University of Innsbruck

Institute for Computer Science
Distributed and Parallel Systems

Master Thesis in Computer Science

Scalable Lighting for Global Illumination

Author:
Michael Walch
Mat.-No.: 0715223

Supervisor:
Dr. Biagio Cosenza

November 28, 2016

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder inhaltlich den angegebenen Quellen entnommen wurden, sind als solche kenntlich gemacht.
Die vorliegende Arbeit wurde bisher in gleicher oder ähnlicher Form noch nicht als Magister-/Master-/Diplomarbeit/Dissertation eingereicht.

Datum	Unterschrift

Abstract

In computer graphics, global illumination models the physical processes of light interaction with objects in a virtual environment to generate photorealistic images. Global illumination takes into account all the effects the objects have on each other as the light travels through the scene. Numerous global illumination algorithms use ray tracing to compute the distribution of light. Usually, the most costly operation in these algorithms is determining whether a ray has intersected an object or not, that is to compute the visibility between points. Reducing or accelerating these visibility tests are some of the main challenges. The majority of tests in this thesis are between a point in the scene and one or more light sources. Lightcuts is a global illumination algorithm that approximates many lights by grouping them into clusters. Therefore the rendering cost is sublinear with the number of lights, which reduces the intersection tests.

This thesis exploits the highly parallelizable structure of ray tracing algorithms by using the OpenCL programming language to accelerate visibility tests, so that the performance of lightcuts is increased. The result is a scalable, data parallel implementation of the lightcuts algorithm designed for massively parallel hardware. The implementation is for heterogeneous systems that is able to run on multiple types of hardware, such as CPUs, GPUs and accelerator cards. The rendering time for our results were reduced from minutes to seconds depending on the resolution of the image. This thesis also implemented a data parallel version of reconstruction cuts, which is an extension to the lightcuts algorithm that exploits spatial coherency, this further reduces visibility tests. Data sharing in the reconstruction cuts algorithm required special thread synchronization because of the limitations in OpenCL. From experimental evaluation, the reconstruction cuts algorithm can achieve speeds that are almost three times faster than the standard lightcuts algorithm. There is perceptually no noticeable difference in the quality of images when rendered with the approximated solution of lightcuts and comparing it to the exact solution where every light source is accounted for, even when increasing the perceptual visibility threshold. However, reconstruction cuts does exhibit a perceptual difference.

Kurzfassung

In der Computergrafik modelliert die globale Beleuchtung die physischen Prozesse der Wechselwirkung des Lichts mit Objekten in einer virtuellen Umgebung, um photorealistische Bilder zu erzeugen. Die globale Beleuchtung berücksichtigt alle Effekte, die Objekte aufeinander haben, während sich das Licht in der Szene ausbreitet. Zahlreiche globale Beleuchtungsalgorithmen verwenden Ray Tracing um die Verteilung des Lichts zu berechnen. Normalerweise ist der teuerste Vorgang bei solchen Algorithmen festzustellen, ob sich ein Lichtstrahl mit einem Objekt schneidet oder nicht, das ist die Sichtbarkeit zwischen zwei Punkten zu berechnen. Die Reduzierung oder Beschleunigung solcher Sichtbarkeitstests ist eine der Haupt Herausforderungen. Die Mehrheit der Tests in dieser Arbeit sind zwischen einem Punkt in der Szene und einer oder mehreren Lichtquellen. Lightcuts ist ein globaler Beleuchtungsalgorithmus der viele Lichter durch ihre Zusammenfassung in Clustern approximiert. Die rendering Kosten sind deshalb sublinear mit der Anzahl der Lichter, somit werden die Schittpunkttests verringert.

Die hohe parallelisierbare Struktur des Ray-Tracing-Algorithmus wird in dieser Arbeit durch die Verwendung der OpenCL Programmiersprache ausgenutzt, um die Sichtbarkeitstests zu beschleunigen und somit die Leistung von Lightcuts zu erhöhen. Das Ergebnis ist eine skalierbare, datenparallele Implementierung des Lightcuts-Algorithmus, dieser ist entwickelt worden für massive parallele Hardware. Die Implementierung ist für heterogene Systeme, die es ermöglicht auf mehreren Typen von Hardware zu laufen, zum Beispiel auf CPUs, GPUs und Beschleunigerkarten. Bei unseren Ergebnissen hat sich die rendering Zeit von Minuten auf Sekunden reduziert, abhängig von der Auflösung des Bildes. Diese Arbeit implementierte auch eine datenparallele Version des Reconstruction-Cuts-Algorithmus, welche eine Erweiterung des Lightcuts-Algorithmus ist, die die Räumliche Kohärenz ausnutzt. Dadurch werden Sichtbarkeitstests weiter reduziert. Gemeinsam genutzte Daten im Reconstruction-Cuts-Algorithmus benötigen eine spezielle Thread Synchronisation wegen der Begrenzungen in der OpenCL Programmiersprache. Durch experimentelle Auswertungen kann Reconstruction Cuts die Geschwindigkeit gegenüber den Standard Lightcuts-Algorithmus fast verdreifachen. Es gibt keine wahrnehmbaren Unterschiede in der Qualität der Bilder, die mit der Näherungslösung Lightcuts gerendert werden, im Vergleich zu der exakten Lösung, bei der jede Lichtquelle berücksichtigt wird, sogar durch Erhöhung der Wahrnehmungsschwelle. Wahrnehmungsunterschiede sind jedoch bei Reconstruction Cuts ersichtlich.

Contents

List of Tables

List of Figures

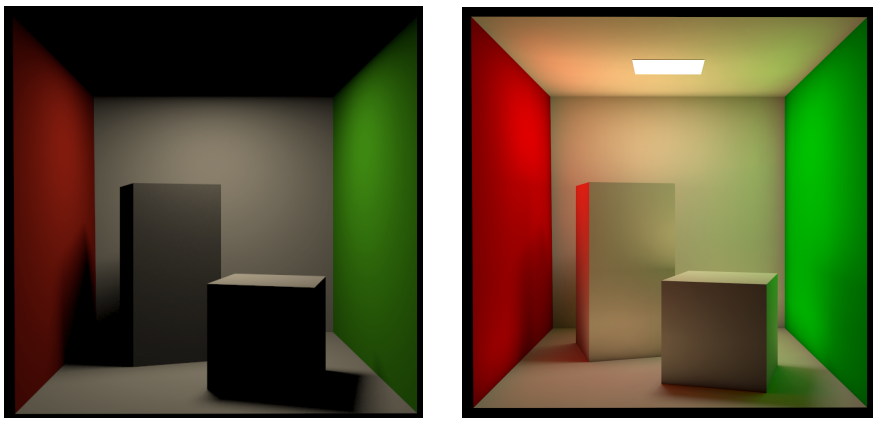
List of Algorithms

Chapter 1

Introduction

The main objective of physically based rendering is to generate photorealistic images by simulating the light distribution in a three-dimensional scene. The modelling of physical processes that describe the behaviour of light as it interacts with different types of surfaces is not a new idea. The recent surge in computational power has made it feasible to simulate light transport and many types of rendering algorithms exist to approximate the distribution of light.

Global illumination refers to the group of algorithms that solve the light transport problem to compute realistic lighting effects. The term global, describes the light received by an object, not just from a light source (direct illumination) but more importantly the light received from all objects in the scene that reflect, scatter or transmit light (indirect illumination). Indirect illumination is the key ingredient to achieve photo realism. Notice the color bleeding on the boxes in Figure 1.1b.



(a) Direct light (b) Direct + indirect light

Figure 1.1: Global illumination

These algorithms can be split into two categories known as Monte Carlo ray tracing or finite element methods [Vea98]. Finite element methods adapt radiative heat transfer

equations to a computer graphics setting. The more common technique known as Monte Carlo ray tracing finds a path by means of a stochastic process that can be traced through the scene.

The rendering equation was introduced by Kajiya [Kaj86] in 1986, who realized that the distribution of light could be formulated as a recursive integral equation and evaluated by sampling paths, this lead to the development of Monte Carlo ray tracing. The rendering equation unifies the approach of solving the light transport problem, since every method has to numerically evaluate the integral, which is impossible to solve analytically except in the simplest of scenarios.

The lightcuts algorithm sublinearly reduces the cost of approximating the rendering equation given many light sources [WFA+05]. The aim of this thesis is to accelerate the lightcuts method through a parallel approach with OpenCL and to evaluate performance. OpenCL is a framework for heterogeneous computing that allows programs to be parallelized and executed on multiple types of hardware, such as CPUs, GPUs and accelerator cards. The light transport problem can be solved by tracing rays through pixels into a three-dimensional scene. Determining whether a ray has intersected an object is usually the most expensive operation when trying to evaluate the rendering equation. Ray tracing is an ideal candidate for parallelization because each pixel can be processed independently. This can be more accurately classified as data-based parallelism because each pixel can be computed simultaneously as a single piece of data to a larger problem, i.e. the rendered image. Modern GPUs have many cores so it is a good match for these type of algorithms because multiple pixels can be evaluated in parallel with each pixel processed on one of the cores. Lightcuts reduces ray intersection tests by grouping lights into clusters and approximating their illumination. Reconstruction cuts [WFA+05] is an extension to lightcuts that approximates nearby pixels so that even fewer rays have to be traced. In this thesis we improve performance of lightcuts and reconstruction cuts by applying a data parallel approach to these algorithms whose purpose is for reducing ray tracing costs when rendering a scene with many lights.

1.1 Thesis organization

In the rest of Chapter 1, a comparison with previous work is discussed and the contributions of this thesis. In Chapter 2, we give an introduction to the theory of light transport and the history of the development of ray tracing, which preceded Monte Carlo ray tracing. Chapter 2 also describes the rendering equation that is the foundation for the following chapter. In Chapter 3, we give a detailed explanation of the lightcuts and reconstruction cuts algorithm that are presented in the lightcuts paper. Chapter 4 discusses the different parallel paradigms and why parallel programming is so important to increasing performance, especially in todays computing environment. An introduction is given to the OpenCL framework and an example of an OpenCL data parallel program, where computations can be offloaded but not limited to the GPU. Chapter 5 presents the parallel implementation of the lightcuts and reconstruction cuts algorithm. This chapter documents the software design decisions, as well as the state machine that was used to

synchronize the threads for reconstruction cuts. Finally, Chapter 6 presents the results of a set of tests, each showing the rendered scenes of our experiments. Accompanying these results is the rendering time and other data that is organized similarly to the original paper, so that comparisons can be drawn. The rest of the chapter finishes with a discussion of the results. Conclusions are drawn in Chapter 7 and possible directions for future work.

1.2 Related work

Similar work to this thesis have already implemented data parallel versions of lightcuts with Nvidia Compute Unified Device Architecture (CUDA). The major drawback of these implementations is that they are limited to running on Nvidia GPUs only. The lightcuts version from Zhang [Zha11] could be considered the CUDA counterpart to this thesis, so instead of implementing a data parallel lightcuts version with OpenCL he chose the CUDA framework. Taubmann [Tau10] implemented a solution with the Nvidia Optix [PBD+10] ray tracing engine, which internally uses the CUDA computing architecture and is optimized for performance on Nvidia GPUs. Arbesser and Mühlbacher [MA09] developed a CUDA version that does not rely entirely on ray tracing methods, such as [Zha11] and [Tau10] but uses Imperfect Shadow Maps (ISM) [RGK+08] to test for visibility of light sources. The big advantage with ISM is the reduction of ray intersection and visibility tests by utilising the graphics pipeline. None of these works deal with the second part of the lightcuts paper, where *reconstruction cuts* exploits the spatial coherence of the scene to reduce costs.

1.3 Contributions

This thesis introduces a data parallel lightcuts version based on OpenCL that allows heterogeneous computing on several types of hardware, such as CPUs, GPUs and accelerator cards and all at the same time. For example, the GPU can render one half of the image while the CPU renders the other half. The code to help bootstrap the process of loading the necessary data into the renderer is based on LuxRays, this work can be considered a stripped down version of this project with the objective of keeping the code as minimal as possible but robust enough to provide a framework to test and implement other solutions apart from the work in this thesis that focus on solving the rendering equation. The light weight code base has the added benefit of requiring less time to understand the complete rendering process making the system more manageable.

The other significant contribution is a data parallel version of the lightcuts extension known as reconstruction cuts, that approximates pixels in close proximity by determining, for example, if they reside on the same surface. This method can simply be summed up as a way to reduce tracing rays. The reconstruction cuts algorithm required a special synchronization strategy to render the image because we are not afforded the flexibility in programming capabilities for computations with the type of hardware accelerated applications in mind.

Chapter 2

Global Illumination

Before we can introduce the lightcuts algorithm we need to establish the theoretical framework for the light transport equation. The general concepts to simulate the light distribution of photorealistic renderers based on the ray tracing algorithm will be roughly outlined in the following section. The theory in this chapter is applied to our stochastic path tracer in Section 2.7.3 and in Section 3.1 we link the theory to the lightcuts algorithm.

Ray casting and recursive ray tracing will first be discussed, since these are the forerunners to modern day Monte Carlo ray tracers. Subsequent sections will show the physical quantities that makeup the rendering equation and then finally the rendering equation itself will be introduced.

2.1 Ray tracing algorithm

The algorithm traces rays in a scene that is a model of our physical world. The scene contains a description of the geometrical objects, light sources and the different materials attached to the objects along with their colors. The materials determine how the light propagates in the scene, for example, matte surfaces reflect light in all directions equally whereas glossy surfaces tend to reflect light in a certain direction (see Section 2.5). Rays are first traced from a virtual camera into the scene and those rays mimic the light by bouncing around the objects. The pinhole camera is one of the simplest devices for taking pictures and is often the starting point of simulating a camera (see Figure 2.1).

The pinhole camera is slightly adapted by placing the pinhole or eye in front of the image film for convenience sake (see Figure 2.2). The eye is the origin and the near plane serves as another endpoint for the direction of the ray. The viewing frustum is a pyramid with a truncated top and the objects that lie within the volume defined by the near and far plane are rendered to the image plane.

More sophisticated camera models with lenses can be developed on top of this abstraction. Just like any camera, the important part is to point the camera to the portion of a scene you want to record. Only the light that enters the camera sensor will make an image.

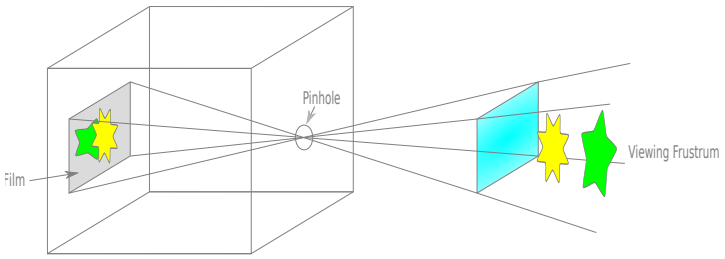


Figure 2.1: Pinhole camera

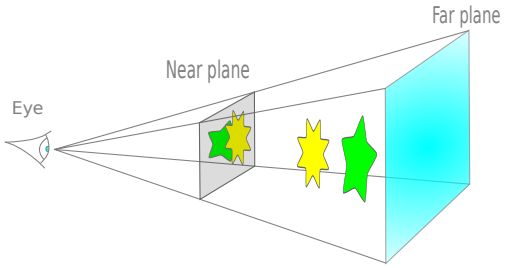


Figure 2.2: Viewing frustum

2.1.1 Ray intersection

When generating a ray, a point on the image plane is selected and the position of the camera relative to the scene serves as the origin. This is expressed as a parametric equation in the form of:

$$r(t) = o + t\mathbf{d}. \tag{2.1}$$

The unit vector $\mathbf{d}$ is the direction vector, and t is a scalar value that should be large enough to hit any object in the scene, if indeed the ray does hit an object.

The sphere will be shown as an example, since it is the simplest way of computing ray intersections. The implicit equation of a sphere is represented as a function $F(x, y, z) = 0$, where x, y, z is a point on the surface. The equation of the sphere with radius r centred at $(0, 0, 0)$ is fully described as

$$x^2 + y^2 + z^2 - r^2 = 0. \tag{2.2}$$

Substituting the parametric ray equation into the implicit equation $F(r(t)) = 0$ gives us

$$(o + t\mathbf{d})_x^2 + (o + t\mathbf{d})_y^2 + (o + t\mathbf{d})_z^2 - r^2 = 0. \tag{2.3}$$

The above equation is a quadratic equation in t so the possible solutions are:

$F(r(t)) = 0$, ray intersects sphere exactly on the surface
$F(r(t)) < 0$, ray intersects sphere
$F(r(t)) > 0$, ray misses sphere

Since triangles are usually the preferred primitive in computer graphics, the maths behind the ray intersection is more involved but follows the same principles shown above. Primitives are the fundamental building block of geometry and some of the possible primitives to build objects are listed here. 2D primitives such as triangles model the geometry of surfaces. Other 2D primitives are quads or quadrics, which can be expressed as quadratic polynomials and higher-order surfaces, such as subdivision surfaces which are refined into more polygons to form a smooth surface. Volume primitives for three-dimensional space are a common application in the medical field. For example, the magnetic resonance imaging (MRI) scanner produces a 3D data set from a group of 2D image slices.

Knowing whether a ray intersects the objects in the scene is the most expensive operation in ray tracing. Accelerating ray intersection code is an active area of research and special data structures exist to quickly find the object in the scene which a ray intersects. If all we had were the the ray intersection equations, then every object in the scene would need to be tested even after we hit an object, since we don't know which object is closer to the camera until every single object has been processed. Acceleration structures allow many objects to be discarded during ray intersection tests. The bounding volume hierarchy (BVH) is one such acceleration structure that will now be briefly described. The BVH divides the scene geometry into primitives and then organizes and sorts the primitives that are spatially close to each other into the leaves of a tree. The nodes and the leaves contain bounding boxes that are wrapped around the primitives to quickly determine if a ray misses all the primitives in the box. The root node then contains a bounding box of the whole scene. This thesis stores triangles as the primitive in the BVH structure and for efficient ray-triangle intersection the Möller-Trumbore [MT97] algorithm is used.

2.1.2 Recursive ray tracing

Ray casting [App68] was the first ray tracing technique, which traced rays only from the camera into the scene known as primary rays. Shadow rays (see Figure 2.3) can make the decision of whether an object is occluded from a light source, but ray casting assumes that objects facing the light sources are exposed to the light and then shaded with the specified colors, lights and materials. In essence, ray casting determines which objects are visible through the pixel. Algorithm 1 depicts the pseudo code for ray casting and shows historically why ray tracing was slow, for every pixel we have to loop over every object.

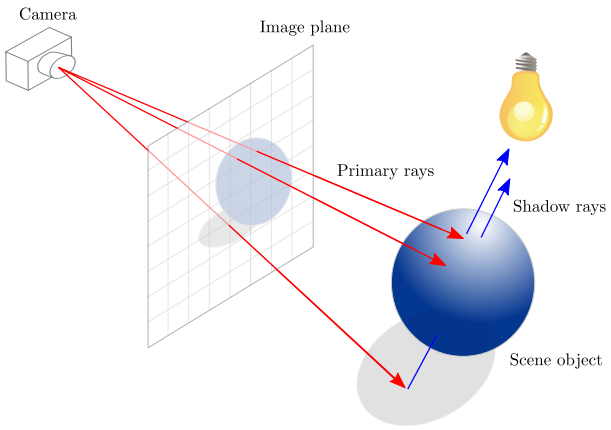


Figure 2.3: Ray tracing

Algorithm 1 Ray casting

```
for each pixel do
    for each object do
        if ray through pixel intersects object and is closest object then
            compute color of pixel
        end if
    end for
end for
```

Three open problems in computer graphics were not solved or done satisfactorily until the landmark paper by Turner Whitted was published in 1980 [Whi80]. Prior to Whitted nobody knew how to properly simulate *shadows*, *reflections* to imitate mirror-like surfaces or *refractions* through objects. Whitted showed how to solve these problems with a recursive ray tracer, which is the precursor to modern Monte Carlo ray tracing. Instead of only tracing rays through a pixel to determine visibility (ray casting), additional rays are used for direct, specular and refractive illumination effects. Algorithm 2 illustrates the mechanics behind the Turner Whitted ray tracer. On line 1 each pixel is processed by generating a ray that runs through the pixel into the scene. The function *Trace* is called along with the ray that is being traced. The argument *depth* to the function *Trace* makes sure the recursive algorithm terminates. In lines 6 and 7 we return, if the maximum depth has been exceeded and use the background color, since no objects have been hit. If the current ray we are tracing does not intersect with any object, we likewise return the background color shown in lines 9 and 10. Lines 12 to 18 calculate the color based on the *Phong reflection model* [Pho75]. Line 12 adds the ambient term and emission. The ambient color is supposed to ”approximate” indirect diffuse lighting.

Algorithm 2 Recursive ray tracing

```
 1: for each pixel do
 2:     r = trace ray from eye through pixel
 3:     shade pixel = TRACE(r, 0)
 4: end for
 5: function TRACE(ray, depth)
 6:     if depth > maxDepth then                          ▷ terminate recursive algorithm
 7:         return background color
 8:     end if
 9:     if ray does not intersect with any object then    ▷ found no intersection
10:         return background color
11:     end if
12:     colour = ambient intensity + own emission
13:     for each light source do                          ▷ direct illumination
14:         if ray from object hits light source then
15:             color += specular intensity
16:             color += diffuse intensity
17:         end if
18:     end for
19:     if object is reflective then
20:         r' = reflect ray
21:         color += TRACE(r', d + 1)
22:     end if
23:     if object is refractive then
24:         r'' = refract ray
25:         color += TRACE(r'', d + 1)
26:     end if
27:     return color
28: end function
```

If the object is not a light source then the emission term is ignored. Line 13 loops over all the direct light sources in the scene and line 14 determines if there is something blocking the light source (shadow rays). If this is not the case then the specular and diffuse terms are added to the color of the object. The values in lines 12, 15 and 16 are read from the model file that contains the necessary information about the objects in the scene. Lines 19 to 25 recursively trace rays to give us the mirror and refractive effects. The direction the rays are propagated depend on the type of surface (see lines 20 and 24). Although it is possible to render all types of effects besides mirrors and glass, the problem is that shooting rays into random directions will lead to a slow convergence of the algorithm and then a decision needs to be made when to terminate recursion.

2.2 Radiometric quantities

In this and the following sections, we introduce the quantities and equations that are necessary for light transport simulation. The rendering equation (see Section 2.4) describes the light transport in a scene, thereby making it possible to simulate effects to render photorealistic images.

Measuring the physical electromagnetic radiation of light is known as radiometry. We

now discuss the most important quantities that represent the physical energy of light or measurements of brightness (see [Vea98]) for further details).

Radiant power

Radiant power, also known as flux is defined as energy per unit time and is measured in Watts (Watts = Joules/sec).

$$\Phi = \frac{\mathrm{d}Q}{\mathrm{d}t} \tag{2.4}$$

This quantity represents the total energy of photons emitted or absorbed on a surface per unit time.

Irradiance

Irradiance is the radiant power incident on a surface per unit surface area and is measured in watts/m^2.

$$E = \frac{\mathrm{d}\Phi}{\mathrm{d}A} \tag{2.5}$$

Exitant radiance or radiosity is a similar quantity but expresses the exitant radiant power per unit surface area.

Solid angle

Before we can understand the last radiometric quantity, *radiance*, we need to introduce the solid angle. The solid angle can be defined as the area covering the unit sphere and is measured in *steradians* (sr). If the radius $r = 1$ and the area covered by the solid angle is the entire hemisphere, then the area is equal to 2π or 6.283 sr. In Figure 2.4 we show that any object can be projected onto the surface of the hemisphere and the solid angle in this case is the area of S.

$$\Omega = \frac{A}{r^2} \tag{2.6}$$

To measure arbitrary surfaces, the solid angle is expressed as a surface integral and the spherical coordinate angles θ and ϕ represent any point in three-dimensional space where $r = 1$ (see Equation 2.7).

$$\Omega = \int\int_S \sin\theta \mathrm{d}\theta \mathrm{d}\phi \tag{2.7}$$

$$\mathrm{d}\Omega = \sin\theta \mathrm{d}\theta \mathrm{d}\phi \tag{2.8}$$

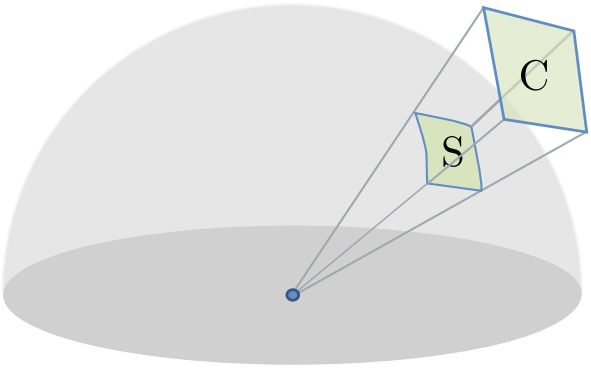


Figure 2.4: The solid angle S is the surface area that is projected onto the hemisphere from object C.

Radiance

This is the most important quantity, since every ray through a pixel tries to measure the radiance. Radiance is the flux per unit projected area, per unit solid angle (watts/(steradian $\cdot$ m^2)).

$$L = \frac{\mathrm{d}^2\Phi}{\mathrm{d}\omega \mathrm{d}A^{\perp}} = \frac{\mathrm{d}^2\Phi}{\mathrm{d}\omega \mathrm{d}A \cos\theta} \tag{2.9}$$

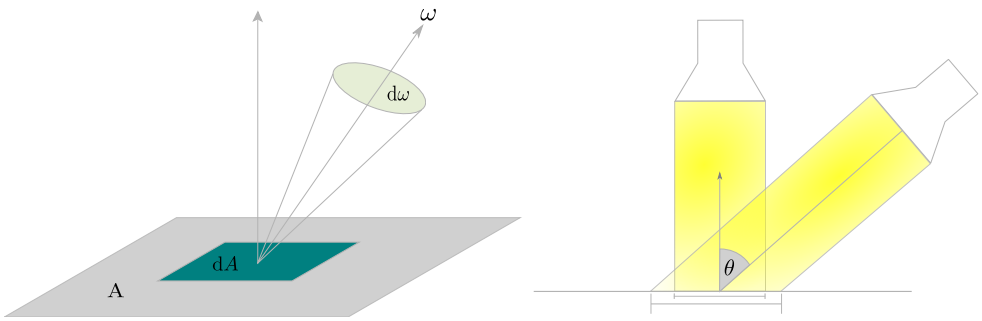


Figure 2.5: Radiance

The projected differential area $\mathrm{d}A^{\perp}$ is the area that is perpendicular to the direction vector ω. The differential solid angle, usually denoted as $\mathrm{d}\omega$ (see Equation 2.8) is centered around the direction vector (see left-hand side of Figure 2.5). When the photons hit the infinitesimal area $\mathrm{d}A$ at an angle, less photons reach the same amount of area because the same amount of power is spread over a larger area (see right-hand side of Figure 2.5). To compensate for this effect the differential area $\mathrm{d}A$ is multiplied by the cosine term (see Equation 2.9). The solid angle becomes smaller the further away an object or light source is, for example, the Sun appears dimmer on Jupiter than when on Earth but it still has the same brightness and the solid angle accounts for this.

2.3 Radiometric integrals

The irradiance [Vea98] is defined as the incident radiance at the point p by integrating over the hemisphere, i.e. over all directions ω. The $\cos\theta$ term stems from the definition of radiance.

$$E = \int_{\Omega} L_i(p,\omega)\cos\theta \mathrm{d}\omega \tag{2.10}$$

The flux [DBBS06] is the actual equation that every global illumination algorithm tries to solve. The parameter S can be thought of as all the (surface) points we are interested in. For example, if we were to compute the flux for a single pixel then the points of interest would be all the points seen through that pixel (see Figure 2.6). At each point p we compute the integral over the hemisphere or the directions ω we are interested in. For a pixel we would only be interested in one direction towards the camera for each point. The hard part in solving this equation is computing the radiance values.

$$\Phi(S) = \int_{A}\int_{\Omega} L_o(p,\omega)\cos\theta \mathrm{d}\omega \mathrm{d}A \tag{2.11}$$

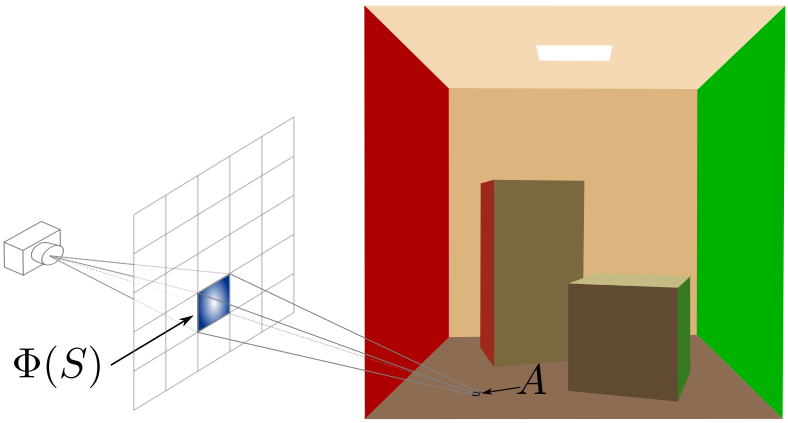


Figure 2.6: Radiant flux

The radiance values L_i and L_o represent the incident and exitant radiance from direction ω respectively. These values can be related to each other based on the law of conservation of energy, thus:

$$L_i(x,\omega) = L_o(y,-\omega). \tag{2.12}$$

The $\cos\theta$ term in the above equations can be distracting so we rewrite it in the form of the differential projected solid angle, which is the solid angle projected onto the unit disk (see Figure 2.7).

$$\Phi(S) = \int_{A}\int_{\Omega} L_o(p,\omega)\mathrm{d}\omega^{\perp}\mathrm{d}A \tag{2.13}$$

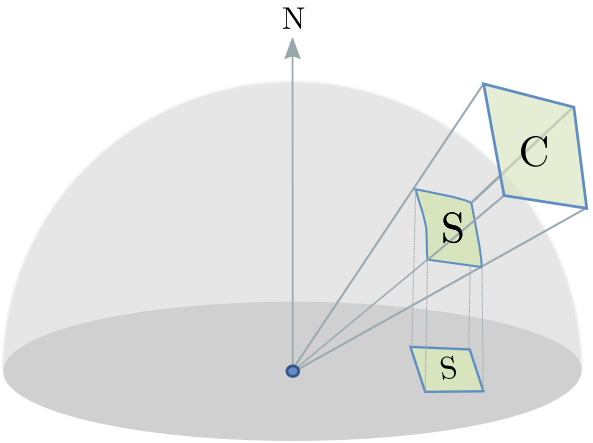


Figure 2.7: The solid angle S is projected onto the plane perpendicular to the surface normal N.

Given the nature of this integral, it is impossible to solve analytically except for the most simplest of scenes, so we sample multiple radiance values by tracing multiple rays through the pixel and then multiply the results by a filter function to help reconstruct the signal. The actual signal over the image plane is a continuous function; according to signal analysis the original function can be reconstructed from a set of discrete samples but in practice we will not be able to sample sufficiently to perfectly reconstruct the original signal. Therefore we use weighted averages (see Equation 2.16). We modify the equation by introducing the importance function, which is a simple analytic expression and will depend on the camera model being used [DBBS06].

$$\Phi(S) = \int_A \int_\Omega L_o(p,\omega) W_e(p,\omega) \mathrm{d}\omega^\perp \mathrm{d}A \tag{2.14}$$

This can also be written as the measurement equation [Vea98], where I_j is the jth pixel.

$$I_j = \int_A \int_\Omega L_i(p,\omega) W_e(p,\omega) \mathrm{d}\omega^\perp \mathrm{d}A \tag{2.15}$$

To solve global illumination, we will approximate the flux by computing the weighted averages [PH10] (see Equation 2.16), where f is the filter function and L is the radiance (see Section 2.7 for how the radiance is computed).

$$I_j \approx \frac{\sum fL}{\sum f} \tag{2.16}$$

2.4 Rendering equation

The theoretical underpinnings of the light transport problem in computer graphics can be distilled into a single recursive equation known as the rendering equation [Vea98] (see

Figure 2.8). This describes the illumination emitted at a single point in all directions of the hemisphere.

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_\Omega f_r(p, \omega_o, \omega_i) L_i(p, \omega_i) \cos\theta_i \, \mathrm{d}\omega_i \tag{2.17}$$

$L_o(p, \omega_o)$	the radiance leaving point p in the outgoing direction ω_o
$L_e(p, \omega_o)$	if p is itself a light source and emits in direction ω_o, we add its contribution
$L_i(p, \omega_i)$	the radiance incident at point p from the incoming direction ω_i
$f_r(p, \omega_o, \omega_i)$	bidirectional distribution function (BRDF) accounts for properties, such as matte and glossy surfaces

This can be written in shorthand operator notation

$$L = L_e + T_{f_r} L.$$

The T_{f_r} is the linear operator on L. By substituting the above equation into itself it can be expanded into a Neumann series [Vea98].

$$L = L_e + T_{f_r}(L_e + T_{f_r}(L_e + T_{f_r}(\dots$$

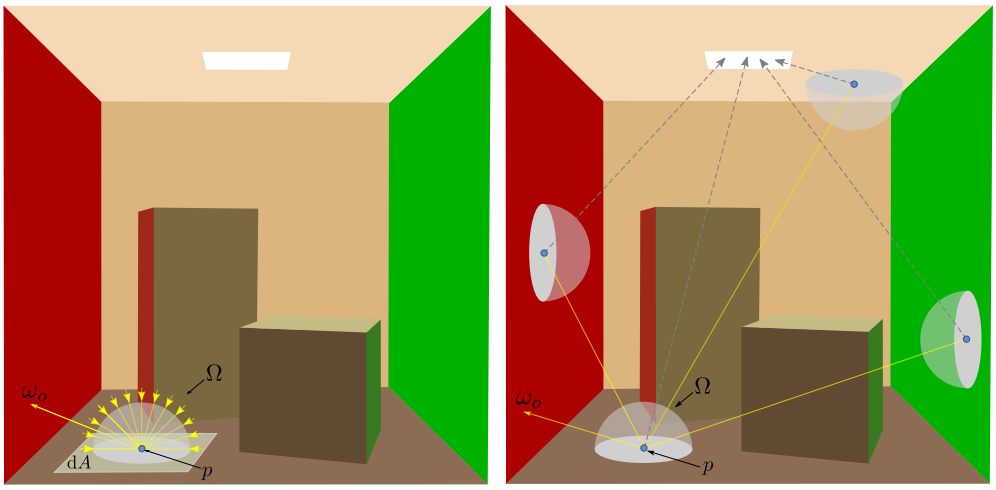
$$L = \sum_{i=0}^{\infty} T_{f_r}^i L_e$$

Physically based BRDFs (see Section 2.5) have a transport operator norm $\|T_{f_r}\| < 1$, which is a consequence of the law of conservation of energy and this property guarantees that the series converges. The first term of the series L_e, is the special case that includes the illumination, if the first point hit by a ray is a light source. The following L_e terms can be estimated by tracing shadow rays to the light sources at the current point in the path. The Equation 2.18 conveys this more succinctly as the sum of the emitted radiance plus the light scattered once, twice, etc.

$$L = \underbrace{L_e}_{\text{light emitter}} + \underbrace{TL_e}_{\text{direct illumination}} + \underbrace{T^2 L_e + \dots}_{\text{indirect illumination}} \tag{2.18}$$

2.4.1 Surface form

The rendering equation can be converted into an integral over all surfaces [Vea98] in the scene instead of over a sphere. As opposed to integrating over all possible directions on



(a) Radiance (hemisphere) (b) Radiance (recursive integral)

Figure 2.8: Radiance

a sphere from a point, we integrate over all possible surface points p that are visible from the point p', where A is all the surfaces in the scene. Switching between the surface and hemispherical form of the rendering equation can be done seamlessly (see Section 2.7.1).

$$L(p' \to p'') = L_e(p' \to p'') + \int_A f_r(p \to p' \to p'')L(p \to p')G(p \leftrightarrow p')\mathrm{dA}(p)$$

$$\mathrm{d}\omega_i^{\perp} = \mathrm{d}\omega^{\perp}(\widehat{p' - p}) = G(p \leftrightarrow p')\mathrm{dA}(p)$$

$$G(p \leftrightarrow p') = V(p \leftrightarrow p')\frac{\cos\theta\cos\theta'}{\|p - p'\|^2}$$

Here $\omega_i = \widehat{p' - p}$ is the unit length vector and $V(p \leftrightarrow p')$ is 1 if p and p' are mutually visible and 0 otherwise. The angles between the unit length vector $\widehat{p' - p}$ and the normals at the point p and p' are θ and θ' (see Figure 2.9).

The measurement equation [Vea98] can be reformulated as,

$$I_j = \int_A \int_A L(p \to p')W_e(p \to p')G(p \leftrightarrow p')\mathrm{dA}(p)\mathrm{dA}(p').$$

We now write the surface form equation as a Neumann series, this reveals that the equation in the surface point form is the sum of all possible paths with all possible lengths [Vea98] (see Figure 2.10).

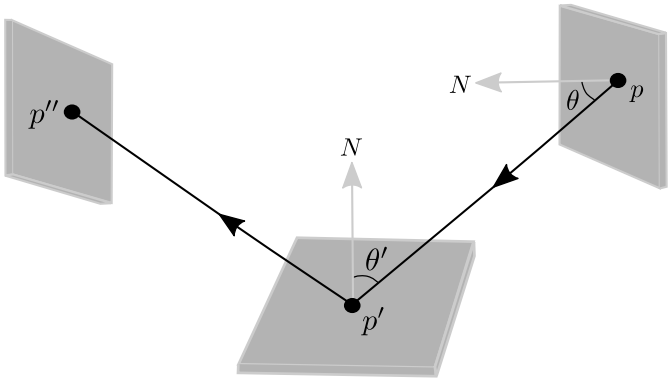


Figure 2.9: Three-point form

$$
\begin{aligned}
L(p_1 \to p_0) = {} & L_e(p_1 \to p_0) \\
& + \int_A L_e(p_2 \to p_1) f_r(p_2 \to p_1 \to p_0) G(p_2 \leftrightarrow p_1) \mathrm{dA}(p_2) \\
& + \int_A \int_A L_e(p_3 \to p_2) f_r(p_3 \to p_2 \to p_1) G(p_3 \leftrightarrow p_2) \\
& \qquad \times f_r(p_2 \to p_1 \to p_0) G(p_2 \leftrightarrow p_1) \mathrm{dA}(p_3) \mathrm{dA}(p_2) + \ldots
\end{aligned}
$$

This can be more succinctly defined as

$$
\begin{aligned}
L(p_1 \to p_0) &= \sum_{i=0}^{\infty} P(i) \\
P(i) &= \int_{A^i} L_e(p_{i+1} \to p_i) \\
&\qquad \times \left(\prod_{j=1}^{i} f_r(p_{j+1} \to p_j \to p_{j-1}) G(p_{j+1} \leftrightarrow p_j) \right) \mathrm{dA}(p_2) \cdots \mathrm{dA}(p_{i+1}).
\end{aligned}
\tag{2.19}
$$

2.5 BRDFs

The bidirectional distribution function (BRDF) describes the scattering of light at surfaces. To define the BRDF we show its relation to the radiometric quantities [Vea98]

$$\mathrm{d}E(p, \omega_i) = L_i(p, \omega_i) \mathrm{d}\omega^{\perp}$$

and

$$\mathrm{d}L_o(p, \omega_o) = f_r(p, \omega_i, \omega_o,) L_i(p, \omega_i) \mathrm{d}\omega_i^{\perp}.$$

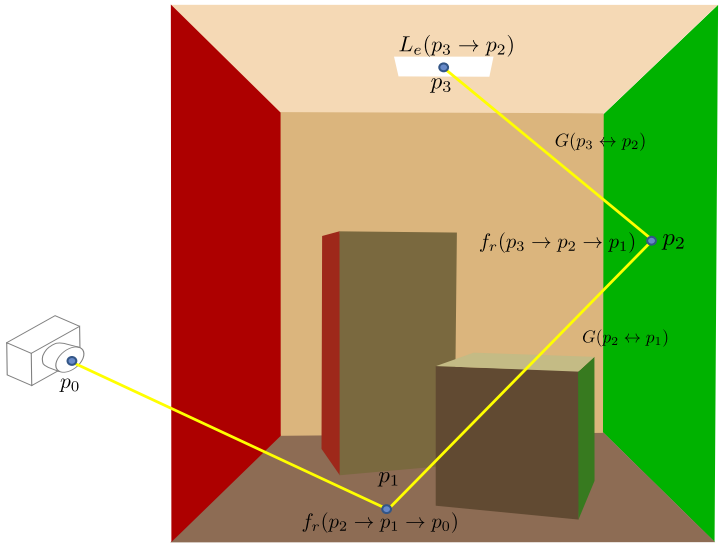


Figure 2.10: Path tracing

There is a proportional increase in $\mathrm{d}L_o(p, \omega_o)$ when increasing $\mathrm{d}E(p, \omega_i)$ [Vea98]:

$$\mathrm{d}L_o(p, \omega_o) \propto \mathrm{d}E(p, \omega_i).$$

So the BRDF just defines the proportionality constant as the measure of radiance per unit of irradiance.

$$f_r(p, \omega_i, \omega_o) = \frac{\mathrm{d}L_o(p, \omega_o)}{\mathrm{d}E(p, \omega_i)}$$

Physical BRDFs have two important properties:

1. The Helmholtz reciprocity states that they will return the same result if the incident and exitant directions are interchanged.

$$f_r(p, \omega_i, \omega_o) = f_r(p, \omega_o, \omega_i)$$

2. They preserve the law of conservation of energy, so that the reflected power in all directions is less than or equal to the amount of power incident on the surface and typified by the equation

$$\int_\Omega f_r(p, \omega_i, \omega_o) \mathrm{d}\omega_o^\perp \leq 1.$$

Diffuse surfaces, such as matte paint scatter light equally in all directions. This means the radiance is equal no matter the direction when looking at a perfectly diffuse surface. A specular BRDF reflects light in a specific direction, such as glass and mirrors. The viewing direction of a mirror will change the object that is reflected. Glossy surfaces scatter light in a set of particular directions (see Figure 2.11).

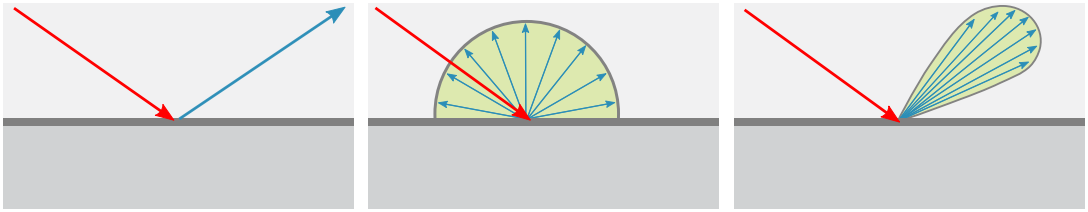


Figure 2.11: From left to right: perfect specular, diffuse and glossy specular BRDFs.

2.5.1 Lambertian reflection

The simplest BRDF that can be modelled is a perfectly diffuse surface. This is known as the Lambertian model and is just a constant. The reflectivity constant p_d is the percentage of light leaving the surface.

$$f_r(p, \omega_i, \omega_o) = k_d = \frac{p_d}{\pi}$$

The Lambertian model is not based on any real-world materials, since no ideal diffuse surfaces exist. However, the simple assumption that the radiance is uniform in all directions is a nice property to compute, i.e a constant.

2.5.2 Modified Blinn-Phong model

The modified Blinn-Phong model describes diffuse surfaces with specular highlights. The specular term is defined by k_s. N is the surface normal at the point p. H is the half-vector $\frac{V+L}{\|V+L\|}$, which approximates the perfect reflection direction. V is the direction towards the camera and L is towards the light source and the exponent n determines the roughness of the surface.

$$f_r(p, \omega_i, \omega_o) = k_s(N \cdot H)^n + k_d$$

The modified Blinn-Phong model does not satisfy Helmholtz's reciprocity and neither the law of conservation of energy. Although the terms in the above equation satisfy the constraints $k_s \leq 1$ and $k_d \leq 1$, it is not sufficient to be energy conserving. Further information of an example of a physically based BRDF can be found in Cook-Torrance [CT82] and a BRDF based on empirical data is given in Ward [War92]. Finding analytical expressions to model materials is not always possible. One way to solve this, is to measure

real materials and then tabularize the data. The value of the BRDF is then read given the directions. Some empirical models derive analytical expressions by trying to fit the actual measurements and thus approximate real materials [LFTG97].

2.6 Monte Carlo methods

In this section we describe the techniques to numerically compute the rendering equation. Monte Carlo integration is based on probability theory along with random samples to solve integrals. A simple example will demonstrate the main concepts behind Monte Carlo methods by computing π. The terms introduced in the following example will be explained in more detail further on in the section. The idea is straightforward, we know that the unit circle has a radius of 1 and therefore its diameter is 2. This means that the square that encloses the unit circle has sides of length 2 and therefore its area is 4. The equation below gives us the ratio we will compute in order to calculate π.

$$p = \frac{\text{area of circle}}{\text{area of square}} = \frac{\pi}{4} = 0.7853$$

So simple algebraic manipulation gives us

$$\text{area of circle} = \text{area of square} \cdot p = 4 \cdot p = \pi.$$

All we need to do is generate random real numbers between -1 to 1 and then make sure that the condition $x^2 + y^2 \leq 1$ is satisfied. If this is the case, then the point lies in or on the circle otherwise the point lies outside the circle but within the square that encloses the circle (see Figure 2.12). Then our ratio is based on the number of points that fall within the circle divided by the number of points we decide to generate.

$$\text{area of circle} = 4 \cdot \frac{\text{number of points in circle}}{\text{number of samples}} \approx \pi$$

Generating 50000 samples gives us an approximation that is correct to three decimal places. Figure 2.12 has 500 samples plotted, 397 points were in the circle and 123 outside, so $\pi \approx 4 \cdot \frac{397}{500} = 3.176$.

The only known feasible way of solving integral equations in higher dimensions are Monte Carlo methods. Samples are drawn from a probability distribution to indirectly compute the expected value which is the solution of the integral. The expected value for a discrete random variable over n possible outcomes is

$$E[x] = \sum_{i=1}^{n} x_i p_i.$$

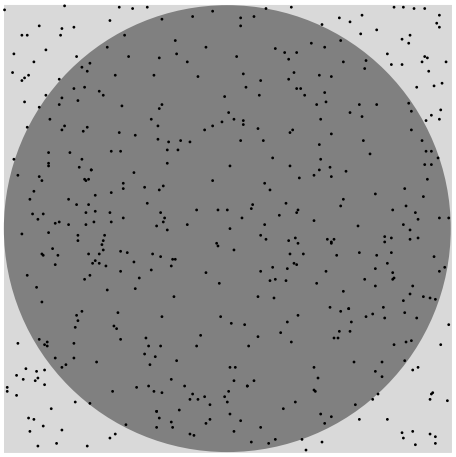


Figure 2.12: π approximated by $4 \cdot \frac{\text{points in circle}}{\text{all points}}$

The fair die has a set of possible values or events for the random variable $x_i = \{1, 2, 3, 4, 5, 6\}$ and the probability of one of these events occurring is $p_i = \frac{1}{6}$. The expected value when the die is thrown is

$$E[x] = \sum_{i=1}^{6} x_i \cdot \frac{1}{6} = 1 \cdot \frac{1}{6} + 2 \cdot \frac{1}{6} + 3 \cdot \frac{1}{6} + 4 \cdot \frac{1}{6} + 5 \cdot \frac{1}{6} + 6 \cdot \frac{1}{6} = 3.5.$$

Throwing the die long enough and averaging the result will give us the expected value according to the law of large numbers. More precisely, the outcome of the arithmetic mean will converge to the expected value by repeating the experiment often enough, the approximation (see Equation 2.20) becomes exact as $\lim N \to \infty$.

$$E[f(x)] \simeq \frac{1}{N} \sum_{i=1}^{N} f(x_i) \tag{2.20}$$

The expected value for a continuous random variable is

$$E[f(x)] = \int f(x)p(x)\mathrm{d}x.$$

The following identities are also useful, where a is a constant.

$$E[af(x)] = aE[f(x)]$$
$$E[\sum_{i=1}^{N} f(x_i)] = \sum_{i=1}^{N} E[f(x_i)]$$

Monte Carlo methods allow us to numerically compute the rendering equation by sampling from a probability distribution, so the function can be continuous and more importantly discontinuous by reason of point sampling.

2.6.1 Monte Carlo estimator

The Monte Carlo integration estimator is defined as the average of the sum of functions divided by the probability distribution function (PDF) $p(x)$ and the random variable x_i is sampled from this distribution [Vea98].

$$F_N = \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{p(x_i)}$$

We show how the expected value of F_N is the integral of the function $f(x)$ we want to evaluate.

$$\begin{aligned} E[F_N] &= E[\frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{p(x_i)}] \\ &= \frac{1}{N} \sum_{i=1}^{N} \int_a^b \frac{f(x)}{p(x)} p(x) \mathrm{d}x \\ &= \frac{1}{N} \sum_{i=1}^{N} \int_a^b f(x) \mathrm{d}x \\ &= \int_a^b f(x) \mathrm{d}x \\ &= I \end{aligned}$$

A large number of N samples will give us the expected value when computing the estimator F_N.

$$I \approx F_N$$

A simple example will show how this works, below is an integral that is equal to 1.

$$F = \int_0^1 4x^3 \mathrm{d}x$$

The estimator of this integral is divided by a uniform PDF where the random variable x_i is in the interval $[0, 1)$. The fair die is an example of a uniform PDF, since each event has the same likelihood (i.e. probability $\frac{1}{6}$). Many programming language libraries provide a pseudorandom number generator (PRNG) that sample from a uniform PDF in the mentioned interval. The PDF should have the property that it integrates to 1 given the domain it is valid for, this is known as a normalized PDF. The PDF $p(x)$ is equal to 1 and the area underneath this graph is trivially 1 (see Figure 2.13).

This will lead us to the following estimator

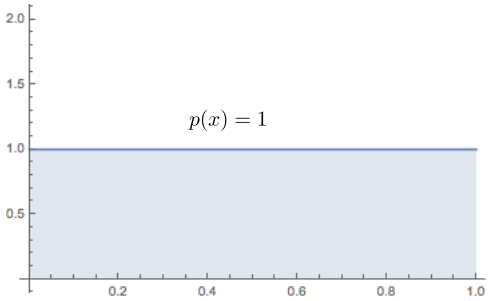


Figure 2.13: Uniform PDF.

$$F_N = \frac{1}{N} \sum_{i=1}^{N} 4x^3.$$

Empirical observation of drawing a 1000 samples from the uniform PDF will give a result close to the expected value of this estimator. As is most often the case, many more samples will need to be drawn from a uniform PDF than when taking a PDF that has the "shape" of the function we are trying to estimate. In other words, the PDF should have a distribution, such that drawing random variables will more likely make the estimator converge quicker, thereby needing less samples. Matching a PDF to more closely resemble the function in question is called importance sampling. For the above example, choosing a PDF with a distribution of x^3 is preferable to a uniform PDF. The reason for this is that we want to draw values from a PDF that are more likely than others, whereas with a uniform PDF all values are equally likely.

Let us look at a non-uniform PDF to make the point more clear, the PDF $p(x) = 2x$ integrates to 1 over the defined domain $[0, 1]$ and is therefore normalized. It is obvious that the right half of the area under the graph (see Figure 2.14) is much larger than the first half.

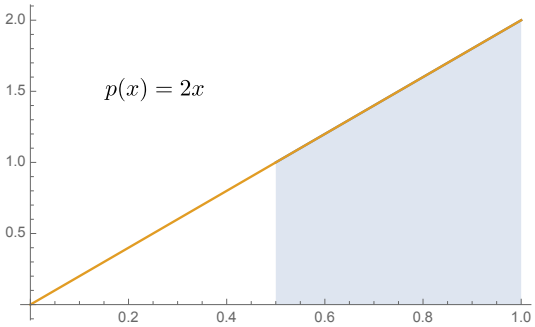


Figure 2.14: Non-uniform PDF.

What this graph is showing us when sampling the PDF, is that the random variable x is much more likely to take on a value in the interval $[\frac{1}{2}, 1]$. The area under the graph

gives us an idea of how likely a random variable will be in a particular domain. The cumulative distribution function (CDF) gives us the answer to how likely such a random variable is. We first define the CDF for a continuous random variable.

$$P(x \in [a, b]) = \int_a^b p(x)\mathrm{d}x$$

As an example to show the reasoning behind the CDF, we ask how likely it is for a random variable to take on a value in the domain $[0, \frac{1}{2}]$. The result could be computed by just looking at the graph for the PDF, which is 0.25. So the chances of the random variable taking on a value in the interval $[0, \frac{1}{2}]$ is 25%. The probability that the random variable will take on a value in the interval from 0.7 to 1 is 51%, so the odds of a value being randomly selected is more than twice as likely than for the first half of the domain. Below we graph the CDF, which shows us exactly how the random variable is distributed; by looking at the vertical axis it can be seen that 75% of the values of the range $[\frac{1}{4}, 1]$ covers half of the domain, i.e. where $x \in [\frac{1}{2}, 1]$. What this means is that we can use a PRNG to generate uniformly distributed random variables and then plug them into the inverse CDF to generate the random variable that represents the distribution we are trying to sample, this is known as the inversion method (see Figure 2.15).

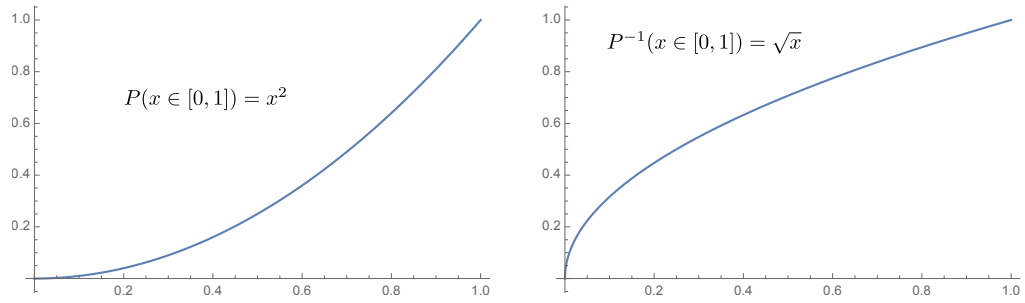


Figure 2.15: CDF and inverse CDF.

Sampling random variables from the PDF $p(x) = 2x$ with the inversion method and plotting their histograms shows the variables are distributed such that they take on the shape of the area underneath the PDF (see Figure 2.16).

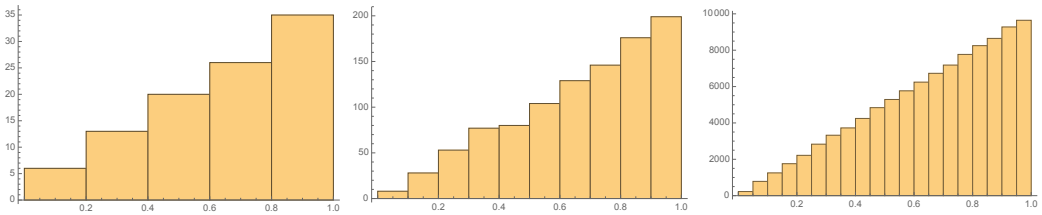


Figure 2.16: Histograms with a 100, 1000, and 100 000 samples respectively.

Ending this section we note the major benefit of Monte Carlo methods is the capability to solve integrals in higher dimensions without having to generate exponentially

many samples when compared to deterministic quadrature techniques. To estimate these integrals, N samples are taken from PDFs that have multiple random variables [PH10].

$$F = \int \int f(x, y) \mathrm{d}x \mathrm{d}y$$
$$F_N = \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i, y_i)}{p(x_i, y_i)}$$

2.6.2 Uniform hemisphere sampling

There are various techniques for selecting distribution or density functions to solve the rendering equation. We show examples of densities for the uniform and importance sampling strategy [PH10]. With uniform sampling we have the option of randomly choosing a point on the hemisphere, which will give us a new direction to propagate the ray. The uniform PDF is a constant, so $p(\omega) = c$ and the density function should necessarily integrate to 1. To find the PDF of the hemisphere, we integrate the solid angles over the area of the hemisphere (see Equations 2.7 and 2.8).

$$\int p(\omega) \mathrm{d}\omega = 1$$
$$c \int \mathrm{d}\omega = 1$$
$$c \int_0^{2\pi} \int_0^{\frac{\pi}{2}} \sin\theta \mathrm{d}\theta \mathrm{d}\phi = 1$$
$$c \int_0^{2\pi} \mathrm{d}\phi \int_0^{\frac{\pi}{2}} \sin\theta \mathrm{d}\theta = 1$$
$$c \int_0^{2\pi} \mathrm{d}\phi \cdot 1 = 1$$
$$c \cdot 2\pi = 1$$
$$c = \frac{1}{2\pi} \Rightarrow p(\omega) = \frac{1}{2\pi}$$

To randomly generate a direction with cartesian coordinates on the hemisphere the

CDF of the density is converted with respect to spherical coordinates.

$$
\begin{aligned}
P(\omega) &= \int p(\omega)\mathrm{d}\omega \\
P(\theta,\phi) &= \frac{1}{2\pi}\int_0^{\phi}\int_0^{\theta}\sin\theta\mathrm{d}\theta\mathrm{d}\phi \\
P(\theta,\phi) &= \frac{1}{2\pi}\int_0^{\phi}\mathrm{d}\phi\int_0^{\theta}\sin\theta\mathrm{d}\theta \\
P(\theta,\phi) &= \frac{\phi}{2\pi}\int_0^{\theta}\sin\theta\mathrm{d}\theta \\
P(\theta,\phi) &= \frac{\phi}{2\pi}(1-\cos\theta)
\end{aligned}
$$

The CDF is separable into θ and ϕ, since the spherical coordinates are independent of each other.

$$
\begin{aligned}
P(\theta) &= 1-\cos\theta \\
P(\phi) &= \frac{\phi}{2\pi}
\end{aligned}
$$

The inverted functions are

$$\theta = \cos^{-1} u_1$$

and

$$\phi = 2\pi u_2,$$

where $1-u$ is replaced with u_1 since the uniform random variable is in the interval $[0,1)$. The final step is to map these functions to cartesian coordinates to finally compute the random direction.

$$
\begin{aligned}
x &= \sin\theta\cos\phi = \cos(2\pi u_2)\sqrt{1-u_1^2} \\
y &= \sin\theta\sin\theta = \sin(2\pi u_2)\sqrt{1-u_1^2} \\
z &= \cos\theta = u_1
\end{aligned}
$$

2.6.3 Cosine lobe sampling

In importance sampling we try to match the shape of the function that is being estimated. In this example, we draw samples from a cosine weighted density function that matches the cosine term in the rendering equation [PH10]. The density is cosine weighted and as in the previous example we first compute the normalizing constant.

$$p(\omega) \propto \cos\theta$$

$$\int c \cdot p(\omega)\mathrm{d}\omega = 1$$
$$c\int_0^{2\pi}\int_0^{\frac{\pi}{2}} \cos\theta \sin\theta \mathrm{d}\theta\mathrm{d}\phi = 1$$
$$c \cdot 2\pi \cdot \frac{1}{2} = 1$$
$$c = \frac{1}{\pi} \Rightarrow p(\omega) = \frac{\cos\theta}{\pi}$$

So we convert the PDF with respect to spherical coordinates with the following property

$$p(\theta, \phi) = \sin\theta p(\omega)$$

and then we obtain

$$p(\theta, \phi) = \frac{\sin\theta\cos\theta}{\pi}.$$

Many useful distributions cannot be separated as easily into its individual densities as the current and the previous example. We show how it is done in the general case by first computing the marginal density for θ and then the conditional density for ϕ. The definition for the marginal distribution for a continuous random variable is

$$p(x) = \int p(x, y)\mathrm{d}y,$$

while the conditional density is

$$p(y|x) = \frac{p(x, y)}{p(x)}.$$

Now, the marginal density for the above example is

$$p(\theta) = \int_0^{2\pi} p(\theta, \phi)\mathrm{d}\phi = \int_0^{2\pi} \frac{\sin\theta\cos\theta}{\pi}\mathrm{d}\phi = 2\sin\theta\cos\theta$$

and the conditional density for ϕ:

$$p(\phi|\theta) = \frac{p(\theta, \phi)}{p(\theta)} = \frac{1}{2\pi}.$$

As before, we compute the CDF for the separable densities:

$$P(\theta) = \int_0^{\theta} 2\sin\theta'\cos\theta'\mathrm{d}\theta' = \sin^2\theta = 1 - \cos^2\theta$$
$$P(\phi|\theta) = \int_0^{\theta} \frac{1}{2\pi}\mathrm{d}\theta' = \frac{\theta}{2\pi}.$$

Again, we map θ and ϕ to cartesian coordinates with the help of the inverted functions:

$$\theta = \cos^{-1}\sqrt{u}_1,$$
$$\phi = 2\pi u_2.$$

2.7 Stochastic ray tracing

The previous sections introduced the theory and methods to estimate the rendering equation. In this section we show a complete path tracer (see [Kaj86] and [Vea98]) in pseudo code that renders with indirect illumination effects. The global illumination path tracer solves the problems of the Whitted ray tracer (see Section 2.1.2). As pointed out in Section 2.1.2 a stopping condition is still needed for the ray, we deal with that in this section. Let us have another look at the rendering equation with the transport operator.

$$L = L_e^{\text{emitter}} + TL_e^{\text{direct}} + T(TL_e + T(TL_e + ...))^{\text{indirect}} \tag{2.21}$$

We split the equation into two terms and ignore the light emitter because it is the easiest part to compute. Note that in the above equation, the indirect term always incorporates the direct illumination every time the light is scattered.

$$L_o = L_o^{\text{direct}} + L_o^{\text{indirect}}$$

Now we estimate the direct term first:

$$L_{direct}(x, \omega_o) = \frac{1}{N} \sum_{i=1}^{N} \frac{L_e(x, \omega_i) f(x, \omega_o, \omega_i) \cos\theta_i}{p(\omega_i)}.$$

The x is the point in the scene for which we are computing the radiance in the direction ω_o, this direction will be towards the camera or eye when retracing the ray back to its origin. The peculiarity of tracing rays in the opposite direction may seem strange, since photons travel from the light source but the conservation of energy allows for this discrepancy. The direction ω_i is the direction from the light source to x and the angle of $\cos\theta$ is between the surface normal at x and the vector ω_i. The hard part about evaluating direct illumination is determining what density to use. Now it is possible to sample a hemispherical uniform PDF or a density that matches the BRDF or cosine term. The problem with sampling either one of these PDFs at x is that the probability of hitting a light source will be zero, since light sources are few and far between the scene (this can be solved with multiple importance sampling [Vea98]). The only way to sample a PDF for the estimator not to equal zero, is to sample a distribution from the light source. The simplest light source is a point light source and as the name suggests it is a single point in the scene that emits light uniformly in all directions. Now we be might be tempted to uniformly sample this light source as a spherical PDF (see Section 2.6.2) but we run into the same problem as before. The likelihood of sampling a direction that will intersect the point x will be zero. The Dirac delta distribution models this scenario exactly, since there is only a single direction from the light source where the point x will be illuminated. The Dirac delta distribution is zero everywhere except for a single point that will be equal to one. We don't need to consider this distribution further, since we trace a shadow ray to the light source to determine the direction ω_i. If the point light

is visible the PDF is 1 and 0 otherwise. The power of light source L_e is read from the scene file, the BRDF for Lambertian and the Phong models can be evaluated according to Section 2.5 and the cosine term is computed from the scene geometry.

The next step is to evaluate the more problematic indirect illumination term.

$$L_{indirect}(x, \omega_o) = \frac{1}{N} \sum_{i=1}^{N} \frac{L_r(x, \omega_i) f(x, \omega_o, \omega_i) \cos \theta_i}{p(\omega_i)} \tag{2.22}$$

The difficult part is to assess the recursive term L_r, if the light is only scattered twice then we can just substitute L_r with L_{direct}, if we continue to propagate the light then we just keep substituting according to Equation 2.21. Note the transport operator in Equation 2.21 can be thought of as the means for propagating the light ray. This time we sample from a uniform, cosine or BRDF modeled distribution to randomly select a new direction for the ray to travel on. The BRDF should be sampled if the surface reflects light only in specific directions such as glossy and specular materials. For further details we refer the reader to [DBBS06].

2.7.1 Surface form conversion

Sampling the probability with respect to the surface area will change the Equations 2.21 and 2.22. As an example, the probability of the surface area can be $\frac{1}{area}$ and a point can be sampled on the surface uniformly. To convert between probabilities of solid angle p_ω and area p_A the terms are related by the equation

$$p_A = p_\omega \frac{\cos \theta'}{\|x - x'\|^2}.$$

The estimator for the surface form

$$L(x' \to x'') = \frac{1}{N} \sum_{i=1}^{N} \frac{L(x \to x') f(x \to x' \to x'') G(x \leftrightarrow x')}{p_A(x)} \tag{2.23}$$

and the modified equation with respect to sampling a solid angle is then

$$L(x' \to x'') = \frac{1}{N} \sum_{i=1}^{N} \frac{L(x \to x') f(x \to x' \to x'') \cos \theta}{p_\omega(x - x')}.$$

For further details we refer the reader to [Vea98].

2.7.2 Russian roulette

The idea of Russian roulette is to select a probability with which to stop the light from bouncing around in the scene [AK90]. For example, if we were to terminate the path half of the time, we would only get half of the brightness. To compensate for the paths that are rejected in this example, the radiance for the path that is sampled is multiplied

by two or divided by a half thereby providing a correct solution for the estimator. The theory allows for paths with infinite length and it may well be that paths of great length add a large contribution to the final estimator. Russian roulette allows for the possibility of such paths. Note that a scene with mostly diffuse surfaces will require smaller paths than scenes with many specular surfaces. The estimator is a sum of recursive estimators where the terms are defined as F_i (see Equation 2.21).

$$F = F_1 + ... + F_N$$

An arbitrary probability p_i is chosen and a random variable $\xi_i \in [0, 1)$ is drawn from a uniform distribution, if the random variable is greater than our chosen probability p_i we skip the path. The integral is computed with probability p_i and the original estimator is weighted by this probability [Vea98].

$$F_i' = \begin{cases} \frac{1}{p_i} F_i & \xi_i < p_i \\ 0 & \text{otherwise} \end{cases}$$

The expected value of the new estimator is shown to prove that the correct result on average is returned.

$$\begin{aligned} E[F_i'] &= p_i \frac{1}{p_i} F_i + (1 - p_i) \cdot 0 \\ &= F_i \end{aligned}$$

The expected value of the number of times the path is reflected is equal to the geometric series of an infinite sum.

$$\sum_{k=0}^{\infty} p^k = \frac{1}{1-p} \qquad 0 < p < 1$$

With probability $p = \frac{1}{2}$ for the above equation the path is *expected on average* to be of length 2. The third term of the above series means that the probability of a path being extended to a length of 3 with the same probability $p = \frac{1}{2}$ is $p^2 = (\frac{1}{2})^2 = \frac{1}{4}$. Note that each time the path is lengthened a substitution or recursive call is being made (see Equation 2.22). Instead of choosing the probability arbitrarily, the contribution (i.e. radiance) of a path could decide whether or not to propagate a path depending on if it is greater or less than some random variable. The estimator for the indirect term (see Equation 2.22) is the mean of N samples at point x and the probability of N paths being continued is $Np < 1$. To clarify this, the first terms of the geometric series are shown.

$$1 + Np + (Np)^2 + (Np)^3 + ...$$

This reduces the probability of each path being propagated by N samples. For example, the probability of propagating a single path of length 2 is 50% given our arbitrary

probability of $p = \frac{1}{2}$. If 100 samples (i.e. 100 paths) were evaluated then the probability for each path of length 2 would be just 0.5% and with length 3 it would be 0.25%. That is why we stick to the spirit of the original Monte Carlo path tracing paper by Jim Kajiya by tracing only a single path. In Kajiya's seminal paper he did the same for direct illumination, only a sinlge ray is traced to the light source. For multiple lights, a source is selected randomly.

2.7.3 Stochastic path tracer

All the theory that has been introduced up until now can be put to use. The pseudo path tracer code described in this section is the simplest ray tracing method to solve global illumination. Many advanced techniques are similar to the path tracer but the main difference is the application of sophisticated statistical methods that usually try to reduce computation time for the estimator or trace specific paths to simulate various effects. Difficult light paths such as caustics is an example of an effect that might be impossible for a path tracer to simulate. Photon mapping is a popular technique to render scenes with caustics (see [JC95] for further details).

Algorithm 3 Stochastic path tracer

```
 1: for each pixel do
 2:     radiance = 0
 3:     weight = 0
 4:     for each path do
 5:         ray = trace ray from eye through pixel
 6:         x = ray.hitpoint
 7:         ω_o = ray.direction
 8:         if point x is an emitter then
 9:             radiance += L_e
10:         end if
11:         radiance += ComputeRadiance(x, ω_o)
12:         filter = FilterFunction(j_th pixel)
13:         radiance *= filter
14:         weight += filter
15:     end for
16:     radiance /= weight
17:     j_th pixel = radiance
18: end for
```

Alogrithm 3 sums up the previous sections of how to solve the global illumination problem. The box filter is the simplest candidate for the filtering function (see line 12). It computes the average of the surrounding pixels at the j_{th} pixel, further details can be found in [PH10]. Note that the direct and indirect computation use only 1 sample, since multiple paths are traced for each individual pixel (see line 4). The probability on line 30 is equal to 1 if the light source is a point light. On the other hand, area light sources are made up of polygons and the probability is sampled with respect to the area it covers. The direct illumination for area lights is then estimated as an integral over the surface area of the scene instead of directions on a sphere (solid angles). Section

2.7.1 shows how the estimator needs to be modified to accommodate surface areas, the indirect term if necessary can be altered in the same way.

```
19: function ComputeRadiance(x, ω_o)
20:     radiance += DirectIllumination(x, ω_o)
21:     radiance += IndirectIllumination(x, ω_o)
22:     return radiance
23: end function
24:
25: function DirectIllumination(x, ω_o)
26:     radiance = 0
27:     shadowRay = trace ray from x to light source
28:     if shadowRay intersects nothing - light source is visible then
29:         radiance = L_e * BRDF * cos θ
30:         radiance /= probability sampling light source
31:     end if
32:     return radiance
33: end function
34:
35: function IndirectIllumination(x, ω_o)
36:     radiance = 0
37:     ξ = sample uniform random variable where ξ ∈ [0, 1)
38:     if ξ < arbitrary probability then
39:         ω_i = sample new direction uniformly on hemisphere
40:         uniformHemispherePdf = 1/2π
41:         ray = trace ray from x in direction ω_i
42:         point = ray.hitpoint
43:         radiance += ComputeRadiance(point, ω_i) * BRDF * cos θ
44:         radiance /= uniformHemispherePdf
45:         radiance /= arbitrary probability
46:     end if
47:     return radiance
48: end function
```

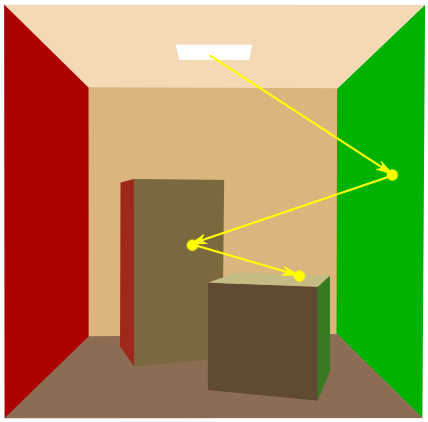
Chapter 3

Lightcuts

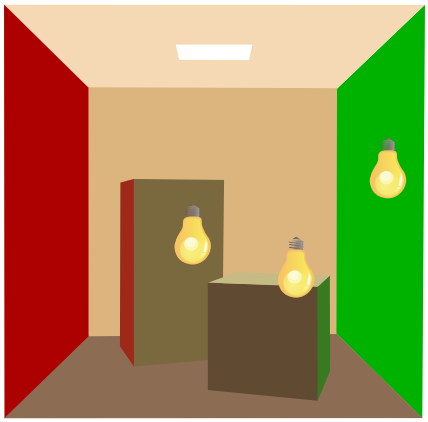
In the previous chapter we alluded to the fact that rays in the stochastic ray tracer were traced from the camera towards the direction of the light source. This may seem counter intuitive but rest assured that randomly tracing rays from the light source and hoping that it'll hit the much more smaller area surface of the camera is much less likely than aiming the camera at the scene and then tracing the rays. More advanced methods such as bidirectional ray tracing, trace rays from both directions and then connects the camera and light paths, it is suitable for more difficult scenes that may not be able to be rendered with path tracing [Vea98]. Before we focus on lightcuts, we will first discuss instant radiosity [Kel97] in the upcoming section, which is a method that generates many light sources called *virtual point lights* (VPLs) and is more akin to bidirectional path tracing than standard path tracing. The overarching idea of lightcuts is to reduce visibility tests given many light sources or VPLs when computing indirect illumination, by selecting only a subset of the light sources that add the biggest contribution to a point in the scene.

3.1 Virtual point lights

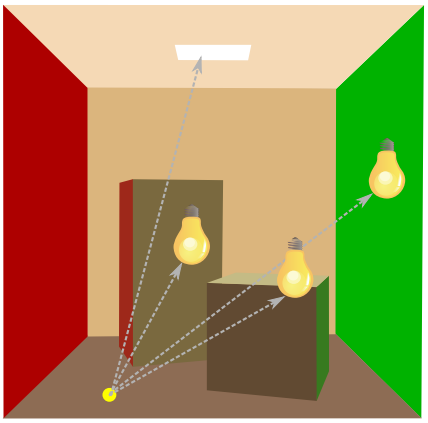
The instant radiosity algorithm by Keller [Kel97] is the bases of all many-lights methods [DKH+14] and has a pre- and post-processing stage. In the pre-processing phase, energy is carried from the light sources by tracing rays and depositing a VPL at the intersection points. The ray is then reflected to deposit another VPL until it is absorbed based on scene reflectivity. The VPLs store the indirect illumination, so that only direct lighting has to be computed in the post-processing phase. When accessing the graphics pipeline only direct illumination can be computed (see Figure 3.1). The central idea of Keller's paper is that we can treat the VPLs as direct light sources which is something modern graphics cards can evaluate very quickly with shadow maps [Wil78]. Ray tracing is responsible for generating the VPLs and the graphics hardware renders the final image. This is a hybrid method juxtaposed to the approach in this thesis, which relies completely on ray tracing for both pre- and post-processing stages. In a fully ray based solution the indirect illumination is computed by tracing only primary and shadow rays.



(a) Trace rays from light source



(b) Deposit VPLs at intersection points



(c) Compute indirect illumination from VPLs as direct light sources

Figure 3.1: Instant radiosity

Shadow rays emitted from the intersection points of the primary rays determine the visibility of the VPLs and then the points are shaded appropriately. We now discuss how all the equations in Chapter 2 that lead up to the surface form (see Section 2.4.1) can be connected to the generation of VPLs. The stochastic path tracer (see Section 2.7.3) computes the radiance for paths of arbitrary length, where the length of a path is determined by Russian roulette (see Section 2.7.2). The stochastic path tracer can be adapted to trace paths from the light source instead of the camera to deposit the radiance transferred along a path as a VPL, this allows a path to be reused multiple times because each point we render will include each VPL that is visible (see Figure 3.1). In summary, the stochastic path tracer must be modified to trace rays that start from the light source and only include the direct illumination from the light source (see Algorithm 4). This means that we can forget about the direct illumination function (see Algorithm 3 lines 20 and 25 to 33) because we are only computing a single path each time we trace a ray from the light source. The reason for this is that we are only

evaluating a single term from the infinite series (see Equation 2.19) where the estimator for a single path $P(k)$ [PH10] with k being the length of the path can be written as

$$P(k) = \frac{L_e(p_k \to p_{k-1}) f_r(p_k \to p_{k-1} \to p_{k-2}) G(p_k \leftrightarrow p_{k-1})}{p_A(p_k)} \times \prod_{j=1}^{k-2} \frac{f_r(p_{j+1} \to p_j \to p_{j-1}) \cos\theta_j}{p_\omega(p_{j+1} - p_j)}. \tag{3.1}$$

The path cannot be fully specified because we still need to connect it to the camera (see Figure 3.2), which is only done in the post-processing or rendering stage, so the Equation 3.1 [PH10] needs to be modified as follows:

$$P(k) = \alpha f_r(p_3 \to p_2 \to p_1) G(p_2 \leftrightarrow p_1) f_r(p_2 \to p_1 \to p_0) \tag{3.2}$$

where

$$\alpha = \frac{L_e(p_k \to p_{k-1}) f_r(p_k \to p_{k-1} \to p_{k-2}) G(p_k \leftrightarrow p_{k-1})}{p_A(p_k)} \times \prod_{j=3}^{k-2} \frac{f_r(p_{j+1} \to p_j \to p_{j-1}) \cos\theta_j}{p_\omega(p_{j+1} - p_j)}.$$

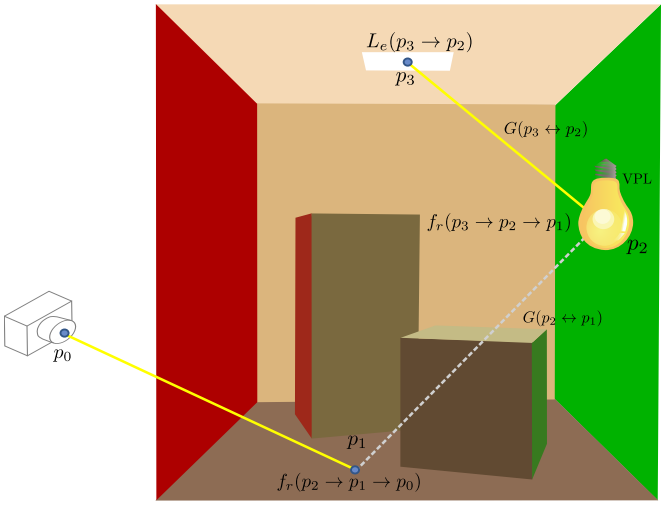


Figure 3.2

The α term in Equation 3.2 is independent of the camera path and the BRDF $f_r(p_3 \to p_2 \to p_1)$ is assumed to be a Lambertian BRDF (see Section 2.5.1), so it is only a constant which means that the VPL is equally bright in all directions [PH10]. This means that only the last two terms $G(p_2 \leftrightarrow p_1)$ and $f_r(p_2 \to p_1 \to p_0)$ need to be

evaluated in the post-processing stage. The BRDF $f_r(p_2 \to p_1 \to p_0)$ does not have to be divided by a probability because the directions from $p_2 \to p_1 \to p_0$ are determined by the VPL and the primary ray. The probability $p_A(p_k)$ in Equation 3.1 is because we are sampling a point on an area light (see Section 2.7.1). When estimating the Equation 3.3 (this is evaluated by instant radiosity), the material M, geometric G and visibility V terms map to the last two terms in Equation 3.2 and each VPL can be mapped to I. The same reasoning applies to Equation 3.4 except that the terms are approximated with the lightcuts algorithm (see Section 3.2). The definition of Equations 3.3 and 3.4 are from the lightcuts paper. Furthermore, we need to divide the radiance by the total number of VPLs because each VPL represents a separate path and is therefore a sample for our estimator (see Equation 2.23). Algorithm 4 shows how we can compute the VPLs in the pre-processing stage.

Algorithm 4 Virtual point lights

for each path **do**
  p = sample point on area light source
  ω = sample new direction uniformly on hemisphere
  ray = trace ray from p in direction ω
  p' = ray.hitpoint
  uniformHemispherePdf = $1/2\pi$
  $pdf_A = \frac{1}{\text{area of light source}} *$ uniformHemispherePdf
  $\alpha = \frac{L_e}{pdf_A} * G(p \leftrightarrow p')$
  while path not terminated **do**
    VPL = $\alpha * \frac{\rho_d}{\pi}$ ▷ deposit VPL at p'
    ω = sample new direction uniformly on hemisphere
    ray = trace ray from p' in direction ω
    p'' = ray.hitpoint
    if first intersection point **then**
      α *= $f_r(p \to p' \to p'')$
    else
      α *= $f_r(p \to p' \to p'') * \cos\theta$
      α /= uniformHemispherePdf
    end if
    $p = p'$
    $p' = p''$
    α /= arbitrary probability
    ξ = sample uniform random variable where $\xi \in [0, 1)$
    if $\xi >$ arbitrary probability **then**
      terminate path
    end if
  end while
end for

3.2 Lightcuts

The performance of instant radiosity is reduced linearly depending on the number of lights, this dependency in lightcuts is sublinear. The direct illumination at a point x,

given a set of point lights S, with the direction ω_o towards the camera and the direction ω_i of the incident light, is the sum of the individual lights $k \in S$ with their material term M multiplied by the geometric G and visibility V terms along with the intensity I. The material term incorporates the cosine term.

$$L_S(x, \omega_o) = \sum_{k \in S} M_k(x, \omega_o, \omega_i) G_k(x) V_k(x) I_k, \tag{3.3}$$

To reduce the above equation to a sublinear method, multiple lights are grouped into a cluster $C \subseteq S$ and the material, geometric and visibility terms of the representative light $j \in C$ are applied to all the lights in the cluster. The direction ω_i is the incident direction of the representative light.

$$\begin{aligned} L_C(x, \omega_o) &= \sum_{k \in C} M_k(x, \omega_o, \omega_i) G_k(x) V_k(x) \sum_{k \in C} I_k \\ &\approx M_j(x, \omega_o, \omega_i) G_j(x) V_j(x) \sum_{k \in C} I_k \end{aligned} \tag{3.4}$$

The term $\sum_{k \in C} I_k$ is the intensity of the entire cluster, this amounts to the evaluation of a single light that is a substitute for multiple lights (see Figure 3.3). Depending on the error, the cluster will most likely need to be partitioned into several smaller clusters, since we start off with a single cluster containing all the lights. For each point x in the scene, the partitioning will need to be recomputed because a different set of lights will be visible to various positions in the scene. This leads to the problem of efficiently determining how to divide the clusters.

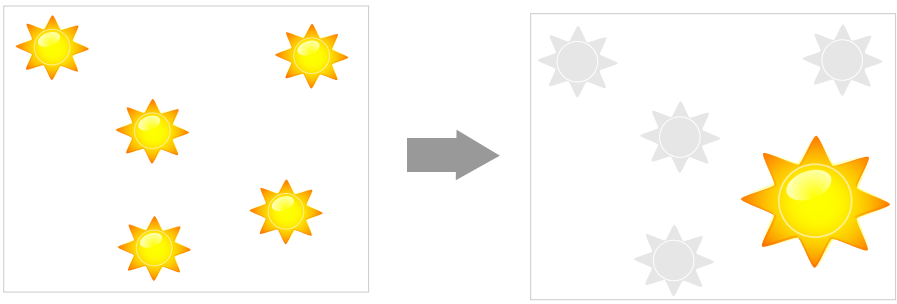


Figure 3.3: Light cluster

3.2.1 Light tree

The job of finding the appropriate clusters is facilitated by a binary tree built once at the pre-processing stage. The lights are inserted into the leaf nodes and the clusters correspond to the interior nodes that contain the contribution or sum of all the lights beneath that node, i.e. the intensity of the cluster and the position of the representative light, as well as the bounding box of the lights. The material, geometric and visibility terms are recomputed for the node for every point x that is rendered. The representative

light is just one of the individual lights, i.e. the VPLs that are stored in the leaf nodes (see Figure 3.4) and chosen from a set of criteria. One way of selecting a representative light is to randomly draw a light from a discrete distribution based on the power of the light sources, which was done in the implementation of this thesis.

For example, if the cluster approximation for the root node were below a defined visibility threshold (see Section 3.2.3), then only a single ray would be traced to the representative light for all the lights in the scene at that specific point. This unfortunately is rarely the case, which is why we need to traverse the tree to find the cut, i.e. the clusters that together are below the threshold. The worst case scenario is the complete failure of the cluster approximation where visibility tests are done for all the lights giving us the exact solution (see Equation 3.3). The lightcuts algorithm must determine if the error is marginal, so that the approximation is as close to the original image as possible. That is to say the estimation should be below a perceptual visibility threshold defined by Weber's Law, which empirically is 2% of the total radiance (see [WFA+05] for details).

In the original lightcuts paper the light tree is constructed with a greedy bottom-up algorithm. The lights are paired together that will create the smallest cluster according to their similarity metric and then paired with the other clusters until no more pairs remain giving us the tree. The performance of the greedy bottom-up algorithm is $\mathcal{O}(n^3)$, so the initial idea was to improve performance with a kd-tree based on the paper by Mikšík [Mik] with a running time of $\mathcal{O}(n \log n)$ [Ben75]. The kd-tree is only used to find the closest clusters but a heap priority queue determines the clusters that are the best matching pairs, which caused the build time to run into minutes for more than several thousand lights. To reduce the construction time from minutes to milliseconds, the decision was to build the light tree based on the thesis from Oliver Taubmann, a brief summary is given here (for further details see [Tau10]). The light tree is constructed with the help of only a kd-tree from the top down. A 3D bounding box is placed around all the lights in the scene and the axis with the greatest length is chosen as the splitting axis. The bounding box is split along this axis giving us two smaller ones. A cluster is then just the sum of the light intensity of the lights in the bounding box, along with the dimensions of the box and a representative light. The measurements of the box are required to compute the bounding errors. The process is repeated until the bounding boxes contain a single light. The tree is left balanced, so that it can be placed into an array without wasting any memory. The benefit of constructing the light tree this way is two fold, top down makes for efficient build times and a balanced tree gives us optimal memory usage because the entire tree can be accessed as an array.

3.2.2 Lightcut selection

All the points in the scene that are rendered will need to choose a lightcut from the global light tree. Every lightcut begins with the root node, if the node or cluster is greater than the predefined visibility threshold, the child nodes are placed onto a priority queue. The node in the queue with the largest error is popped off the queue and its siblings are pushed there instead. This process is repeated until all the nodes combined meet the error criterion. The algorithm begins with a coarse cut, the root node and keeps

progressively refining until a cut is found (see Figure 3.4).

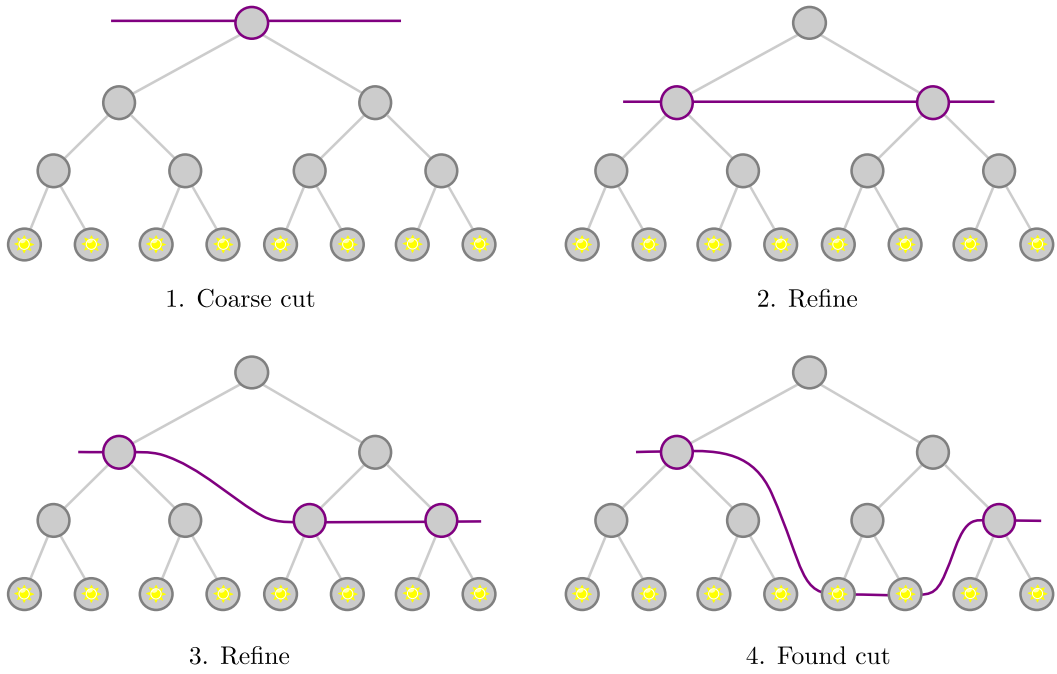


Figure 3.4: Lightcut

Algorithm 5 clarifies the steps precisely. The representative light is shared between the parent and one of its children, so that ray visibility tests can be cut in half when computing the radiance (see lines 17 and 18). For further details of radiance computation see Section 3.2.3. The additional benefit of sharing a representative light is that the material and geometric terms can be reused.

3.2.3 Error bounding

The error caused by the approximation of a cluster comprised of many lights needs to be accounted for. To accomplish this an upper bound is computed for the visibility, geometric and material terms, so that the error is maximized ensuring a close estimation of the orginal image. This will cause Algorithm 5 to run longer to fulfill the condition of being equal to or below the perceptual visibility threshold (see line 10). These bounded terms are multiplied with the cluster intensity to give us the bounded error, which is an estimation of the maximum radiance. Therefore the bounded error is also an upper bound for the exact solution (see Equation 3.3). The most straightforward term to bound is visibility, since the assumption is made that the cluster is visible no matter what. Remember we are only interested in the bounded error as a means of fulfilling the error criterion (see line 10), when computing the radiance at the point x we proceed as we normally would, i.e. we compute equation 3.4 and a visibility test is done for the representative light. The difficulty of bounding the visibility term is why we resort to

Algorithm 5 Finding the lightcut

```
1: function LightCut(x, lightTree, normal, ω_o, material)
2:     k = 0.02                                          ▷ 2% according to Weber's Law
3:     root = lightTree                                  ▷ light tree points to root node
4:     ComputeRadiance(x, root, normal, ω_o, material)   ▷ compute radiance and error bound for node
5:     totalRadiance = root.radiance
6:     heap = priority queue
7:     heap.push(root)                                   ▷ node with greatest error at the top
8:     while heap != empty do
9:         cluster = heap.top()
10:        if cluster.errorBound <= totalRadiance*k then
11:            break
12:        end if
13:        heap.pop()                                    ▷ remove top node from heap
14:        totalRadiance -= cluster.radiance
15:        leftChild = cluster.leftChild
16:        rightChild = cluster.rightChild
17:        ComputeRadiance(x, leftChild, normal, ω_o, material)
18:        ComputeRadiance(x, rightChild, normal, ω_o, material)
19:        totalRadiance += leftChild.radiance
20:        totalRadiance += rightChild.radiance
21:        heap.push(leftChild)
22:        heap.push(rightChild)
23:    end while
24:    return radiance
25: end function
```

the trivial case of setting this value to the upper bound of 1. The geometric term for VPLs is

$$G(x) = \frac{1}{\|y - x\|^2}.$$

The term y is the closest point from the bounding box of the cluster to x, giving us the smallest distance, thus increasing the geometric term. The material term is the BRDF times a cosine factor. The angle of the cosine is the most difficult part to bound, we could set cosine to 1, which would be the upper bound but in order to reduce ray visibility tests we apply a tighter upper bound to this value. What we want to do is find the minimum angle between our point x and the bounding box of a cluster, since the smallest angle gives us the largest cosine value to maximize the error. The cosine of the angle θ between the z-axis and a point p is

$$\cos\theta = \frac{p_z}{\sqrt{p_x^2 + p_y^2 + p_z^2}}.$$

For a bounding box the cosine of θ will be bounded above by:

$$\cos\theta \leq \begin{cases} \frac{max(p_z)}{\sqrt{min(p_x^2)+min(p_y^2)+(max(p_z^2))^2}} & max(p_z) \geq 0 \\ \\ \frac{max(p_z)}{\sqrt{max(p_x^2)+max(p_y^2)+(max(p_z^2))^2}} & \text{otherwise} \end{cases}$$

The sides of the bounding box have length unlike the dimensionless point, which is why the minimum or maximum value along a particular axis of the bounding volume must be chosen. For example, along the x-axis the point p_x may range from -2 to 1, so $max(p_x)$ is 1 and $(max(p_x))^2$ is also 1. When the point is squared $max(p_x^2)$ is 4 and $min(p_x^2)$ is 0.

The simplest case of applying the above formula to compute the bounded cosine term is when the normal of the point x and the z-axis are aligned. The more frequent case involves rotating the z-axis so that it matches the normal of x, this rotation is applied to the bounding box or more specifically the points that mark the corners. The first step is to translate the bounding box to the point x, so that it is the new origin for the sake of the z-axis now simply being the unit vector $(0, 0, 1)$ that passes through the origin. Then the angle between the normal of x and the z-axis is computed. Next we determine the axis through which the rotation is performed. The normal of point x, which should be normalized is projected onto the xy-plane, this simply involves setting the z value of the vector to zero. The axis of rotation is the cross product of the projected normal and z-axis, i.e. the vector that is perpendicular to these two vectors. The rotation matrix given this angle and axis is applied to the bounding box and then the above formula can be used.

The BRDF may contain terms that themselves need to be bounded, if the material is Lambertian then no further bounding has to be computed as this is just a constant. The other material that was implemented in this thesis was the modified Blinn-Phong model that contains a specular term with a cosine factor $\cos\theta = N \cdot H$, where $\cos\theta$ is the angle between the normal and the half-vector (see Section 2.5.2). The way to bound this term is almost exactly as described above except that after we translate the bounding box, the corners c are set to $\frac{\omega_o + c}{\|\omega_o + c\|}$.

The lightcuts paper describes additional light sources, such as oriented lights that need to have their cosine term bounded as well (see [WFA$^+$05] for details). The same principle above is applied yet again but this time when rotating the bounding box the angle of rotation must be adjusted to include the orientation and angle of the bounding cones of the oriented lights (see [WFA$^+$05] for details).

Algorithm 6 computes the radiance for each point x of the current node, the arguments are the direction towards the camera ω_o, the material properties and the normal at point x. Note that we are estimating only the indirect illumination in contrast to Algorithm 3, so the direct illumination can be computed before or after finding the cut. On line 9 the numerator of the geometric term is evaluated (see Section 2.4.1), if the result is negative we know the light from the point x is not visible and therefore the value is set to zero. On line 11 the term is clamped because the value may become infinite

when the distance between the point x and the VPL is very small (see line 10), which is known as weak singularity [KK04] and is especially noticeable at corners as bright spots (see Figure 3.5). Clamping this term removes artifacts, such as bright spots (see Figure 3.5) in an image but adds bias to the final result, meaning that the value of the estimator will not be exactly equal to the integral [DBBS06]. The bias can be compensated for by additional visibility tests (for further details see [PH10]) and as to the value of $\frac{2}{3}$ see [DGS12]. In the lightcuts paper the authors deal with these bright spots by reducing the maximum contribution of each light.

Algorithm 6 Radiance computation at point x for the current node

```
 1: function ComputeRadiance(node, x, normal, ω_o, material)
 2:     intensity = node.intensity
 3:
 4:     materialTerm = GetMaterialTerm(node, x, normal, material, ω_o)
 5:
 6:     vpl = vector from x to representative light
 7:     vplNormal = normal at representative light
 8:     repPoint = node.representative.point                ▷ position of representative light
 9:     geometricTerm = max(0.f, (Dot(vpl, normal)*Dot(vpl, vplNormal)))
10:     geometricTerm /= DistanceSquared(x, repPoint)
11:     geometricTerm = min(2/3, geometricTerm)
12:
13:     visibilityTerm = GetVisibilityTerm(node, x)          ▷ trace ray to representative light
14:
15:     estimatedRadiance = materialTerm * geometricTerm * visibilityTerm * intensity
16:     node.radiance = estimatedRadiance
17:
18:     boundMaterialTerm = GetBoundMaterialTerm(node, x, normal, material, ω_o)
19:     boundGeometricTerm = GetBoundGeometricTerm(x, node)
20:     node.errorBound = boundMaterialTerm * boundGeometricTerm * 1.0 * intensity
21: end function
```

3.3 Reconstruction cuts

The purpose of reconstruction cuts is to reduce shadow rays to the clusters by grouping pixels next to each other that have similar properties, thereby computing the radiance for only a subset of the pixels and interpolating the radiance for the rest of the group. The scene or image is divided into 4x4 blocks of pixels and all points seen through those pixels must share similar features. Three preliminary tests determine if a 4x4 block is eligible for interpolation. The points should all have normals that do not differ by more than 30 degrees, this will ensure that the pixels in the block will be more or less on a level surface. The material of all points in the block should be the same. The cone test makes sure that shadows at boundaries or sharp features are rendered properly, so that points which fall within the cone of another point fail the test (see Figure 3.6). If any of the tests fail then we default to lightcuts for all the pixels.

The four corner pixels of the 4x4 block serve as the samples that are for interpolating

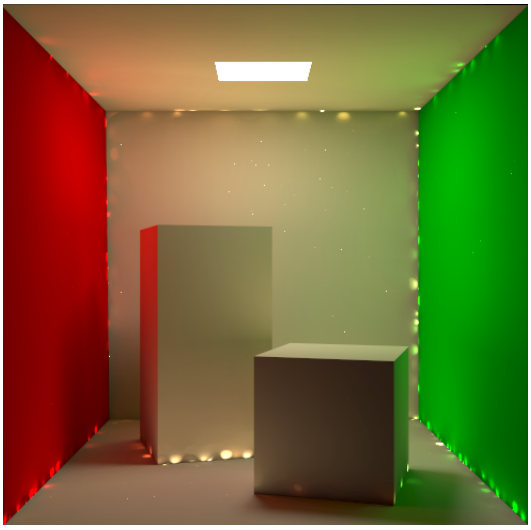


Figure 3.5: Weak singularity

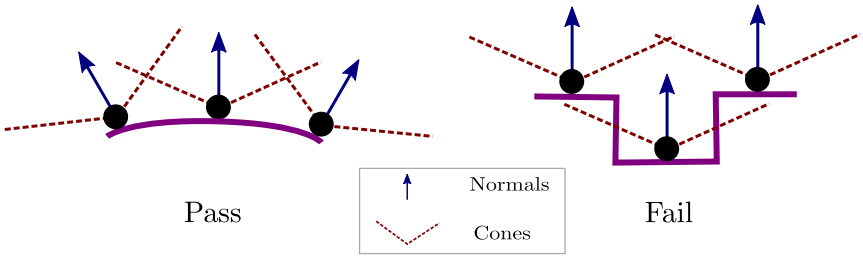


Figure 3.6: Cone test

the remaining pixels. When the lightcut is computed for the sample k, we have the estimated radiance of the node n on the cut defined as L_n^k and then we compute the radiance L_n^k of every node above the cut by adding the estimated radiance L_n^k of the nodes on the cut which are its descendants. The total estimated radiance of the root node is L_T^k, which is the sum of the radiance of all the nodes on the cut. Then we compute the average direction d_n^k, which is the average direction of the descendant nodes on the cut that point to the representative light weighted by their radiance. Then we can compute the intensity and the total intensity for each node above and on the cut for each of the four samples.

$$\gamma_n^k = \frac{L_n^k}{M_n^k(x, w_o, d_n^k)}$$
$$\Gamma_n^k = \frac{L_T^k}{M_n^k(x, w_o, d_n^k)}$$

Now we can determine if it is possible to interpolate the remaining pixels. To compute

the reconstruction cut, the minimum and maximum intensity γ_n^k and the minimum total intensity Γ_n^k over the samples are selected. The threshold τ_n is 2% of the minimum total intensity Γ_n^k. We start from the top of the global light tree and process each node on the heap until it is empty, by doing one of the following steps for each of the remaining pixels:

1. **Discard:** If $max(\gamma_n^k) = 0$ then drop node from heap.
2. **Interpolate:** If $max(\gamma_n^k) - min(\gamma_n^k) < \tau_n$ then compute d_n which is the average of the directions d_n^k weighted by γ_n^k. Compute γ_n with bilinear interpolation using the samples γ_n^k. Multiply γ_n by the material term $M_n(x', w_o, d_n)$ and add this result to the total radiance.
3. **Evaluate Cluster:** If $max(\gamma_n^k) < \tau_n$ or we have a leaf node then compute Algorithm 6 and add this result to the total radiance.
4. **Refine:** Place children of current node on the heap.

We conclude this chapter by having defined the lightcuts and reconstruction cuts algorithms. Before we can parallelize these algorithms we introduce the general concepts of parallelizing a program with the OpenCL framework in Chapter 4.

Chapter 4

OpenCL Framework

In this chapter, we describe how a program can be parallelized with the OpenCL framework and then in Chapter 5 we focus on the parallel implementation of lightcuts and reconstruction cuts. In this section, we explain why we have to parallelize programs in order to increase performance and the problems that usually come with it.

The days of ever increasing clock rates are gone and with it the benefits of accelerating existing applications that are not able to exploit modern day mainstream multi-core systems. The clock frequencies of CPUs can no longer be significantly increased without higher power consumption leading to thermal issues known as the *Power Wall*. Since the frequency speed can no longer be the deciding factor, processor vendors have essentially just shrunk the CPU and put more on a single chip to better manage heat dissipation.

Apart from physical boundaries, multi-core systems require attention to detail because the programmer is forced to think differently about how to solve problems that did not previously exist with single CPU applications. The core problem of programming these systems is sharing and synchronising data.

Ray tracing is an *embarrassingly parallel* problem, so it is very easy to divide the workload into its individual components that can be simultaneously processed by multiple cores, which is to say that we can estimate each pixel independently. Thus the problems associated with parallel programming, such as sharing and synchronising data can be avoided.

4.1 Parallel programming

In order to exploit modern hardware we need to process multiple things at the same time and different paradigms exist to take advantage of this situation. Parallel programming should not be confused with concurrent programming, which does multiple things and possibly at the same time but are unrelated. A web server servicing multiple clients is an example of a concurrent program, the clients are unrelated as they may be accessing different content from other users. In contrast, parallel programs work towards a common goal by increasing throughput to solve a problem.

There are **data** and **task** based parallel programs. For example, we take a set of

numbers that we add to and then multiply by some arbitrary number. In a data parallel based program, each processor would take a chunk of numbers from the set and add to them until all the numbers have been processed. Then each processor would take the results and multiply them to finish the last step. In a task based program, each processor would take an allotted portion of numbers from the set and apply both operations in one step. Any remaining numbers would be processed in the same way. A data parallel program is suited for SIMD (single instruction, multiple data) [Fly72] platforms because a single operation is applied at the same time to as much data as possible, which for the above example is simply the set of numbers. A task parallel program is ideal for MIMD (multiple instruction, multiple data) [Fly72] systems because multiple operations (addition and multiplication in the above example), as well as multiple data are processed at the same time. SIMD may seem inferior to MIMD but less transistors are devoted to cache and control flow, so that it possible for GPU vendors to squeeze thousands of cores on to their systems.

4.2 GPGPU

The graphics pipeline on GPUs is no longer fixed and parts of the pipeline are programmable, which has allowed GPUs to become more versatile as they can also execute computations traditionally meant for the CPU, this is known as GPGPU (general-purpose computing on graphics processing units). GPUs are well suited to solve data parallel problems given that the architecture is based on SIMT (single instruction, multiple thread) [Nvi]. The programming model Single Program Multiple Data (SPMD) allows an instance of the same program to be executed on different portions of data. Loop-sectioning or strip-mining combined with SPMD is a common technique for parallel programming.

Algorithm 7 Potential loop optimization

```
1: for i ← 1 to N do
2:     C[i] ← A[i] + B[i]
3: end for
```

Loops transformed according to this scheme are split into multiple partitions that are executed in parallel, it is preferable to have no data dependencies between iterations. The pseudo code in Algorithm 7 is an example of a loop that could be effectively optimized for SPMD and strip-mining because line 2 can be run in parallel on as many cores as possible (see Figure 4.1) and note that there are no data dependencies between the iterations of the loop. The code in Algorithm 7 is very similar to the code we will parallelize in Section 4.4. In this thesis we apply a data based parallel approach to our problem and in a similar fashion, just as every iteration on line 2 in Algorithm 7 can be executed in parallel, we process each pixel of our image in parallel on as many cores as possible. However, this does not limit our programs to GPUs, it just means that we are able to take advantage of them.

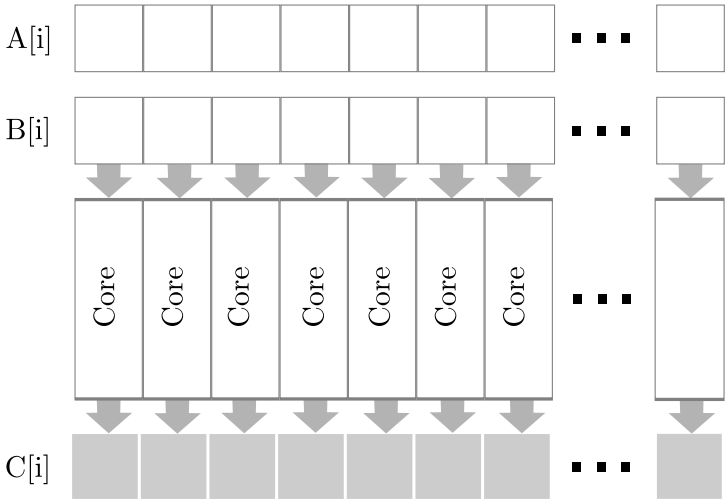


Figure 4.1: Strip mining

4.3 OpenCL

OpenCL provides a standardized API framework that helps developers write data or task based parallel software for heterogeneous systems and is portable across operating systems, such as Windows and Linux. OpenCL provides a C type language, compiler and the runtime environment to execute the C like code on OpenCL capable devices, referred to simply as the compute device or just device. The major limitations of this extension to the C language is that many standard library functions are not available and recursive functions are not permitted because of hardware limitations on some platforms particularly the GPU, other platforms do not have this restriction but for the sake of portability it is limited on all hardware. This restriction may be removed in future because newer GPUs do in fact support recursion. There are two components to an OpenCL program, the kernel code that runs on a device and the host program that will usually run on the CPU to bootstrap the kernel code so that it can be executed. The host can control multiple devices and can itself be a device. The host program is also responsible for dispatching commands, fetching and pushing data to the devices that execute the kernels. The compute unit in OpenCL is composed of one or more processing elements (PEs) and a device can have multiple compute units. The compute units determine how to run the programs on the PEs under their control. The kernel is basically a C like function that is executed on a single compute unit and is run as a copy on each PE as a single thread.

An important concept in OpenCL is work-items, which are the threads that are executed on the PEs. Work-items are part of a work-group executing on a compute unit. The compute unit can manage one or more work-groups. Work-items within a work-group share local memory and they execute the same kernel. For example, we could see a work-item as computing a single iteration of the loop in line 2 of Algorithm 7 or as a single column in Figure 4.1.

4.4 OpenCL example program

In order to get a kernel to run on a device some scaffolding is required to setup the code to be executed. The complete code example below shows the general steps that are necessary for every OpenCL program. The first step is to get the available platforms; every vendor provides a single platform as of this writing. The OpenCL function *clGetPlatformIDs* is usually called twice, once to retrieve the number of platforms and then on the second call we retrieve the platforms as a data structure containing the necessary information that we would like to query (see lines 25 and 28). Should the value returned be unequal to the OpenCL defined macro *CL_SUCCESS* which is simply 0, then an error has occurred. Error handling has been removed from the code to keep it to a minimum.

Depending on the implementation, calling this function is not mandatory, we could go straight to getting a specified device by calling *clGetDeviceIDs*. On line 31 we call *clGetDeviceIDs* to retrieve the number of devices provided by supplying the first available platform as an argument, the function is called again on line 36 to fill the data structure *cl_device_id*. The next step is to create a context with which we manage device resources, such as kernels, memory and command queues. The lines 40-44 attach all the devices of the platform that we supplied as part of an argument to a context. Next we call *clCreateCommandQueue* by supplying a context and a device to which we wish to send commands, in this case the *device[2]* happens to be a GPU and may differ on your platform (see lines 46 and 47).

Lines 49 to 50 creates a program object with the context and source string which is shown in lines 4 to 11, then on line 52 the program is compiled. The source string (lines 4 to 11) may be hard to read but this is to keep the length of the example program to a minimum, otherwise we would require additional code to load a more human readable format from a separate file. Lines 54 to 60 generate random numbers as test input for our device code. Line 62 creates the kernel object, the name of the kernel must be given, in this case it is *vector_add* and note the key word prefix of *__kernel* in the source string. The kernel is the code a device will begin to execute and it can call other functions defined in the source. The lines 64 to 73 create memory objects that will copy our test data to the device when we begin queueing commands. The line 74 creates another memory object to store the results of our kernel. The lines 78 to 80 set the arguments for our kernel which requires three arrays or pointers to arrays. Note the prefix of the arguments in the kernel source code __global, this indicates the type of memory region. Global refers to the memory that all work-groups can access and local, i.e. prefixed with __local refers to memory accessible only to work-items within a work-group. Local memory is preferable because of speed but much less of it is available. The built-in function in the kernel code *get_global_id(0)* identifies each thread or work-item uniquely on the device and each thread will execute the kernel adding the values associated with its thread number to the output array (see lines 9-10 and Figure 4.1). After all the threads or work-items are finished, they will be given new thread identifiers to process the next set of values until the range given by the value global_work_size passed to the function *clEnqueueNDRangeKernel* is finished. To uniquely identify threads in multi-

dimensional arrays, such as images we could define *get_global_id(0)* as the column and *get_global_id(1)* as the row.

Finally, line 83 will begin executing our kernel which only adds the floats from two arrays together and places the result in another array. The line 88 reads the results from the memory object *output* into an array on the host. The lines 92 to 101 just perform cleanup and deallocation of memory.

Listing 4.1 C++ OpenCL example program

```
1  #include <iostream>
2  #include <OpenCL/opencl.h>
3
4  const char *source =
5  "__kernel void vector_add (__global const float *inputA,\n" \
6  "                          __global const float *inputB,\n" \
7  "                          __global float *output)\n"\
8  "{\n"\
9  "   int gid = get_global_id(0);\n" \
10 "   output[gid] = inputA[gid] + inputB[gid];\n" \
11 "};";
12
13 const uint DATA_SIZE = 65536;
14
15 int main(int argc, const char * argv[]) {
16
17     cl_context context;
18     cl_command_queue commands;
19     cl_program program;
20     cl_kernel kernel;
21
22     cl_int error;
23     cl_uint nplatforms;
24
25     error = clGetPlatformIDs(0, NULL, &nplatforms);
26
27     cl_platform_id* platforms = new cl_platform_id[nplatforms];
28     error = clGetPlatformIDs(nplatforms, platforms, NULL);
29
30     cl_uint ndevices;
31     error = clGetDeviceIDs(platforms[0],
32                            CL_DEVICE_TYPE_ALL, 1,
33                            NULL, &ndevices);
34
35     cl_device_id *devices = new cl_device_id[ndevices];
36     error = clGetDeviceIDs(platforms[0],
37                            CL_DEVICE_TYPE_ALL,
38                            ndevices, devices, NULL);
39
40     cl_context_properties
41     properties[] = {CL_CONTEXT_PLATFORM,
42                     (cl_context_properties)platforms[0],0};
43     context = clCreateContext(properties, ndevices, devices,
44                               NULL, NULL, &error);
```

```
45
46      commands = clCreateCommandQueue(context, devices[2],
47                                      0, &error);
48
49      program = clCreateProgramWithSource(context, 1, &source,
50                                          NULL, &error);
51
52      error = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
53
54      float dataA[DATA_SIZE];
55      float dataB[DATA_SIZE];
56      srand(time(NULL));
57      for(int i=0; i < DATA_SIZE; i++) {
58          dataA[i] = rand() / (float)RAND_MAX;
59          dataB[i] = rand() / (float)RAND_MAX;
60      }
61
62      kernel = clCreateKernel(program, "vector_add", NULL);
63
64      cl_mem inputA, inputB, output;
65      inputA = clCreateBuffer(context,
66                              CL_MEM_READ_ONLY |
67                              CL_MEM_COPY_HOST_PTR,
68                              sizeof(cl_float) * DATA_SIZE,
69                              dataA, NULL);
70      inputB = clCreateBuffer(context,CL_MEM_READ_ONLY |
71                              CL_MEM_COPY_HOST_PTR,
72                              sizeof(cl_float) * DATA_SIZE,
73                              dataB, NULL);
74      output = clCreateBuffer(context,CL_MEM_WRITE_ONLY,
75                              sizeof(cl_float)*DATA_SIZE,
76                              NULL, NULL);
77
78      error  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &inputA);
79      error |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &inputB);
80      error |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &output);
81
82      size_t global_work_size = DATA_SIZE;
83      error = clEnqueueNDRangeKernel(commands, kernel, 1, NULL,
84                                     &global_work_size, NULL, 0,
85                                     NULL, NULL);
86
87      float results[DATA_SIZE];
88      error = clEnqueueReadBuffer(commands, output, CL_TRUE, 0,
89                                  DATA_SIZE * sizeof(cl_float),
90                                  results, 0, NULL, NULL);
91
92      clReleaseMemObject(inputA);
93      clReleaseMemObject(inputB);
94      clReleaseMemObject(output);
95      clReleaseProgram(program);
96      clReleaseKernel(kernel);
97      clReleaseCommandQueue(commands);
```

```
98      clReleaseContext(context);
99
100     delete [] platforms;
101     delete [] devices;
102
103     return 0;
104 }
```

To summarize, in this chapter we have shown how a data parallel program can be written, so as to offload computations to an OpenCL capable device.

The implementation in this thesis follows the same principles as in Listing 4.1 but now we pass an array that is the size of the image we wish to render, along with the necessary data to a device that executes our kernel. Each work-item is mapped to a single pixel that traces a ray from the camera into the scene and then computes the Equation 3.4 for a point. A work-item computes a single sample or the radiance at a point. The radiance values of all the pixels are returned as the output to the host system and then for each sample we estimate Equation 2.16. We normally require multiple samples for a pixel to render pleasing images, so the process is repeated for each pixel in the image.

Chapter 5

Implementation

The parallel OpenCL version of lightcuts and reconstruction cuts in this thesis is based on the LuxRays system, which is the part of the popular open source software LuxRender dedicated to accelerating ray intersection on GPUs. LuxRender is itself based on the acclaimed Physically Based Rendering software (see [PH10] for further details). The code used from LuxRays in this thesis is to help bootstrap the rendering process, so that we do not have to concern ourselves with how to load the data to an OpenCL device. The lightcuts and reconstruction cuts algorithm to render a scene is then written from scratch as an OpenCL kernel that estimates Equation 3.4. The BVH (see Section 2.1.1) acceleration structure and intersection code from LuxRays was integrated into our parallel kernel implementations. The code is dependent on only a few 3rd party software libraries: Boost, FreeImage and tinyobjloader. FreeImage is for the purpose of saving the rendered scenes to an image format on disk and tinyobjloader is for loading the scene file and its material properties. The source code of tinyobjloader has been included in this project, since it is only two files and one less dependency to worry about. The scene files supported are in the Wavefront file format and have the extension .obj that contain the geometry of the scene and references to a separate .mtl file describing the material properties. The right-handed coordinate system is used with the z-axis in the up direction and y-axis facing forward, this helps with orientation when loading and modifying scenes in a program such as Blender and then later rendering it with the software in this thesis. The OpenGL and GLUT libraries are only necessary when we want to view the rendering process in real-time, otherwise we can resort to the option of saving the image to disk when the program has finished. OpenCL is of course the last dependency, on Mac OS X it is already a part of the system.

Before proceeding to write the data parallel implementations, stock C++ applications were first developed to test the code, so without hardware acceleration before porting the algorithms to OpenCL. Again, for the C++ versions, LuxRays was used to load the data and find intersections, the algorithms to compute the rendering equations were written from scratch. This was to help alleviate the shortcomings of the OpenCL debugging facilities provided on Mac OS X. The only way to debug the kernels were with *printf* statements. Debugging the stock versions by stepping through the program was vital

to verifying the correct functioning of the implemented algorithms, so as to simplify porting the code to OpenCL. Development began by estimating the exact solution (see Equation 3.3) by tracing a ray to every VPL, this is known as IGI (Instant Global Illumination) [WKB+02] and is a much simpler method than computing Equation 3.4 (i.e. lightcuts). Instant radiosity also computes the exact solution and is equivalent to IGI but the visibility of the VPLs are determined with shadow maps [Wil78] instead of tracing rays.

The best way to debugging OpenCL programs in my opinion, is probably with Windows and using hardware from either Intel or AMD because they provide plugins for Visual Studio. Nvidia provides the Nsight plugin for Visual Studio which can only profile OpenCL code, debugging can be done for CUDA which is very similar to OpenCL but proprietary and limited to Nvidia hardware. To maintain backwards compatibility with all vendors and devices the code was implemented in OpenCL 1.0.

5.1 Software architecture

LuxRays has a large code base containing plenty of functionality that was not required for this thesis, therefore the code that was not used was removed. However, the design of the system remains the same (see Figure 5.1). The system is very light weight and as such the whole rendering process is much easier to grasp.

We now describe how the rendering software in this thesis works (see Figure 5.1). The main entry point of the program begins by reading the path to a config file named render.cfg that is passed on the command line and contains information of which renderer and scene files to load. Next, a RenderSession object is created which holds the path to the config file. If no config file is supplied a default option is automatically provided, which will cause the Cornell box to be rendered. If the preview window is enabled by compiling with OpenGL support, then the RenderSession object is passed to OpenGL for the dimensions of the window and to retrieve the updated pixels from the renderer. The RenderSession is then initialized by querying the OpenCL platform and creating the Scene object that loads the scene into memory. The options from the config file are then read, such as the image dimensions, OpenCL devices and the position of the camera in the scene. The devices are then mapped to the OpenCL context (see Section 4.4). The RenderSession creates the OpenCLIntersectionDevice object that takes the Scene object and loads the vertices and BVH data structure into OpenCL buffers. The kernel code to trace rays and eliminate intersection tests with the BVH structure were originally managed by OpenCLIntersectionDevice, so it compiled and set the kernel arguments. The renderer would then interface to this code and run the kernel. The reason for separating the ray tracing code in another kernel is the ability to profile the efficiency and simplify debugging. The IGI OpenCL version consists of two kernels, the first kernel traces a block of rays and the results of whether the rays intersected something are passed to the integrator kernel that ultimately computes the rendering equation. For lightcuts and reconstruction cuts we only fetch the buffers from the OpenCLIntersectionDevice and pass these as arguments to the integrator kernels that include the intersection code

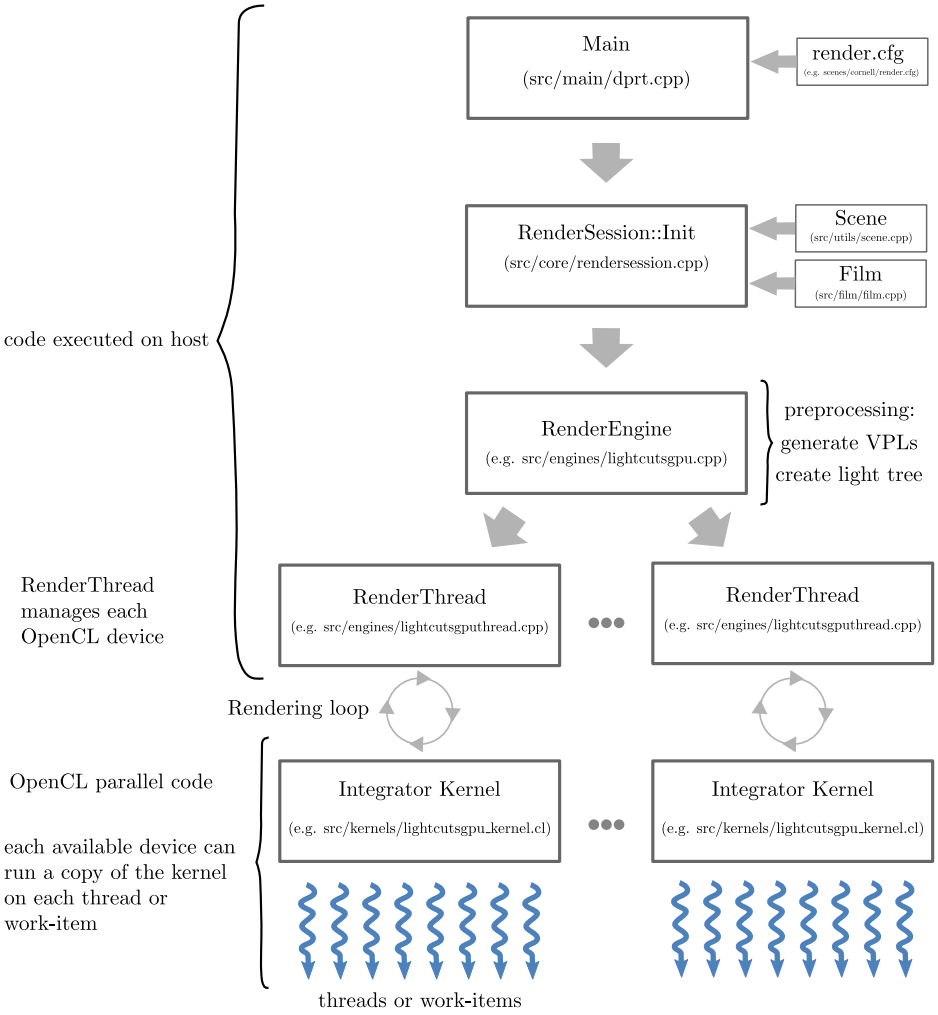


Figure 5.1: Architecture overview

directly. The reason for this is to keep all the threads on the OpenCL device occupied because these algorithms do not require all the rays that are passed in as a buffer to the intersection kernel to be traced, thereby causing many threads to sit by idly.

The type of RenderEngine object created is specified by a config option, this could be any light transport algorithm that has been implemented. The Scene, Device, Film and film mutex objects are passed to the RenderEngine. Depending on the type of RenderEngine, the VPLs and light tree will be generated at this stage. The RenderEngine then creates a *task based thread* on the **host system** for each device (see Figure 5.1) we want to manage and run the kernel on. For example, if we have two GPUs each GPU would be managed by a separate task based thread and then execute its own copy of the kernel on as many threads or work-items as possible and return the results, in this case each GPU would process one half of the image. The Film object holds the image

buffer and the mutex makes sure that only one task based thread at a time can write to this buffer. The program creates at least two task based threads, the main thread determines when the image buffer has been changed and updates the pixels of the rendered image, while all the other threads manage the devices. The RenderSession then finishes by prompting the RenderEngine to start the task based threads. Developing a light transport algorithm requires the RenderEngine and RenderThread objects to be implemented.

5.1.1 The RenderThread

The RenderThread implements the thread function that is started by the RenderEngine and then manages and calls the kernels in a loop (see Figure 5.1) until the work is done. The kernels have to execute as quickly as possible because on Windows, for example, the GPU is reset if it does not respond within a certain amount of time, thereby prematurely preempting the kernel. This behaviour can be avoided by adjusting the TDR (Timeout Detection and Recovery) delay but kernels should complete their work quickly so the OS remains responsive. The main thread then opens up an OpenGL window or saves the image to a file depending on the runtime options and then runs indefinitely by polling the Film object and updating the pixels. Most of the work for a new light transport algorithm is contained in the RenderThread and kernel code, the RenderEngine can be copied from other algorithms with almost no change. The RenderSession source needs to be updated to include the new algorithm. The RenderThread creates the OpenCL objects that hold the data required by the kernels that are compiled and then executed. The first kernel to be run by the RenderThread is the InitFrameBuffer, which is passed an array called the FrameBuffer containing the RGB colors for each pixel. The kernel just sets the FrameBuffer to zero, so the image is initially black. The FrameBuffer will eventually hold the rendered image which is copied to the host, so that the main thread can then write the pixels to the Film object, which applies a filter (see Equation 2.16) and then the image is finally output to a window or file. The Init kernel runs after the InitFrameBuffer and is passed an array with data structures of type Ray (see Equation 2.1), so it fills the buffer with the direction for the rays that are going to be traced from the camera by the integrator kernel.

5.1.2 Integrator kernel buffers

The integrator kernel is where most of the action happens, this is where the rendered image is written to the FrameBuffer, all the infrastructure that has previously been setup is only for the purpose of being able to execute this code. The RenderThread runs this kernel repeatedly until the image has been rendered, and the FrameBuffer is periodically read from the device to be copied to the Film object. InitFrameBuffer and Init kernels are only run once before the above loop (see Figure 5.1), so their runtime is negligible. The integrator kernel is a state machine, so every thread or work-item run on the device is responsible for tracing and updating the state of a single ray, where each work-item is mapped to a pixel. The state machine allows the work-items to continue rendering

the scene each time the integrator kernel is called in the loop. There are three data structures (C Struct) that maintain the state machine, below we list their names and the most important data they hold:

1. **Ray:** The origin and direction of the ray.
2. **RayHit:** The coordinates of the ray intersection.
3. **Path:** The current state, radiance and pixel index into the framebuffer.

A 1D array for each of the three data types is passed to the integrator, so that each work-item is responsible for a single element in the three arrays (see Figure 4.1). The data passed to the kernels are written to global memory, this is essential for the state machine to operate because the data in local memory would be wiped out on reentry of the kernel, however, this does not mean we don't make use of local memory at all. The common data or OpenCL buffers that are shared between different types of integrators are listed below along with their main purpose, they all are pointers to 1D arrays and the data is made primarily up of floats, integers and the float3 and float4 vector data types, these are just 3 or 4 floats grouped together respectively and are more efficient on SIMT platforms than when working with floats individually:

- **State Machine:** Ray, RayHit, Path
- **Scene Data:** Materials, Vertices, VertexColors, VertexNormals, Triangles, TriangleLights (area lights), BVH
- **Framebuffer:** Pixels

5.2 Lightcuts kernel

The lightcuts integrator kernel requires additional buffers for the light tree and this is the only kernel called by the RenderThread in the rendering loop (see Figure 5.1). Again, these are three 1D arrays, with every work-item requiring a heap for the lightcut and an index into the heap, the last array is the light tree, which is read-only like the scene data buffers. Now that we have dealt with all the data, we discuss the implementation of the lightcuts kernel.

Algorithm 8 closely models the essential parts of the OpenCL code, its main purpose is to show the implementation of the state machine. Each work-item retrieves its unique global identifier amongst all the work-items. Every work-item then fetches the data from the arrays indexed by its global work-item ID from the Ray, RayHit, Path, Heap and HeapIndex buffers. The data for a ray is loaded into local memory, since this is modified frequently and when the kernel exits we write the data back to the buffers in global memory. A work-item will have one of four states every time the kernel is executed, so either on reentry of the kernel the work-item will have a new state or continue in the current state (see Figure 5.2). The integrator begins with every work-item in the

initial Camera start state, which is set in the Init kernel. The Camera start state calls the intersection function to trace rays from the camera into the scene and the state is changed. The work-item returns and upon reentry the next state Direct Light computes

Algorithm 8 Pseudocode: lightcuts kernel

```
function LightcutsKernel(paths, rays, raysHit, heaps, heapIndices, lightTree)
    gid = get_global_id(0)                                   ▷ identify each work-item
    ray = rays[gid]
    rayHit = raysHit[gid]
    path = paths[gid]
    heap = heaps[gid]
    id = heapIndices[gid]
    if path.state = STATE_CAMERA_RAY then
        traceRayFromCamera(ray, rayHit)
        path.state = STATE_DIRECT_LIGHT
        return
    end if
    if path.state = STATE_DIRECT_LIGHT then
        if rayHit = hit something then
            computeDirectLight
            path.state = STATE_ROOT_LIGHT
        else
            path.state = STATE_CAMERA_RAY
        end if
        return
    end if
    if path.state = STATE_ROOT_NODE then
        node = lightTree[0]                                  ▷ root node indexed by 0
        computeRadiance(node, rayHit, normal, material, w_o)
        push(heap, id, node)
        path.state = STATE_LIGHT_CUTS
        return
    end if
    if path.state = STATE_LIGHT_CUTS then
        while heap not empty do
            if errorBound <= totalRadiance * 2% then
                path.state = STATE_CAMERA_RAY
                return
            end if
            compute radiance for left and right node and push to heap
        end while
        path.state = STATE_CAMERA_RAY
        return
    end if
end function
```

the radiance from the visible light sources in the scene at the intersection point. If a ray does not intersect a point in the scene then a new direction from the camera is setup, so that the next time the kernel is called the work-item will begin in the Camera state. All other rays that intersect a point will proceed to the Root Node state which computes the radiance of the root node of the light tree (see Section 3.2.2) and places it on the

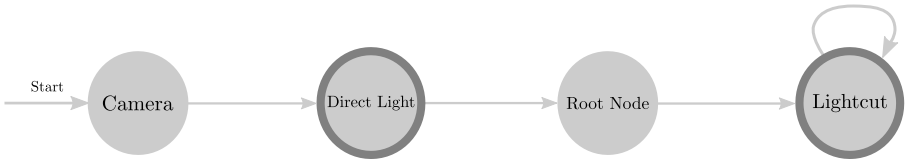


Figure 5.2: Lightcuts state machine

heap. The Lightcut state then finds the lightcut below the threshold (see Algorithm 5 from lines 8 to 23). After this, a new ray from the camera is traced to repeat the process. When the integrator is run on a CPU, the loop (see Algorithm 5 from lines 8 to 23) can be run until the lightcut is found, however, on the GPU the loop must occasionally be preempted for the OS to remain responsive, this is why the state machine (see Figure 5.2) in the Lightcut state has an arrow pointing to itself.

5.3 Reconstruction cuts kernel

The reconstruction cuts kernel follows the same pattern as in Algorithm 8, except that we test for additional states modeled by Figure 5.3. What makes this algorithm more difficult is that we need to synchronize between work-items that process a 4x4 block of the image, so we have 16 work-items for each block (see Section 3.3). The RecontructionCuts integrator requires two additional parameters to those that are passed to the Lightcuts kernel. The first parameter is an array that provides space for every work-item that processes a sample, i.e. the corner pixels of the 4x4 block, to store the intensity γ_n^k, the total intensity Γ_n^k which can be stored as a float and the direction d_n^k of every node on and above the cut (see Section 3.3). The other parameter is an array of booleans shared by the work-items of a block to determine whether they can interpolate the radiance or if the standard lightcuts method should be applied. Similar to the Lightcuts kernel, all the work-items enter the Camera state, the difference is that the next state is set to Cuts Method (see Figure 5.3). In the Cuts Method state we test the rays to see if the surfaces they hit have normals that do not differ by more than 30°, if they share the same material type and if they pass the cone test. When these conditions are met the computation for the block in question can proceed with reconstruction cuts. Failing any of these tests simply means we default to the lightcuts method for every work-item in the block. In the Direct Light state, if a ray does not intersect anything in the scene, then the work-item goes to the Wait state until all the other work-items of a block reach this state. The work-item sits by idly until all the work-items converge to the same state in order to process the image block by block. If the standard lightcuts method is applied, then all the work-items in the block that reach the Root Node state will proceed to the Lightcut state. From there they will go to the Wait state until all the work-items are in the same state, in which case they can continue to the Terminate state and then to the Camera start state to repeat the process. If the reconstruction cuts method is used, the work-items in the Root Node state that are not a sample work-item go to

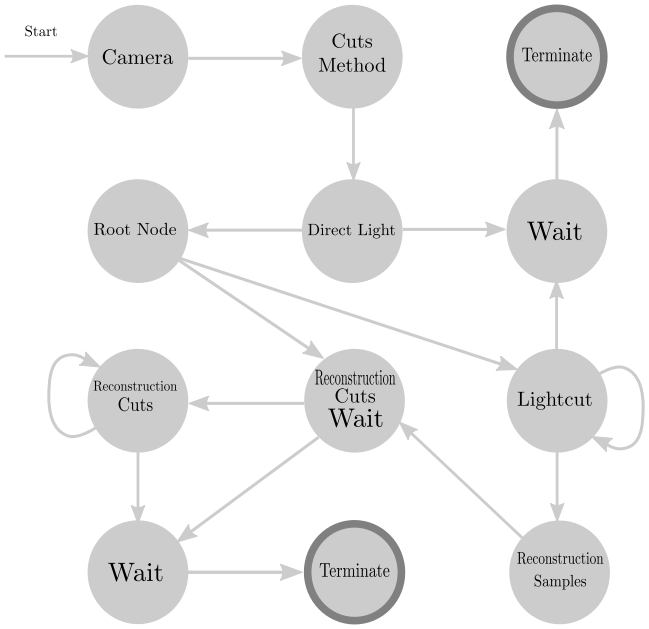


Figure 5.3: Reconstruction cuts state machine

the Reconstruction Cuts Wait state where they wait for the samples to be computed. The sample work-items go to the Lightcut state and from there to the Reconstruction Samples state. When the sample work-items reach the Reconstruction Cuts Wait state they trigger the waiting work-items in the block to move to the Reconstruction Cuts state where they use the samples to interpolate the radiance, while the sample work-items go to the Wait state. Finally, all the work-items within a block will reach the Wait state where the first work-item makes sure they all are in the same state. They then proceed to the Terminate state where the pixels of the image are updated and then they go back to the Camera state.

In OpenCL the *barrier* function synchronizes work-items within a work-group. Every work-item in the work-group has to reach the barrier function in order for the work-items to continue, for example, if the barrier function can only be called when a certain condition is true then other work-items may not be able to reach that point causing undefined behaviour. Barriers for our intents and purposes are useless for synchronization because some work-items in the work-group may have to voluntarily preempt the kernel for the host system to remain responsive and will never reach the barrier while other work-items would have, this situation causes the kernel to crash. The benefit of the state machine is that we do not have to wait for all the work-items being executed in a work-group, as we would have with a barrier, since work-items within a block can proceed to the next state. Therefore, we have more granular control, as the synchronization primitives available in OpenCL do not suffice.

Chapter 6

Results and Discussion

In this chapter, we demonstrate the results of the parallel implementation of lightcuts and reconstruction cuts, by applying the algorithms to three test sets with three scenes each, while recording the time it takes to render an image. We end the chapter with a discussion by drawing conclusions in reference to these tests. There is only one parameter for each test set that is adjusted to see the effect it has on time, for the first test we vary the number of lights, the second test changes the perceptual visibility threshold and the third test alters the resolution of the image. The format of the tests is as follows. For every test set we briefly explain why the test was conducted. All test sets are accompanied with two tables where the first table shows the render time for each of the scenes and the algorithms with the parameter settings. The second table depicts the per pixel averages of the cuts size and shadow rays. The second table is an extension of the first, where the entry of the scene corresponds to the first table. The shadow rays represent the rays that are traced to both the indirect and direct light sources. The first two tests show images with the cut size and a bar beneath to determine the size. The cut size is per pixel with red areas showing the maximum and blue the minimum cut size.

The first test displays graphs that show the render time with a varying number of VPLs. The first test also includes the naive approach or the IGI method, where shadow rays test every VPL for occlusion. The IGI algorithm has also been parallelized with OpenCL so that an accurate comparison can be made with the other two algorithms that approximate the VPLs as clusters. We then show the rendered scenes for each of the three algorithms with 20000 VPLs, the cut size and error images. The error images reveal differences between IGI and the other two algorithms. The first test set is concluded with the tables and histograms summarizing the results with 20000 VPLs.

For the two remaining tests we place between 100000 and 200000 VPLs in the scenes. The time to build the light tree is negligible, since constructing a tree with 650000 VPLs for the Sponza scene takes 1869ms and is done once compared to 46s for the same tree size in the lightcuts paper (see Figures 9 in [WFA+05]). The last two tests do not include IGI because we cannot adjust the threshold or reduce the shadow rays at higher resolutions, since every VPL is accounted for and not approximated. Even at a higher

threshold, lightcuts shows no visible differences when compared to IGI. However, visible differences for reconstruction cuts can already be seen at 2% in the first test.

The second test adjusts the threshold for four settings $1\%, 2\%, 5\%$ and 7%. We show the rendered scenes for the most interesting results and conclude with the tables and histograms showing all of the settings. The final test just shows the two tables and histograms for all the different resolutions.

6.1 Results

Timing is for a dual Intel Xeon E5-2690 v2 each with 10 cores for a total of 20 cores. The scenes were rendered at a resolution of 1024x768. The Cornell box and Sibenik scenes required a 100 samples per pixel (spp) with jittered sampling, as undersampling with this technique produces noisy or grainy images. The Sponza scene required 500 spp for lightcuts and IGI to produce visually pleasing images. Resonstruction cuts was able to produce higher quality images, i.e. less noise with just 200 spp for the Sponza scene. The lower sampling rate for reconstruction cuts is because we render 4x4 blocks of pixels instead of individual pixels. The lightcuts paper was able to produce higher quality images with even fewer samples for reconstruction cuts because the results are adaptively anti-aliased [PS89]. The timing for our results is for 1 sample per pixel at a resolution of 1280x960 except where otherwise stated. The results in the lightcuts paper for lightcuts is also for 1 spp at a resolution of 640x480 and can be compared to the results in the final test in this section with the same resolution. The reconstruction cuts results in the original paper are for more samples but we can still make an estimate by converting the time to 1 spp.

For example, in the Grand Central model, reconstruction cuts traces an average of 318 shadow rays per pixel at 1280x960 compared to 475 for the lightcuts implementaion at 640x480 with a time of 407s at 1 spp (see Figures 9 and 13 in [WFA+05]). The time for the Grand Central scene with lightcuts is scaled to a resolution of 1280x960 by multiplying 407s by 4 giving us 1628s, the actual time would be slightly less because of the higher average shadow rays per pixel at 640x480 than with higher resolutions (see Table 6.3a and 6.3). Reconstruction cuts can trace 1.49 ($\frac{475}{318} = 1.49$) times as many shadows rays per pixel as the standard lightcuts method in the same amount of time for this example, so for 1280x960 reconstruction cuts will require approximately 1093s ($\frac{1628s}{1.49} = 1093s$) to render the Grand Central scene with 1 spp in the lightcuts paper.

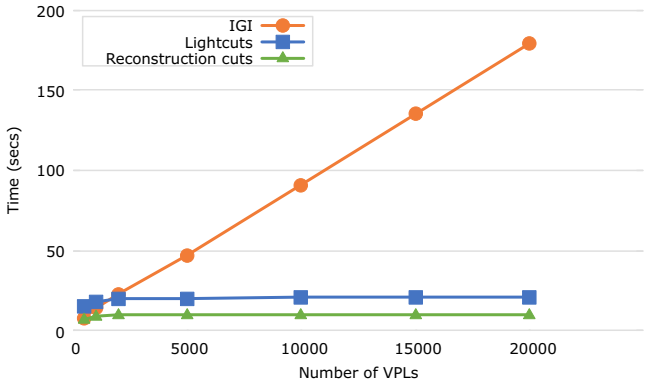
The maximum cut size is set to 1000 with a 2% error ratio for all the images except where otherwise stated, so we stick to the settings in the orginal lightcuts paper.

6.1.1 Varying number of VPLs

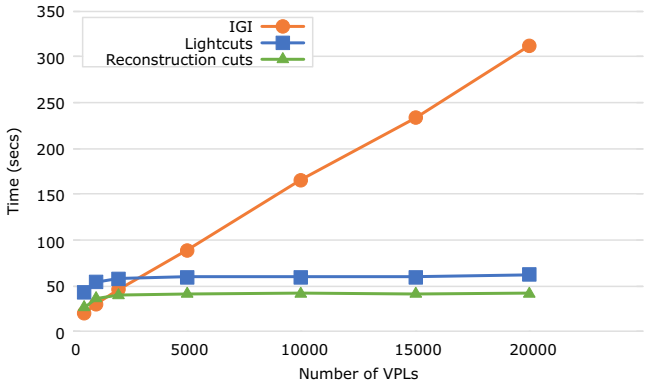
The purpose of the results in this set is to show that the lightcuts and reconstruction cuts algorithm scales sublinearly with increasing number of lights, whereas IGI increases linearly.

The graphs below are for the time it takes to render each pixel once (1 spp) at

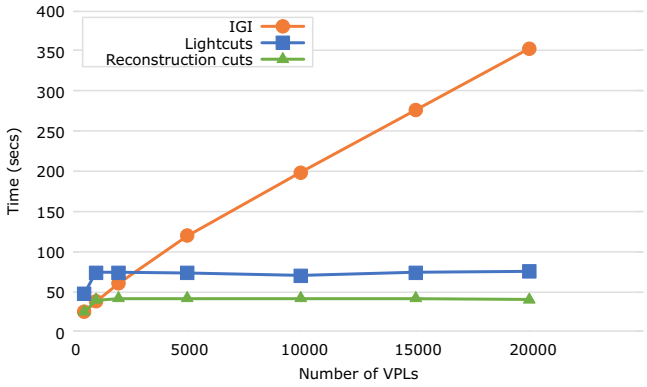
1280x960. The error images (d) in the figures are between IGI and lightcuts, while (f) is for IGI and reconstruction cuts. The error images for figure (d) have to be magnified by 4 in order to show any differences between IGI and lightcuts, whereas reconstruction cuts shows visible differences without magnification.



(a) Cornell

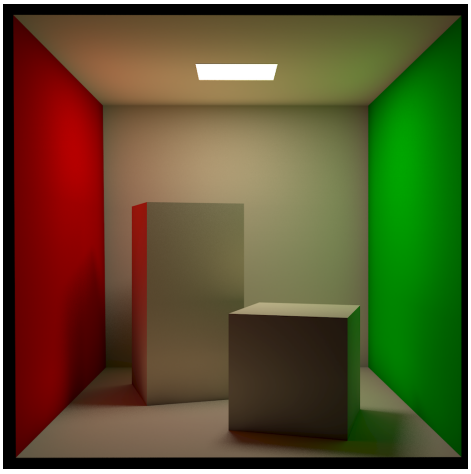


(b) Sibenik

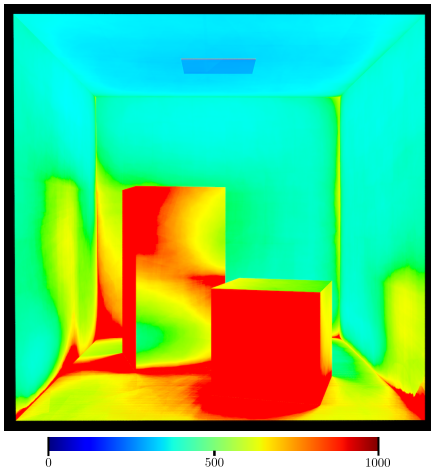


(c) Sponza

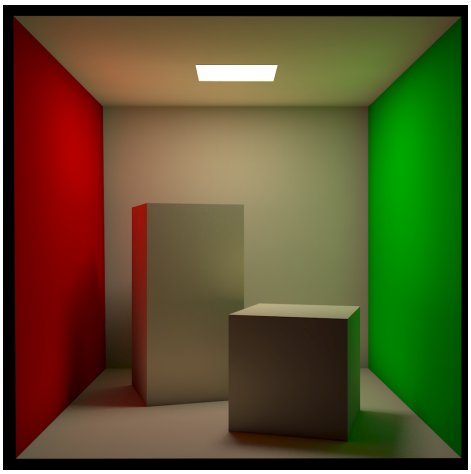
Figure 6.1: Increasing number of VPLs



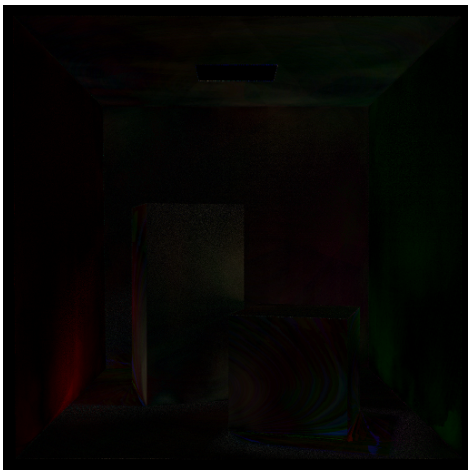
(a) IGI



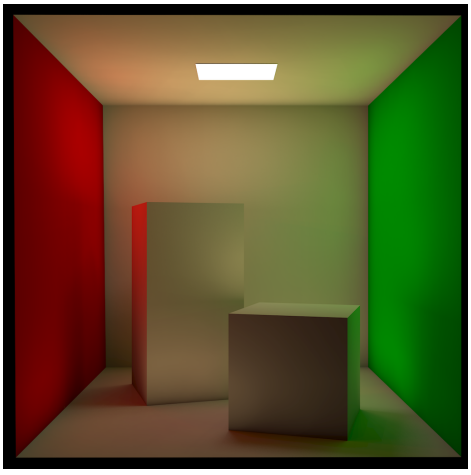
(b) Cut size



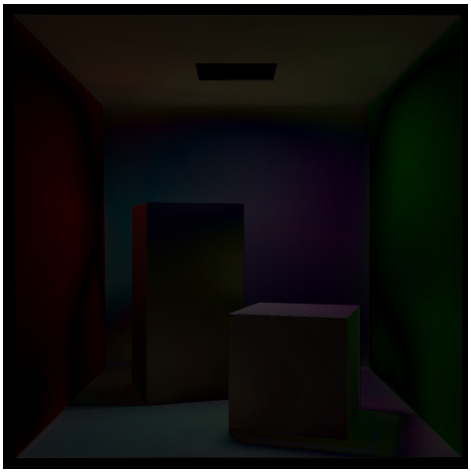
(c) Lightcuts



(d) 4 x Error Image

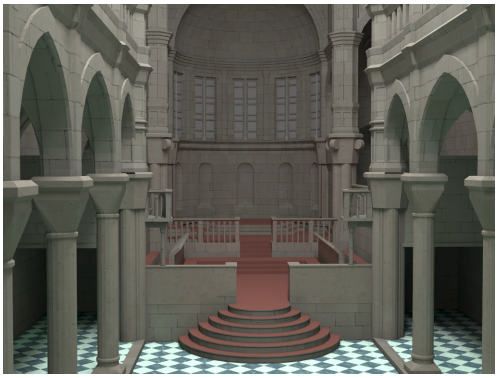


(e) Reconstruction cuts

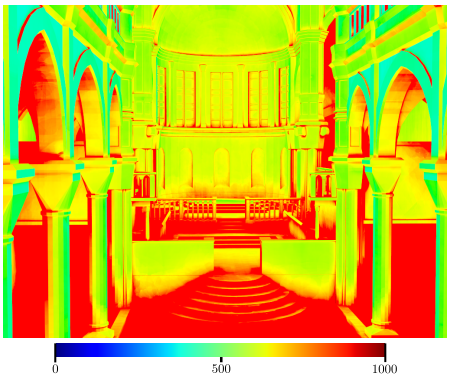


(f) Error Image

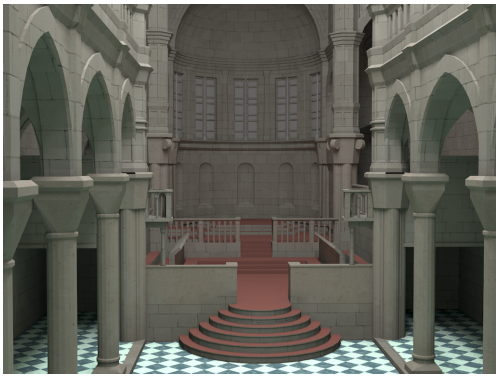
Figure 6.2: Cornell box with 20000 VPLs



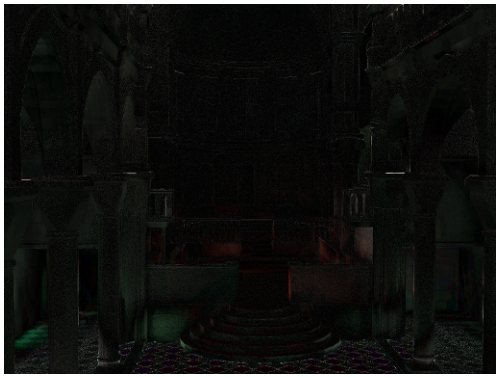
(a) IGI



(b) Cut size

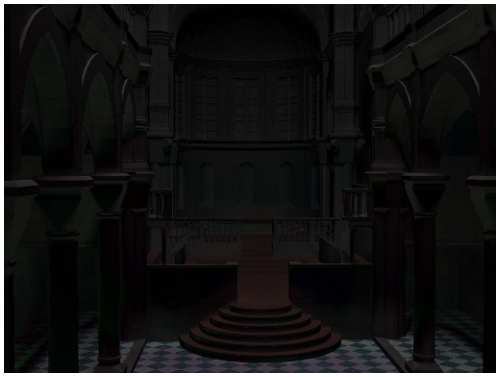


(c) Lightcuts



(d) 4 x Error Image

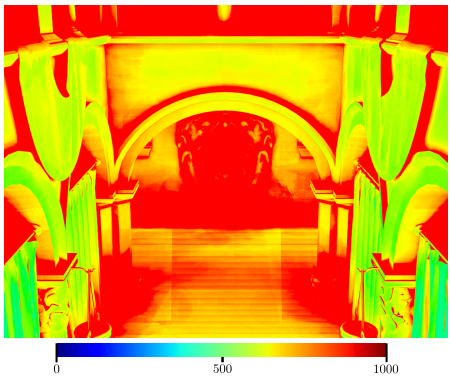
(e) Reconstruction cuts



(f) Error Image

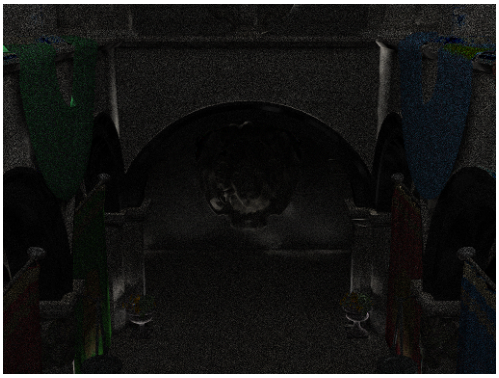
Figure 6.3: Sibenik with 20000 VPLs

(a) IGI



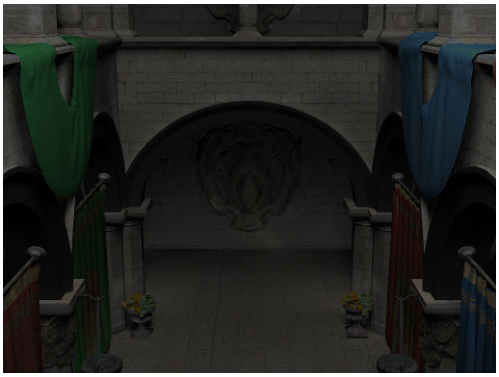
(b) Cut size

(c) Lightcuts



(d) 4 x Error Image

(e) Reconstruction cuts



(f) Error Image

Figure 6.4: Sponza with 20000 VPLs

Scene	Polygons	Resolution	No. of lights		Image time		
			Area lights	Point lights	IGI	Lightcuts	Reconstruction cuts
Cornell box	18	1280x1280	2	20000	179s	21s	10s
Sibenik	108699	1280x960	2	20000	312s	62s	42s
Sponza	262213	1280x960	8	20000	353s	75s	40s

(a) Image time

Scene	Resolution	No. of lights		Per pixel averages			
				Lightcuts		Reconstruction cuts	
		Area lights	Point lights	Cut size	Shadow rays	Cut size	Shadow rays
Cornell box	1280x1280	2	20000	552	382	203	132
Sibenik	1280x960	2	20000	776	534	476	322
Sponza	1280x960	8	20000	867	585	439	282

(b) Per pixel averages

Table 6.1: Summarized results for 20000 VPLs

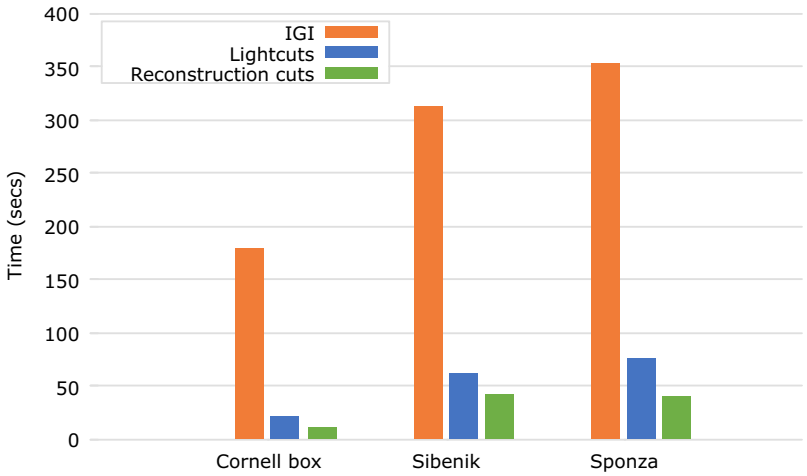


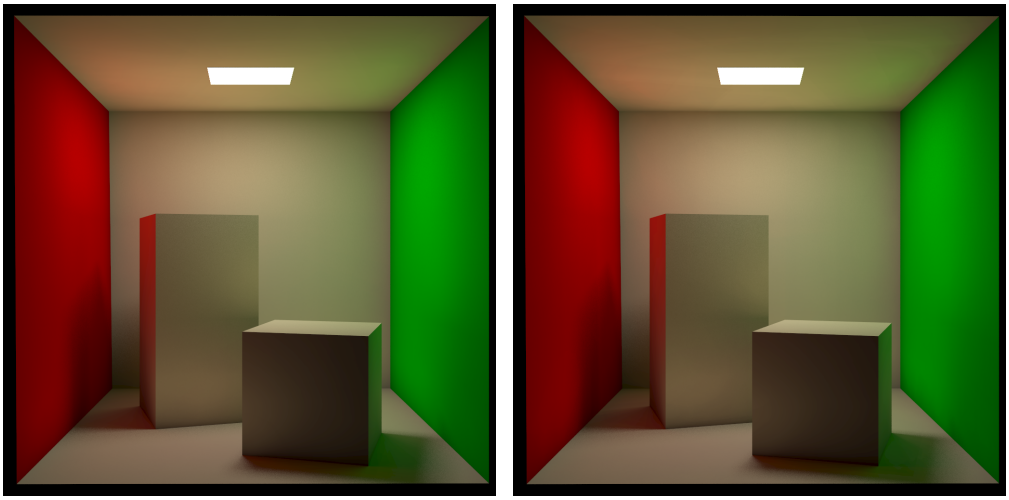
Figure 6.5: Summarized results for 20000 VPLs

6.1.2 Adjusting the perceptual visibility threshold

The objective of this test is to decrease rendering time by increasing the threshold while still producing images that are visually pleasing.

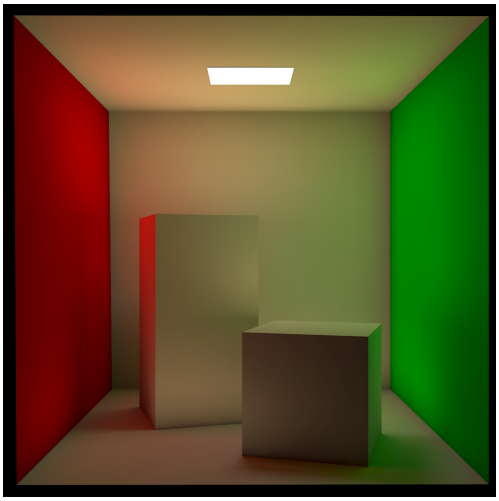
Below we show images with two threshold settings for each of the scenes, where we always provide images with the reference benchmark of 2%. Although humans can detect changes with a threshold of just under 1% in the worst possible conditions, we did not choose 1% as the benchmark setting, since there are no visible artifacts when compared to 2% for our results, this was verified by pitch black error images. We first show the rendered images, then the cut size images and the error images for each of the scenes with the two chosen settings. The second threshold setting is chosen as the highest possible threshold that is still visually pleasing with next to no artifacts. The error images in the figures (c) and (e) are between the two threshold settings for each

of the algorithms, where we abbreviate lightcuts and reconstructions cuts as Lc and Rc respectively. To show any differences in this case, the error images are magnified by 4. For example, 4 x Error Lc 2% vs 7% indicates the difference image multiplied by 4 between the rendered scene at a threshold of 2% and 7% for the lightcuts algorithm. The error images (d) and (f) are between the two algorithms for each of the threshold settings, and the rendered scenes is for each of the algorithms and threshold settings. The scenes presented here are rendered with 100000 VPLs at 1024x1024 for the Cornell box and 200000 VPLs at 1024x768 for the Sibenik and Sponza scenes. The tables and histograms at the end of this test quantify these results.

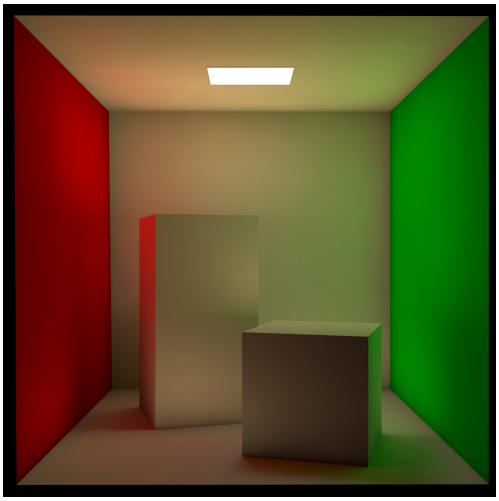


(a) Lc 2%

(b) Lc 5%

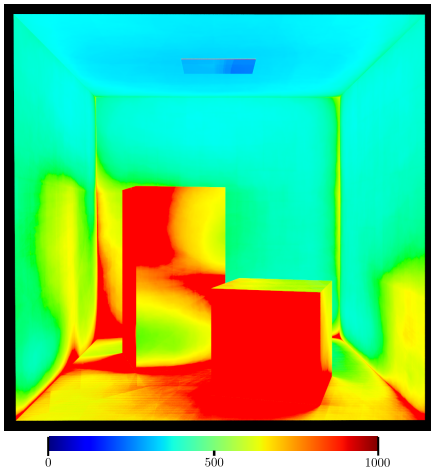


(c) Rc 2%

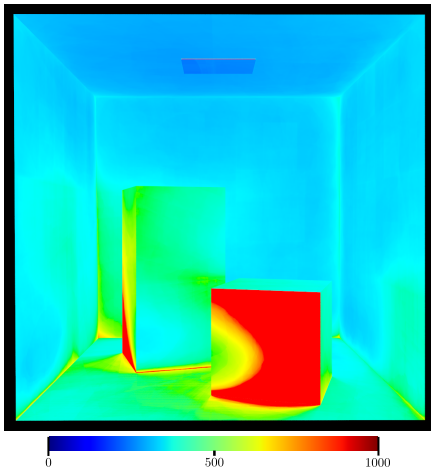


(d) Rc 5%

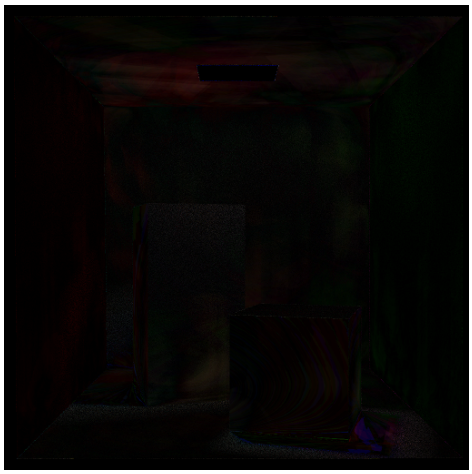
Figure 6.6: Cornell box visibility threshold



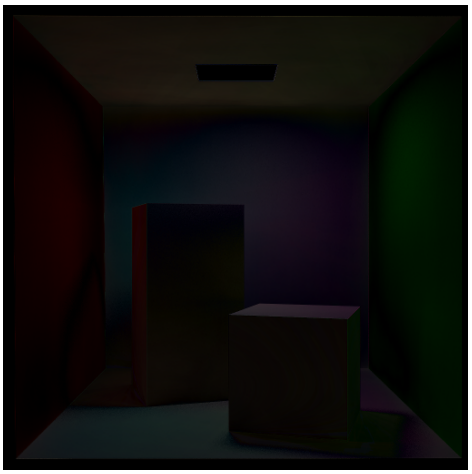
(a) Cut size 2%



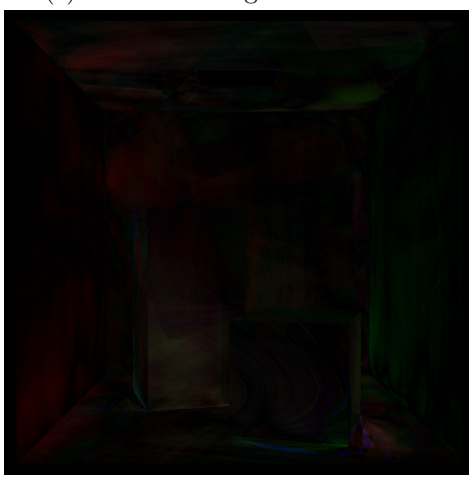
(b) Cut size 5%



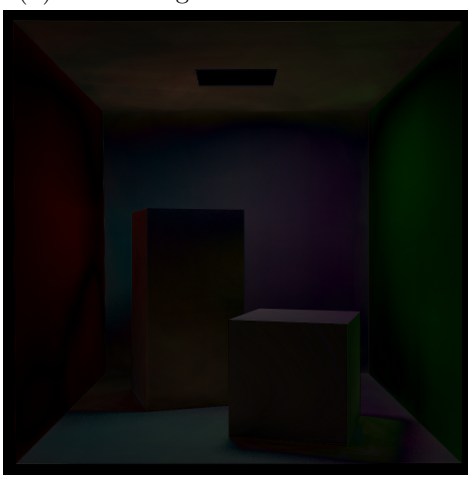
(c) 4 x Error image Lc 2% vs 5%



(d) Error image Lc vs Rc 2%

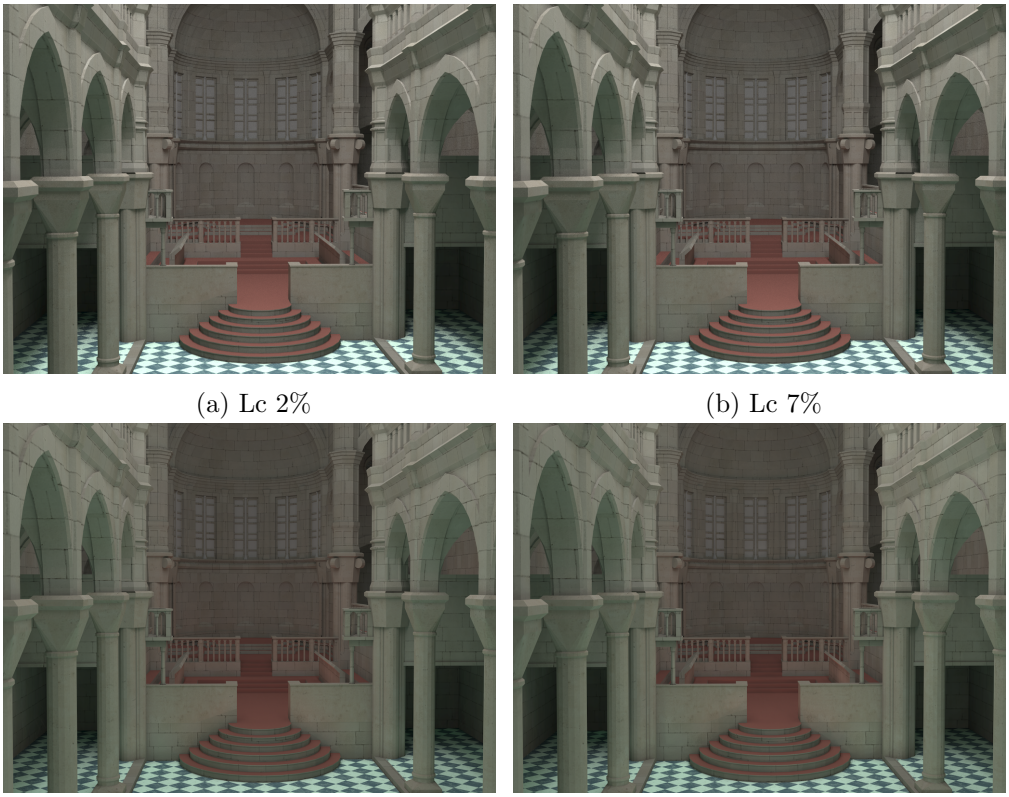


(e) 4 x Error image Rc 2% vs 5%



(f) Error image Lc vs Rc 5%

Figure 6.7: Cornell box visibility threshold



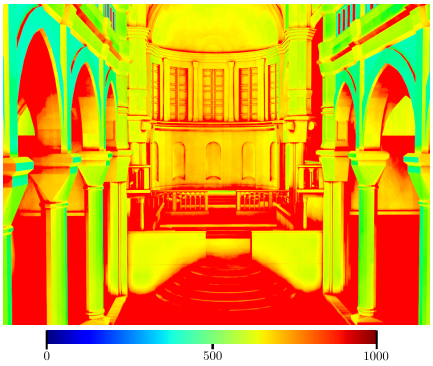
(a) Lc 2%

(b) Lc 7%

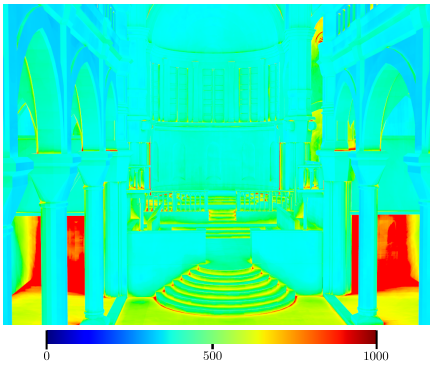
(c) Rc 2%

(d) Rc 7%

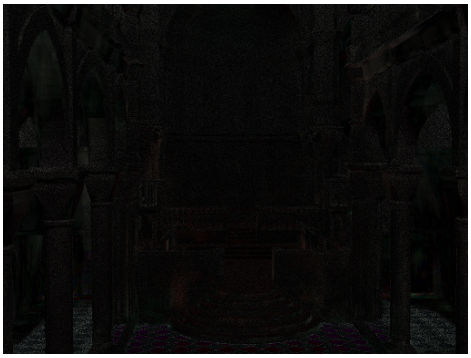
Figure 6.8: Sibenik visibility threshold



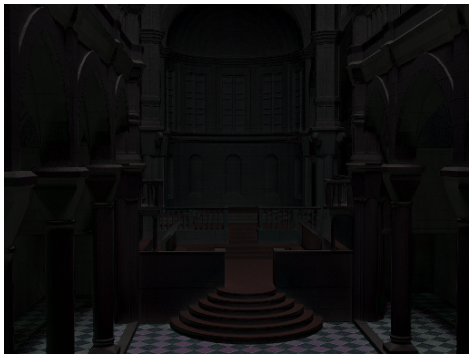
(a) Cut size 2%



(b) Cut size 7%

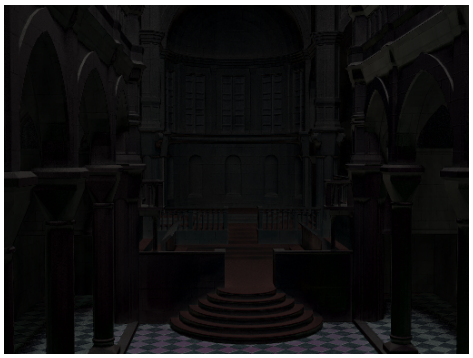


(c) 4 x Error image Lc 2% vs 7%



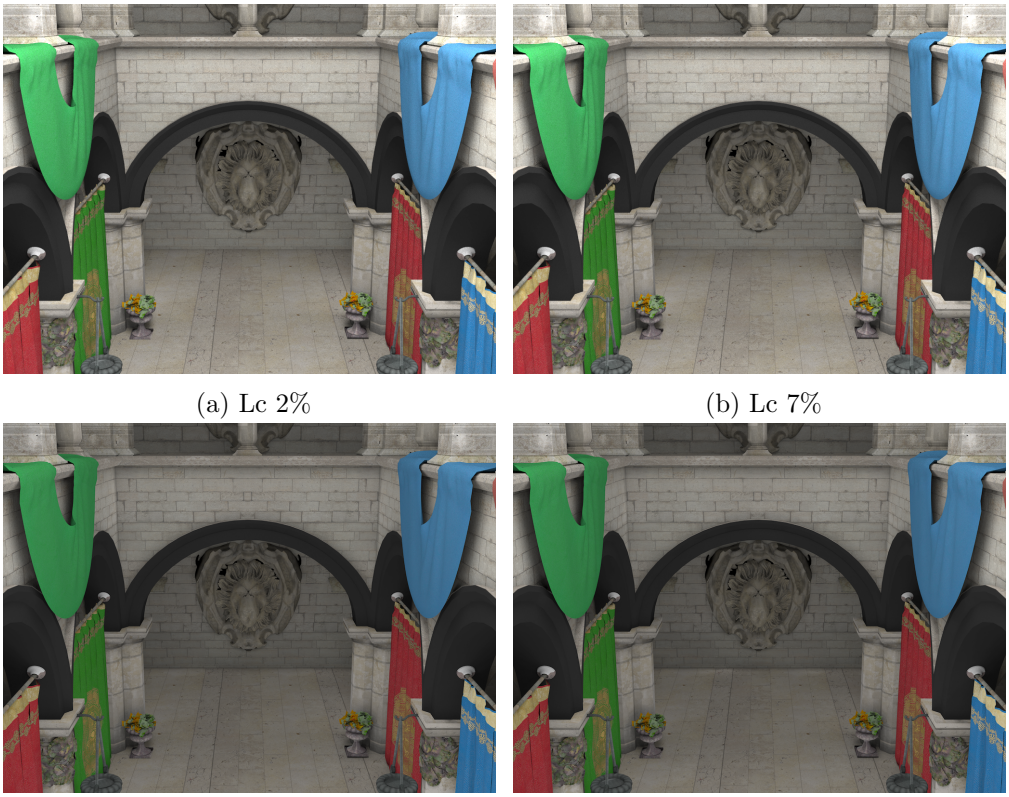
(d) Error image Lc vs Rc 2%

(e) 4 x Error image Rc 2% vs 7%



(f) Error image Lc vs Rc 7%

Figure 6.9: Sibenik visibility threshold



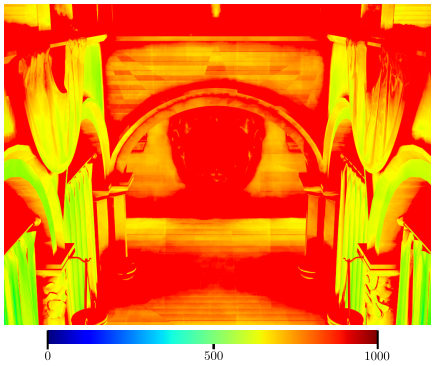
(a) Lc 2%

(b) Lc 7%

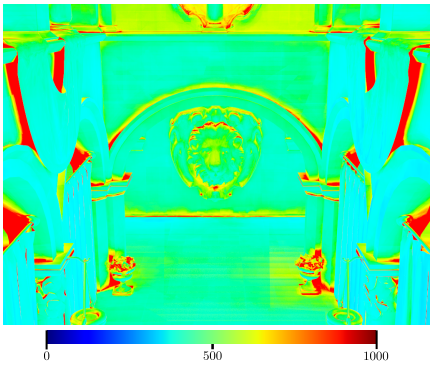
(c) Rc 2%

(d) Rc 5%

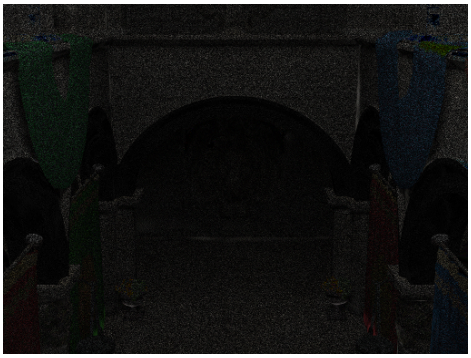
Figure 6.10: Sponza visibility threshold



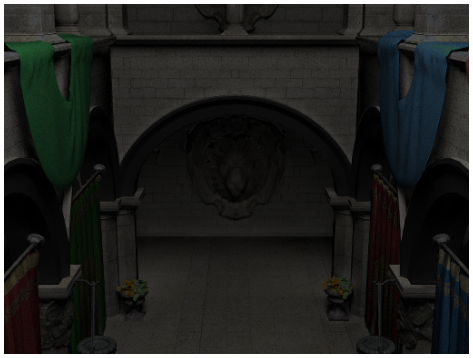
(a) Cut size 2%



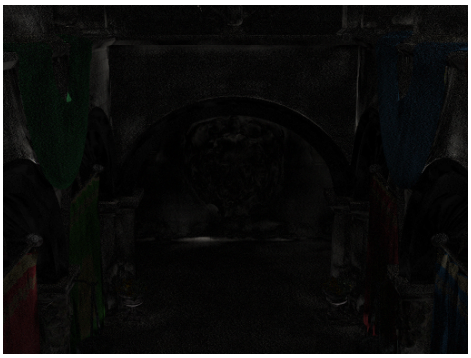
(b) Cut size 7%



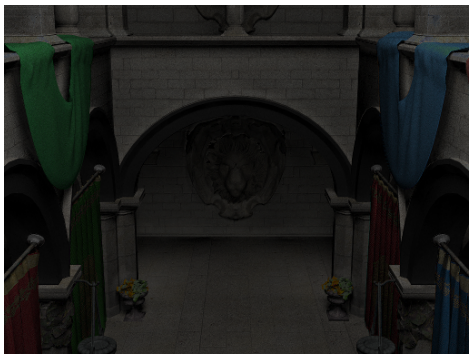
(c) 4 x Error image Lc 2% vs 7%



(d) Error image Lc vs Rc 2%



(e) 4 x Error image Rc 2% vs 5%



(f) Error image Lc 7% vs Rc 5%

Figure 6.11: Sponza visibility threshold

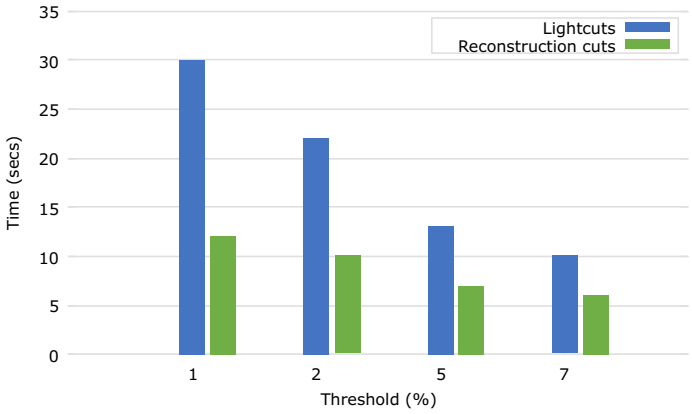
Scene	Resolution	Polygons	No. of lights		Image time (Visibility threshold)							
					Lightcuts				Reconstruction cuts			
			Area lights	Point lights	1%	2%	5%	7%	1%	2%	5%	7%
Cornell box	1280x1280	18	2	100000	30s	22s	13s	10s	12s	10s	7s	6s
Sibenik	1280x960	108699	2	200000	74s	63s	36s	29s	42s	41s	25s	20s
Sponza	1280x960	262213	8	200000	83s	75s	44s	36s	40s	39s	28s	23s

(a) Image time

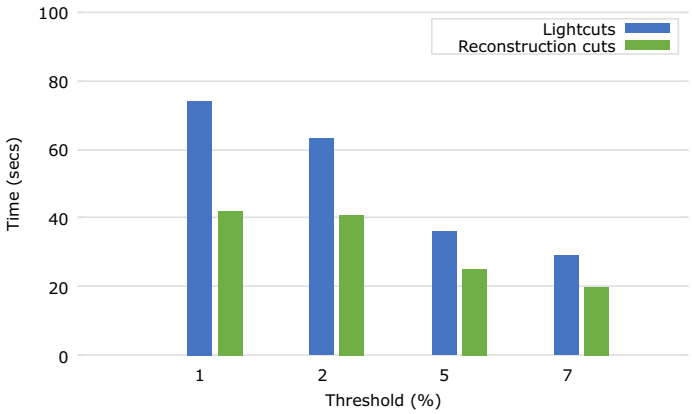
Scene	Per pixel averages (Visibility threshold)															
	Lightcuts								Reconstruction cuts							
	Cut size				Shadow rays				Cut size				Shadow rays			
	1%	2%	5%	7%	1%	2%	5%	7%	1%	2%	5%	7%	1%	2%	5%	7%
Cornell box	765	555	329	264	569	383	200	156	256	199	113	88	165	130	71	54
Sibenik	979	822	480	381	642	542	327	262	535	496	281	214	338	322	184	141
Sponza	997	893	531	422	650	585	345	275	482	457	316	253	282	276	195	156

(b) Per pixel averages

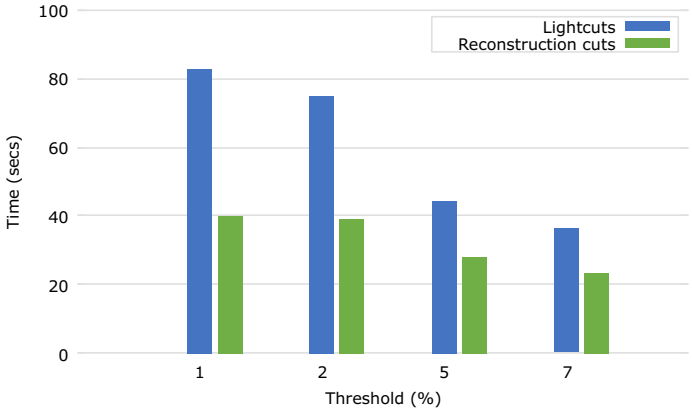
Table 6.2: Visibility threshold results



(a) Cornell box



(b) Sibenik



(c) Sponza

Figure 6.12: Visibility threshold results

6.1.3 Adjusting the resolution

This test shows the timing for lightcuts and reconstruction cuts at different resolutions with 1 spp and a 2% threshold.

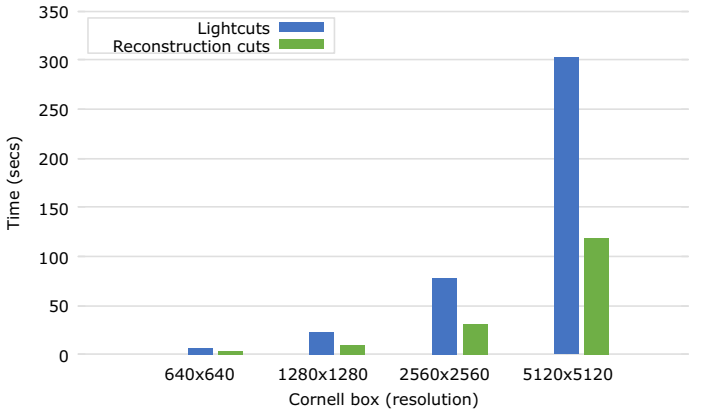
Scene	Polygons	Resolution	No. of lights		Image time	
			Area lights	Point lights	Lightcuts	Reconstruction cuts
Cornell box	18	640x640	2	100000	6s	3s
Sibenik	108699	640x480	2	200000	17s	12s
Sponza	262213	640x480	8	200000	25s	12s
Cornell box	18	1280x1280	2	100000	22s	10s
Sibenik	108699	1280x960	2	200000	62s	41s
Sponza	262213	1280x960	8	200000	79s	38s
Cornell box	18	2560x2560	2	100000	78s	31s
Sibenik	108699	2560x1920	2	200000	247s	117s
Sponza	262213	2560x1920	8	200000	296s	118s
Cornell box	18	5120x5120	2	100000	302s	118s
Sibenik	108699	5120x3840	2	200000	982s	394s
Sponza	262213	5120x3840	8	200000	1185s	406s

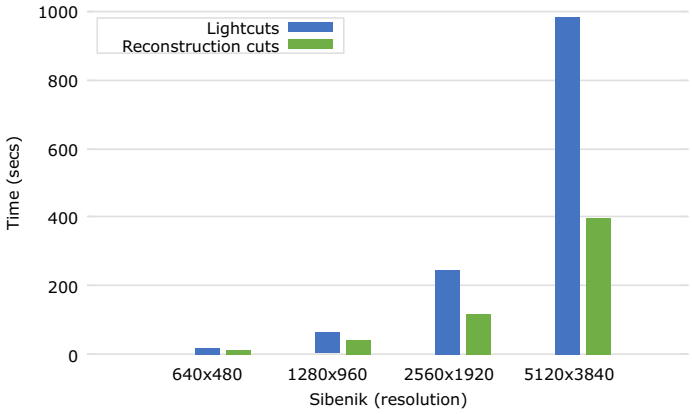
(a) Image time

Scene	Resolution	No. of lights		Per pixel averages			
				Lightcuts		Reconstruction cuts	
		Area lights	Point lights	Cut size	Shadow rays	Cut size	Shadow rays
Cornell box	640x640	2	100000	590	395	222	139
Sibenik	640x480	4	200000	824	544	636	409
Sponza	640x480	4	200000	891	598	556	342
Cornell box	1280x1280	2	100000	560	389	201	131
Sibenik	1280x960	4	200000	817	539	497	323
Sponza	1280x960	4	200000	896	592	454	267
Cornell box	2560x2560	2	100000	540	376	194	127
Sibenik	2560x1920	4	200000	812	536	395	259
Sponza	2560x1920	4	200000	898	585	371	230
Cornell box	5120x5120	2	100000	537	369	178	120
Sibenik	5120x3840	4	200000	811	534	324	213
Sponza	5120x3840	4	200000	886	577	310	190

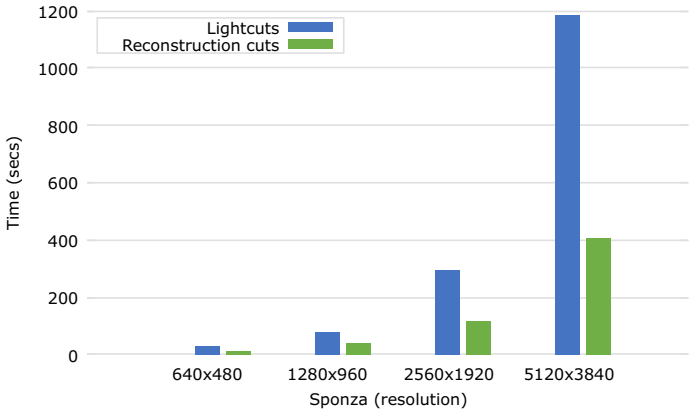
(b) Per pixel averages

Table 6.3: Resolution results





(a) Sibenik



(b) Sponza

Figure 6.13: Resolution results

6.2 Discussion

The claims of the original lightcuts paper hold up to scrutiny, since we were able to reproduce the results by reducing the time sublinearly with increasing number of lights, as this is shown clearly by the first test in Section 6.1.1 when viewing the graphs and comparing the performance to the naive (i.e. IGI) method. Even when increasing the number of VPLs from 20000 to 200000 the time remains roughly the same for lightcuts and reconstruction cuts. When a low number of VPLs (up to 1000) is enough to render a scene then the IGI method is preferable, since it is quicker than lightcuts and a little slower than reconstruction cuts with 1000 VPLs but much easier to implement.

The second test shows us that the 2% threshold is a very conservative setting but one that is meant to work well with most scenes. For the Cornell box the threshold can be adjusted to 5% and for the Sibenik model we can adjust the threshold all the way to 7% and still render decent looking images. The Sponza scene works well with 7% for lightcuts but for reconstruction cuts the threshold has to be set more accurately to 5%, which is still not perfect when looking at the floor near the corners where we have some patches that are brighter juxtaposed to 2%. From this we can conclude that the same threshold setting may not work equally well as it does for lightcuts than for reconstruction cuts and that it is scene dependent. Adjusting the threshold from 2% to 7% cuts the render time in half for lightcuts, the same cannot be said of reconstruction cuts although it is at least 1.45 faster than lightcuts at 7% for our test scenes. For the Sponza model, lightcuts would be just slightly slower at 7% than reconstruction cuts at 5%.

The final test shows us that the shadow rays for the per pixel averages actually decrease with higher resolutions. Reconstruction cuts can reduce the shadow rays by 48% from 640x480 to 5120x3840 for the Sibenik model. The reason for this is that reconstruction cuts is able to interpolate more pixels at higher resolutions. Reconstruction cuts is almost 3 times faster than lightcuts at rendering the Sponza scene at 5120x3840 in contrast to 2 times as fast at 640x480. In most cases reconstruction cuts is between 2 and 2.5 times faster than lightcuts and in all cases it is at least 1.4 times quicker from the results shown.

In summary, lightcuts and reconstruction cuts scale sublinearly with increasing number of lights. Reconstruction cuts is always faster than lightcuts with the same visibility threshold settings but lightcuts is in some cases more capable of handling higher thresholds as is seen with the Sponza model. At higher resolutions reconstruction cuts has a clear speed advantage versus lightcuts.

Chapter 7

Conclusion and Future Work

In this thesis, we have presented a data parallel version of the lightcuts and reconstruction cuts algorithm. The aim of these two algorithms are to reduce the number of shadow rays, since visibility tests (i.e. shadow rays) are usually the dominant cost, by grouping lights into clusters. The heterogeneous data parallel versions of these two algorithms can run on multiple types of hardware, such as CPUs, GPUs and accelerator cards. Although it should be no surprise that the parallelized verisons are faster when comparing the results to those in the lightcuts paper [WFA+05], since we have more computational power. However, we can conclude that the algorithms are scalable when ported to the OpenCL framework. In addition, the reconstruction cuts algorithm is almost three times faster than the standard lightcuts algorithm, where a state machine had to be designed for the purpose of synchronizing the threads because of the limitations of OpenCL. There is visibly no loss of quality in the images rendered with lightcuts when comparing it to IGI (exact solution), which is evidenced by the error images that need to be magnified to show any differences at all. Even when increasing the perceptual visibility threshold, no differences can be seen between IGI and lightcuts. The images rendered with reconstruction cuts are darker when compared to IGI and the error images show differences without magnification.

Future work could use more effective sampling techniques for the point lights or VPLs by placing them at more advantageous positions, where the paths from the camera are more likely to find the point lights and reduce the cut size [SIP07] and [SIMP06]. Adaptive progressive refinement [PS89] could be used to produce high quality antialiased images with fewer samples, as was done in the lightcuts paper, this would significantly reduce the amount of time to render the reference images in this thesis. Some enhancements to the work in this thesis could be additional materials such as the Cook-Torrance [CT82] and Ward [War92] BRDFs and different light sources, such as directional lights and environment maps.

Bibliography

[AK90] James Arvo and David Kirk. Particle transport and image synthesis. *SIGGRAPH Comput. Graph.*, 24(4):63–66, September 1990.

[App68] Arthur Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 37–45, New York, NY, USA, 1968. ACM.

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[CT82] R. L. Cook and K. E. Torrance. A reflectance model for computer graphics. *ACM Trans. Graph.*, 1(1):7–24, January 1982.

[DBBS06] Philip Dutre, Kavita Bala, Philippe Bekaert, and Peter Shirley. *Advanced Global Illumination*. AK Peters Ltd, 2006.

[DGS12] Tomáš Davidovič, Iliyan Georgiev, and Philipp Slusallek. Progressive lightcuts for gpu. In *ACM SIGGRAPH 2012 Talks*, SIGGRAPH '12, pages 1:1–1:1, New York, NY, USA, 2012. ACM.

[DKH+14] Carsten Dachsbacher, Jaroslav Křivánek, Miloš Hašan, Adam Arbree, Bruce Walter, and Jan Novák. Scalable realistic rendering with many-light methods. *Computer Graphics Forum*, 33(1):88–104, 2014.

[Fly72] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(9):948–960, September 1972.

[JC95] Henrik Wann Jensen and Niels Jørgen Christensen. Photon maps in bidirectional monte carlo ray tracing of complex objects. *Computers & Graphics*, 19(2):215–224, 1995.

[Kaj86] James T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, August 1986.

[Kel97] Alexander Keller. Instant radiosity. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 49–56, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[KK04] Thomas Kollig and Alexander Keller. Illumination in the Presence of Weak Singularities. In *MCQMC Methods*, 2004.

[LFTG97] Eric P. F. Lafortune, Sing-Choong Foo, Kenneth E. Torrance, and Donald P. Greenberg. Non-linear approximation of reflectance functions. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 117–126, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[MA09] Thomas Mühlbacher and Clemens Arbesser. Lightcuts in cuda. Technical report, Vienna University Of Technology, 2009.

[Mik] Miroslav Mikšík. Implementing lightcuts.

[MT97] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *J. Graph. Tools*, 2(1):21–28, October 1997.

[Nvi] Nvidia. *Fermi Compute Architecture Whitepaper.*

[PBD+10] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. Optix: A general purpose ray tracing engine. *ACM Trans. Graph.*, 29(4):66:1–66:13, July 2010.

[PH10] Matt Pharr and Greg Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2010.

[Pho75] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, June 1975.

[PS89] J. Painter and K. Sloan. Antialiased ray tracing by adaptive progressive refinement. *SIGGRAPH Comput. Graph.*, 23(3):281–288, July 1989.

[RGK+08] T. Ritschel, T. Grosch, M. H. Kim, H.-P. Seidel, C. Dachsbacher, and J. Kautz. Imperfect shadow maps for efficient computation of indirect illumination. *ACM Trans. Graph.*, 27(5):129:1–129:8, December 2008.

[SIMP06] B. Segovia, J. C. Iehl, R. Mitanchey, and B. Péroche. Bidirectional instant radiosity. In *Proceedings of the 17th Eurographics Conference on Rendering Techniques*, EGSR '06, pages 389–397, Aire-la-Ville, Switzerland, Switzerland, 2006. Eurographics Association.

[SIP07] Benjamin Segovia, Jean-Claude Iehl, and Bernard Péroche. Metropolis Instant Radiosity. *Computer Graphics Forum*, 26(3):425–434, September 2007.

[Tau10] Oliver Taubmann. Lightcuts on modern graphics hardware. Bachelor thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, November 2010.

[Vea98] Eric Veach. *Robust Monte Carlo Methods for Light Transport Simulation.* PhD thesis, Stanford, CA, USA, 1998. AAI9837162.

[War92] Gregory J. Ward. Measuring and modeling anisotropic reflection. *SIGGRAPH Comput. Graph.*, 26(2):265–272, July 1992.

[WFA+05] Bruce Walter, Sebastian Fernandez, Adam Arbree, Kavita Bala, Michael Donikian, and Donald P. Greenberg. Lightcuts: A scalable approach to illumination. *ACM Trans. Graph.*, 24(3):1098–1107, July 2005.

[Whi80] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, June 1980.

[Wil78] Lance Williams. Casting curved shadows on curved surfaces. *SIGGRAPH Comput. Graph.*, 12(3):270–274, August 1978.

[WKB+02] Ingo Wald, Thomas Kollig, Carsten Benthin, Alexander Keller, and Philipp Slusallek. Interactive global illumination using fast ray tracing. In *Proceedings of the 13th Eurographics Workshop on Rendering*, EGRW '02, pages 15–24, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[Zha11] Tong Zhang. Gpu-based global illumination using lightcuts. Master's thesis, Purdue University, 2011.