleration Analysis

	Simple(1)	Venus(1418)	Galleon(4698)
Pass 1	-88.262%	160.201%	369.681%
No Lightcuts - 2	81.109%	269.807%	425.165%
No Lightcuts - all	58.122%	265.889%	423.311%

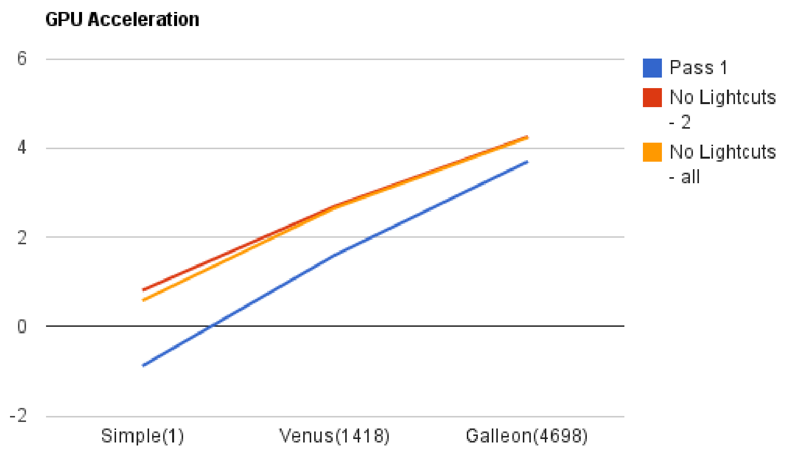


Figure 5.4 GPU Acceleration Chart

Please notice that to emphasize the GPU performance boost effect, Lightcuts has not been considered. And what is noticeable, if there are too many geometry primitives in the scene, the GPU driver on Windows will halt because of long kernel function execution.

5.2.2. Lightcuts Acceleration

Like the GPU testing data, Lightcuts performance increase effect data can be calculated too, as shown in Table 5.5. Figure 5.5 shows this data as a chart.

Table 5.5

Lighcuts Acceleration Analysis

Pass 2	Simple(1)	Venus(1418)	Galleon(4698)
CPU	-82.1%	-10.433%	22.378%
GPU	-33.67%	5.54%	10.553%

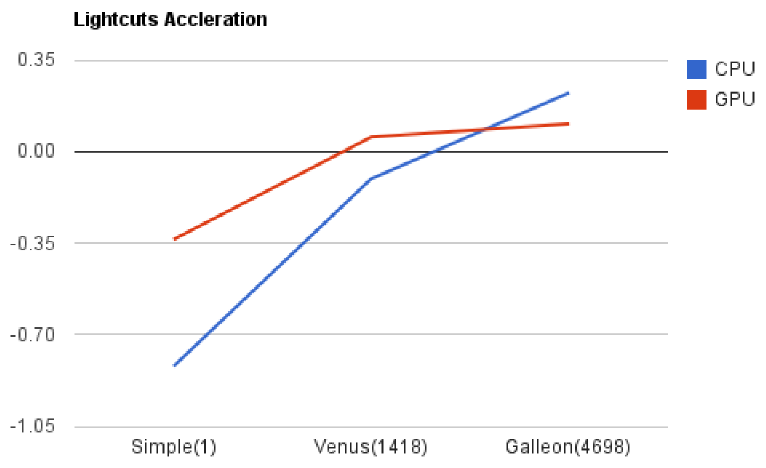


Figure 5.5 Lightcuts Acceleration Chart

To make it clear how much Lightcuts makes rendering faster, CPU and GPU configurations are individually considered. As with as the GPU situation, if there were too many virtual point lights, pixel computation would cause Windows GPU driver to halt.

5.2.3. Overall Acceleration

At last, a combined effect of both GPU and Lightcuts is provided in Table 5.6. Figure 5.6 shows the overall acceleration graphically.

Table 5.6

Overall Acceleration Testing Results

	Simple(1)	Venus(1418)	Galleon(4698)
Overall	9.941%	284.721%	476.524%

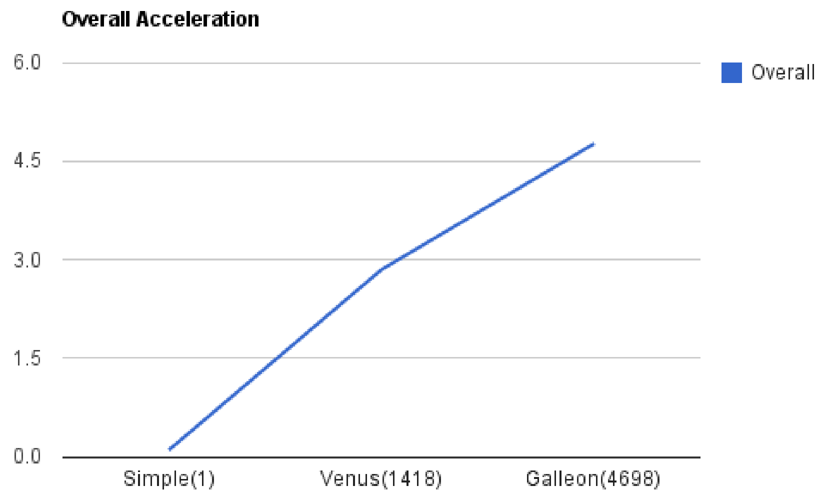


Figure 5.6 Overall Acceleration Chart

These overall data derive from the time cost of the CPU without Lightcuts and the GPU with Lightcuts.

5.3. Summary

From the testing results analysis in last section, it is not difficult to make some judgments on the performance acceleration effect of the GPU and Lightcuts:

- The GPU is able to offer satisfactory acceleration for scene rendering for relatively complex scenes, but not for a simple scene with only a few geometries, due to the memory manipulation cost between CPU and GPU.

- Lightcuts can only provide effective acceleration on complex scenes; for relatively simple scenes, it even makes rendering slower because of its Lightcuts tree traversal cost.

- With the GPU and Lightcuts combined, rendering speed-up ratio is almost linear with scene complexity (geometry primitives’ number).

CHAPTER 6. CONCLUSIONS AND FUTURE WORK

6.1. Conclusions

Encouraging results have been produced by research in this thesis, as demonstrated in Chapter 5. From these results, several conclusions about GPU acceleration on Ray-tracing, and Lightcuts algorithm acceleration can be stated.

- Ray-Tracing parallelism using the GPU can enhance rendering performance effectively for complex scenes.

  The assumption in Chapter 2 has partially been proved by the solution proposed in this thesis. Hardware parallelism handles the pixel independence essence of Ray-Tracing and gives satisfactory results for relatively complex scenes. However, the GPU acceleration will make rendering simple scenes even worse.

- Lightcuts can further improve the rendering performance significantly for complex scenes.

  According to testing results in Chapter 5, Lightcuts can cut unnecessary virtual point lights off from the second pass so that rendering is reduced significantly. However, as with the GPU, Lightcuts has to maintain its light tree at runtime, so this cost will make simple scene rendering even slower.

6.2. <u>Future Work</u>

Better performance cannot guarantee a perfect rendering engine. There are some other important aspects for a good rendering engine those are not emphasized or involved in this thesis. So there is still room for further improvements.

- A proper CUDA kernel design is expected.
  The current kernel function implementation suffers from Windows Driver Watchdog mechanism. So the kernel function could be further splitted, so its execution time on GPU could be shorter. Thus the GPU driver halting problem could be alleviated.

- More global illumination effects.
  Besides diffuse-diffuse illumination, there are many other more interesting global illumination effects, such as color bleeding, participating media etc. Such effects can make images more realistic and attracting. But they require more computation efforts of course.

- More shading material effects, as metal, plastic etc.

  In current research, only one shading model has been implemented: the Phong shading model, which is considered too simple to represent complex materials, such as metal, plastic, wood etc., which possess much more complex lighting interactions with objects.

LIST OF REFERENCES

LIST OF REFERENCES

Akenine-Moller, T., & Haines, E. (2002). *Real-time rendering*. Natick, MA: A. K. Peters Press.

Cohen, M., & Wallace, J. (1993). *Radiosity and realistic image synthesis* San Francisco, CA: Mogan Kaufmann Publishers, Inc.

Jenson, H. (1996). *Global illumination using photon maps.* Natick, MA: A. K. Peters Press

Keller, A. (1997). Instant Radiosity. *Proceedings of SIGGRAPH 97, Computer Graphics Proceedings, Annual Conference Series*, 49 – 56.

Kontkanen, J., & Laine, S. (2005) Ambient occlusion fields. *Proceedings of the ACM SIGGRAPH 2005 Symposium on Interactive 3D Graphics and Games, ACM Press,* 41 - 48.

Meyers, S. (2005). *Effective C++ 3rd Edition*. Indianapolis, IN: Addison-Wesley Professional Press.

Ryoo, S., Rodrigues, C., Baghsorkhi, S., Stone, S., Kirk, D., & Hwu, W. (2008). Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM Press, 73 – 82.

Segovia, B., Lehl, J.C., Mitanchey, R., & Peroche. (2006). Bidirectional instant radiosity. *Eurographics Symposium on Rendering*, 389 – 398.

Shirley, P. (2005). *Fundamentals of computer graphics*. Natick, MA: A. K. Peters Press.

Shreiner, D., Woo, M., Neider, J., & Davis, T. (2007). *OpenGL programming guide*. Indianapolis, IN: Addision-Wesley Professional Press.

Turner, W. (1980). An improved illumination model for shaded display. *Communications of the ACM 23(6)*, 343-349.

Walter, B., Arbree, A., Bala, K., & Greenberg, D. (2006). Multidimensional Lightcuts. *ACM Transactions on Graphics - Proceedings of ACM SIGGRAPH 2006, 25(3)*, 1081 – 1088.

Walter, B., Fernandez, S., Arbree, A., Bala, K., Donikian, M., & Greenberg, D. (2005). Lightcuts: A scalable approach to illumination. . *ACM Transactions on Graphics - Proceedings of ACM SIGGRAPH 2005*, *24(3)*, 1098 – 1107.

APPENDICES

Appendix A. Scene Description Files

The scenes are described in text format files, in which all information like geometry positions, sizes, scales and lighting parameters are indicated.

Three testing scenes' description files are included here:

Simple Scene:

```
# Cornell Box
# VPL(0.0001/0.7/GPU:0.08-CPU:0.075) - LC(1*10^-6)
S BK{0.0, 0.0, 0.0}|
# camera
C STRAT|PERS(10)|C{0, 0, 960}|V{0, 0, -1}|U{0, 1, 0}|
# lights
L OMNI|POS{0, 100, 0}|DIR{0, 0, -1}|ATTEN(1)|S{1, 1, 1}|D{1, 1, 1}|A{1, 1, 1}|
# Sphere
P SPH|FRKE{0, 1, 0.1, 0}|C{-80, -100, 70}R(60)|S{0.35, 0.35, 0.1}D{0.35, 0.35,
0.1}A{0.35, 0.35, 0.1}SH(20)|
P SPH|FRKE{0.0, 0.0, 0.7, 1}|C{100, -100, -40}R(60)|S{0.2, 0.2, 0.2}D{0.2, 0.2,
0.2}A{0.2, 0.2, 0.2}SH(20)|
# Walls
# Bottom
P SQU|FRKE{0.0, 0.0, 0.6, 1}|C{0, -200, 0}N{0, 1, 0}H{0, 0,
1}|W400:H400|S{0.14, 0.14, 0.14}D{0.14, 0.14, 0.14}A{0.0, 0.0, 0.0}SH(20)|
# Top
P SQU|FRKE{0.0, 0.0, 0.6, 1}|C{0,  200, 0}N{0,-1, 0}H{0, 0,
1}|W400:H400|S{0.14, 0.14, 0.14}D{0.14, 0.14, 0.14}A{0.0, 0.0, 0.0}SH(20)|
# Back
P SQU|FRKE{0.0, 0.0, 0.6, 1}|C{0, 0, -200}N{0, 0, 1}H{0, 1,
0}|W400:H400|S{0.14, 0.14, 0.14}D{0.14, 0.14, 0.14}A{0.0, 0.0, 0.0}SH(20)|
# Left
P SQU|FRKE{0.0, 0.0, 0.6, 1}|C{-200, 0, 0}N{1, 0, 0}H{0, 1, 0}|W400:H400|S{0.0,
0.0, 0.0}D{0.88, 0.05, 0.05}A{0.0, 0.0, 0.0}SH(20)|
# Right
P SQU|FRKE{0.0, 0.0, 0.6, 1}|C{200, 0, 0}N{-1, 0, 0}H{0, 1, 0}|W400:H400|S{0.0,
0.0, 0.0}D{0.05, 0.05, 0.95}A{0.0, 0.0, 0.0}SH(20)|
```

Venus Scene:

```
# Cornell Box
# VPL(0.00003/0.5/GPU:0.15-CPU:0.16) -  VPL\LC(5*10^-7)
S BK{0.0, 0.0, 0.0}|
# camera
C STRAT|PERS(10)|C{0, 0, 960}|V{0, 0, -1}|U{0, 1, 0}|
# lights
L OMNI|POS{0, 50, 250}|DIR{0, 0, -1}|ATTEN(1)|S{1, 1, 1}|D{1, 1, 1}|A{1, 1, 1}|
# venus : G:/RenderT/glm-data/venus.obj
O FRKE{0, 0.0, 0.9, 0.8}|PATH:venus.obj|SMTH(1)|TRAN{0, 0, 0}|SCAL{1.3, 1.3,
1.3}|ROT{0, 1, 0}:0|MAT(1)|S{0.2, 0.2, 0.2}D{0.2, 0.2, 0.2}A{0.2, 0.2, 0.2}SH(30)|
# Walls
# Bottom
P SQU|FRKE{0.0, 0.0, 0.6, 1}|C{0, -200, 0}N{0, 1, 0}H{0, 0,
1}|W400:H400|S{0.14, 0.14, 0.14}D{0.14, 0.14, 0.14}A{0.0, 0.0, 0.0}SH(20)|
# Top
P SQU|FRKE{0.0, 0.0, 0.6, 1}|C{0,  200, 0}N{0,-1, 0}H{0, 0,
1}|W400:H400|S{0.14, 0.14, 0.14}D{0.14, 0.14, 0.14}A{0.0, 0.0, 0.0}SH(20)|
# Back
P SQU|FRKE{0.0, 0.0, 0.6, 1}|C{0, 0, -200}N{0, 0, 1}H{0, 1,
0}|W400:H400|S{0.14, 0.14, 0.14}D{0.14, 0.14, 0.14}A{0.0, 0.0, 0.0}SH(20)|
# Left
P SQU|FRKE{0.0, 0.0, 0.6, 1}|C{-200, 0, 0}N{1, 0, 0}H{0, 1, 0}|W400:H400|S{0.0,
0.0, 0.0}D{0.95, 0.05, 0.05}A{0.0, 0.0, 0.0}SH(20)|
# Right
P SQU|FRKE{0.0, 0.0, 0.6, 1}|C{200, 0, 0}N{-1, 0, 0}H{0, 1, 0}|W400:H400|S{0.0,
0.0, 0.0}D{0.05, 0.05, 0.95}A{0.0, 0.0, 0.0}SH(20)|
```

Galleon Scene:

```
# Cornell Box
# VPL(0.000035/0.5/GPU:0.3 -  VPL\LC(10^-7)
S BK{0.0, 0.0, 0.0}|
# camera
C STRAT|PERS(10)|C{0, 0, 960}|V{0, 0, -1}|U{0, 1, 0}|
# lights
L OMNI|POS{0, 50, 350}|DIR{0, 0, -1}|ATTEN(1)|S{1, 1, 1}|D{1, 1, 1}|A{1, 1, 1}|
# venus
O FRKE{0.0, 0.0, 0.9, 0.9}|PATH:galleon.obj|SMTH(1)|TRAN{0, -80, 60}|SCAL{4,
4, 4}|ROT{0, 1, 0}:45|MAT(1)|S{0.1, 0.1, 0.05}D{0.1, 0.1, 0.05}A{0.1, 0.1,
0.05}SH(30)|
# Walls
# Bottom
```

```
P SQU|FRKE{0.0, 0.0, 0.6, 1}|C{0, -200, 0}N{0, 1, 0}H{0, 0,
1}|W400:H400|S{0.14, 0.14, 0.14}D{0.14, 0.14, 0.14}A{0.0, 0.0, 0.0}SH(20)|
# Top
P SQU|FRKE{0.0, 0.0, 0.6, 1}|C{0,  200, 0}N{0,-1, 0}H{0, 0,
1}|W400:H400|S{0.14, 0.14, 0.14}D{0.14, 0.14, 0.14}A{0.0, 0.0, 0.0}SH(20)|
# Back
P SQU|FRKE{0.0, 0.0, 0.6, 1}|C{0, 0, -200}N{0, 0, 1}H{0, 1,
0}|W400:H400|S{0.14, 0.14, 0.14}D{0.14, 0.14, 0.14}A{0.0, 0.0, 0.0}SH(20)|
# Left
P SQU|FRKE{0.0, 0.0, 0.6, 1}|C{-200, 0, 0}N{1, 0, 0}H{0, 1, 0}|W400:H400|S{0.0,
0.0, 0.0}D{0.95, 0.05, 0.05}A{0.0, 0.0, 0.0}SH(20)|
# Right
P SQU|FRKE{0.0, 0.0, 0.6, 1}|C{200, 0, 0}N{-1, 0, 0}H{0, 1, 0}|W400:H400|S{0.0,
0.0, 0.0}D{0.05, 0.95, 0.05}A{0.0, 0.0, 0.0}SH(20)|
```

Appendix B. Data Structures

In this section, the core data structures used in the system are listed in detail, which are supposed to be references of Chapter 4.

Ray Struct:

```
struct Ray
{
	long id;

	vect3d start_point;
	vect3d direction_vec;
	vect3d color;

	float fDeltaX, fDeltaY;
	bool bIsInObj;

	vect3d _hitPoint;
	vect3d _hitNorm;
};
```

Parent Light class:

```
class Light
{
public:
	float _fAttenuate;
	vect3d _ambientColor;
	vect3d _diffuseColor;
	vect3d _specularColor;
};
```

Sub-class of Light for directional point light:

```
class DirPointLight : public Light
{
public:
	vect3d _pos;
	vect3d _dir;
};
```

Sub-class of Light for directional light:

```
class DirLight : public Light
{
public:
        vect3d _dir;
};
```

Struct for lights on GPU:

```
struct LightGpu
{
        LightType eType;     // Light type

        // common
        float _fAttenuate;

        vect3d_gpu _ambientColor;
        vect3d_gpu _diffuseColor;
        vect3d_gpu _specularColor;

        //      DirPoint
        vect3d_gpu _dirp_pos;
        vect3d_gpu _dirp_dir;

        //      Dir
        vect3d_gpu _dir_dir;
};
```

Struct for object material:

```
struct material
{
        vect3d specColor;
        vect3d diffColor;
        vect3d ambiColor;
        float fShininess;
};
```

Parent class for all objects:

```
class Object
{
public:
        unsigned _id;
        material _mat;

protected:
        float _fReflectionRatio;
        float _fRefractionRatio;
        float _fRefractionK;
        float _fEmitRatio;
};
```

Sub-class of Object for triangles:

```
class Triangle : public Object
{
public:
        vect3d_vertices[3];
        vect3d_normal;
        vect3d _vnormal[3];

        bool _bSmooth;
        bool _bHasVNorm;
};
```

Sub-class of Object for sphere:

```
class Sphere : public Object
{
public:
        float _fRad;
        vect3d _ctr;
};
```

Sub-class of Object for square:

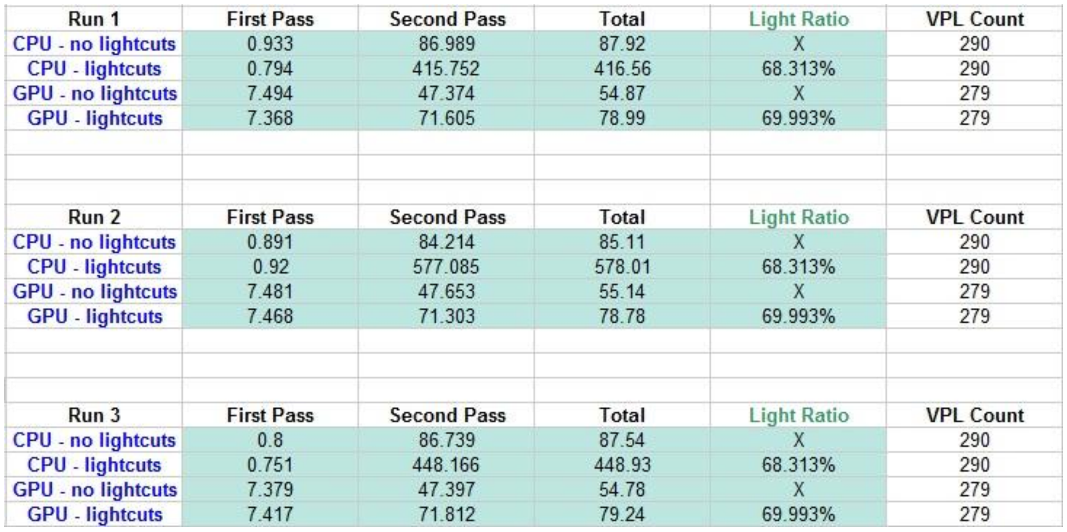
```
class Square : public Object
{
public:
        vect3d _vNormal;    //      Directions

        //      Positions
        vect3d _vCenter;
        float _nWidth;
        float _nHeight;

        //      For Calc.
        vect3d _v2HeightVec;
        vect3d _v2WidthVec;
};
```

GPU struct and the enum for objects:

```
        enum GpuObjType {TRI_GPU, SQU_GPU, SPH_GPU,
NONE_GPU};

        struct PrimGpuObj
        {
                //      common
                float _fReflectionRatio;
                float _fRefractionRatio;
                float _fRefractionK;
                float _fEmitRatio;
                material_gpu _mat;

                GpuObjType  eType;          // for GPU use
                int nId;

                //      Triangle
                vect3d_gpu   _vertices[3];
                vect3d_gpu   _normal;
                vect3d_gpu   _vnormal[3];
                bool _bSmooth;
                bool _bHasVNorm;

                //      Sphere
                float _fRad;
                vect3d_gpu _ctr;

                //      Square
                vect3d_gpu _vNormal;
                vect3d_gpu _vCenter;
                float _nWidth;
                float _nHeight;
                vect3d_gpu _v2HeightVec;
                vect3d_gpu _v2WidthVec;
        };
```

Appendix C. Original Testing Data

In the testing analysis part, the average values are considered. Those values derive from original testing data from three times running for each scene:

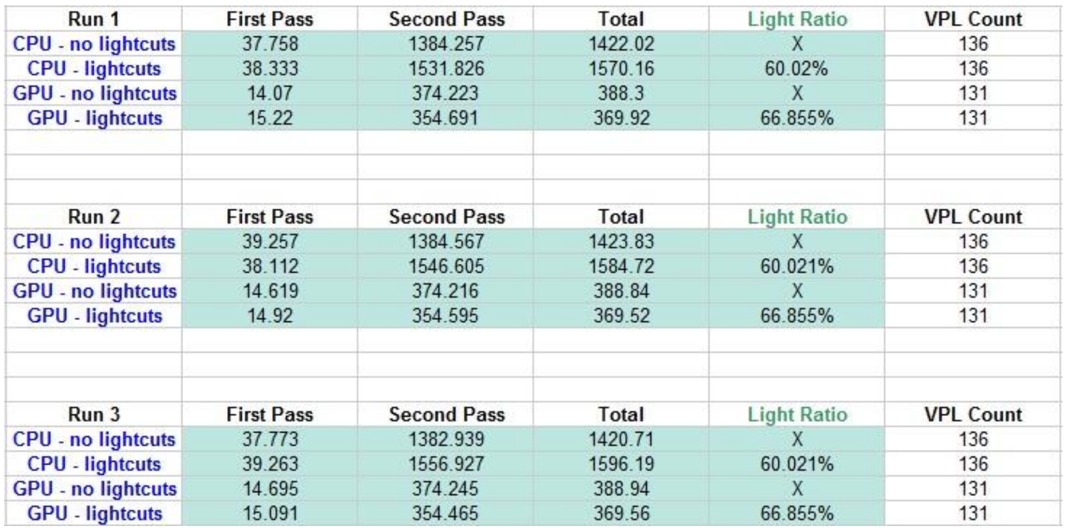


Run 1	First Pass	Second Pass	Total	Light Ratio	VPL Count
CPU - no lightcuts	0.933	86.989	87.92	X	290
CPU - lightcuts	0.794	415.752	416.56	68.313%	290
GPU - no lightcuts	7.494	47.374	54.87	X	279
GPU - lightcuts	7.368	71.605	78.99	69.993%	279

Run 2	First Pass	Second Pass	Total	Light Ratio	VPL Count
CPU - no lightcuts	0.891	84.214	85.11	X	290
CPU - lightcuts	0.92	577.085	578.01	68.313%	290
GPU - no lightcuts	7.481	47.653	55.14	X	279
GPU - lightcuts	7.468	71.303	78.78	69.993%	279

Run 3	First Pass	Second Pass	Total	Light Ratio	VPL Count
CPU - no lightcuts	0.8	86.739	87.54	X	290
CPU - lightcuts	0.751	448.166	448.93	68.313%	290
GPU - no lightcuts	7.379	47.397	54.78	X	279
GPU - lightcuts	7.417	71.812	79.24	69.993%	279

Figure C.1 Original Testing Results for Simple Scene

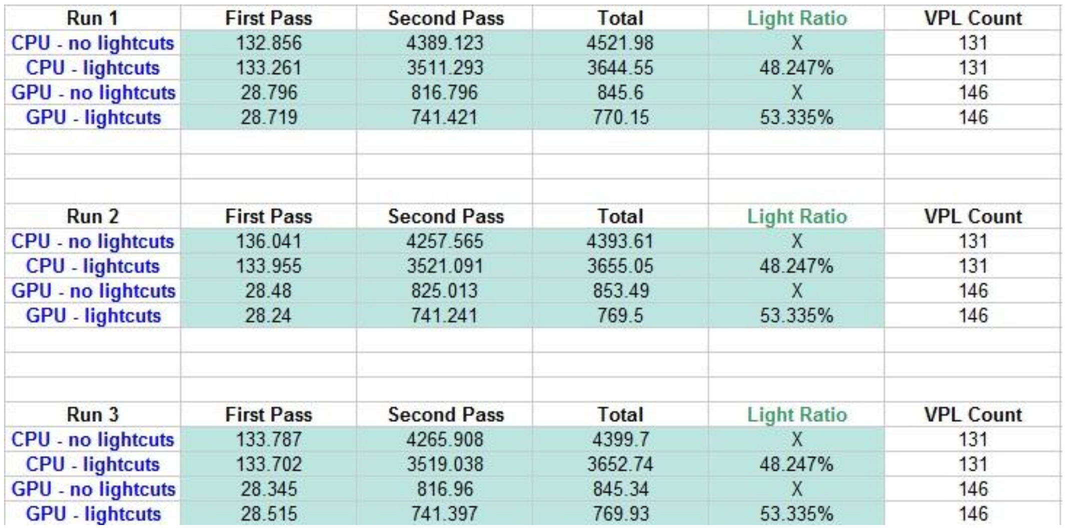


Run 1	First Pass	Second Pass	Total	Light Ratio	VPL Count
CPU - no lightcuts	37.758	1384.257	1422.02	X	136
CPU - lightcuts	38.333	1531.826	1570.16	60.02%	136
GPU - no lightcuts	14.07	374.223	388.3	X	131
GPU - lightcuts	15.22	354.691	369.92	66.855%	131

Run 2	First Pass	Second Pass	Total	Light Ratio	VPL Count
CPU - no lightcuts	39.257	1384.567	1423.83	X	136
CPU - lightcuts	38.112	1546.605	1584.72	60.021%	136
GPU - no lightcuts	14.619	374.216	388.84	X	131
GPU - lightcuts	14.92	354.595	369.52	66.855%	131

Run 3	First Pass	Second Pass	Total	Light Ratio	VPL Count
CPU - no lightcuts	37.773	1382.939	1420.71	X	136
CPU - lightcuts	39.263	1556.927	1596.19	60.021%	136
GPU - no lightcuts	14.695	374.245	388.94	X	131
GPU - lightcuts	15.091	354.465	369.56	66.855%	131

Figure C.2 Original Testing Results for Venus Scene



Run 1	First Pass	Second Pass	Total	Light Ratio	VPL Count
CPU - no lightcuts	132.856	4389.123	4521.98	X	131
CPU - lightcuts	133.261	3511.293	3644.55	48.247%	131
GPU - no lightcuts	28.796	816.796	845.6	X	146
GPU - lightcuts	28.719	741.421	770.15	53.335%	146

Run 2	First Pass	Second Pass	Total	Light Ratio	VPL Count
CPU - no lightcuts	136.041	4257.565	4393.61	X	131
CPU - lightcuts	133.955	3521.091	3655.05	48.247%	131
GPU - no lightcuts	28.48	825.013	853.49	X	146
GPU - lightcuts	28.24	741.241	769.5	53.335%	146

Run 3	First Pass	Second Pass	Total	Light Ratio	VPL Count
CPU - no lightcuts	133.787	4265.908	4399.7	X	131
CPU - lightcuts	133.702	3519.038	3652.74	48.247%	131
GPU - no lightcuts	28.345	816.96	845.34	X	146
GPU - lightcuts	28.515	741.397	769.93	53.335%	146

Figure C.3 Original Testing Results for Galleon Scene