Globale Beleuchtungsberechnung mit Lightcuts

Diplomarbeit im Fach Informatik

vorgelegt von

Thomas Rudolf Kemmer

geb. am 29. Oktober 1980 in Miltenberg

angefertigt am

Institut für Informatik
Lehrstuhl für Graphische Datenverarbeitung
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer: Manfred Ernst

Betreuender Hochschullehrer: Prof. Dr. Marc Stamminger

Beginn der Arbeit: 22. Juni 2006

Abgabe der Arbeit: 08. Januar 2007

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Idea of the Project

Many photo-realistic rendering systems supporting global illumination were created until today. Most of them use prominent algorithms like ray tracing, path tracing, radiosity and metropolis light transport. The algorithms have in common the ability to simulate a complex illumination environment. Among all of them ray tracing is supposed to be the first choice for modern scientific and commercial rendering architectures. It can process highly detailed scenes due to its logarithmic complexity. Additionally, ray tracers can handle optical phenomena like diffuse global illumination and subsurface scattering. During the last years, many theses and papers have been published with ideas to improve ray tracing effects and its efficiency. There are several fields of optimizations regarding the ray tracing algorithm. Many of them focus on efficient intersection testing which involves sophisticated space subdivision methods. This thesis offers a different approach by using an algorithm called *lightcuts*. Its main idea is to optimize rendering speed by reducing the number of necessary shadow rays. This is achieved by creating a hierarchical bounding structure for light sources in the scene.

1.2 Goals and History of the Project

The primary goal of my thesis is to implement the *lightcuts* algorithm into a modern ray tracing system and analyse its performance. I chose the ray tracing framework accompanying the ”Physically Based Rendering” book from Matt Pharr and Grep Humphreys [22] due to its plugin structure and detailed documentation. At the beginning I examined how ray tracing works within this system and solved problems occurring when using version 4 of the g++ compiler. Then I started to create the lightcuts integrator code which is based on PBRT’s direct lighting integrator. This implicates following steps for each supported light source type:

- hierarchical clustering of light sources into light trees
- error estimation for light tree nodes

I started to implement the algorithm for point light sources. Therefore, a wrapper class for light trees was generated, which should be an abstraction to be used by any kind of light type. To cluster the

lights I used the greedy clustering algorithm mentioned in the first lightcuts paper [28]. It performed poorly and I soon realized it is not possible to optimize the bottom-to-top light clustering approach. As a consequence I simply replaced it by a top-down clustering mechanism using the kd tree algorithm which is explained in chapter 3. After this was accomplished, I started to think about how infinite area light could be used by the lightcuts algorithm. The paper proposed to replace the infinite area light by a selection of distant lights. I thought it might be a better idea to dynamically generate distant lights, because this would perfectly fit the dynamic nature of the lightcuts algorithm to descend the light tree as far as necessary. I realized this by sampling the radiance map with an algorithm proposed by Paul Debevec [8], which is described in 6. Any generated distant light represents a patch of directions on the unit sphere. The directions are then used for error estimation in the lightcuts algorithm.
In dependency of the light type the maximum error has to be approximated by using a light cluster instead of real lights. Therefore, I had to find the error's upper bound introduced by using an interval of incident light directions instead of a real light. The error estimation will be described in chapter 4 for the $\cos\theta$ term and in chapter 5 for the BRDF models.. It was very complicated to find a bound for the infinite area light source. Using a bounding cap offered a reasonable approximation with reduced computational effort. In general, calculations with spheres are very difficult and end up easily with large terms. Finally, it was possible to find a method to efficiently create bounding caps by reducing the problem to a two dimensional case. The main challenge within my thesis was error estimation for light tree nodes. I started with approximating the worst incident light direction and ended up with bound for three BRDF models: Lambertian, Oren-Nayar and the Torrance-Sparrow microfacet model. Especially Torrance-Sparrow was extremely complex due to its usage of the half-angle vector. In the end I had a PBRT plugin which supports efficient rendering with a large number of light sources.

1.3 Previous Work

The following sections give a brief outline on the most influencial papers with respect to my thesis.

1.3.1 Lightcuts: A Scalable Approach to Illumination

The lightcuts paper from Bruce Walter, Sebastian Fernandez et al. [28] firstly introduced a method for calculating very complex illumination situations with strongly sublinear costs with respect to the participating light sources. At this level, their ray tracer could handle point lights, area lights, HDR environment maps, sun/sky models and indirect illumination. The idea behind the initial lightcuts framework is to generate light clusters which are large amounts of individual light sources incorporated by a bounding structure. This introduces an error to the final image. Therefore it is necessary to find the error's upper bound. Lighting is done by descending a hierarchical light tree and estimate efficiently an upper bound for the error present at this level. The calculation is finished as soon as the error falls below a previously defined threshold. The results demonstrated in the paper are impressive: the reference renderer has to evaluate each participating light separately, which results in thousands of shadow rays. On the contrary, the lightcuts method accomplishes to reduce the number of shadow rays to a few hundreds per pixel. Despite the relatively small number of shadow rays, the example renderings look very promising.

1.3.2 Multidimensional Lightcuts

The succeeding paper to the previously mentioned lightcuts appeared at this year's SIGGRAPH [27]. Bruce Walter et al. from Cornell University further improved their rendering engine and transported the idea from illumination calculation to other domains. It was achieved to efficiently render rich visual effects such as motion blur, participating media, depth of field and spatial anti-aliasing. Therefore, they used a method to discretize the integrals into sets of gather points and light points to adaptively approximate the sum of all possible gather-light pair interactions. The given examples show a surprisingly efficient rendering system which scales very well. This paper was the first to mention the usage of a kd tree algorithm for building the light tree. In comparison to the previous lightcuts greedy tree algorithm, this should have improved performance significantly. Unfortunately, this aspect was not discussed in the paper.

1.3.3 Notes on the Ward BRDF

Bruce Walter's paper [26] addresses some interesting aspects of the Ward BRDF. It is explained how the BRDF can be efficiently evaluated and what needs to be considered for a correct sampling. The description of a bounding mechanism for the Ward BRDF over a region of given directions was the more important aspect regarding my thesis. Although I do no use this BRDF, it is also based on the half-angle vector direction. This makes the bounding mechanism compatible to be used with the Torrance-Sparrow model.

1.3.4 Median Cut Sampling for HDR Light Probes

Paul Debevec created a poster accompanying a paper at SIGGRAPH 2005 [8]. The paper describes an algorithm to split an HDR environment map into regions with similar light energy. Its purpose is to create a certain amount of directional lights representing the map. The algorithm works by splitting patches recursively along the bigger side of the actually treated patch. This algorithm inspired me to use it in my implementation for creating an efficient and dynamic sampling technique. This allows handling large maps and saving preprocessing time for my implementation.

1.3.5 Recovering High Dynamic Range Radiance Maps from Photographs

Debevec and Malik presented a method of recovering high dynamic range radiance maps from photographs at SIGGRAPH 1997 [5]. This can be done by using conventional imaging equipment, which is the main benefit of the procedure. It works by taking multiple photographs of a real scene with different amounts of exposure. Afterward, the images are merged by an algorithm to create a single high dynamic range radiance map. This enables the authentic recording of illumination settings in combination with an adequate effort. Paul Debevec created numerous maps and made them publicly available in the internet [6]. Remarkably, most scientific work about high dynamic range (HDR) uses the maps from his gallery. At SIGGRAPH 1998 a photo realistic rendering system was presented which allows rendering of synthetic objects into real scenes [7]. This uses the HDR radiance maps to illuminate a synthetic object appropriately thus it is not recognized as artificial. Of course, this also

involves local lighting effects, but due to the use of a reflection model for nearby objects the method achieves very good results.

Chapter 2

Fundamentals

Photo realistic rendering implies a very complex simulation system based on physical principals. So it is a necessity to know and understand some fundamental concepts. This chapter is dedicated to explain a selection of the most important general algorithms used in the implementation accompanying this thesis. The first section describes how to calculate with spherical patches and solid angles. These basics are required for calculations with respect to the infinite area light in chapter 6. The subsequent sections explain the general idea of *ray casting* and *ray tracing*. To understand the optimization proposed by the lightcuts system, this knowledge is essential. This chapter is closed by a description of different reflection models used by modern ray tracers like reflection, refraction and subsurface scattering. Additionally, some prominent models including the Torrance-Sparrow microfacet model are explained in great detail.

2.1 Geometry

Calculations in a 3D environment can only be done by the help of a decent portion of geometrical knowledge. For the ray tracing algorithm described later in this chapter you need to know how to subtend rays and any kind of objects present in the scene. This would most likely be a set of polygons and quadric surfaces. For some parts of the lightcuts implementation spherical objects were a major challenge, so this is the first item to be examined.

2.1.1 Spheres and Spherical Coordinates

Often it is very handy to describe a point on the unit sphere by Cartesian coordinates. On the contrary, a better representation for many applications are spherical coordinates. These coordinates consist of two angles and a distance from the origin. The distance r is the radius of the sphere centered at the origin of the coordinate axis. θ denotes the zenith angle between the positive z-axis and the vector from the origin to the point P on the sphere. ϕ is defined as the azimuth angle from the positive x-axis and the projected vector $\vec{p}$ on the xy-plane. With this definition the valid range for θ is $[0; \pi]$ and for ϕ the valid range is $[0; 2\pi)$. It is sometimes necessary to switch between the two representations. The formulas for the mapping from $f(r, \theta, \phi) \rightarrow f(x, y, z)$ are:

$$\begin{aligned} x &= r \sin\theta \cos\phi \\ y &= r \sin\theta \sin\phi \\ z &= r \cos\theta \end{aligned}$$

The formulas for the inverse mapping function from $f(x, y, z) \rightarrow f(r, \theta, \phi)$ are:

$$\begin{aligned} \theta &= \arccos(\frac{z}{r}) \\ \phi &= \arctan(\frac{y}{x}) \end{aligned}$$

If the radius of the sphere is equal to 1.0, then it can be referred as unit sphere and r can be left out whenever it is a factor or denominator. Since infinite area lights are defined by incoming radiance from all directions, the unit sphere plays an important role.

2.1.2 Solid Angles

The solid angle Ω is the geometric equivalent to an angle in a plane. It represents a set of directions and its unit is *steradian (sr)*. The solid angle of a surface in space with respect to a location can be obtained by projecting it on a sphere with radius r around this location. Consequently it is defined as the size of the projected surface divided by the squared radius:

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

The differential angle $d\omega$ can also be written as an integral depending on the differential size of the considered surface dA:

$$d\omega = \frac{dA \cos\theta}{r^2}$$

The θ denotes the angle between the surface normal and the vector from the origin. This ensures that the orientation of the surface is treated correctly. The divisor is the squared distance of the surface from the origin.

2.1.3 Spherical Patches

The infinite area light in combination with the median cut algorithm uses special lights which describe a whole set of directions on the unit sphere. For this case it is essential to be able to calculate the spherical angle for such a light source. These patches can be expressed by intervals of θ and ϕ in spherical coordinates. Matching this prerequisite, it is possible to calculate the area of such a patch as seen in figure 2.1 and thus finding the solid angle it describes:

$$\begin{aligned} A &= \int_{\theta_1}^{\theta_2} \int_{\phi_1}^{\phi_2} \sin\theta d\theta d\phi \\ &= (-\phi_1 + \phi_2)(\cos\theta_1 - \cos\theta_2) \end{aligned}$$

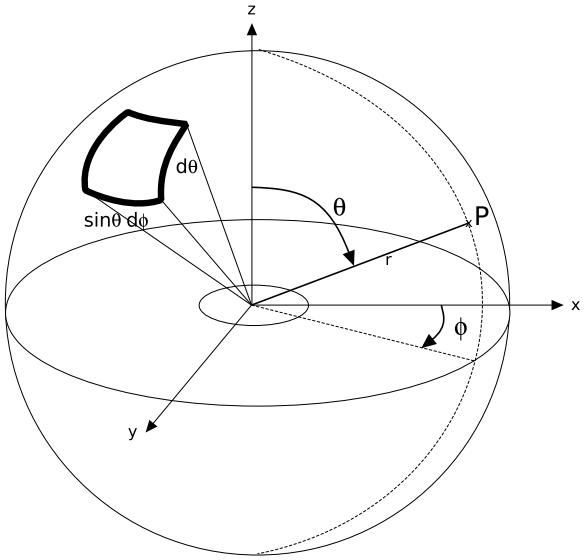


Figure 2.1: Spherical Patch.

2.2 Ray Casting

The ray casting algorithm was first presented by Arthur Appel in 1968 [2]. It works by shooting rays from an eye point into the scene. The ray then tries to find the closest object blocking its path. When an object was hit by the ray, the lighting and shading at the intersection point can be determined. The surface color is determined by the properties of the lights in the scene, the material of the surface and the interaction of both.

2.3 Ray Tracing

The more advanced Ray Tracing algorithm was developed by Turner Whitted in 1979 [29]. The previous ray casting algorithm has the big deficiency of only shooting *primary rays*. This means only rays from the eye point are cast. Whitted's idea was to recursively call the algorithm again, if the ray hits a shiny object like a mirror (see figure 2.2).
The ray is reflected by the surface normal at the intersection point to retrieve the arriving light. Almost the same applies to transparent objects with the difference that light is refracted. Then the ray enters matter and might be reflected and refracted again inside. The reflection/refraction process is repeated until a predefined recursion depth is reached or a solely diffuse object is hit. Diffuse objects reflect

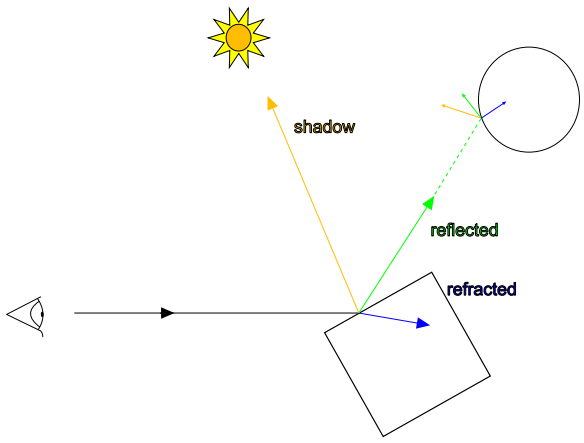


Figure 2.2: The figure shows the general idea of recursive ray tracing.

light in any direction, hence the reflection direction is not representing the properties of a surface adequately. At each intersection point *shadow rays* are generated to check, if a surface is visible to the lights in the scene. This test is very time-consuming, since all light sources have to be considered for opaque objects lying between the surface and the position of the light source. In general, the *visibility* denominates if objects are situated between two points in the scene. Whitted's method is based on evaluating the perfect specular reflection and refraction direction. Since there are few real surfaces like mirrors and glass which satisfy this criteria, the method was extended by using a function representing the reflective properties of a surface. This function is called *bidirectional reflection distribution function* (BRDF). It describes how incident radiance emitted by a light source from direction ω_i is reflected in direction ω_o. This issue is addressed in great detail in subsection 2.3.5.
Many techniques have been developed for speeding up this process. One approach is the lightcuts algorithm that reduces the number of shadow rays necessary to approximate the lighting.

2.3.1 The Rendering Equation

The rendering equation introduced by Kajiya [13] is the basic principle of modern global illumination algorithms. It is also known as *light transport equation* (LTE) which might be a better term for its meaning. The equation describes how much light arrives at the viewer's position L_o from a visible surface. This surface can reflect (L_r) and emit (L_e) radiance described by the formula below:

$$L_o(p, \vec{\omega}) = L_e(p, \vec{\omega}) + L_r(p, \vec{\omega})$$

The vector ω denotes the direction from surface position p towards the observer's position. The amount of reflected light is substituted by the integral of arriving radiance reflected by the BRDF. Additionally, this term depends on the incident angle of the arriving light. The expanded version of the LTE is formulated as follows:

$$L_o(p, \vec{\omega_o}) = L_e(p, \vec{\omega_o}) + \int_S \underbrace{f_r(p, \vec{\omega_o}, \vec{\omega_i})}_{\text{BSDF}} \underbrace{L_i(p, \vec{\omega_i})}_{\text{incident radiance}} \underbrace{(\vec{\omega_i} \cdot \vec{n})}_{\text{attenuation}} \, d\vec{\omega_i}$$

Since the principle of light transport is energy conservation, it may not be forgotten that incident light $L_i(p, \vec{\omega_i})$ has a recursive nature. Every illuminated surface emits light, which is scattered in the scene and illuminates another surface and so on... Radiance is constant along a ray in vacuum. For this reason it is possible to define the exitant radiance at p in direction $\vec{\omega}$ to be equal to the incident radiance at p' for a ray arriving from direction $-\vec{\omega}$:

$$L_i(p, \vec{\omega}) = L_o(p', -\vec{\omega})$$

Of course, there must not be any surface intersecting the ray between p and p'. To find the first intersection of a ray starting from p in direction $\vec{\omega}$, the trace function t is used. This changes the equation to:

$$L_i(p, \vec{\omega}) = L_o(t(p, \vec{\omega}), -\vec{\omega})$$

If this result is inserted into the LTE, the final equation suffers from recursive definition, because radiance leaving the surfaces appears on both sides:

$$L(p, \vec{\omega_o}) = L_e(p, \vec{\omega_o}) + \int_S f_r(p, \vec{\omega_o}, \vec{\omega_i}) L(t(p, \vec{\omega_i}), -\vec{\omega_i})(\vec{\omega_i} \cdot \vec{n})\, d\vec{\omega_i}$$

It is plausible that the equation can only be solved analytically for very primitive scenes with few participating surfaces. Many algorithms have been developed to estimate the value of the integral on the right side of the equation. A simple method is to be only interested in the radiance arriving directly from light sources in the scene. In the *physically based ray tracer* (PBRT) from Matt Pharr and Greg Humphreys the algorithms calculating the estimates are called *integrators*. The term is used in this thesis as well for the $lightcutsintegrator$.

2.3.2 Light Attenuation

Most ray tracing systems have the common assumption of light traveling in a vacuum if it does not intersect any geometry situated in the scene. Without a medium present, all emitted photons hit a surface or fly towards infinity. Due to energy conservation the photons never disappear as long as they are not absorbed by a surface thus transformed into heat. It is the point of interest to calculate the amount of light arriving at a differential area located at a surface in the scene using the setting from figure 2.3.
A light source is shining at a surface with its surface normal n and the inclination angle θ. The distance to the light source is denoted by r. Since a definite quantity of photons are shot from a light source the inward angle of the arriving light determines how many photons arrive at the differential area. As *Lambert's law* tells, the amount of energy is proportional to the cosine of the angle between the surface normal and the incident light direction. The incoming radiation is scientifically referred to as *irradiance* E and is measured by the unit $[wm^{-2}]$ (area density of flux). The initial energy of a light source is defined as Φ, so the irradiance arriving at the differential area is:

$$E = \Phi \cos\theta$$

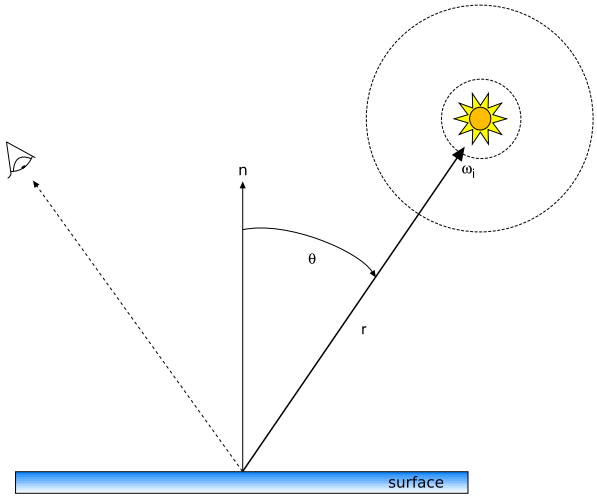


Figure 2.3: Light attenuation for omni directional light sources.

Point lights distribute light equally in all directions, hence the distance r from the light attenuates the number of photons hitting the differential area. This can be imagined easily: The energy spreads spherically around the originating light source. Simultaneously, the energy on the surface of the sphere stays the same. Due to the surface area's quadratic development, the irradiance arriving at the spherical surface is proportional to $\frac{1}{r^2}$:

$$E = \frac{\Phi \cos\theta}{4\pi r^2}$$

In more general terms, if a differential area is considered, the irradiance arriving at a point p from the hemisphere over the surface normal n is defined by the following integral:

$$E(p,n) = \int_{\Omega} L_i(p,\omega_i)|\cos\theta_i| d\omega_i$$

2.3.3 Bidirectional Surface Scattering Distribution Function (BSSDF)

The scattering behavior of light is the most important aspect to take into consideration when simulating physically correct global illumination. As mentioned before it is a part of the LTE. If you imagine some photons arriving at a random surface, it is a matter of course that these photons are reflected, transmitted or absorbed. The BSSDF is the most general model that simulates this behavior of light. Its initial nomenclature has already been defined in 1977 [19]. The definition for the BSSDF over the whole sphere of directions S^2 at position p looks like this:

$$dL_o(p,\omega_o) = \int_A \int_{S^2} S(p',\omega_i,p,\omega_o)\cos\theta_i d\omega_i dA$$

Solving this equation is not trivial at all, because it also contains the differential irradiance arriving at p' from direction ω_i. Therefore, the BSSRDF requires integration over surface area and incoming direction. Its complexity is due to the fact, that light which is entering matter at position p' might travel for some distance underneath until it leaves it again at position p Subsurface scattering is a problem

domain of its own and is heavily worked on. So far the publications of Jensen [12] at SIGGRAPH in 2001 and subsequent work showed that it is worth the additional effort. Translucent materials like skin, fluids and marble appear much more realistic than before. The subsequent sections describe simplified versions of bidirectional reflection functions commonly used by ray tracers.

2.3.4 Bidirectional Scattering Distribution Function (BSDF)

The union of both previously mentioned BRDF and BTDF is defined as BSDF. Usually it is a set of four functions: two treating the light reflected at both sides of the surface and two for its transmitted amount. From the programmer's point of view, it is an advantage, because you only need to refer to an abstract reflection interface, that knows itself how light is reflected in dependence of the viewer's direction.

2.3.5 Bidirectional Reflectance Distribution Function (BRDF)

As mentioned before, the *Bidirectional Reflectance Distribution Function* is a simplified *BSSRDF* which assumes that reflected light arrived at the same position before. To be physically correct, BRDFs have two fixed qualities: *reciprocity* and *energy conservation*. The first one ensures that incident and exiting light directions may be switched and the result stays the same:

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

Energy conservation is the basic principle of all physically based simulation systems. In this specific case, it means that the total reflected light energy is equal or less than the incident light energy arriving from a hemisphere H^2 around the surface normal n:

$$\int_{H^2} (n) f_r(p, \omega_o, \omega_i) \cos\theta_i d\omega_i \leq 1$$

The BRDF represents the probability that an incident photon will be reflected in a certain direction and does not pay attention to the photons entering the matter. This implicates that the surface normal $\vec{n}$ and the incident light direction ω_i are situated on the same side of the surface, in other words the dot product of the two vectors is always positive. Thus, the BRDF only defines the relation of incoming irradiance and leaving radiance at a position p and not the interaction with the material itself, which is referred to as *subsurface scattering*. It can be observed experimentally, that reflected radiance is proportional to the incoming irradiance, i.e. if a light source emission is increased, the irradiated objects also appear brighter. This relationship can be expressed as:

$$dL_o(p, \omega_o) \propto dE(p, \omega_i)$$

This proportionality leads to the definition of the BRDF:

$$f_r(p, \omega_o, \omega_i) \quad = \quad \frac{dL_o(p, \omega_o)}{dE(p, \omega_i)} = \frac{dL_o(p, \omega_o)}{L_i(p, \omega_i) \cos\theta_i d\omega_i}$$

By integrating this relationship over the surface area it is possible to calculate the total light leaving the surface:

$$dL_o(p, \omega_o) = \int_{S^2} f_r(p, \omega_o, \omega_i) L_i(p, \omega_i) \cos\theta_i d\omega_i$$

Assuming the light arriving from all directions is the same, the equation can be further simplified. Usually, this term is denoted by ρ_{hd} and represents the *hemispherical directional reflectance*:

$$\rho_{hd} = \int_{H^2(n)} f_r(p, \omega_o, \omega_i) \cos\theta_i \; d\omega_i$$

2.4 Reflection Models

Many models have been developed trying to simulate certain effects. The common models are derived from measured data, certain phenomenons, optics or simulations. I will start with the simple models and basics to reflection first and later on describe the more sophisticated ones.

2.4.1 Lambertion Reflection (perfect diffuse reflection)

Lambert's Cosine Law postulates that an ideal diffuse surface, also known as a "Lambertian" surface, scatters light equally in all directions. By looking at such a surface, the perceived brightness does not change with an altering viewing direction. The reflected number of photons depends on the differential viewing angle and the differential size of the surface area. The following example helps to better understand this issue: An observer looking from the surface normal direction sees a differential area dA with a determined differential solid angle $d\Omega$. Now the observer moves around the differential area keeping the same distance. The intention of the viewer to only keep his differential solid angle enables him to see a bigger area of the surface. Luckily, the surveyed size of the patch is proportional to $\frac{1}{\cos\theta}$. As defined by Lambert, the amount of reflected light in the viewing direction is proportional to the cosine of the angle the viewer is looking at it. This is the key to cancel the cosine in the numerator and denominator of the equation, leading to an equally perceived number of photons independent of the viewing direction. That means for the BRDF a constant value independent of the viewer's direction.

$$f_r(\omega_i, \omega_o) = \frac{\rho}{\pi}$$

2.4.2 Specular Reflection and Transmission

Mirrors reflecting light are the ideal example for specular reflection. Regarding computer graphics this type of reflection plays an important role for the microfacet models described in the following section. In geometrical terms, perfect specular reflection implies incident light being scattered in a single outgoing direction:

$$\theta_o = \theta_i$$

If light arrives at a translucent surface, a certain amount will be refracted. This behavior of direction change is explained by *Snell's law*. At the boundary between the two participating media, the wave direction is altered. The so-called *refraction direction* angle θ_t from the mirrored surface normal $-\vec{n}$ can be derived using *Snell's law*:

$$\eta_i \sin\theta_i = \eta_t \sin\theta_t$$

The change in direction depends on the participating media, which are categorized by their index of refraction. This parameter η tells how much slower light travels inside a specific medium compared to the speed of light measured in vacuum. Unfortunately, the index of refraction depends on the wavelength of the incident light, as it can be observed whenever white light is dispersed by a prism. To save computational time and effort this effect is usually ignored in a ray tracing system.

2.4.3 Fresnel Equations

The previous subsection was denoted to the reflection and transmission direction of light for specular surfaces. Another interesting part of reflection theory is the behavior of semi-transparent surfaces. Augustin-Jean Fresnel developed a model based on the refractive indices of the media the light is traveling through. It is known as the *Fresnel equations* and denotes the reflection coefficient. In general, the result also depends on the polarization of light and additionally on the conductive behavior of the surface. *Conductors* (metals) and *dielectric media* (non-conductors) have their own set of equations due to the fact, that metals are not translucent, but absorb a certain amount of light energy. This effect is controlled by the *absorption coefficient* k. The formula for dielectrics is:

$$\begin{aligned} r_{\parallel} &= \frac{\eta_t \cos\theta_i - \eta_i \cos\theta_t}{\eta_t \cos\theta_i + \eta_i \cos\theta_t} \\ r_{\perp} &= \frac{\eta_i \cos\theta_i - \eta_t \cos\theta_t}{\eta_i \cos\theta_i + \eta_t \cos\theta_t} \end{aligned}$$

Table 2.1 offers many examples for indices of refraction. Refraction is wavelength-dependent, so these values can only be used as an approximation of the behavior of visible light.

Material	η at $\lambda = 589.3nm$	Material	$\lambda = 589.3nm$
Vacuum	1.0	Air at sea level	1.0002926
Water (20°C)	1.333	Ice	1.31
Rock salt	1.516	NaCl	1.544
Bromin	1.661	Diamond	2.419
Cinnabar	3.02	Silicon	4.01

Table 2.1: Measured values for the indices of refraction. Values taken from [23].

For conductors this formula is commonly used:

$$
\begin{aligned}
r_{\parallel} &= \frac{(\eta^2 + k^2)\cos\theta_i{}^2 - 2\eta\cos\theta_i + 1}{(\eta^2 + k^2)\cos\theta_i{}^2 + 2\eta\cos\theta_i + 1} \\
r_{\perp} &= \frac{(\eta^2 + k^2) - 2\eta\cos\theta_i + \cos\theta_i{}^2}{(\eta^2 + k^2) + 2\eta\cos\theta_i + \cos\theta_i{}^2}
\end{aligned}
$$

Some examples for the absorption coefficient and the index of refraction for conductors are given in table 2.2.

Material	η	k
Copper	0.617	2.630
Gold	0.370	2.820
Silver	0.177	3.638
Steel	2.485	3.433

Table 2.2: Measured example values for the indices of refraction and the absorption coefficients of real materials.

Most ray tracers do not calculate with polarized light. So it is assumed that the polarization of light is randomly distributed. This leads to the formula for the “unpolarized“ Fresnel reflection coefficient:

$$F_r = \frac{1}{2}(r_{\parallel} + r_{\perp})$$

2.4.4 Microfacet Models

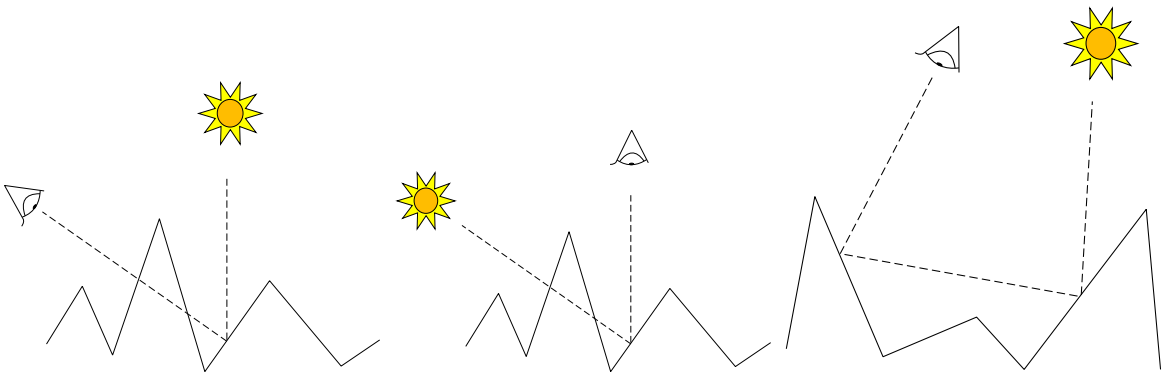
Real life surfaces are not plane at all. With the proper magnification surfaces can be imagined as regions with many pits and falls. The differences in height and angle determine the perceptual roughness an observer recognizes when looking at it. A model using this geometric-optics based approach is called a *microfacet model*. In general, the microfacets are treated as tiny mirrors with respect to the differential area being illuminated. As a consequence, the total amount of light scattered towards the viewer’s direction is determined by the number of mirrors providing the ideal reflection direction. Some local lighting effects may occur which reduce or increase the arriving light. These anomalies can be sorted into three groups, demonstrated in figure 2.4:

Masking A microfacet is not visible to the viewer due to another one occluding it.

Shadowing The inversion to masking: Reflected light does not reach the viewer because a microfacet occludes its path.

Inter-reflection Light reaching the viewer after bouncing between multiple facets.

There exist several ways to reduce these effects by simplifications to the model. A most common one is to assume that all microfacets are *V-shaped* with pits of equal height. This way, it is sufficient to consider the direct microfacet neighbor only. The models always try to use a good trade-off between simulating the anomalies and calculation efficiency.



(a) Masking (b) Shadowing (c) Interreflection

Figure 2.4: Geometric effects caused by microfacets.

2.4.4.1 Oren-Nayar Diffuse Reflection

In 1992 Michael Oren and Shree K. Nayar from Columbia University [20] proposed a *microfacet* model that eliminates some shortcomings of the primitive Lambertian model for diffuse reflection. For this reason, they observed the reflection behavior of real life objects. Their microfacet model is based on a surface consisting of symmetric V-shaped grooves - all being perfect Lambertian reflectors of their own. This is a main difference to other microfacet models simulating specular reflection. The parameter σ is given to modify the roughness of the surface following a Gaussian distribution. It simply changes the standard deviation of the orientation angle. To give some examples, a σ of 0° simulates a perfect Lambertian surface while a σ of 40° simulates a surface appearing much flatter due to the reduced dependency of the surface orientation angle. The final equation for the model is:

$$f_r(\omega_i, \omega_o) = \frac{\rho}{\pi}(A + B\ max(0, \cos\phi_i - \phi_o)\sin\alpha\tan\beta)$$

$$\begin{aligned} A &= 1 - \frac{\sigma^2}{2(\sigma^2 + 0.33)} \\ B &= \frac{0.45\sigma^2}{\sigma^2 + 0.09)} \\ \alpha &= max(\theta_i, \theta_o) \\ \beta &= min(\theta_i, \theta_o) \end{aligned}$$

2.4.4.2 Torrance-Sparrow Microfacet Model

The micorfacet model introduced by Torrance and Sparrow has already been developed in 1967 (see [25]). They use the assumption of surfaces consisting of perfect specular reflecting microfacets. Microfacets having a surface normal equal to the half way vector reflect light towards the viewer. Therefore the surface normals need to fulfill the following equation:

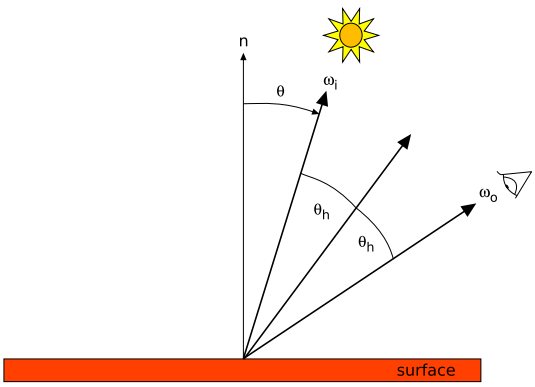


Figure 2.5: Halfway Vector.

$$\omega_h = \frac{\omega_i + \omega_o}{|\omega_i + \omega_o|}$$

The ω_h denotes the halfway vector, i.e. the vector in *half way between* the incident light ω_i and the viewer ω_o as it is shown in figure 2.5. The orientation of the microfacet surface normals is described by a distribution function $D(\omega_h)$ controlling the portion of halfway vectors that perfectly match the previously defined equation. In 1977 Blinn [4] proposed a microfacet distribution function with an exponential falloff starting from the direction of the surface normal. The properly normalized Blinn microfacet distribution with its exponent e is formulated as:

$$D_{Blinn}(\omega_h) = \frac{e+2}{2\pi}(\omega_h \cdot n)^e$$

To further elaborate the model it is assumed that the reflecting microfacets are surfaces in accordance to Fresnel's law of reflection. This means that the Fresnel term $F_r(\omega_o)$ in dependence of the viewer's direction must be added. In order to account for shadowed and masked microfacets an additional term is added, called the *geometric attenuation* term $G(\omega_o, \omega_i)$. The attenuation term consists of two elements. The first tries to simulate the masking effect from 2.4:

$$G_{mask}(\omega_o, \omega_i) = \frac{2(n \cdot \omega_h)(n \cdot \omega_o)}{\omega_o \cdot \omega_h}$$

The shadowing effect can be simulated this way:

$$G_{shadow}(\omega_o, \omega_i) = \frac{2(n \cdot \omega_h)(n \cdot \omega_i)}{\omega_o \cdot \omega_i}$$

Usually a combined version of the terms is used:

$$G(\omega_o, \omega_i) = min(1, min(\frac{2(n \cdot \omega_h)(n \cdot \omega_o)}{\omega_o \cdot \omega_h}, \frac{2(n \cdot \omega_h)(n \cdot \omega_i)}{\omega_o \cdot \omega_i}))$$

Putting it all together, the final equation for the Torrance-Sparrow BRDF is:

$$f_r(p, \omega_o, \omega_i) \quad = \quad \frac{D(\omega_h)G(\omega_o, \omega_i)F_r(\omega_o)}{4\cos\theta_o \cos\theta_i}$$

Chapter 3

Illumination by Lightcuts

The lightcuts framework first presented by Bruce Walter et al. on SIGGRAPH 2005 is a method for efficiently computing realistic illumination. The initial idea was to approximate the direct illumination provided by many point lights by clustering them into groups. These groups are organized as a light tree with the root node representing the accumulated illumination in the whole scene. With an algorithm to compute the bound of the approximation error for each cluster, it is possible to determine a *light cut* through the tree without falling below a predefined error threshold. The whole chapter is denoted to explain the lightcuts system in theory, starting from the basics of light transport.

3.1 The Rendering Equation Revisited

The previous chapter introduced the rendering equation and the properties of surface reflection theory in great detail. For most scenes it is practically impossible to solve the light transport equation in its full generality. Therefore, an algorithm is needed to calculate an approximation for the lights integral that gives the reflected radiance. The algorithm of choice is *direct lighting* which is a simple approximation for the LTE:

$$L_o(p, \vec{\omega_o}) = L_e(p, \vec{\omega_o}) + \int_S f_r(p, \vec{\omega_o}, \vec{\omega_i}) L_d(p, \vec{\omega_i}) \cos\theta_i \; d\vec{\omega_i}$$

The direct lighting integrator represents this part of the previous equation:

$$\int_S f_r(p, \vec{\omega_o}, \vec{\omega_i}) L_d(p, \vec{\omega_i}) \cos\theta_i \; d\vec{\omega_i}$$

Gladly, reflections from the light sources are independent of each other. As a consequence it is possible to break it down into a sum over all lights:

$$\sum_l \int_S f_r(p, \vec{\omega_o}, \vec{\omega_i}) L_{d(l)}(p, \vec{\omega_i}) \cos\theta_i \; d\vec{\omega_i}$$

The ability to evaluate the contribution of each light independently is the key requirement for the lightcuts algorithm.

3.2 The Lightcuts Algorithm

Bruce Walter and his associates reformulated the previously mentioned direct lighting integral. They introduce four terms for the final lighting computation:

M (material term) The BRDF of the surface multiplied by $\cos\theta_i$

G (geometric term) Attenuation of light due to geometrical distance.

V (visibility term) Visibility of the light source from the regarded point on the surface.

I (intensity) Emitted power from the light.

The point x is illuminated by a set of lights $\mathbb{L}$, gathering the contribution of each individual light:

$$L_x = \sum_{i \in \mathbb{L}} M_{x,i} G_{x,i} V_{x,i} I_i$$

Since accurate approximation of global illumination requires a large set of lights, it is very time-consuming to evaluate the illumination by using all lights. Lightcuts provides a scalable solution for this problem by arranging lights into clusters. The radiance reflected into the viewers direction is expressed as estimate from all cluster light sources $\mathbb{C}$:

$$L_x \approx \sum_{c \in \mathbb{C}} M_{x,c} G_{x,c} V_{x,c} I_c$$

To bring efficiency to the next level, the clustering of light sources is organized, referred to as *hierarchical light tree*. This step is done by preprocessing all the lights available in the scene, which is explained below in this chapter. When the ray tracing algorithm shoots rays into the scene and hits a surface, it is essential to have a method to find a light partition contributing the greatest amount of radiance for the considered surface location. Therefore an error estimation routine it used to compute an upper bound on the error introduced by using a light cluster compared to the individual lights. The direct lighting integral L_c for a cluster of lights is the sum of all radiance added by the lights in the cluster:

$$L_c = \sum_{i \in \mathbb{C}} M_{x,i} G_{x,i} V_{x,i} I_i$$

A good approximation for this term can be found by creating a representative light r for the cluster.

$$L_r \approx M_{x,r} G_{x,r} V_{x,r} \sum_{i \in \mathbb{C}} I_i$$

It is situated inside of the cluster and introduces a certain amount of error BE_c. The crucial part of the algorithm is fast and precise error estimation. If the calculated error yields a value above the predefined perceptual visibility threshold, a refinement of the cluster must be initiated. This is done by ascending the hierarchical light tree and performing the error estimation again for the child nodes.

These refinement steps are repeated until the error estimate drops beneath the threshold. For leaf nodes in the tree the error is always 0 and does not need to be evaluated. The selected nodes represent a cut through the light tree, thus it is called *lightcuts*. The source code describing this algorithm can be obtained from Appendix A.3. In the subsequent sections I will explain the process of error estimation for light clusters and the structure of the light tree.

3.3 Lightcuts Bounding Cluster Error Estimation

An upper bound for the error introduced by using a cluster instead of each individual light has to be approximated. The error is defined by the term BE_c and is the difference between the direct lighting integral L_c and the direct lighting approximation L_r using a representative light:

$$BE_c = |L_c - L_r|$$

Since direct lighting works by multiplying M, G, V and I it is necessary to find the error's upper bound introduced by any of these terms. The intensity term I is also part of the error estimation, because the magnitude of incident radiance determines how much error could possibly be done. The error bounds have to be generated for a surface position x and a light cluster $\mathbb{C}$.

The visibility term V defines the visibility of a light source. It is 1, if a light is visible from the surface position. Otherwise it is 0. In case of semi transparent surfaces, a fraction value is also conceivable. The term's upper bound needs to be exact, which is not easy to achieve for arbitrary scenes. Therefore all lights are declared potentially visible. This means using the trivial upper bound of 1.0:

$$VE = 1.0$$

The geometric error term G describes the attenuation each light in the cluster suffers from. For point lights the term depends on the distance d between the surface position x and each light's position $lpos_i$ in the cluster $\mathbb{C}$:

$$G_{x,i} = \frac{1}{d_i} = \frac{1}{|lpos_i - x|^2}$$

An upper bound for the attenuation term can be found by taking the minimum possible distance:

$$max(G_x) = \frac{1}{min(d_i)}$$

For large clusters of point lights this is hard to evaluate, so the minimal distance is approximated using a bounding box, which inherits all the light sources. Figure 3.3 shows a hit surface and the vector with minimal distance from the surface position to the light cluster box.
The minimal distance from the surface position to the axis aligned bounding box (AABB) is then determined by calculating the minimal distance for each geometric dimension. This means the x-value of the surface position p is compared to the x-interval $X[x_{min}; x_{max}]$ of the bounding box. If the coordinate is outside the interval, the point with the shortest distance from p is at the limit of the interval closest to the coordinates of p. Otherwise the coordinate is between the limits of the interval.

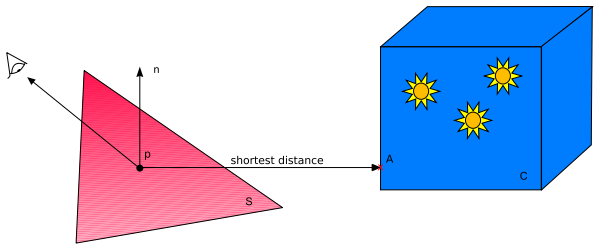


Figure 3.1: The minimal distance between the surface position p and the light cluster C is used to calculate the geometric term's upper bound.

In this case, simply the value from p is taken over. The hit point A on the bounding box surface consequently has the coordinates:

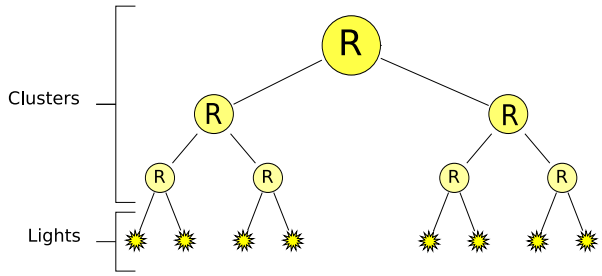
$$A = \{Closest(x), Closest(y), Closest(z)\}$$

The geometric bound for the infinite area light is 1.0, because there is no attenuation with increasing distance.

The material term M inherits both the cosine falloff due to the incident light direction as well as the reflection properties of a material defined by the BRDF of the hit surface position p. Determining an upper bound for the cosine term means to calculate the minimum angle between the surface normal at p and the bounding structure for the clustered light sources. For point lights this means calculating a bound for a bounding box. The special infinite area light uses a bounding cap. The idea behind it and notes on the implementation of the bounding process will be described in the subsequent chapter. Creating upper bounds on the BRDF is an even more challenging task. My implementation supports bounds on some models already existing in the PBRT rendering system. Starting with the Lambertian diffuse reflection, which is trivial to bound, I developed a bounding mechanism for the microfacet Oren-Nayar diffuse reflection model and for the Torrence-Sparrow microfacet model. The Torrence-Sparrow microfacet model contains the viewer dependent halfway vector. The paper [26] offered a reasonable solution for bounding the halfway vector. This will be explained in chapter 5.

3.4 Light Tree

Light trees are hierarchical data storages for lightcuts compatible light sources. In my implementation there are two different types of light trees. One tree is generated for an infinite area light and an additional one for the point lights in the scene. These two types of light trees have many things in common, so I will describe the general functionality first. The light tree consists of interior nodes and leaves. Each leaf is an individual light source, not different to those used by general direct lighting calculations. Interior nodes in the tree are light clusters representing all the lights below. The nodes contain a *representative light* that embodies the total emitted radiance of its child nodes and a bounding structure to help approximating the geometric and material term in the *error estimation* routine. An example tree with 8 lights can be examined in figure 3.4.



(a) A light tree with 8 individual lights. The interior nodes denoted by R represent the lights below. The emitted radiance of a node is the accumulation of its child nodes.

3.4.1 Representative Lights

A representative light emits the accumulated radiance of its children. The total emission of a cluster is defined as the sum of all individual lights the cluster represents:

$$I_R = \sum_{i \in \mathbb{C}} I_i$$

The point light tree uses an AABB to represent the spatial extent of the incorporated light sources. Axis aligned bounding boxes are defined by their limit in each dimension. The figure 3.4.1 shows an axis aligned bounding box for a 2 dimensional setting:

$$BBox(x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, z_{max})$$

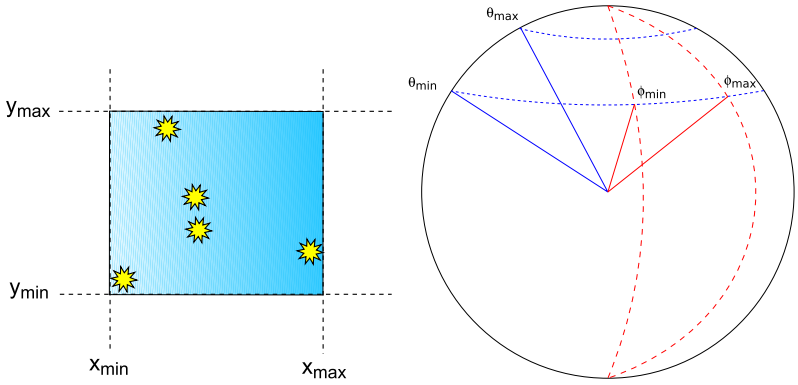
Infinite area lights (IAL) do not need to store bounding boxes. The intended use of an IAL is to define radiance arriving from all possible directions. The implementation of the infinite area light tree uses a variable number of directional light sources to replace common infinite area light source sample algorithms. Therefore, representative directional lights store the radiance arriving from a limited area on the unit sphere. The area is expressed by two intervals, one for the zenith distance θ and one for the azimuth angle ϕ:

$$BDir(\theta_{min}, \theta_{max}, \phi_{min}, \phi_{max})$$

The next sections describe two methods of creating a light tree for point lights. Mainly there are two possibilities for clustering: The bottom-up approach, which was described in the initial lightcuts paper and a top-down attempt mentioned in the *multidimensional lightcuts* paper from 2006 [27]. I will show that the latter leads to a much better light tree, than the first one with its *similarity metric*.

3.4.2 Greedy Bottom-up Light Tree Generation

The first lightcuts paper [28] proposed to build the light tree in bottom to top manner by clustering lights following a similarity metric. This metric was defined by the emitted radiance $I_{\mathbb{C}}$, the diagonal length of the cluster bounding box $\alpha_{\mathbb{C}}$, the half-angle of the bounding cone $\beta_{\mathbb{C}}$ and a constant c to control the relation between spatial and directional similarity:

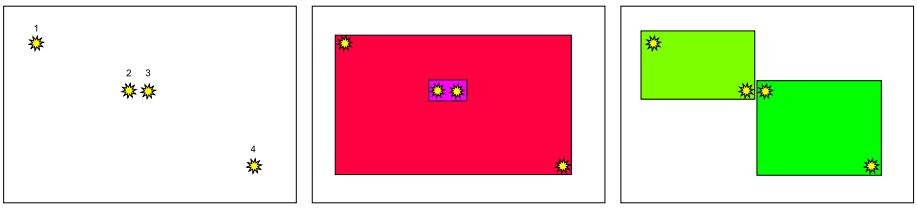


(b) Lights inherited by an axis aligned 2D Bounding Box.

(c) Represents incident light directions in the intervals for θ and ϕ.

$$I_{\mathbb{C}}(\alpha_{\mathbb{C}}^2 + c^2(1 - \cos\beta_{\mathbb{C}})^2)$$

The algorithm was not described any further. It was only mentioned to be a greedy clustering approach trying to combine lights with minimal value due to a similarity metric. This method of clustering is very slow, because you need to compare each light with all remaining lights in order to find the minimal similarity metric value. For the first level of clusters already $\frac{n^2}{2}$ comparisons have to be done and stored for efficiency. For any level of the tree, the number of elements is bisected, so in the end by obeying the result of the arithmetic series of $1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots$ the number of comparisons reaches n^2. Nevertheless, I implemented the algorithm to see how it actually performs. The result was a slow evaluation and unnecessary big clusters. Whenever two elements are grouped due to their similarity, other pairs might become less optimal. Figure 3.2 demonstrates the problem.



(d) 4 point lights in the scene.

(e) When the two lights nearest to each other get grouped first, the created clusters are too big.

(f) The optimal clustering.

Figure 3.2: Possible problems occurring by light clustering.

Assuming that all point light sources have an equal emission of light, the similarity metric tells, that the two lights in the middle are grouped first. After taking out the two lights the two ones left get clustered to an unnecessary big cluster. Building a minimum spanning tree could possibly help to solve this problem, but there is a more reasonable approach for the generation of clusters.

3.4.3 kd Light Tree

The kd-tree is a very well suited top-down data structure to partition a big number of light sources into clusters. Remembering the lightcuts system's operating mode, it should be the main task to generate clusters, whose error estimates soon fall below the perceptual threshold. Considering point lights the main factors of influence are the distance to a light source, the minimum angle between the surface normal and the cluster and its intensity. The squared distance is part of the geometric term of the lightcuts error estimate. For this reason a binary space subdivision algorithm, e.g. octree space partitioning could be used. The octree algorithm splits the space into eight equally big regions in each step of execution. This is achieved by splitting with three axis perpendicular planes, selecting the position of the plan as the middle of the currently considered box. In contrast to this, the kd-tree nodes only contain one axis perpendicular splitting plane, not necessarily situated at the median of the boxes geometric extent. As a consequence, during kd-tree generation a heuristic can be used to determine which split position offers the best trade-off depending on the future usage. I implemented it as a special balanced kd-tree where each leaf node is at the same level of the tree and only the leaves store real light sources. Wikipedia tells [24], that these special trees are also called kd-tries. The difference between a kd-trie and a kd-tree is that interior nodes of kd-trie do not store data. This has consequences for tree building and data storage. The next chapter will cover this issue thoroughly.

3.4.4 kd Point Light Trie Construction

In the beginning it is necessary to define k, which is the number of dimensions the initial data has. For the purpose of generating a point light tree, there are three dimensions. These are the x, y and z coordinate of the point lights positions.

The algorithm building the tree works this way:

- determine split dimension.
- determine split position.
- elements left of the split position must be smaller than those on the right.
- recursive call of the algorithm for the split elements.

There are several strategies possible to determine, which axis should be split first. The simplest one is to split in a certain predefined order. Dealing with geometric positions, a better strategy is to split the axis with the maximum extent first. This reduces the volume of the bounding box by the maximum immediately known value.
Finding the split position is a similar problem. The use of the *spatial median* as primitive splitting strategy leads to a cubically subdivisioned space, which generates a good kd-tree for many scenes. An even simpler method is the usage of the *object median* splitting strategy. It generate a splitting plane with each side containing the same or similar number of objects. This is the splitting rule of choice, since it is easy to implement and generates a balanced tree. Splitting heuristics for kd-trees could be a chapter on its own. Depending on the usage, there exist many ideas how splitting could be controlled. Since these models were mainly developed for ray tracing, they operate with triangles and cannot be used directly to create light trees. There is still some room for further optimization.

A splitting heuristic that detects particular small regions with many lights would be fine. Since the algorithm works impressively well, this part has not been a priority so far. For an optimal light tree, the concept would be to try finding a split plane, whose children have a similar estimated error. This could be achieved by trying to generate clusters with similar intensity and size. For scene files with a large number of equally distributed point light sources the standard kd-trie satisfies this proposal.

Figure 3.3 shows how light tree generation works.

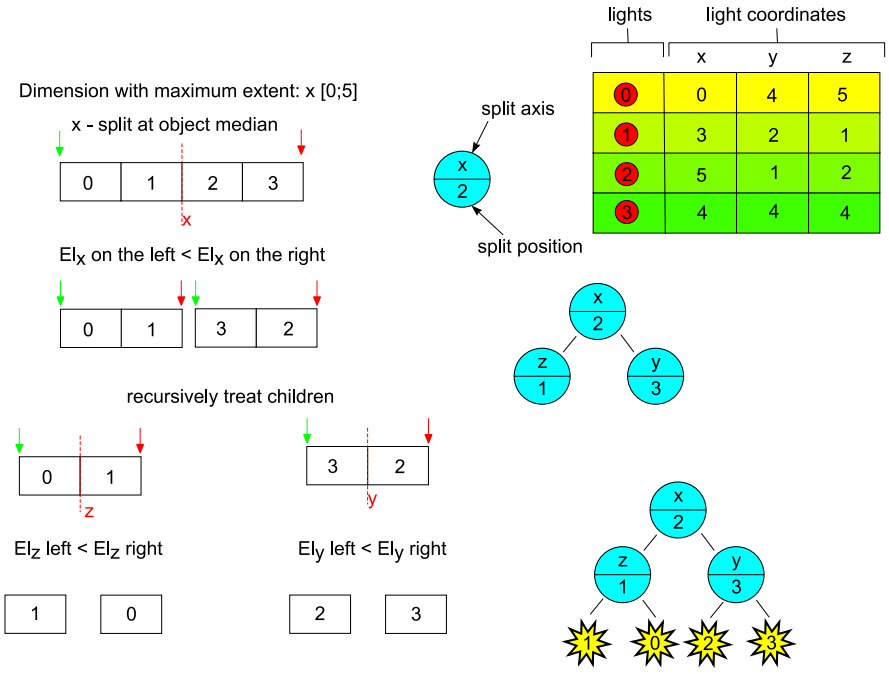


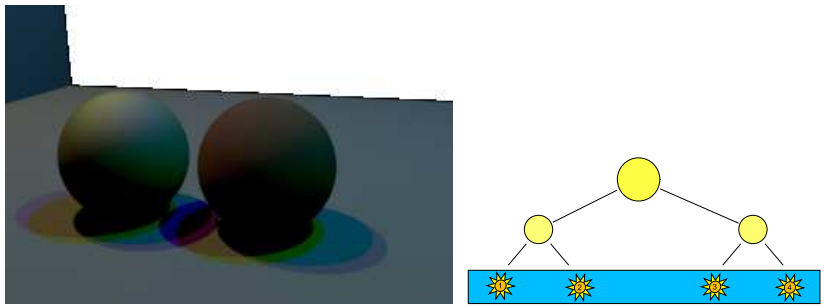
Figure 3.3: Overview of the construction algorithm of a kd-trie: First, the split dimension is determined, afterwards the split position. Regrouping of the elements to ensure, that any element in the left branch has a smaller value with respect to the split dimension compared to those in the right branch. A node is created and the algorithm calls itself recursively for both branches.

The algorithm used for light tree generation starts to determine the split axis. The dimension with the maximum extent is selected. Therefore, all elements must be compared for their minimum and maximum coordinates for each dimension. For the error estimation of a light cluster, a bounding box will be needed. It is also possible to generate a bounding box for all lights contained in this cluster. As a side effect, the box can be queried for its dimension with maximum extent. It is nevertheless necessary to iterate over all elements in one step. Each split creates a new node of the tree. The nodes contain the split information, which is the *split axis*, *split position*, the data necessary for the lightcuts error estimation, the bounding box and the representative light. The split position is chosen by selecting the median of elements enclosed by the node. Afterwards, it is necessary to make sure that all elements in the left branch are smaller than the elements in the right branch with respect to the split dimension. This action is cheaper than sorting them in ascending order. The algorithm can now call itself recursively for the just created child branches. The recursion stops, when there is only one element left. This is a leaf of the tree and contains only the real light source. For real lights,

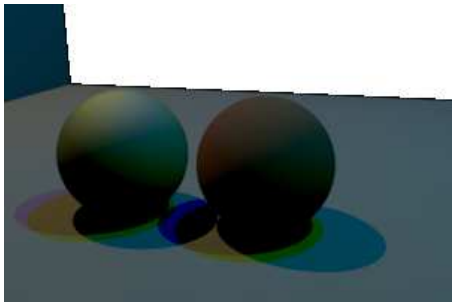
no bounding construct or representative light is necessary. The source code for kd-trie generation is presented in appendix 2.

3.5 Lightcuts in Action

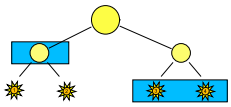
Finally, the selection of nodes leads to a cut across the light tree. To demonstrate the error for some cases I created some renderings for a scene with a very reduced lighting setting. Four point lights are present, two on each side of the spheres. The following pictures show the changes and introduced error zones by clustering the lights.



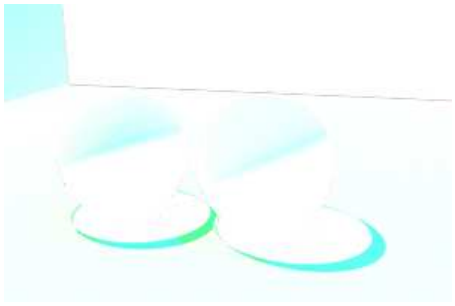
(a) The scene rendered with all lights. (b) The light tree used for rendering.



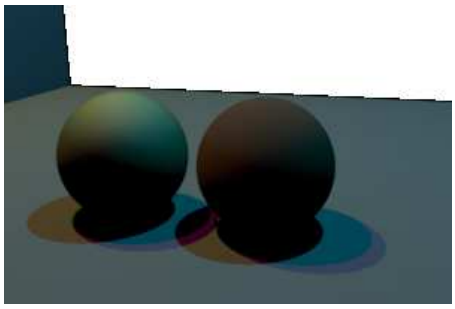
(c) The scene rendered with one representative light and two real lights



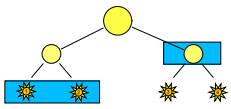
(d) Light tree



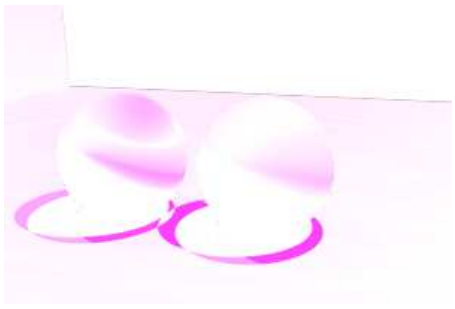
(e) Error zones



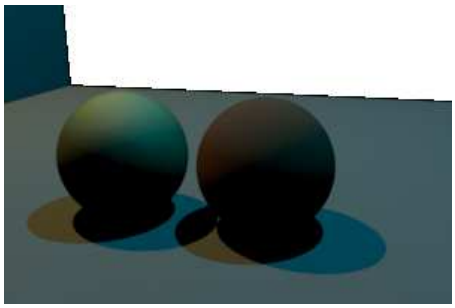
(f) The scene rendered with two real lights and one representative light.



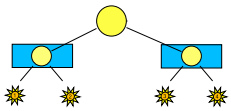
(g) Light tree



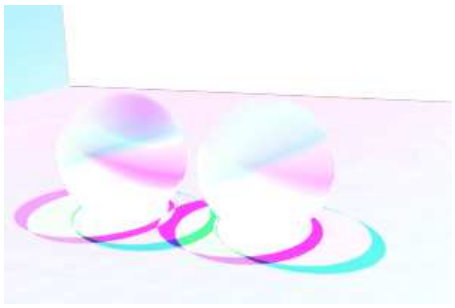
(h) Error zones



(i) The scene rendered with two representative lights.



(j) Light tree



(k) Error zones

Chapter 4

Bounding the $\cos\theta$

This chapter explains how the cosine of θ_i can be bound for incident light directions. In the lighting equation, the $\cos\theta$ is an important part. It describes the scattering of photons due to incident angle. This phenomenon has already been clarified in 2.3.2. The bound for the cosine of θ is a very grateful term, because it can be computed efficiently and reduces the maximum possible error of the lightcuts error estimation term significantly for laterally situated light clusters.

4.1 Algorithm for the Infinite Area Light Source

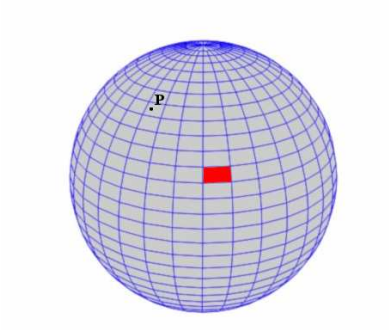


Figure 4.1: The point P is the normal direction of the hit surface. The red patch is the set of considered incident light directions ω_i.

The goal is to calculate a cheap and tight bound on the minimum angle between the vector of the surface normal p and the incident light direction ω_i. Not to be confused with the term used for the spherical angle θ, the minimum angle is denoted as α in this section. The following equations will solve the problem mathematically. For spherical coordinates the θ is in the range $[0; \pi)$. In this interval the cosine function is invertible, because it is continuous and strictly monotonic decreasing. The direct relationship between the angle and its cosine allows using both expressions synonymously. This means, bounding the minimal angle is equivalent to bound the maximum cosine.

$$\theta_{min} \sim max(cos(\theta_i))$$

The same applies to any angle between two vectors like the α, the angle between the patch and p. Since both are situated on the unit sphere and thus already have a length of 1.0, the cosine of the angle is the dot product:

$$p \cdot \omega_i$$

To calculate the bound, it is necessary to find the nearest position inside the patch with respect to p.

4.2 Mathematical Solution

The mathematical solution for the problem is based on the dot product between the point p and the set of incident light directions:

$$\cos\alpha = p \cdot \omega_i = p_x * i_x + p_y * i_y + p_z * i_z$$

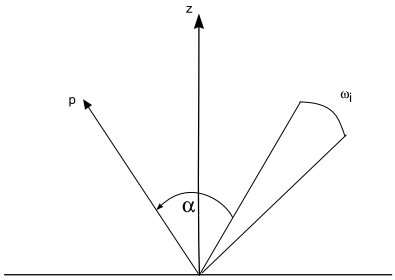


Figure 4.2: $alpha$ is the angle between the surface normal p and the incident light direction ω_i at a surface location. The coordinate system is in world space.

It is possible to omit the trivial case, because this can be computed previously. If the vector p directs inside the set of incident light directions ω_i, the result for $\cos\alpha$ equals 1.0. Otherwise the nearest vector is situated somewhere at the outer edge of the cluster. The cosine of the angle between an arbitrary point P on the sphere and the outer edges of the patch can be described by the following four equations. Each equation solves the problem for one of the edges. The correct solution is the maximum result, i.e. the minimum θ obtained by any of these for the given intervals.

$$\begin{aligned}
D_1(\theta) &= p_x * \sin\theta * \cos\phi_{min} + p_y * \sin\theta * \sin\phi_{min} + p_z * \cos\theta \\
D_2(\theta) &= p_x * \sin\theta * \cos\phi_{max} + p_y * \sin\theta * \sin\phi_{max} + p_z * \cos\theta \\
D_3(\phi) &= p_x * \sin\theta_{min} * \cos\phi + p_y * \sin\theta_{min} * \sin\phi + p_z * \cos\theta_{min} \\
D_4(\phi) &= p_x * \sin\theta_{max} * \cos\phi + p_y * \sin\theta_{max} * \sin\phi + p_z * \cos\theta_{max}
\end{aligned}$$

To determine whether there is a local maximum in the considered range of possible θ and ϕ it is necessary to calculate the first derivative:

$$D_1'(\theta) = p_x * \cos\phi_{min} * \cos\theta + p_y * \sin\phi_{min} * \cos\theta - p_z * \sin\theta \quad (4.1)$$

$$D_3'(\phi) = -p_x * \sin\theta_{min} * \sin\phi + p_y * \sin\theta_{min} * \cos\phi \quad (4.2)$$

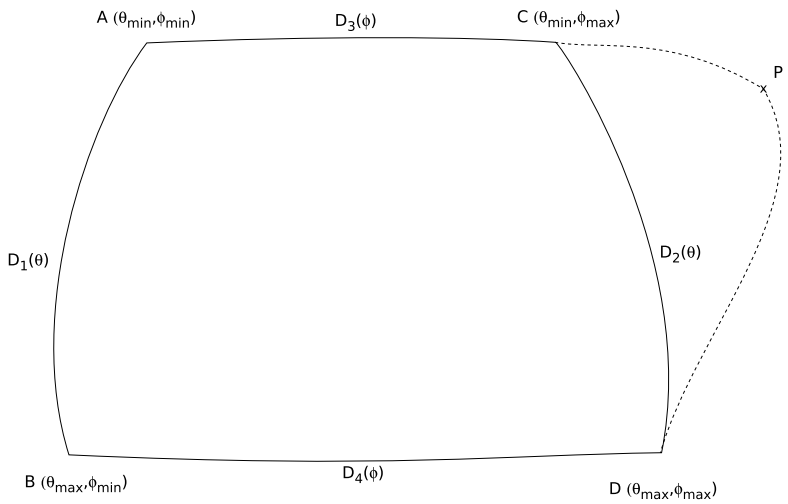


Figure 4.3: The geodetic distance between a point P and a spherical patch $ABCD$ is equal to the included angle.

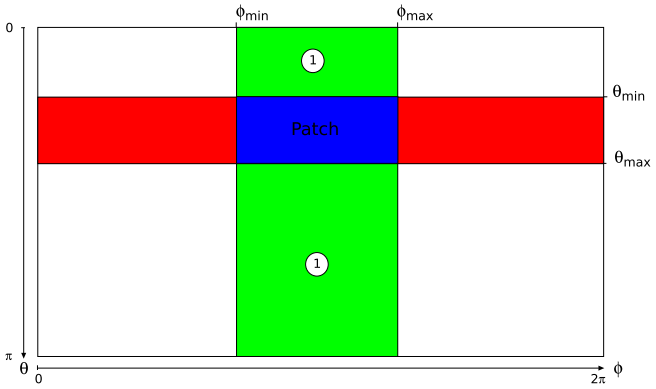
D_2 and D_4 can be solved analogously. The extrema are determined by setting the equations equal to 0 and solve them for θ assuming that $a = p_x * \cos\phi_{min}$ and $b = p_y * \sin\phi_{min}$ and $c = p_z$:

$$\begin{aligned}
D_1'(\theta) &= 0 \\
0 &= a * \cos\theta + b * \cos\theta - c * \sin\theta \\
0 &= \underbrace{(a+b)}_{=d}\cos\theta - c * \sin\theta \\
c * \sin\theta &= d * \cos\theta \\
c^2 * (1 - \cos^2\theta) &= d^2 * \cos^2\theta \\
cos^2\theta &= \frac{c^2}{c^2 + d^2} \\
\cos\theta &= \pm\frac{c}{\sqrt{c^2 + d^2}} \\
\Rightarrow \theta &= \pm\arccos\left(\pm\frac{c}{\sqrt{c^2 + d^2}}\right)
\end{aligned}$$

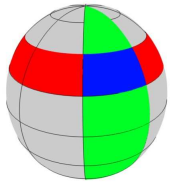
It is now possible to test if an extremum exists inside the θ interval of the patch. If there is none, the function is monotone and one of the boundary points is the shortest distance. The same applies for the D_3' and D_4' equations, assuming that $e = p_x * \sin\theta_{min}$ and $f = p_y * \sin\theta_{min}$.

$$\begin{aligned}
D_3'(\phi) &= 0 \\
0 &= f\cos\phi - e\sin\phi \\
\Rightarrow \phi &= \pm\arctan\left(\pm\frac{e}{\sqrt{f^2 + e^2}}\right)
\end{aligned}$$

These results lead to the conclusion, that it is possible but not fast to calculate the nearest distance from an arbitrary point to the patch. Computing the extrema must be performed for any edge to determine, if the shortest distance is in between the boundary points. Luckily, there are some special cases for whom it is possible to evaluate a quick and easy solution. If the sphere is represented by a latitude/longitude mapping as it is usually done for environment mapping, a section of the sphere can be marked where evaluation is easy.



(a) The sphere as 2D ll-texture.



(b) The colored areas attached to the sphere.

Figure 4.4: The green area marked with 1 can be treated in an efficient way.

If the point P is inside the area marked with 1, the point Q with the nearest distance to P has the same ϕ value as Q. This can be observed in figure 4.4). It is possible to proof that the equation describing the distance between the point P and the edge next to it has an extremum at Q. If the equation from 4.2 is used and $\phi_Q = \phi_P$, this leads to:

$$\begin{aligned} D_3'(\phi) &= 0 \\ 0 &= -\sin\theta\cos\phi_P \sin\theta_{min}\sin\phi_Q + \sin\theta\sin\phi_P\sin\theta_{min}\cos\phi_Q \\ 0 &= 0 \end{aligned}$$

For the 1-marked area the cosine of α is the maximum dot product of the vector p and the two q vectors, one for the upper and one for the lower patch boundary:

$$max(\cos\alpha) = max(q_1 \cdot p, q_2 \cdot p)$$

.
It is possible to simplify the dot products to:

$$max(\cos\alpha) = max(\cos(\theta_P - \theta_{min}), \cos(\theta_P - \theta_{max}))$$

It depends on the implementation and the cached data which one is evaluated faster. For the remaining parts of the sphere it is more difficult to find the maximum cosine. Most of the time one of the vertices A, B, C and D has the minimum angular distance with respect to the point P. Especially if

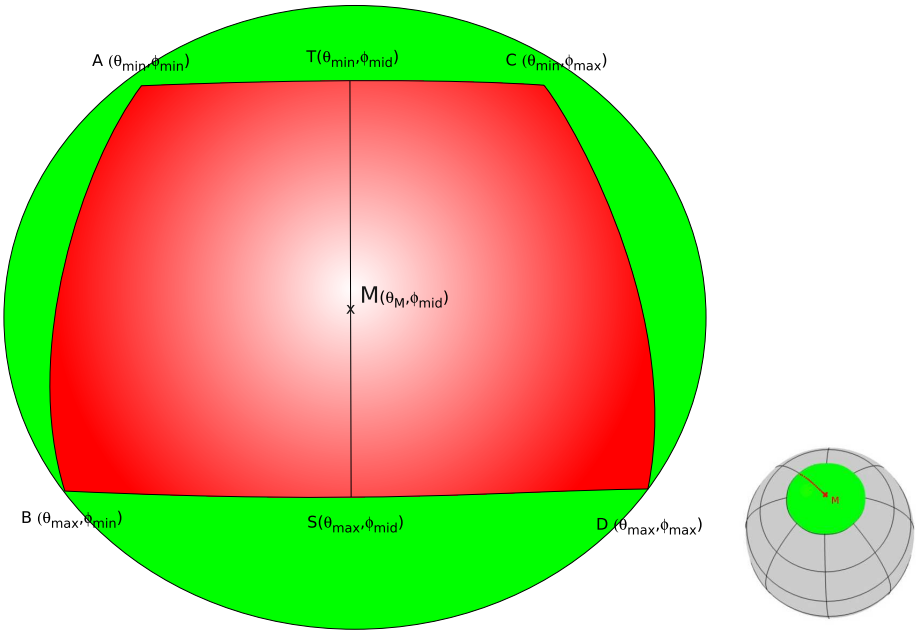
P is located in the red marked area, the point Q with the nearest distance is sometimes between the boundary points. Unless the equations introduced in 4.1 are solved for determination of the extrema, Qs position cannot be told. Altogether, it is complex to find an exact solution for an upper bound on $\cos\alpha$. Therefore, it seems convenient to search for a fast approximation. Previously, I already thought of bounding cones for clustering distant lights for the lightcuts algorithm. A similar clustering approach can be used for directions on the unit sphere. The structure will be called *bounding cap* and is explained in the next section.

4.2.1 Bounding Cap Approximation

The *bounding cap* is a very simple bounding structure. It represents a cap-like region on the unit sphere defined by a midpoint M and an apex angle β. This facilitates computing, because any calculation can be done with respect to the midpoint. Afterwards, the apex angle is regarded to finish the calculation. In comparison to the introduced patch algorithm, a bound for the maximum cosine can be defined by just two lines:

$$\begin{aligned} \alpha_m &= \arccos(p \cdot m) \\ max(\cos\alpha) &= \cos(\alpha_m - \beta) \end{aligned}$$

Picture 4.5 visualises the idea of creating a bounding cap which inherits the spherical patch.



(a) The green bounding cap fits the red patch.

(b) Spherical view of the bounding cap with midpoint M and apex angle β.

Figure 4.5: Schematics for the bounding cap approximation.

The bounding cap has some prerequisites that must be satisfied:

- The patch is completely inside the cone.
- The midpoint M is situated inside the patch.
- The distance from M to the left and right border is the same, because the patch is a symmetric figure: $m \cdot a = m \cdot c, m \cdot b = m \cdot d$.
- The midpoint M has the spherical coordinates θ_M and $\phi_M = \phi_{mid} = \frac{\phi_{min}+\phi_{max}}{2.0}$.
- The apex angle of the cone does not exceed π or even $\frac{\pi}{2}$, if spherical patches are not allowed to cross the equator.

4.2.2 Creation of a Tight Bounding Cap

It is very important to find a tight bounding cap with a preferably large overlapping area of the spherical patch and the bounding cap. In order to find a midpoint M allowing a small apex angle, it is possible to simplify the problem: Find the midpoint for the circumcircle of a triangle created by three vertices of the patch and project this midpoint to the unit sphere afterwards. Scaling the vector does not change the relation of the distances towards each other. Figure 4.6 shows the idea. It is based on the assumption that the cap covers the patch entirely if all its vertices are inside. Gladly, it does not matter which vertices of the patch are used to create the triangle. The circumcenter of triangle is determined by the help of the median line of any edge. For all four possible triangles the median line of AB or CD is participating, so the result stays the same.

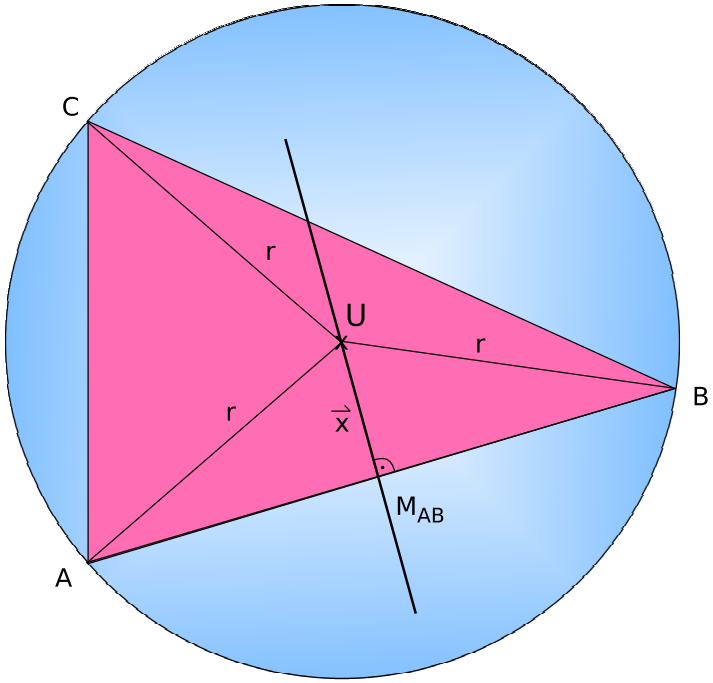


Figure 4.6: The circumcircle of the $\triangle$ ABC.

The pythagorean theorem helps to determine the distance $\|x\|$ from the base line to the circumcenter U:

$$\|x\| = \sqrt{r^2 - \left(\frac{\|b-a\|}{2}\right)^2} \tag{4.3}$$

r is the radius of the circumcircle and can be expressed in dependency of the triangles' surface area S:

$$r = \frac{\|b-a\|\|c-b\|\|a-c\|}{4S}$$

U's position can be expressed as:

$$U = a + \frac{b-a}{2} + x$$

The vector x can be obtained by rotating the base line of the triangle by 90 degrees to the left and scaling it by the length obtained in equation 4.3. Finally, the vector u is scaled to a point situated at the bounding sphere by normalizing:

$$M = \frac{u}{\|u\|}$$

There is one pitfall left, if the initial ϕ extent of the considered spherical patch is larger than half the sphere. This leads to a midpoint on the wrong side of the origin. For implementing the algorithm, it is important to keep this in mind.

4.2.3 Algorithm for the oriented bounding box

Point light sources are clustered by generating bounding boxes to represent the lights' spread. Similar to the previously explained bounding procedure, it is also necessary to find the minimum angle between the surface normal n and a bounding box. Therefore the problem is transformed into a coordinate system, where the hit surface position is the origin and the z-axis is the direction of the surface normal. The problem is visualized by the subsequent figure:
The method to determine the minimum angle between the bounding box C and the hit surface normal was introduced in the paper "Notes on the Ward BRDF" by Bruce Walter [26]. I will present his ideas and calculations in the following section and enhance them.

4.2.3.1 Minimum Angle - Maximum Cosine

Bruce Walter describes an easy way to calculate a bound for the angle α with a minor cutback in exactness. This disadvantage is compensated by the ability to handle arbitrary rotated boxes. Figure 4.7 shows a schematic where the minimum incident light direction with respect to the bounding box is denoted by the red x. The maximum cosine of θ can be written as the dot product between the surface normal and the normalized incident light direction:

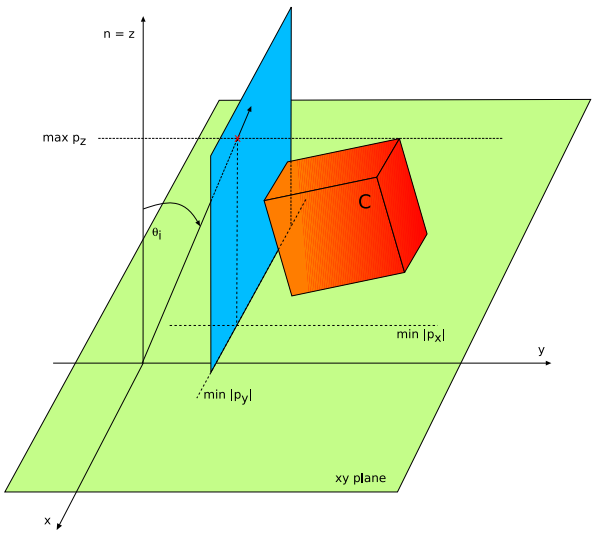


Figure 4.7: The surface normal is aligned with the z-axis. The bounding box C is anywhere but subtending the z-axis.

$$\cos\theta_i = n \cdot i$$

In a coordinate system where the orientation of the surface normal is the same as the z-axis the problem can be reformulated as:

$$\cos\theta_i = \frac{i_z}{\sqrt{i_x^2 + i_y^2 + i_z^2}}$$

Replacing i_z by its maximum possible value for the bounding box helps to maximize the term in a first step:

$$\cos\theta_{i_{min}} \leq \frac{max(i_z)}{\sqrt{i_x^2 + i_y^2 + max(i_z)^2}}$$

Depending on i_z the closest or furthest i_x and i_y in relation to the z-axis is selected to minimize or maximize the denominator. It is noteworthy, that expressions like $max(i_x)^2$ denote the maximum obtainable squared value.

$$\cos\theta_{i_{min}} \leq \begin{cases} \frac{max(i_z)}{\sqrt{min(i_x)^2+min(i_y)^2+(max(i_z))^2}} & \text{if } max(i_z) \geq 0 \\ \frac{max(i_z)}{\sqrt{max(i_x)^2+max(i_y)^2+(max(i_z))^2}} & \text{otherwise} \end{cases}$$

The trivial case can be checked previously and sorted out: if the bounding box subtends the positive z-axis, then the cosine of θ equals 1.0, of course.

4.2.3.2 Maximum Angle - Minimum Cosine

The minimum cosine can be computed in an analogous way. Therefore the minimum i_z has to be determined first. Then again, it depends on its sign, if the closest or furthest values of i_x and i_y are used for minimizing the expression. The following drawing 4.8 shows examples for both cases:

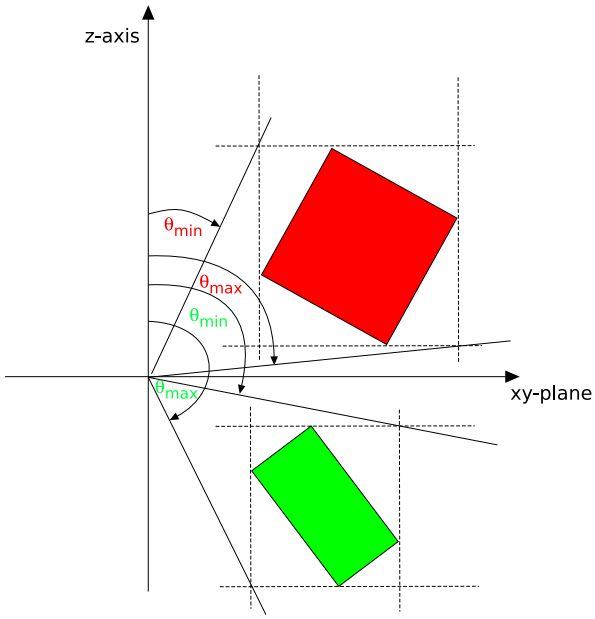


Figure 4.8: Bounds for the point light cluster boxes. The xy-plane is mapped to the horizontal axis.

The formula for calculating the minimum cosine is denoted as:

$$\cos\theta_{i_{max}} \geq \begin{cases} \frac{min(i_z)}{\sqrt{max(i_x)^2+max(i_y)^2+(min(i_z))^2}} & \text{if } max(i_z) \geq 0 \\ \frac{min(i_z)}{\sqrt{min(i_x)^2+min(i_y)^2+(min(i_z))^2}} & \text{otherwise} \end{cases}$$

Again, the trivial case can be handled separately, if the bounding box subtends the negative z-axis. For both the maximum and minimum angle it is not sufficient to look at the vertices of the oriented bounding box. It is furthermore a necessity to consider the intervals created by vertices. Since the oriented box is situated at an arbitrary position in the local coordinate system this can be easily evaluated by iterating over the eight vertices of the box. Knowing the extent with respect to each dimension allows an easy evaluation of the minimum and maximum values of i_x, i_y and i_z.

Chapter 5

Bounding the BRDF

This chapter is denoted to explain how the most difficult part of the lightcuts error estimation computes an upper bound on the *material term* M of the lightcut integrator. This term is $\cos\theta$ times the BRDF of the surface. The previous chapter dealt already with bounding the $\cos\theta$ term, the maximum and minimum values for the cosine were calculated. For this chapter, these values are taken for granted. All BRDFs depend on the reflection direction ω_o and the incident light direction ω_i. Due to light clustering the incident light arrives from a set of directions. To bound the BRDF it is necessary to find a local maximum of the BRDF for the given intervals. The task is easier, if the function is monotone or at least monotone for the considered interval. The first step to bound the BRDF is to calculate the extent of the interval of the incident light direction. For the zenith angle θ this was already demonstrated in the previous chapter. Many BRDFs are symmetric around the z-axis. Then it is sufficient to calculate the θ interval. If a BRDF depends on the viewers direction, things really get complicated. Usually this involves the usage of the half-angle vector. I will demonstrate a method which allows bounding the half-angle vector. The subsequent sections address different types of BRDFs with increasing complexity.

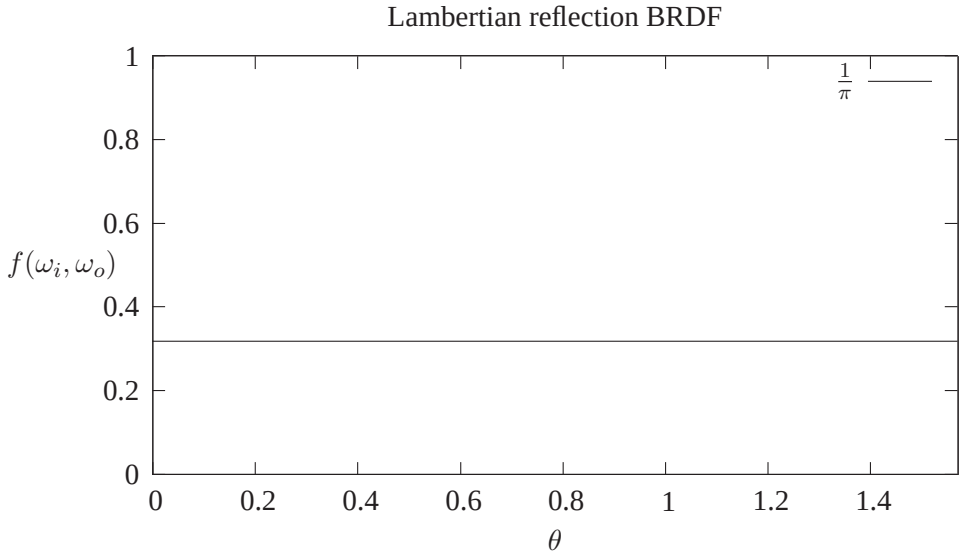
5.1 Lambertian

The Lambertian reflection is constant over the hemisphere of directions. Because of it simplicity, I used the Lambertian BRDF as a kind of debug BRDF during development of the lightcuts integrator. The value returned by the Lambertian BRDF is the incident radiance over π:

$$f_r(\omega_i, \omega_o) = \frac{\rho}{\pi}$$

The upper bound B_L for the constant Lambertian BRDF is trivial to compute. The figure 5.1 shows the development of the BRDF for the interval $[0; \frac{\pi}{2})$:

$$B_L(\omega_{\mathbb{C}}, \omega_o) = \frac{1}{\pi}$$



5.2 Oren-Nayar

Oren and Nayar created a microfacet based model for diffuse reflection. A detailed description of the model can be found in 2.4.4.1. The BRDF is defined by this formula:

$$f_r(\omega_i, \omega_o) = \frac{\rho}{\pi}(A + B\ max(0, \cos\phi_i - \phi_o) \sin\alpha \tan\beta)$$

α and β offer the best possibility for computing an upper bound:

$$\begin{aligned} \alpha &= max(\theta_i, \theta_o) \\ \beta &= min(\theta_i, \theta_o) \end{aligned}$$

In contrast to the Lambertian BRDF this model also depends on the viewer's direction ω_o. The interval of θ_i has a lower bound of 0 and an upper bound of $\frac{\pi}{2}$. Otherwise the illumination would arrive from a position behind the surface. The viewing direction also suits the same interval. Therefore, it is just necessary to find the minimum and maximum θ in the interval θ_i and the angle θ_o that yields the maximum value of $\sin\alpha \tan\beta$. There are three angle constellations, which can be observed in figure 5.1.

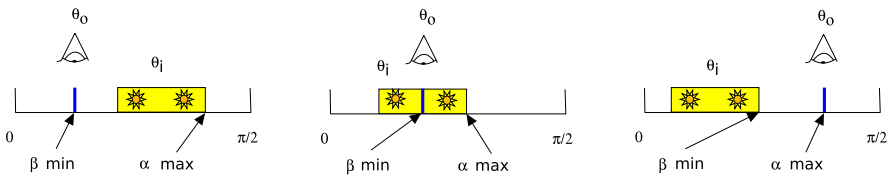
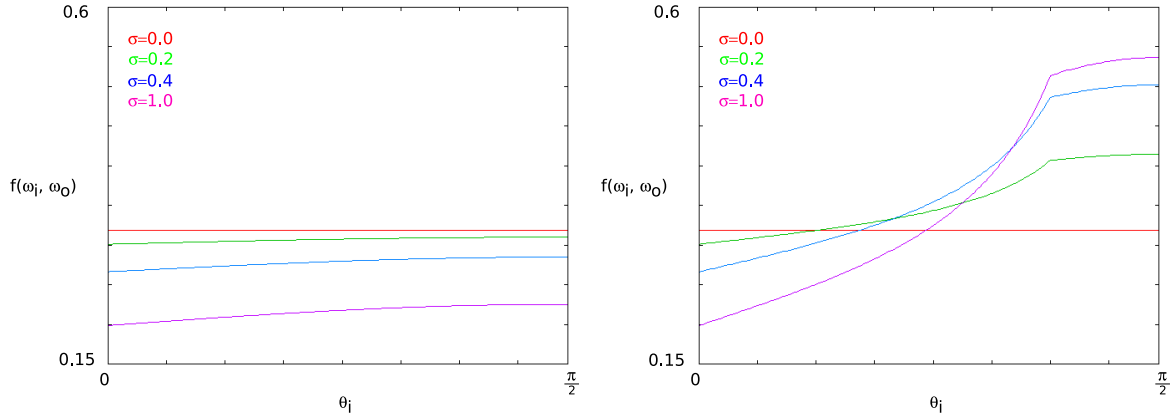


Figure 5.1: The incident light interval θ_i and the viewing direction θ_o are used to bound the Oren-Nayar BRDF. The minimum and maximum values are chosen to retrieve the maximum BRDF term.

The maximum reflection happens, when both α and β are as big as possible, because the sine and tangent are monotone and continuous in the maximum possible interval $[0; \frac{\pi}{2})$. Taking the maximum value of the θ_i and θ_o would be sufficient. To tighten the bound, the usage of the maximum possible minimum is advised as shown in figure 5.1. Finally it is possible to denote the bound B_{ON} as follows:

$$B_{ON}(\omega_{\mathbb{C}}, \omega_o) = \frac{1}{\pi}(A + B) \sin\alpha \tan\beta$$



(a) $\theta_o = 0.2$ (b) $\theta_o = 1.2$

Figure 5.2: Two example plots of the Oren-Nayar bound with varying roughness (σ). In the left figure the viewer's direction in relation to the surface normal is very steep compared the example on the right.

5.3 Strategies for the Halfway Vector

The Torrance-Sparrow microfacet model determines the maximum reflection by the help of the halfway vector. This vector is well known from the Phong lighting model, which is a simple approximation for calculating reflected light. The Phong model achieved its prominent status, due to its easy nature and great results - without being physically correct at the same time. The model proposed by Torrance and Sparrow simulates the reflection behavior of real surfaces much better by comparing the halfway vector to the orientation of the microfacet. Since this reflection algorithm is mainly based on this comparison, it is crucial to know the halfway vector. The lightcuts error estimation has to deal with an interval of incident light directions. As a consequence, the halfway vector also points at a certain range. So, the first task is to determine a bound for the possible halfway vector directions ω_h as illustrated in figure 5.3. The method depends on the type of light cluster. The following subsections explain this for two bounding structures. The point of interest with respect to the Torrence-Sparrow BRDF is to calculate the maximum angle between the viewing vector and the halfway vector denoted by $max(\theta_h^*)$ and additionally the minimum angle between the surface normal and the halfway vector $min(\theta_h)$. I will focus primarily on the computation of these two angles or rather their cosine, which is equivalent due to their direct correlation.

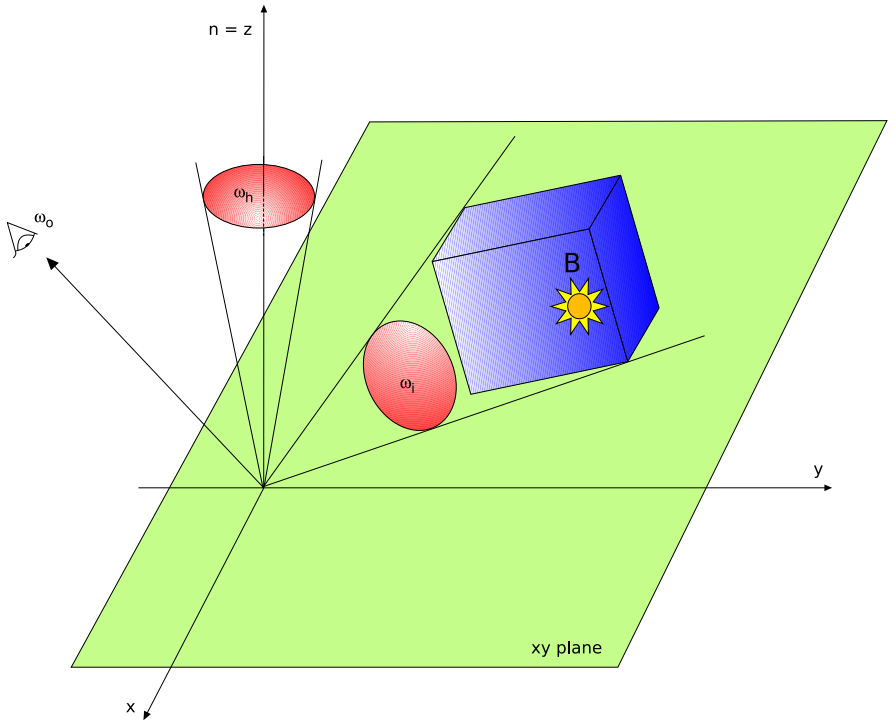


Figure 5.3: Incident radiance ω_i arrives from light cluster B. In combination with the viewing direction ω_o a bunch of halfway vectors ω_h is generated. θ_h is the interval of angles between n and ω_h, whereas θ_h^* is the interval of angles between ω_o and ω_h (compare to figure 5.3.

5.3.1 Bound For The Point Light Cluster

To bound the halfway vector, it is necessary to find its set of directions between a bounding box and the viewing vector. A nice method for bounding the half-way vector was presented by Bruce Walter in his paper called “Nodes on the Ward BRDF” from 2005 [26]. He describes a transformation into another coordinate system where calculation of the minimum and maximum angle between ω_o and the bunch of directions ω_h can be done easily. The scenery is rotated in order to match the viewing direction to the z-axis. Figure 5.4 shows a scenery where this transformation was applied. The relationship between the incident light direction and the halfway vector can then be formulated as:

$$\begin{aligned} \theta_i^* &= 2\theta_h^* \\ \phi_i^* &= \phi_h^* \end{aligned}$$

The star indicates the usage of a different coordinate system. At this point it is already achieved to bound the maximum and minimum angle between the viewing vector and the incident light direction, as the bounding mechanism from 4.2.3 can be used again. The bound on the cosine returned by the algorithm can then be used to calculate the half-angle bound with respect to the viewer’s direction:

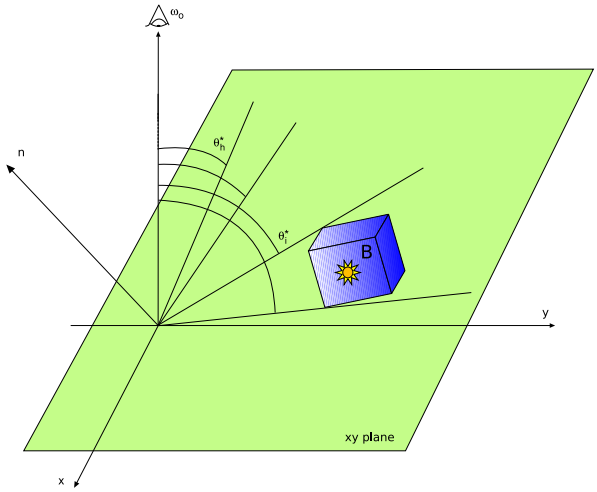


Figure 5.4: A rotated coordinate system is used to bound θ_h^*. ω_o is used as z-axis, the surface normal lies on the xz-plane.

$$\cos\theta_h^* = \cos\frac{\theta_i^*}{2}$$

Applying the trigonometric formula for multiple angles this leads to:

$$\cos\theta_h^* = \pm\sqrt{\frac{1}{2}(1+\cos\theta_i^*)}$$

It is safe to omit the negative sign in the previous equation, because $\theta_i^* \in [0;\pi]$ and therefore $\theta_h^* \in [0;\frac{\pi}{2}]$.
After the minimum θ_h^* has been determined, this leaves only the θ_h to be bound. It is the angle between the surface normal and the halfway vector. This can be achieved by finding the maximum cosine of the angle:

$$\cos\theta_h = h\cdot n = h^*\cdot n^*$$

In the new coordinate system this can be expressed as:

$$\cos\theta_h = \begin{pmatrix}\sin\theta_h^*\cos\phi_h^*\\ \sin\theta_h^*\sin\phi_h^*\\ \cos\theta_h^*\end{pmatrix}\cdot\begin{pmatrix}\sin\theta_n^*\cos\phi_n^*\\ \sin\theta_n^*\sin\phi_n^*\\ \cos\theta_n^*\end{pmatrix}$$

With the surface normal lying on the xz-plane ($\phi_n^* = 0$) the equation can further be simplified:

$$\cos\theta_h = \begin{pmatrix}\sin\theta_h^*\cos\phi_h^*\\ \sin\theta_h^*\sin\phi_h^*\\ \cos\theta_h^*\end{pmatrix}\cdot\begin{pmatrix}\sin\theta_n^*\\ 0\\ \cos\theta_n^*\end{pmatrix} \tag{5.1}$$

$$= \sin\theta_h^*\cos\phi_h^*\sin\theta_n^* + \cos\theta_h^*\cos\theta_n^* \tag{5.2}$$

The angles of the transformed normal n^* are fixed and result by applying the transformation matrix to the initial surface normal direction $\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$. The task is to find appropriate values for ϕ_h^* and θ_h^* that maximize the previous expression. It would be safe to replace the ϕ_h^* with its maximum possible value of 1.0. Yet, it is necessary to find the maximum ϕ_h^* for the whole light cluster $\mathbb{C}$, as the goal is to obtain a bound as tight as possible. This can be done analogously to the way of bounding the $\cos\theta$ in 4.2.3. Figure 5.5 illustrates the idea.

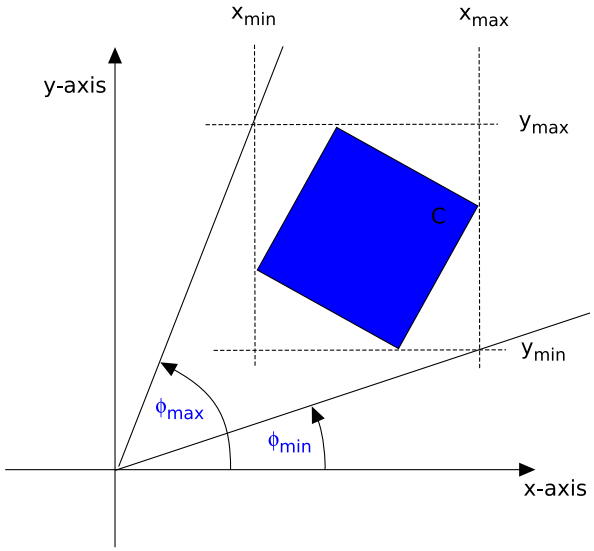


Figure 5.5: Determining the maximum and minimum values for x and y leads to a bound for the angle ϕ by using the Pythagorean equation.

The last step after the computation of the $max(\cos\phi_h^*)$ is the selection of the appropriate θ_h^* out of its previously calculated interval. This can be achieved by calculating the derivative with respect to θ_h^* and finding the maximum:

$$\begin{aligned} \frac{\partial\langle h,n\rangle}{\partial\theta_h^*} &= 0 \\ 0 &= \cos\theta_h^* max(\cos\phi_h^*)\sin\theta_n^* - \sin\theta_h^*\cos\theta_n^* \end{aligned}$$

If this equation is solved for $\cos\theta_h^* \in [0;1]$, there is one solution left:

$$\cos\theta_h^* = \sqrt{\frac{\cos^2\theta_n^*}{\cos^2\theta_n^* + sin^2\theta_n^*[max(\cos\phi_h^*)]2}}$$

It is still necessary to verify the solution, because the extremum could also be a minimum.

$$\frac{\partial^2\langle h,n\rangle}{\partial(\theta_h^*)^2} = -\sin\theta_h^* max(\cos\phi_h^*)\sin\theta_n^* - \cos\theta_h^*\cos\theta_n^*$$

If the result of the second partial derivative is < 0 then the result is a maximum and it can be used for the following step. Otherwise 1.0 is used as upper bound. The solution for θ_h^* and $max(\phi_h^*)$ is put into the equation 5.2. The final bound for the halfway vector can now be calculated. If all these computations pay off - it depends. Especially computing the ϕ_h^* can be omitted by taking the upper bound of 1.0.

5.3.2 Bound For The Directional Light Cap

The creating of a bound for the light cap is much easier compared to the previous box bounding approach. A bounding cap consists of a normalized direction vector ω_i and an apex angle α. The halfway bound is really simple. At first it is necessary to define the possible halfway directions. This can be done by using the initial definition of the halfway vector and the direction vector of the cap:

$$\omega_h = \frac{\omega_o + \omega_i}{2}$$

The apex angle for the halfway vector is half the apex angle of the incident light directions:

$$\alpha_h = \frac{\alpha}{2}$$

Now it is possible to calculate directly the minimum angle between the surface normal and the halfway vector, denoted by θ_h and the maximum angle between the viewing vector and the halfway vector, denoted by θ_h^*:

$$\begin{aligned} min(\theta_h) &= \arccos(n \cdot \omega_h) - \alpha_h \\ max(\theta_h^*) &= \arccos(\omega_h \cdot \omega_o) + \alpha_h \end{aligned}$$

5.3.3 Microfacet Modell

All the previous preparation steps are needed to find a bound for the Torrance-Sparrow microfacet model. The model uses this BRDF:

$$f_r(p, \omega_i, \omega_o) = \frac{D(\omega_h)G(\omega_o, \omega_i)F_r(\omega_o)}{4\cos\theta_o \cos\theta_i}$$

The advantage of the model is the possibility to bound each term separately. All terms in the numerator get maxed and the only variable term in the denominator $\cos\theta_i$ is bound for its minimum value. A minimum bound for the $\cos\theta_i$ can be calculated using the methods described in the previous chapter, so it is just omitted here. Altogether this results in the final bound for the Torrance-Sparrow BRDF:

$$B_{TS}(p, \omega_{\mathbb{C}}, \omega_o) = \frac{max(D(\omega_h))max(G(\omega_o, \omega_{\mathbb{C}}))max(F_r(\omega_o))}{4\cos\theta_o min(\cos\theta_i)}$$

The following three subsections describe the bounding mechanism of the three terms:

5.3.3.1 $Max(D(\omega_h))$

If the Blinn microfacet distribution is used, the upper bound for $D(\omega_h)$ is:

$$\begin{aligned} D(\omega_h) &= \frac{e+2}{2\pi}(\langle n, \omega_h \rangle)^e \\ D(\omega_h) &\leq \frac{e+2}{2\pi}(max(\cos\theta_h))^e \end{aligned}$$

5.3.3.2 $Max(G(\omega_o, \omega_i))$

An upper bound for the geometric attenuation term G is:

$$\begin{aligned} G(\omega_o, \omega_{\mathbb{C}}) &= min\left(1, min\left(\frac{2(n\cdot\omega_h)(n\cdot\omega_o)}{(\omega_o\cdot\omega_h)}, \frac{2(n\cdot\omega_h)(n\cdot\omega_i)}{(\omega_o\cdot\omega_h)}\right)\right) \\ G(\omega_o, \omega_{\mathbb{C}}) &\leq min\left(1, min\left(\frac{2max(\cos\theta_h)\cos\theta_o}{min(\cos\theta_h^*)}, \frac{2max(\cos\theta_h)max(\cos\theta_i)}{min(\cos\theta_h^*)}\right)\right) \end{aligned}$$

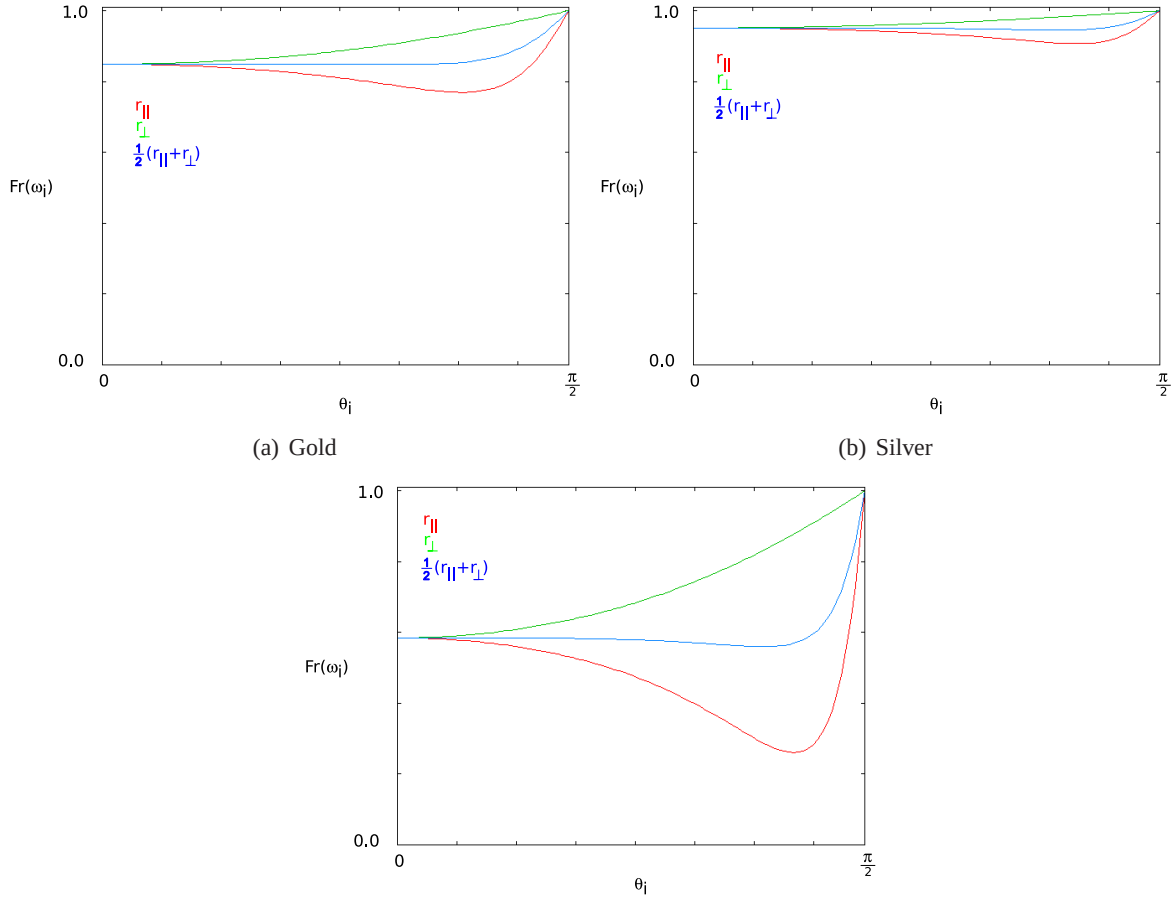
5.3.3.3 $Max(F_r(\omega_o))$

The Fresnel function for the Torrance-Sparrow BRDF uses the halfway vector as replacement for the surface normal. So, for the incident light direction, the interval of the angle θ_h^* is applied. It was discussed previously how to create a bound on this interval. This is why it is treated as already known here. To solve the bound for the Fresnel term in a more general way, I denote the incident light direction by θ_i and not θ_h^*.
The Fresnel dielectrics term inherits the transmission angle θ_t. It depends on the materials participating at the position where the light hits the surface. The sine of the reflection direction θ_t can be defined in terms of θ_i:

$$\sin\theta_t = \frac{\eta_i}{\eta_t}\sin\theta_i \tag{5.3}$$

$$\cos\theta_t = \pm\sqrt{1 - \frac{\eta_i}{\eta_t}\sin\theta_i} \tag{5.4}$$

θ_t is defined to be the angle between the negative z-axis and the transmission direction, which has to obey the interval $[0; \frac{\pi}{2}]$. It is just necessary to be sure that the transmitted light really enters the material. Then the negative sign can simply be ignored. If the expression for $\cos\theta_t$ is put into the Fresnel dielectric equation from 2.1 the resulting equation gets really complicated. It gets even worse, if the derivative is calculated and set to 0 to determine the extrema. It is so complex, that I could not solve it by hand - so, for the Fresnel dielectric term, I always use the upper bound of 1.0. For the Fresnel conductor term, it is much easier: It is worth to examine some example plots of the Fresnel term for a few selected materials. I used the η and k values given by table 2.2.
As it can be observed in the plots in figure 5.6 it is very likely, that the Fresnel function has at most one extremum in the interval $[0; \frac{\pi}{2}]$. This can be confirmed by determining its position. Since this leads to a huge mathematical term with many solutions, it is better to examine the two functions for



(a) Gold

(b) Silver

(c) Steel

Figure 5.6: Three example plots of the Fresnel term with different materials. The red curve demonstrates the amount of reflected parallel polarized light. The green curve is the equivalent for perpendicular polarized light. The blue curve is the fresnel reflection for unpolarized light.

perpendicular and parallel polarized light separately. The partial derivative of the Fresnel equation for parallel polarized light leads to the following equation:

$$\frac{\partial r_{\parallel}}{\partial \theta_i} = -\frac{2\eta(-2\eta + k^2 + \eta^2 + (k^2+\eta^2)\cos(2\theta_i))\sin\theta_i}{1 + 2\eta\cos\theta_i + (k^2+\eta^2)(\cos\theta_i)^2)^2}$$

If this equation is solved for its extrema, the following solutions are acquired:

$$\begin{aligned} \frac{\partial r_{\parallel}}{\partial \theta_i} &= 0 \\ \theta_i &= \pm\arccos\left(\pm\frac{1}{\sqrt{k^2+\eta^2}}\right) \end{aligned}$$

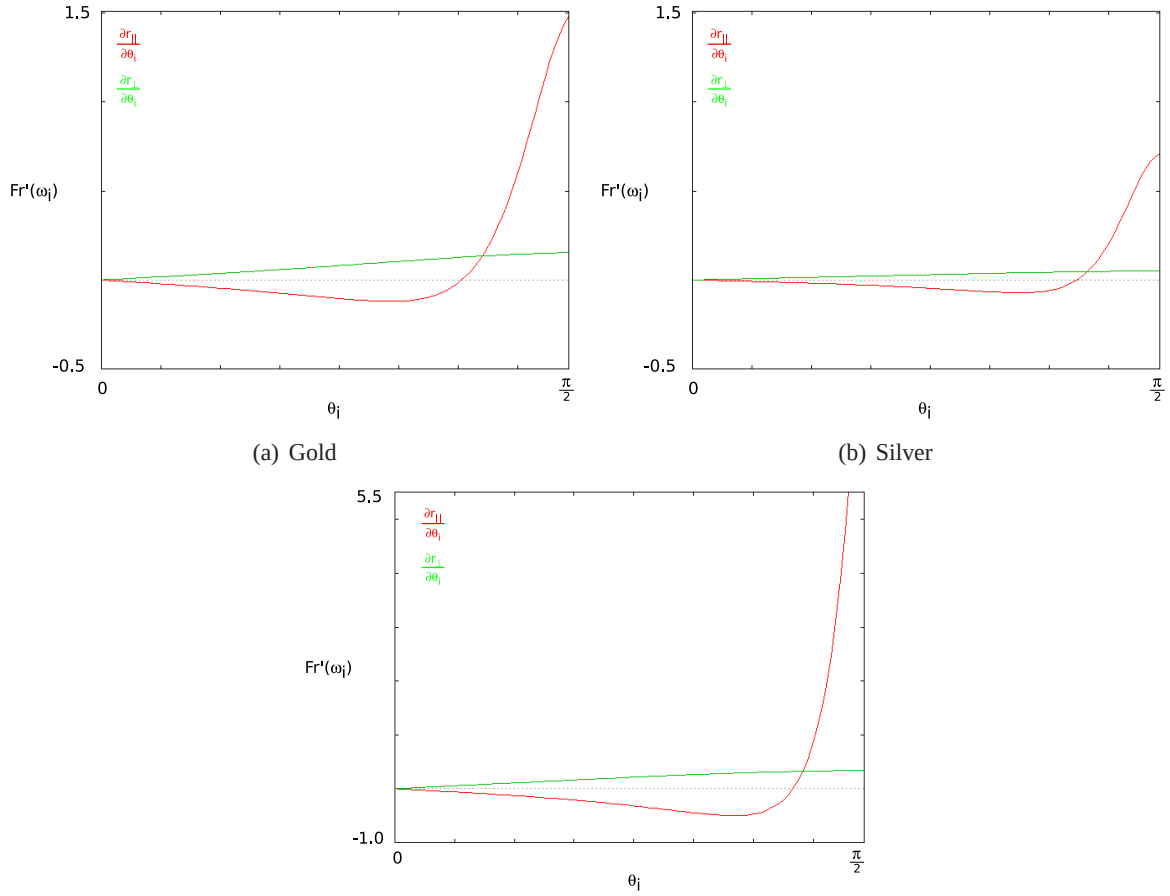
This results in four solutions. The plots show that it is very likely for one solution of θ_i to be situated in the interval $[0; \frac{\pi}{2}]$. The next step is to analyze, the partial derivative of the fresnel equation for perpendicular polarized light:

$$\frac{\partial r_\perp}{\partial \theta_i} = -\frac{2\eta(1 - 2k^2 - 2\eta^2 + \cos(2\theta_i)) \sin\theta_i}{(k^2 + \eta^2 + 2\eta\cos\theta_i + (\cos\theta_i)^2)^2}$$

Again, the first derivative is solved for the extrema:

$$\begin{aligned} \frac{\partial r_\perp}{\partial \theta_i} &= 0 \\ \theta_i &= \pm\arccos(\pm\sqrt{k^2 + \eta^2}) \end{aligned}$$

The procedure is quite the same as demonstrated with the parallel polarized light before. Yet there is one important difference: the extremum is less likely to be in the interval $[0; \frac{\pi}{2}]$. This is the key that allows bounding the fresnel term for conductors. If the extrema of the perpendicular polarized light equation are outside of the interval $(0; \frac{\pi}{2})$, then the function is monotone inside. Of course, the extrema can be precalculated for the initialization values of η and k using the above formula to make sure that this is definitely the case. The second assumption that the parallel polarized light equation has a minimum inside the interval $(0; \frac{\pi}{2})$ does not harm the bounding procedure, because the goal is to determine the maximum value in the range for a set of light directions defined by the help of the incident light's bounding structure. Therefore, it is sufficient to calculate the maximum for the two limits of the given interval for θ_i.



(a) Gold

(b) Silver

(c) Steel

Figure 5.7: The plots show the development of the first partial derivative with respect to θ_i for the Fresnel term with different materials. The red curve demonstrates the amount of reflected parallel polarized light. The green curve is the equivalent for perpendicular polarized light.

Chapter 6

Algorithms for the Infinite Area Light Source

The purpose of this chapter is to show the general usage of infinite area lights and how it is used in combination with the lightcuts rendering system. Also, some algorithms are presented, which demonstrate how to convert an infinite area light into a set of directional light sources.

6.1 Fundamentals of Infinite Area Light Sources

The infinite area light source is nothing else than a huge light source surrounding an entire scene. So, there is just one infinite area light possible in each scene file. It can be imagined as a sphere casting light from any direction into the scene. This method is commonly used to realistically illuminate synthetic objects as if they were in a given environment. It requires of course, that someone actually captured the illumination situation in this environment. For a good estimate of the illumination an image with high dynamic range should be generated. Paul Debevec describes a method how this can be done using standard digital photo equipment [5]. The data structure storing the light information is mostly referred to as *light probe* or *radiance map*. This image based lighting approach had a big impact on the realism of computer graphics. The initial usage of environment maps was for reflection mapping first introduced by Blinn and Newell in 1976 [3]. They used a self-drawn image representing the environment of a room to illuminate the Utah teapot. Later on, the reflection mapping (also called *environment mapping*) was used to efficiently simulate complex glossy and mirroring surfaces by the help of a precomputed texture image. The method is widely used in up-to-date computer games and other real time applications on recent raster-graphics based hardware. It works by looking up the reflection direction in the environment map to calculate the color of the incident light at the rasterized surface position. This is only done to simulate reflection effects, but not used for global illumination. The PBRT ray tracer and other global illumination systems can use environment maps as infinite area lights affecting the lighting of all objects, whether they are glossy or not. This means that diffuse surfaces, which reflect received light from any direction, must sample the light source in an appropriate way and not just look up the reflection direction. The term *radiance map* is generally used for the storage texture of incident light in contrast to *environment map*.

6.2 Data Storage for Infinite Area Lights

The radiance maps store incident light from all directions. They are accessed via spherical coordinates (θ, ϕ), texture coordinates (u, v) or discrete integer coordinates (x, y) depending on the actual usage. I tried to use the spherical representation as often as possible. A common representation for the radiance map is the latitude-longitude mapping. That means, the map can be accessed via spherical coordinates, as it is shown in figure 6.1. The mapping allows for partitioning the map into regions along the latitudes and longitudes. This property is useful later on for the median cut algorithm to fractionise the light.

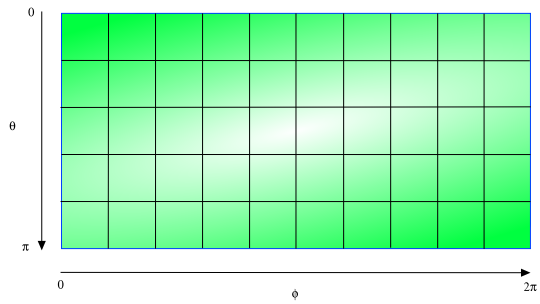


Figure 6.1: Environment map with latitude-longitude mapping.

6.3 Median Cut Algorithm for Infinite Area Lights

Paul Debevec presented a method to approximate an infinite area light source by using a predefined number of point light sources. I adopted this technique to create representative distant light source for the lightcuts rendering system. The idea of Debevec's algorithm is to create lights at positions where incident illumination is likely to occur. These directions are already given by the light probe image itself! Light probe sampling already was the topic of many papers and research in the past. Beginning with simple stratified sampling up to structured importance sampling by Argawal et al. [1], there are many algorithms solving the sampling problem. The proposed method by Debevec tampers with its simplicity and well conditioned splitting behavior which creates regions with equal energy. Another advantage is the option to use it as a progressive splitting algorithm, which can further refine a specific region by splitting it again. A split result in 64 single lights by Debevec's algorithm can be observed in figure 6.2.

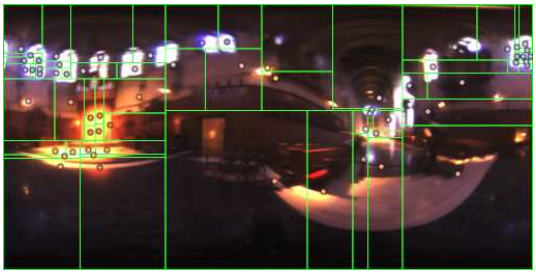


Figure 6.2: Split results obtained with the original median cut algorithm for light probe sampling.

Before I start to explain the split algorithm, it is necessary to know how to calculate the total radiance emitted by a region. Therefore the light probe (usually available in ll mapping) is read at a per pixel level. Each pixel of the light probe represents a differential solid angle on the sphere of incident light directions. The extent of a pixel can be expressed in intervals for θ and ϕ:

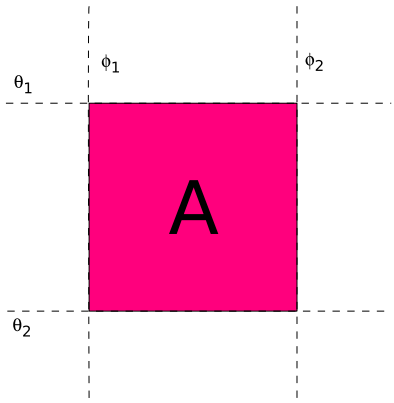


Figure 6.3: Intervals for a pixel of a ll map.

The formula from 2.1.3 helps to calculate the solid angle for all the pixels. If the solid angle (which is equivalent to the size of the differential area of a pixel on the unit sphere) is multiplied by the emitted radiance. The result is the relative emitted light energy from these directions of the light source:

$$\begin{aligned} \Phi_{Pix} &= AL \\ &= (-\phi_1 + \phi_2)(\cos\theta_1 - \cos\theta_2) * Y \end{aligned}$$

The previous formula is an addition to the algorithm introduced by Debevec. He suggested to scale the size of the patches simply by multiplying with $\cos\theta_m$. I think using the solid angle is a better measure. The emitted radiance is most likely expressed as RGB value. Due to human perception anomalies it is necessary to express the emitted light as weighted average of the color channels. By following an ITU-Recommendation these weights are $Y = 0.2125R + 0.7154G + 0.0721B$.
Meeting all the prerequisites, it is now possible to calculate the relative emitted light energy for each pixel. This implies that the same can be done for the entire sphere of directions just as for a distinct region. The proposed algorithm by Paul Debevec generates 2^n regions of similar energy and works as follows:
The radiance emitted by the light source is calculated by summing up the radiance from all participating pixels in that particular region:

$$\Phi_{Region} = \sum_{pix} \Phi_P ix$$

In each summand the solid angle of the patch is used. Thus I recommend to precalculate the solid angles used for scaling and store them in a table. The position of the light source is determined as the center of the region:

Algorithm 1 MEDIAN_CUT_ALGORITHM_FOR_LIGHT_PROBE_SAMPLING(n)

Require: $n \geq 0$

add entire light probe as single $region$ to $regionlist$
for i = 0 to n **do**
  for all $regions \in regionlist$ **do**
    subdivide $region$ along its longest dimension such that its light energy is divided evenly
    add new $regions$ to $resultregionlist$
  end for
  if $i < n - 1$ **then**
    regionlist = resultregionlist
  else
    for all regions $\in$ resultregionlist) **do**
      lights.add(GENERATE_LIGHT_SOURCE(region))
    end for
  end if
end for
return lights

$$\begin{aligned} \theta_m &= \tfrac{\theta_1+\theta_2}{2} \\ \phi_m &= \tfrac{\phi_1+\phi_2}{2} \\ L_{pos} &= \begin{pmatrix} \sin\theta_m \cos\phi_m \\ \sin\theta_m \sin\phi_m \\ \cos\theta_m \end{pmatrix} \end{aligned}$$

6.4 Improving the Position of the Light Sources

It is possible to improve the position of the light sources inside their region. The center is a good approximation when the regions are already small. For big regions, imagine the entire image divided only once, the center would be the direction all light energy is arriving from. This can yield a big error, as in the image there might be a dark spot. This leaves some space for two methods which try to improve this shortcoming.

6.4.1 Centroid

This is the first method, which tries to improve the position of the directional light source. The centroid simulates the characteristic of a region much better than the center. Each patch on the sphere emits a certain amount of radiance, which is precomputed and stored in an array for fast access. This algorithm is used to calculate the centroid:

6.4.2 Random Sampling of the Spherical Patch

As an alternative to the centroid method, it is also possible to calculate a random position inside the region, denoted by θ_r and ϕ_r. It can be used as a static position for the light source of even better

Algorithm 2 CENTROID

```
for x = x_min to x_max do
    for y = y_min to y_max do
        accu ← accu + radiance(x, y)
        accu_x ← accu_x + x * radiance(x, y)
        accu_y ← accu_y + y * radiance(x, y)
    end for
end for
x_centroid ← accu_x / accu
y_centroid ← accu_y / accu
```

re-evaluated for each lighting request. This makes the light source even less biased in a Monte Carlo sense. To compute a random sample for the region, it is necessary to bear in mind that sampling a spherical patch is not uniform. Unfortunately, built-in functions in programming languages can easily create uniform distributed random integer values. Therefore the inversion method is used to map a uniform random variable to the real distribution created by the properties of the spherical patch.
Much literature can be found about the correct sampling of a sphere. I already wrote about sampling techniques in my student thesis work [17], so I will not repeat the basics of sampling theory here. The presented method for sampling a spherical patch, is a modified version of the algorithm introduced by Matt Pharr and Greg Humphreys in a paper about *importance sampling of infinite area lights* (see [21]). The considered spherical patch has its extent in the intervals $[\phi_1; \phi_2]$ and $[\theta_1; \theta_2]$. These intervals represent a range of pixels of the radiance map, which can be interpreted as a 2D distribution function $f(u, v)$ over $[u_1, u_2] \times [v_1, v_2]$ ($u_1 < u_2$; $v_1 < v_2$; $u, v \in \mathbb{N}_0$). The solid angle of each pixel is then accessible via $f(u, v)$ (u represents the column and v the row of the pixel). That is the key to define the probability $p(u, v)$ of a pixel to be picked, also called the probability density function (PDF) for an outcome of u and v:

$$p(u, v) = \frac{f(u, v)}{\sum_{u=u_1}^{u_2} \sum_{v=v_1}^{v_2} f(u, v)}$$

It is the solid angle of a pixel divided by the total solid angle of the region. Of course, the integral over all probabilities equals one ($\int p(u, v)$), because one event is definitely happening. Gladly, the spherical patches have the same size in each column, so the probability to pick each column is the same. This means the marginal density function for the columns is constant:

$$p_u(u) = \frac{\sum_{v=v_1}^{v_2} f(u, v)}{\sum_{u=u_1}^{u_2} \sum_{v=v_1}^{v_2} f(u, v)} = \frac{1}{u_2 - u_1}$$

This definition helps to define the conditional density for the rows:

$$p_v(v|u) = \frac{f(u, v)}{\sum_{v=v_1}^{v_2} f(u, v)}$$

I previously mentioned the inversion method to map uniform random variables to a defined distribution. The method needs a cumulative distribution function (CDF) to work. A CDF $F(x)$ is defined as the probability of the random variable X to get a value less or equal to x. Therefore the probability for x is accumulated with all previous probabilities:

$$F_v(x) = P_v(X \leq x) = \sum_{v=v1}^{x} p(u, v)$$

The inversion method can now draw a sample by using a uniform random number $\xi \in [0..1]$. It is compared to the function values of the CDF (which is monotone) and thus the initial x can be looked up. In practice, an array is created which holds the accumulated probabilities up to the x-th element. Then the values stored in this array can be searched for ξ. The algorithm stops, when a value greater or equal to ξ was found and returns its position in the array. Afterwards, the value is transformed into the correct sample position for the light source position. The source code for this algorithm can be looked up in 1

6.4.3 Dynamic Infinite Area Light for Lightcuts

In the process of implementing the lightcuts algorithm I thought much about the negative effects of fixed, precomputed light trees. The dynamic generation of a light tree might be a much better approach, because it can adapt to the needs of the rendered scene. I already knew the median cut paper from P. Debevec and explored, if the algorithm could be used in a progressive way. In comparison to a simple split-at-spatial-median algorithm, the proposed median cut algorithm uses the energetic median as split position. This is a major advantage for the lightcuts error estimation routine, since after each split the error is at least bisected due to the direct correlation of error and a light's emitted energy. When the lightcuts renderer is started and an infinite area light is used, only two representative lights are created: one for the upper and another one for the lower hemisphere. Each light stores some information about its size, emitted radiance and representative direction. For any ray hitting a surface, the lightcuts integrator starts evaluating the illumination by pushing the root nodes of the lights on the lightcuts stack. If the error estimation routine decides that the representative light's error is too big, it will be refined. This is done by median cut splitting the light and adding it to the infinite area light tree at its appropriate position. Refinement can also be denied by the light source, if it already reached a minimal predefined size. This can either be a predefined solid angle or simply a bound given by the radiance map resolution. The dynamic trees help to reduce unnecessary preprocessing time. Additionally, it is possible to use very high resolution environment maps and sample them dynamically! Most ray tracers always use a fixed number of infinite area light source samples. The lightcuts system decides by the help of the error estimation for any hit surface how many samples are necessary to stay within a predefined error bound.

6.4.4 Comparison of Dynamic Light Trees after Rendering

Figure 6.4 shows the radiance map used to illuminate the scene in figure 6.5. The scene was rendered with the three different light sample strategies described in the previous sections. This results in a

change of sample positions as visualised by the plots in figure 6.6. The data used for the plots are the leaf node positions of the dynamic light tree created while rendering: i.e. the fully expanded tree without the positions of the representative lights. Figure 6.6(a) shows the results for using the *center* of the regions which leads to samples situated at rather grid-like positions. The *centroid* strategy creates samples next to brighter regions. This can be easily observed by watching the samples at the lower hemisphere ($\theta > \frac{\pi}{2}$). In comparison to the previous center strategy image, the lonely samples disappear because the lower hemisphere of the radiance map is entirely black. The last figure 6.6(c) shows the result for using the static *random* sampling strategy. Of course, this looks different for each rendering process, since the sample positions are drawn randomly. All strategies have in common to create a light tree with heavily sampled bright regions, due to a high potential error introduced by very bright light sources. A further discussion with respect to the quality of the rendered images can be obtained in chapter 8.

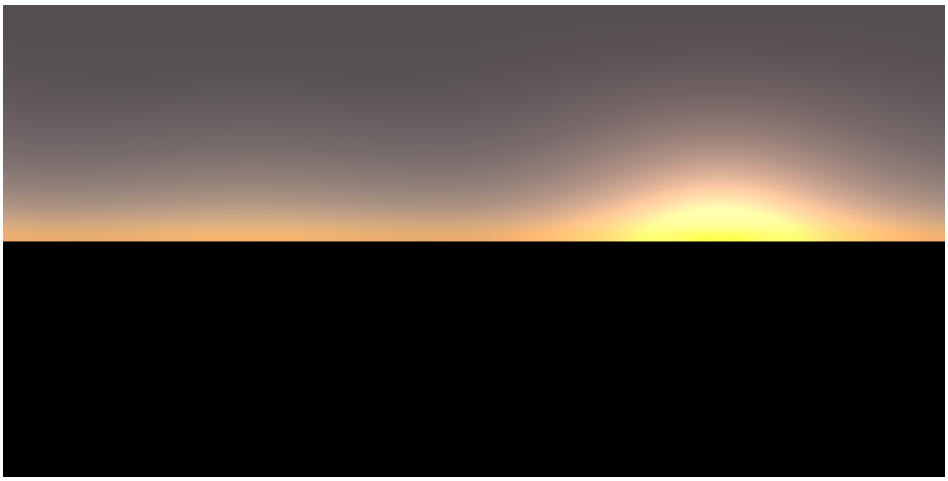


Figure 6.4: Sunset radiance map.

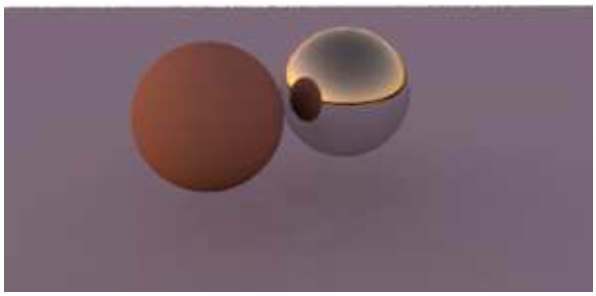
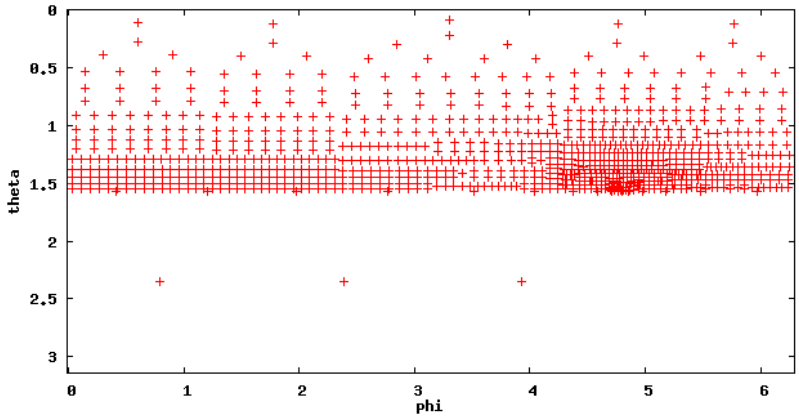
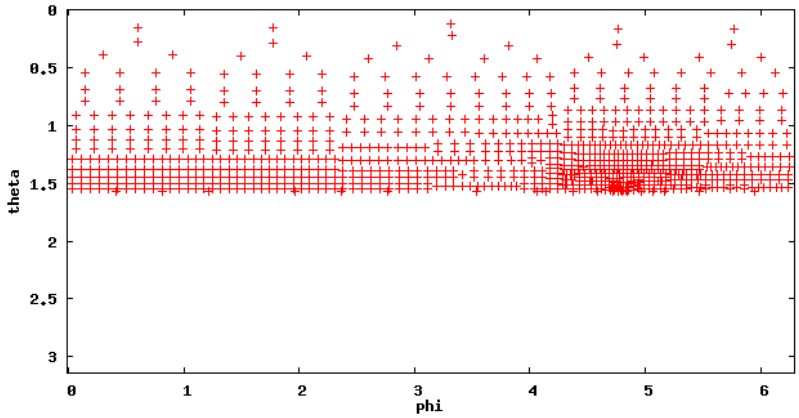


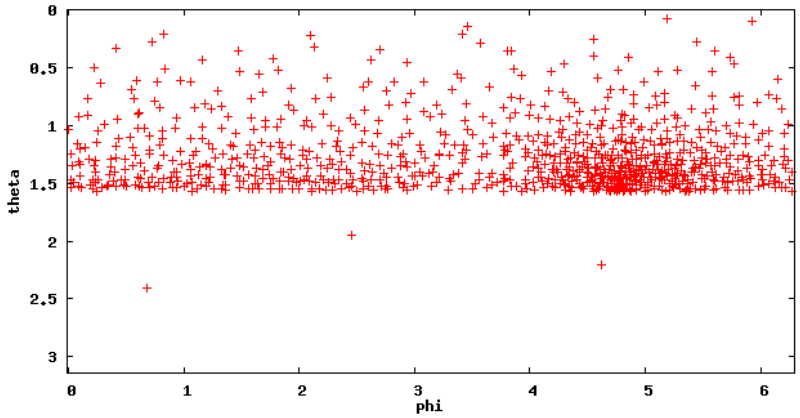
Figure 6.5: Scene with two spheres illuminated by the sunset radiance-map.



(a) Center



(b) Centroid



(c) Random

Figure 6.6: Samples created by three different strategies.

Chapter 7

The PBRT Rendering System

The *PBRT* rendering system is a full featured ray tracer. It accompanies Matt Pharr's and Greg Humphrey's book on physically based rendering [22]. The ray tracer was initially used for didactic purposes in a computer science course at Stanford University. Over the years it evolved to a robust, rendering system with rich features. I chose it for my own implementation, because it is plugin-based, extremely well documented and extensible. To be able to integrate the lightcuts idea, it was necessary to understand how PBRT works in the first place. This chapter gives a rough overview of the system and additionally some implementation details of the lightcuts integrator.

7.1 General Overview

The basic mode of operation is demonstrated in figure 7.1.

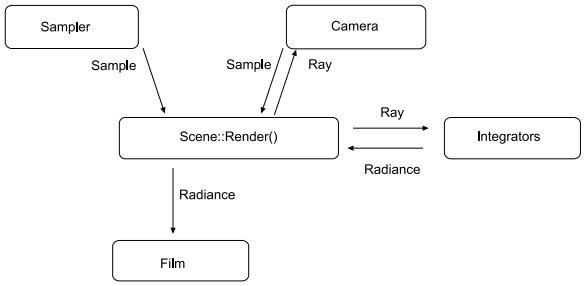


Figure 7.1: Diagram for PBRT's main rendering loop.

Rendering is started in the main render loop by retrieving samples for each image sample from the *Sampler*. The *Camera* takes a sample and generates a ray direction. Then the ray direction is given to the *Integrator*, which calculates the radiance arriving for this direction. The *Film* stores the retrieved radiance in an image. Rendering is complete when the *Sampler* created enough samples to generate the final image. All the components are abstractions with an interface. Thus they may be replaced by an algorithm fitting the minimal interface requirements. Therefore PBRT is realized as a rendering core with additional plugins. Less memory consumption is another advantage of this structure: The program dynamically loads the plugins it needed and leaves out unnecessary ones.

7.2 Integrators

The most interesting part of the PBRT rendering system are the integrator plugins. They handle rays shot into the scene and need to determine the radiance. Of course, this is also an abstraction for all ray casting based algorithms. In version 1.02 of the ray tracing system there already exist integrators for *bidirectional path tracing*, *direct lighting*, *photon mapping*, *irradiance caching* and the original *Whitted algoritm*. The general functionality of an integrator can be looked up in figure 7.2.

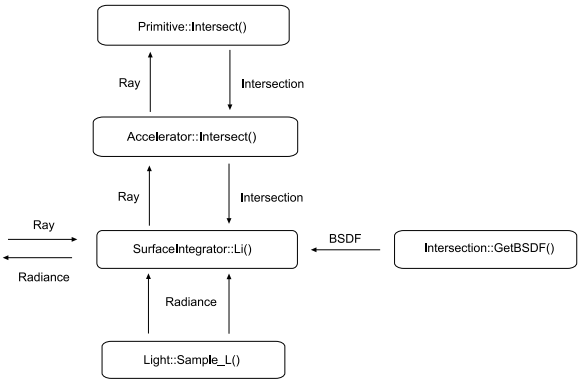


Figure 7.2: The diagram shows the class relationship of the Integrator abstraction.

A ray created by the camera and sent by the main rendering loop is processed by the integrator to obtain the radiance along that ray. That is the task for the integrator: find the closest object the ray intersects with. This can be done by asking the *Accelerator*. It is an abstraction for all objects situated in the scene. Testing each object separately is very expensive, so bounding structures are used to speed up this process. If an object was determined, the result is returned as an *Intersection*. This is an abstraction to store properties of the intersected surface. By the help of the *GetBSDF()* method these properties are evaluated to add the material properties for the intersection point. The *Lights* are used to calculate the illumination. Finally the accumulated reflected radiance for the ray is returned to the main rendering loop.

7.3 Direct Lighting Integrator

The *direct lighting integrator* is a rather simple method to approximate the LTE. I introduced the theory already in chapter 3. To be able to start with the lightcuts implementation, it is essential to know how shading is done by the direct lighting integrator. Remembering the direct lighting approximation, it is possible to break the LTE down into a sum over all lights in the scene:

$$\sum_l \int_S f_r(p, \vec{\omega_o}, \vec{\omega_i}) L_{d(l)}(p, \vec{\omega_i}) \cos\theta_i \; d\vec{\omega_i}$$

To estimate the direct lighting integral for one light, it is necessary to choose directions for sampling the light source and the BSDF. Sampling can be done efficiently by using multiple importance sampling thus incident light directions and reflective BRDF directions are generated. *Importance sampling* is a method for estimating integrals generally faster compared to the Monte Carlo method. Plenty of

work has been done to explore sampling strategies. The interested reader may look it up in [22], [17] or [1]. The problem can be simplified, if the participating light sources are *delta lights*. Then the light source direction is fixed, because light is received from only one direction. The pseudo code 3 explains how lighting with delta lights is done:

Algorithm 3 DIRECT_LIGHTING(ray)

```
1:  L = 0.0
2:  if ray intersects object then
3:      calculate intersection
4:      L+ = emittedlight by intersected object
5:      for all lights ∈ scene do
6:          L+ = BSDF * L_i * cos θ_i / lightpdf
7:      end for
8:      if actualraydepth < maximumraydepth then
9:          if BSDF is reflecting light then
10:             L+ = DIRECT_LIGHTING( reflected ray )
11:         end if
12:         if BSDF is refracting light then
13:             L+ = DIRECT_LIGHTING( refracted ray )
14:         end if
15:     end if
16: else
17:     L = background
18: end if
```

7.4 Lightcuts Integrator Plugin

This section informs about the general functionality of the lightcuts integrator plugin. The *lightcuts integrator* was developed in several steps.

7.4.1 Preprocess()

At first I implemented a preprocessing method to ensure that all lightcuts compatible light sources are detected and appropriate light trees are built. The `Preprocess()` method of an integrator is called after the scene file was parsed completely and all objects have been instantiated. By checking all lights it is possible to determine lightcuts compatible light sources: point lights and distant lights. It would also be possible to use explicit lightcuts lights, but then existing scene files could not be used by simply switching the integrator. If an infinite area light is used for lighting, this is different and must be explicitly changed in the scene description due to dynamic tree expansion. To be able to modify the light tree while rendering, the integrator needs to access the light for retrieving new child nodes. That is generally impossible with PBRT's plugin design, because plugins can only access each other via basic interface classes defined in the rendering core. Afterwards, the preprocessing method

generates a `LightTree` for each light type. This involves generating a kd tree for point lights and a minimal binary tree for the dynamic infinite area light tree as well as a binary tree for distant lights. Tree generation is started by calling the `buildTree()` method of the adequate `LighTree`.

7.4.2 doLightcut()

The second step was the adjustment of the direct lighting integrator to use light trees instead of real light sources. The direct lighting code calls the `doLightcut()` method to evaluate the lighting at the intersection point. This is done separately for each available light tree. Evaluation begins by inserting the root node into the lightcuts queue, which is used to store the actual participating nodes of the light tree. At this point the function `estErr()` is called to calculate the maximum possible error introduced by using the representative light of the actual node instead of the real ones. It depends on the result of the error term how the algorithm continues. If the error is below the predefined perceptual threshold, the representative light is used for the lighting calculation and completely evaluated. Otherwise, the children of the light node are added to the lightcuts queue and the error estimation algorithm starts again for those. The method `erstErr()` uses the intersection point with its surface normal, the direction of the ray and additional information from the light nodes to determine the error. Each component of the final term is computed separately. For computing an approximation of the material term the `bsdf->BRDFBound()` method is called. The implementation was realized in a third step in the global reflection class `BSDF`.

Algorithm 4 DO_LIGHTCUT(ray)

```
1:  workqueue.insert(lighttree.root)
2:  lightqueue.clear();
3:  while workqueue not empty do
4:      light = workqueue.pop();
5:      error = CALL ESTIMATEERROR(light)
6:      if error ≤ perceptualthreshold then
7:          lightqueue.push(light).
8:      else
9:          workqueue.push(light.children)
10:     end if
11: end while
12: for all light ∈ workqueue do
13:     L+ = EvaluateDirectLighting for light
14: end for
```

7.4.3 Helpers of the Lightcuts Integrator

Several classes and methods are used to assist the lightcuts integrator.

7.4.3.1 Lighttree

The `Lighttree` class and derived classes for each type of light tree are used to intially build and finally delete it. Especially the `InfiniteLighttree` has a method `extendTree()` to dynamically create new child nodes if necessary. The generation of representative lights is also realized in the `Lighttree` class.

7.4.3.2 InfiniteAreaLightLC

The `InfiniteAreaLightLC` implements the infinite area light for lightcuts. It reads a HDR light map in $.exr$ file format to generate new median cut sampled representative distance lights. The light source position is determined by using one of the following four different strategies: mid, $centroid$, $staticrandom$ and $dynamicrandom$. The latter differ in dynamic vs static placement of the light's position.

7.4.3.3 DistantLightLC

Special distant lights are used in the infinite area light tree. Each `DistantLightLC` knows its representative extent on the light map. The constructor creates the bounding cap by the help of the procedure described in chapter 4. The method `boundCosTheta(const Normal& n, Vector* wi)` computes the bound for the cosine of θ.

7.4.3.4 Modifications to the core

Bounding the BRDF works best, when it is done in the BRDF coordinate system. Therefore, the required methods have been implemented in the basic reflection class `BSDF`. Each lightsource type uses its own function to bound the bsdf term. The method `BRDFBound()` is called with a referenced data structure for exchanging related information. Because PBRT uses a wrapper class to enable multiple BRDFs for a surface, the `BRDFBound` collects the bounds of all participating BRDFs. If a reflection model is not yet supported, it returns an upper bound of 1.0 by default. Otherwise the bound on the BRDF is calculated by calling the `bound()` method. Additional modifications were necessary on the default light interface, since the PBRT developer designed it for shading and not for light object modifications.

Chapter 8

Results and Discussion

This chapter is denoted to present rendering results for selected scenes. It shows the major achievements of the lightcuts algorithm. Additionally, I will also discuss some disadvantages and draw a conclusion indicating aspects to be further investigated.

8.1 Benchmarks and Evaluation

All scenes were rendered by an Intel Pentium M machine with 1400 Mhz core clock and 512 MB of RAM using Ubuntu/Linux. I created and rendered some scenes with many light sources, to show the benefits of the algorithm, i.e, the reduction of shadow-rays in comparison to the standard direct lighting integrator.

8.1.1 Scenes with many Light Sources

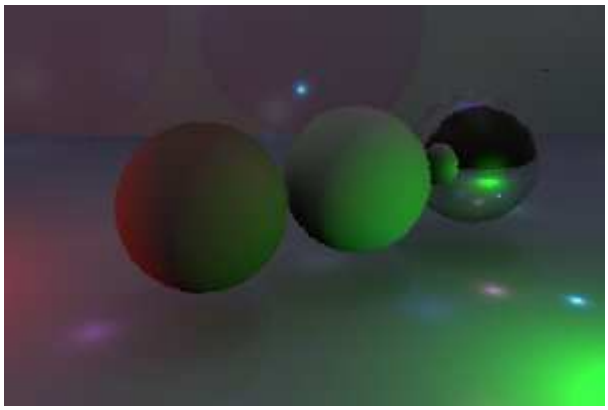
The first examined scene visualizes three balls with different materials and is rendered with one primary ray for each pixel. The left one uses a perfectly diffuse material (Lambertian BRDF), the middle one uses a material similar to plastic (Oren-Nayar BRDF) while the right one uses a shiny metal (Torrance-Sparrow BRDF). Figure 8.1 shows the scene rendered with direct lighting and the estimate by using the lightcuts integrator. The following table presents the data obtained by rendering the scene:

Point Lights	Shaded Points	Shadow Rays	Shadow Rays Per Sample	Image Time
10006	63.9k	5.765M	90,2	76.7s
10006	63.9k	374.454M	5860	1389.7s

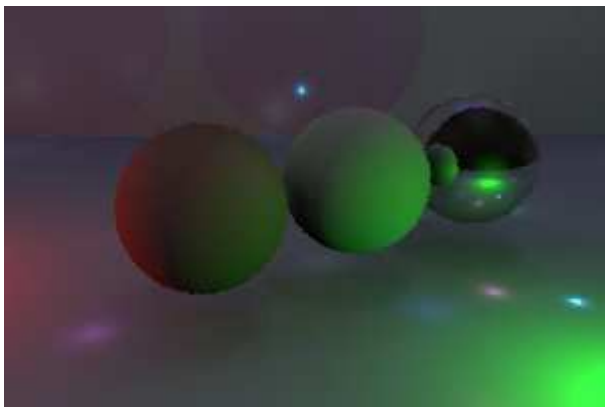
Table 8.1: Scene1, rendered in a resolution of 300x200. The object parameters used for the spheres: Oren-Nayar $\sigma = 0.15$, Microfacet(Blinn) $e = 45.3kr = (0.7, 0.7, 07)ks = (0.5, 0.45, 0.35)$.

This scene uses a setting with almost perfect preconditions for the lightcuts method. The number of lights is very high, which enables efficient clustering. In comparison to direct lighting using each light source separately, the lightcuts method can often use the representative light to generate the final image. This leads to over fifty times less the number of shadow rays which results in impressively

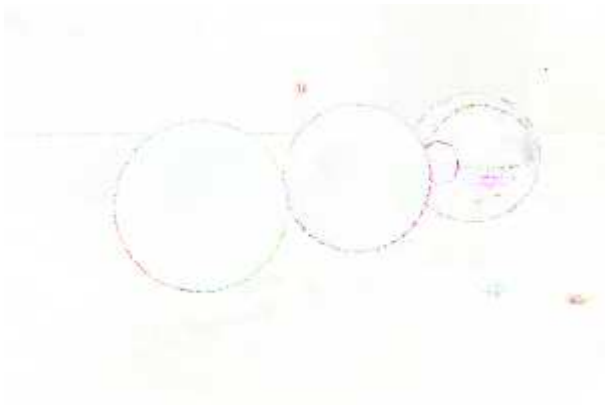
reduced rendering time. Even a close examination of both images reveals no visible error. The differential image from figure 8.1 shows a strong enhancement of the error introduced by the clusters. This phenomenon meets the expectations, which were already described by Ward et al. in the lightcuts paper [28]. The observed error appears to be greater in very bright regions.



(a) Reference image



(b) Image generated with lightcuts



(c) Differential image with magnified error

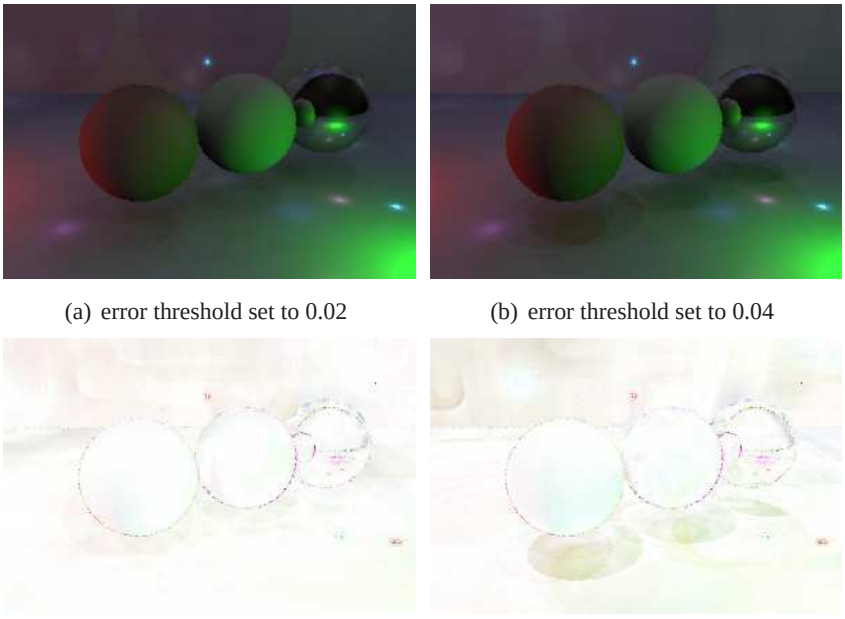
Figure 8.1: Three images showing a scene rendered with direct lighting and lightcuts. The maximum allowed error threshold was set to 0.01.

8.1.2 Modifying the Error Threshold

By noticing the magnified error in the previous section it seems appropriate to analyse the expansion with different error threshold values. Figure 8.2 illustrates the visual effects by increasing the light-cuts' error threshold. The accompanying table 8.2 demonstrates the change in shadow ray count and rendering time:

error threshold	Shadow Rays	Shadow Rays Per Sample	Image Time
0.01	5.765M	90,2	76.7s
0.02	4.045M	63,3	56.4s
0.04	2.917M	45,6	41.2s

Table 8.2: The settings are equal to those in in 8.1. The scene uses 10006 light sources.



(a) error threshold set to 0.02

(b) error threshold set to 0.04

(c) Differential image with magnified error, threshold set to 0.2

(d) Differential image with magnified error, threshold set to 0.4

Figure 8.2: Modulation of the error threshold leads to increasing error and thus reduced quality.

Unlike the previously rendered image, where no visible error could be determined, these images prove that the error threshold must be chosen carefully. Additionally, the error varies in both images due to random positioning of the light sources during kd tree building. In default mode the kd-tree builder selects one of the children's positions to be the representative light's coordinates. This explains the visible discrepancy between the differential images as it can be seen in figure 8.2(a) compared to figure 8.2(b).

8.1.3 Using Lightcuts for Optimal Area Light Sampling

The second scene uses 100, 1000, 10000 and 100000 individual lights to simulate an area light source above the two killeroo models. One of the models uses diffuse reflection and the other one the Torrance-Sparrow microfacet model. Table 8.3 shows render results in numbers:

Point Lights	Shaded Points	Shadow Rays	Shadow Rays Per Sample	Image Time
100	43.5k	2.911M	66,9	32.0s
1000	43.5k	2.471M	56,8	38.1s
10000	43.5k	2.475M	56,9	41.1s
100000	43.5k	2.474M	56,9	45.9s

Table 8.3: Scene2, rendered in a resolution of 200x200 and an error threshold of 0.01.

The results show that the lightcuts algorithm efficiently avoids to oversample the area light. The number of light sources is irrelevant as long as its total count is sufficient. Even 100 light sources are sufficient to generate perfect soft shadows. Error bounds enable the algorithm to decide for itself how far it needs to descend the light tree to select the right number of samples. This is an important proof that the algorithms works as expected, because the number of shadow rays stays almost the same. The slight rise in rendering time is due to additional preprocessing effort. Light tree building uses up more time while the pure rendering time almost stays the same.

(a) 100 light sources (b) 1000 light sources (c) 10000 light sources

Figure 8.3: The scene is rendered with an increasing number of light sources representing an area light above the models.

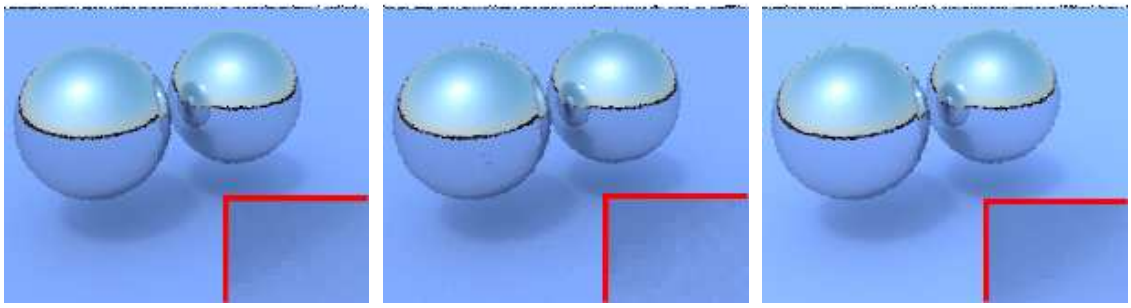
8.1.4 Using Lightcuts for Infinite Area Lights

My lightcuts implementation replaces the infinite area light by a dynamic light tree, which inherits distant lights to represent the illumination situation. The method of sampling the IAL has already been explained in chapter 6. I produced example renderings to demonstrate the pros and cons of the lightcuts method. Figure 8.4(a) shows the reference scene using the *daylight* radiance map. It was rendered with the direct lighting integrator using one eye ray per pixel. To obtain optimal reference images, I gradually increased the number of samples until no perceptual image noise was left. Since

random sampling is not a good competitor to the lightcuts integrator, I also used the importance sampler plugin from Pharr and Humphreys [21] for benchmarking. Evaluation of the results is very subjective due to a light brightness scaling factor necessary to compensate the power heuristics used by PBRT for sampling light sources and BRDFs. Nevertheless, the lightcuts method has the advantage of automatically using the necessary samples for an intersection position to be illuminated correctly. In contrast to this, the number of samples used by importance sampling and random sampling is generally predefined. This enables lightcuts to generate almost perfect results, if the error threshold is selected carefully. Importance sampling trusts in the predefined sample count, which can be chosen low for radiance maps using only one or few bright regions like the daylight setting in figure 8.4(a). If the radiance map uses many bright regions, the same number of samples will not be sufficient for a noise free image as seen in figure 8.6.

IAL-sampler	Shaded Points	Shadow Rays	IAL-Samples	Image Time
importance	38.4k	4.427M	100	46.9s
random	38.4k	68.631M	1000	306.8s
lightcuts	38.4k	13.014M	variable	112.0s

Table 8.4: Difference in rendering time for a high quality image.



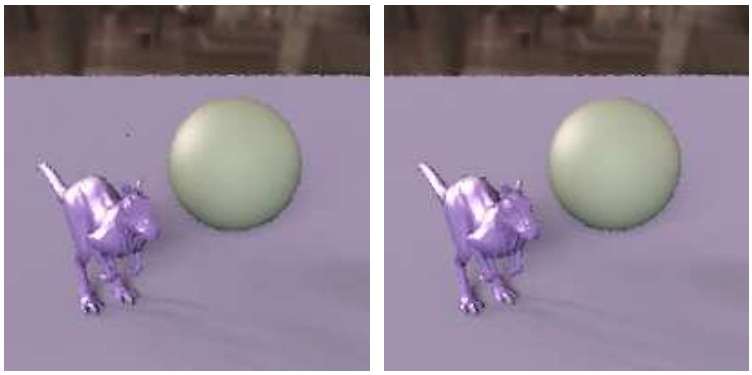
(a) Importance sampling (100) (b) Random sampling (1000) (c) Lightcuts (error thresh=0.01)

Figure 8.4: Two shperes rendered with different sampling techniques using the daylight radiance map.

IAL-sampler	Shaded Points	Shadow Rays	IAL-Samples	Image Time
importance	33.3k	3.716M	100	52.0s
lightcuts	33.3k	14.184M	variable	118.8s

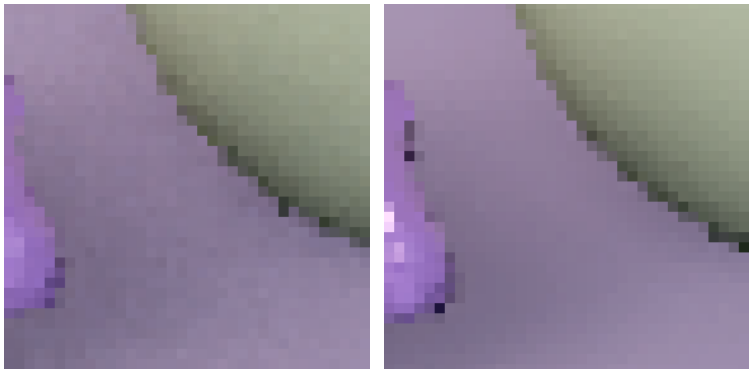
Table 8.5: Render statistics for lightcuts vs importance sampling.

Table 8.5 shows the necessary rendering time and shadow ray count per shaded point, which is equal to the average number of light source samples for all available settings. As expected, the random sampling method performs worst. It uses as many as 1000 samples to create an image free from noise. Importance sampling seems to be twice as fast as lightcuts for many tested scenes Due to efficient selection of bright regions and correct stochastic evaluation the image converges to a good result with fewer samples. This deficiency of the lightcuts' IAL implementation will be examined in the next



(a) Importance sampling (100) (b) Lightcuts

Figure 8.5: Images rendered with importance sampling and lightcuts using the galileo radiance map.



(a) Importance sampling (100) (b) Lightcuts

Figure 8.6: Magnified image area to show the weakness of importance sampling with a fixed sample count. The contrast of both images was slightly increased to visualize the difference in the printout.

subsection.

8.1.5 The Sampling Weakness of the IAL

The median cut algorithm for light probe sampling generates samples in regions of equal light energy. This results in big dark regions represented by only a few lights. In some cases this leads to shadow artifacts. If the error threshold it set too low, the lightcuts algorithm cannot determine this error, because it is introduced by multiple nodes. This can be seen as worst case scenario for lightcuts, since it provides a stochastic error bound rather than an absolute one. Figure 8.7 shows a rendered image with an error threshold set to 0.2.

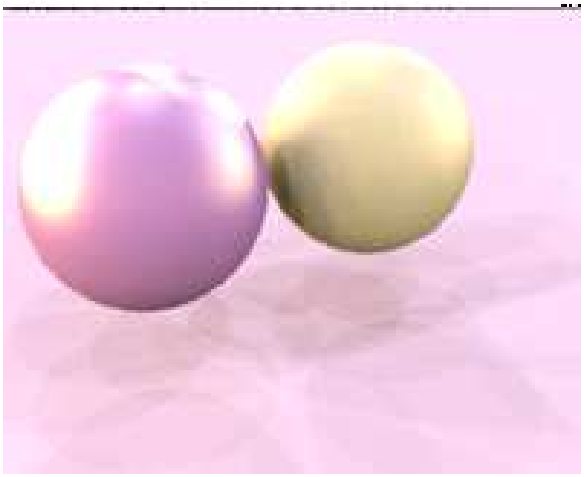


Figure 8.7: The image shows the IAL's sample weakness due to high error threshold.

8.1.6 Visualizing the Size of a Cut

It is very interesting to examine how many light nodes are used to calculate the approximated illumination of a pixel. Figure 8.8 and 8.9(b) show the number of shadow rays used for scenes 8.1 and 8.9(a). As expected, bright parts of the scene use a large number of shadow rays to avoid a big error whereas darker parts use only few shadow rays.

Figure 8.8: Point light scene cut size

(a) IAL scene

(b) IAL scene cut size

Figure 8.9: Visualized cut size: brighter means more shadow rays.

8.2 Conclusion

The task for this thesis was to implement and analyse the lightcuts algorithm, which approximates the illumination by clustering light sources. Additionally, a method is proposed to handle large radiance maps by using a dynamic light tree. I will summarize the topics examined:

- The lightcuts algorithm with basic functionality was implemented as a PBRT plugin. The implementation was more complicated than initially expected due to PBRT's encapsulating design. It finally resulted in interface changes to its rendering core.
- For efficient generation of point light trees, a kd-trie cluster algorithm was developed and implemented. It also supports creation of representative lights and a bounding structure for each light node.
- The theoretical background for lightcuts' error estimation was examined. This includes the

derivation of a bound on the $\cos\theta$ as well as the bound on three BRDF models. Therefore a method for bounding the halfway vector had to be analysed and implemented. The findings can be used as basis for future work.

- An adaptive light tree for infinite area lights was designed and implemented using the median cut algorithm. The light tree supports four different flavors of sample position generation.
- Several mathematical obstacles were introduced by the dynamic infinite area light tree. Since nodes represent light arriving from spherical patches, it was not possible to calculate the error approximation with reasonable effort. The bounding cap and its generation was developed and implemented to ease computation.

The initial findings of the lightcut paper [28] could be confirmed. Rendering time for scenes with a large set of light sources is significantly reduced by using the lightcuts approach. This works especially well for clustered point lights as demonstrated previously in this chapter. Rendering time increases logarithmic with the number of participating point lights due to the lightcuts behavior of error driven shadow ray usage. In contrast to this, the infinite area light performance was quite disappointing, since I expected it to be always faster than importance sampling of the radiance map. This is mainly due to the selected progressive energy median splitting algorithm, which can result in undersampled regions. As a consequence, the general error threshold has to be very conservatively chosen to obtain images yielding no visible error. Of course, this results in additional rendering time. In comparison to importance sampling the lightcuts method offers the benefit to generate noise-free results independent of the used radiance map. I think, it would be worth the effort to explore the possibility to combine both methods: i.e. using lightcuts to estimate the samples needed for a good result and afterwards sampling the radiance map by using multiple importance sampling. Presumable, this will reduce the problem with undersampled areas.

8.3 Future Work

There are several interesting directions left, which could be explored further. My lightcuts implementation could be extended to improve anti-aliasing efficiency. This could be done easily, if intersection information is used by all sample rays shot for one pixel. If the hit surface is the same or extremely similar in angle and material, it is possible to share the error information and thus reduce time for error estimation. Additionally, this also leads to a reduced shadow ray count per sample. Most of the time this is anti-aliasing for (almost) free - similar to other adaptive anti-aliasing algorithms. The limitation of maximum descent in the light tree is another idea to improve rendering speed, especially when it is more important to be fast than accurate. Priority queues can be used to split those nodes first, which yield a high error. This should generate fast and accurate results, because nodes producing large possible error are replaced first. The most important extension to the lightcuts implementation is the support of real global illumination. So far, diffuse inter-reflection is not possible with direct lighting, which would be an interesting feature. Alexander Keller's instant radiosity algorithm [15] could be used to enable indirect lighting for diffuse materials. Due to its mode of operation to distribute point lights to generate indirect illumination, it fits the lightcuts approach perfectly.

Appendix A

Source Code Snippets

A.1 Random Sampling of Spherical Patches

Listing 1 Random Sampling of Spherical Patches

```
1  SphericalSample(int xmin, int xmax, int ymin, int ymax ) {
2
3      float u = RandomFloat();
4      float v = RandomFloat();
5
6      // find out the range for the y-values
7      float kummin = m_Valskum[ymin];
8      float kummax = m_Valskum[ymax];
9      float kumdiff = kummax - kummin;
10
11     // Scale v to cover the entire range
12     v *= kumdiff;
13
14     // Make sure the search with v starts at the correct position
15     v += kummin;
16
17     // apply the inversion method by finding the xth element in the cdf
18     float *ptr = std::lower_bound(m_Valskum+ymin, m_Valskum+ymax+1, v);
19     int offset = (int) (ptr-m_Valskum-1);
20
21     // scale theta and phi to  match spherical coordinates
22     theta = offset * m_invheight * M_PI;
23     phi = (xmin  + u * (xmax-xmin)) * m_invwidth * TWOPI;
24 }
```

A.2 kd-Trie Generation

Listing 2 Random Sampling of Spherical Patches

```
KdTreeLC<NodeData, LookupProc>recursiveBuild(u_int nodeNum,
                int start, int end,
                vector<const NodeData *> &buildNodes) {

    // Create leaf node of kd-tree
    if (start + 1 == end) {
        nodes[nodeNum].initLeaf();
        nodeData[nodeNum] = *buildNodes[start];
        return;
    }
    BBox bound;
    // Compute bounds of data from start to end
    for (int i = start; i < end; ++i)
        bound = Union(bound, buildNodes[i]->p);

    // Use the bounding box's maximum extent as split axis
    int splitAxis = bound.MaximumExtent();
    int splitPos = (start+end)/2;
    // Sort elements by the selected split axis
    std::nth_element(&buildNodes[start], &buildNodes[splitPos],
        &buildNodes[end], CompareNode<NodeData>(splitAxis));

    // create internal kd-tree node
    nodes[nodeNum].init(buildNodes[splitPos]->p[splitAxis], splitAxis,NULL);
    nodeData[nodeNum] = *intNodes[nextFreeIntNode++];

    // Copy the bounding box into the internal nodes Data
    nodeData[nodeNum].bound = new BBox(bound);

    if (start < splitPos) { // Recursive call for remaining left children
        nodes[nodeNum].hasLeftChild = 1;
        u_int childNum = nextFreeNode++;
        recursiveBuild(childNum, start, splitPos, buildNodes);
    }
    if (splitPos < end) { // Recursive call for remaining right children
        nodes[nodeNum].rightChild = nextFreeNode++;
        recursiveBuild(nodes[nodeNum].rightChild, splitPos,
                       end, buildNodes);
    }

    // After children creation, create a representative light
    NodeData::buildRep(&nodeData[nodeNum], &nodeData[nodeNum+1],
                       &nodeData[nodes[nodeNum].rightChild] );
}
```

A.3 Lightcut Algorithm

Listing 3 Lightcut algorithm for point light sources

```
1
2       queue<u_int> workqupls; // Work Queue
3       queue<KDLightEl*> lightqupls; // Light Queue
4
5       // Start evaluation if the point light tree exists
6       if ( pointLT ) workqupls.push(rootPL);
7
8       // As long as potential nodes are left in the queue
9       while ( workqupls.size() > 0 ) {
10
11          // recieve light
12          u_int nodeNum = workqupls.front(); workqupls.pop();
13          KDLightEl* act = pointLT->LookupLightEl( nodeNum );
14
15          // Use leaf nodes directly for lighting
16          if ( ! act->bound ) {
17              lightqupls.push( act ); continue;
18          }
19
20          float est_error = estErrNPL( act, bsdf, p, wo, n );
21
22          // negative error is recieved if light from a cluster is
23          // completely arriving from behind
24          if (est_error < 0.0 ) continue;
25
26          // the node is directly used for lighting if the error is smaller
27          // than the predefined error threshold
28          if ( est_error < errorthresh ) {
29
30              lightqupls.push( act );
31
32          } else {
33              // Recieve the children of the node
34              u_int* cdrn = pointLT->getChildren(nodeNum);
35              if ( cdrn[0] ) {
36                  if ( cdrn[0] ) workqupls.push(cdrn[0]);
37                  if ( cdrn[1] ) workqupls.push(cdrn[1]);
38                  delete[] cdrn;
39      }     }
40
41      // Calculate direct lighting for collected light sources
42      while ( lightqupls.size() > 0 ) {
43          L_PL += estimateDirectLC( lightqupls.front()->repLight, p, n, wo,
                bsdf);
44          lightqupls.pop();
45      }
```

APPENDIX A. SOURCE CODE SNIPPETS

Bibliography

[1] S. Agarwal, R. Ramamoorthi, S. Belongie, and H. Jensen. Structured importance sampling of environment maps, 2003.

[2] Arthur Appel. Some techniques for shading machine renderings of solids. In *AFIPS Spring Joint Computer Conference*, pages 37–45, 1968.

[3] J. F. Blinn and M. E. Newell. Texture and reflection in computer generated images. *Communications of the ACM*, 19(10):542–547, October 1976.

[4] James F. Blinn. Models of light reflection for computer synthesized pictures. In *SIGGRAPH '77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pages 192–198, New York, NY, USA, 1977. ACM Press.

[5] P. Debevec and J. Malik. Recovering high dynamic range radiance maps from photographs, 1997.

[6] Paul Debevec. http://www.debevec.org/probes/.

[7] Paul Debevec. Rendering synthetic objects into real scenes: Bridging traditional and image-based graphics with global illumination and high dynamic range photography. *Computer Graphics*, 32(Annual Conference Series):189–198, 1998.

[8] Paul Debevec. A median cut algorithm for light probe sampling. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Posters*, page 66, New York, NY, USA, 2005. ACM Press.

[9] Manfred Ernst. Photo-realistic rendering on programmable graphics hardware. Diploma thesis, University of Erlangen-Nuremberg, Erlangen, July 2003.

[10] James A. Ferwerda, Sumanta N. Pattanaik, Peter Shirley, and Don Greenberg. A model of visual adaptation for realistic image synthesis. In *SIGGRAPH 1996*, pages 249–258, 1996.

[11] Rob Shakespeare Greg Ward, Larson Shakespeare. *Rendering With Radiance: The Art And Science Of Lighting*. Booksurge Llc, 2004.

[12] Henrik Wann Jensen, Stephen R. Marschner, Marc Levoy, and Pat Hanrahan. A practical model for subsurface light transport. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 511–518, August 2001.

[13] J.T. Kajiya. The rendering equation. In *ACM SIGGRAPH '86 Proceedings, vol.20*, pages 143–150, 1986.

[14] A. Keller. Quasi-monte carlo methods in computer graphics: The global illumination problem, 1995.

[15] Alexander Keller. Instant radiosity. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 49–56, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[16] Alexander Keller. *Quasi-Monte Carlo Methods for Photorealistic Image Synthesis*. PhD thesis, University of Kaiserslautern, Kaiserslautern, June 1997.

[17] Thomas Kemmer. Globale beleuchtungsberechnung in virtuellen szenen. Student thesis, University of Erlangen-Nuremberg, Erlangen, September 2005.

[18] Tomas Möller and Eric Haines. *Real-Time Rendering*. A K Peters, Natick, Massachusetts, 1999.

[19] F.E. Nicodemus, J.C. Richmond, and J.J. Hsia. Geometric considerations and nomenclature for reflectance. 1977.

[20] Michael Oren and Shree K. Nayar. Generalization of Lambert's reflectance model. *Computer Graphics*, 28(Annual Conference Series):239–246, 1994.

[21] Matt Pharr and Greg Humphreys. Infinite area light source with importance sampling, 2004.

[22] Matt Pharr and Greg Humphreys. *Physically Based Rendering*. Morgan Kaufmann Publishers, 2004.

[23] Wikipedia the free encyclopedia:. List of indices of refraction. http://en.wikipedia.org/wiki/List-of-indices-of-refraction, December 31st 2006.

[24] Wikipedia the free encyclopedia:. kd-trie. http://en.wikipedia.org/wiki/Kd-trie, January 4th, 2007.

[25] K. Torrance and E. Sparrow. Theory for off-specular re¿ection from roughened surfaces. *Journal of the Optical Society of America*, 57(9):1105–1114, 1967.

[26] Bruce Walter. Notes on the ward brdf. Technical report PCG-05-06, Program of Computer Graphics, Cornell University, April 2005.

[27] Bruce Walter, Adam Arbree, Kavita Bala, and Donald P. Greenberg. Multidimensional lightcuts. *ACM Trans. Graph.*, 25(3):1081–1088, 2006.

[28] Bruce Walter, Sebastian Fernandez, Adam Arbree, Kavita Bala, Michael Donikian, and Donald P. Greenberg. Lightcuts: a scalable approach to illumination. 24(3):1098–1107, July 2005.

[29] Turner Whitted. An improved illumination model for shaded display. In *ACM vol. 23, no. 6*, pages 43–349, 1980.

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Ich bin damit einverstanden, dass die Arbeit veröffentlicht wird und dass in wissenschaftlichen Veröffentlichungen auf sie Bezug genommen wird.

Der Friedrich-Alexander-Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Graphische Datenverarbeitung, wird ein (nicht ausschließliches) Nutzungsrecht an dieser Arbeit sowie an den im Zusammenhang mit ihr erstellten Programmen eingeräumt.

Erlangen, 08. Januar 2007

_____

(Thomas Rudolf Kemmer)