Karlsruhe Institute of Technology

Importance Sampling in Many Lights Trees

Bachelor's Thesis of

Beini Ma

at the Department of Informatics
Computer Graphics Group

Reviewer:	Prof. Dr.-Ing. Carsten Dachsbacher
Second reviewer:	Prof. B
Advisor:	M.Sc. Alisa Jung

23. April 2018 – 23. August 2018

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 23. August 2018

. . . . . .

(Beini Ma)

Abstract

When rendering complex scenes with a high requirement for image quality, path tracing has been the most popular tool in the last few years. Compared to scanline techniques like rasterization, path tracing offers a higher image quality with the trade-off of longer rendering times. That is the main reason why path tracing is the preferred algorithm when creating pictures or videos in beforehand. Recently, real-time ray tracing has also become a big subject in computer graphics. Keeping the rendering times as short as possible is desirable in both cases.

This work discusses an acceleration structure that allows for faster rendering of scenes with a large number of lights. Instead of choosing a random light source in the scene, we will try to sample lights that have a high contribution to the point to be lighted more often. That way, our algorithm converges faster and the image quality with similar rendering times will be better than using conventional light sampling strategies.

Zusammenfassung

Wenn wir Szenen rendern, in denen wir hohe Anforderungen auf die Qualität des Bilds setzen, ist path tracing das beliebteste Werkzeug in den letzten Jahren gewesen. Verglichen mit Rasterisierung liefert path tracing eine höhere Bildqualität mit einer längeren Renderzeit. Das ist der Hauptgrund, warum path tracing der bevorzugte Algorithmus ist, wenn man Bilder oder Videos im Voraus erstellt. Neuerdings ist Echtzeit ray tracing ein großes Thema in der Computergrafik geworden. Die Renderzeiten so kurz wie möglich zu halten, ist in beiden Fällen wünschenswert.

In dieser Arbeit führen wir eine Beschleunigungsstruktur ein, die es uns erlaubt, schneller Szenen mit eine großen Zahl von Lichtquellen zu rendern. Statt ein zufälliges Licht der Szene zu nehmen, werden wir versuchen, öfters Lichtquellen abzutasten, die einen hohen Einfluss auf den Punkt haben, den wir beleuchten wollen. Somit konvergiert unser Algorithmus schneller und die Bildqualität wird bei gleichen Renderzeit besser sein als mit konventionellen Methoden, die Lichtquellen aussuchen.

Contents

List of Figures

1 Introduction

1.1 Problem/Motivation

Path tracing is one of the most important rendering techniques when creating highly realistic pictures. It allows us to render the scene much closer to reality when compared to typical scanline rendering methods at the cost of more computations. In situations where the images can be rendered ahead of time, such as for visual effects or films, we can take advantage of the better results of ray tracing. Recently, there have been talks about real time ray tracing. NVIDIA claims that real-time ray tracing with a single GPU in games and other graphic applications was possible. Keeping the rendering times as short as possible is not exclusive for real-time ray tracing. Animated films like the in 2014 released *Big Hero 6*, have exploded with more and more complicated scenes. Disney released statistics showing the movie was rendered with 1.1 million computational hours per day, distributed on a 55.000-core computer across four geographic locations. If the rendering times are so long, they are important even when the system does not need to provide the rendering of the movie in real-time. *Big Hero 6* plays in a city called San Fransokyo. It contains around 83.000 buildings, 215.000 streetlights and 100.000 vehicles. At nighttime, all these objects can be light sources, which leads to a gigantic number of emitters. When rendering with path tracing, we have to light certain points of the scene with light sources. With this huge number of lights, it is clearly not practical to calculate the lighting of each point for every single light in the scene. Also, choosing random light sources will definitely not achieve the image quality we want for a pleasant user experience in a reasonable time. [Vol14; 18]

There are other sampling approaches that try to limit the time required to render these scenes with a big amount of lights. For instance, we could say that the probability of a point of the scene sampling a certain light is only dependent on the emission power of said light. We would make a distribution that only takes into account the emission power of the lights. To light a specific point we would then sample a single light according to the distribution function we built earlier. Obviously there are a lot of problems with this approach. An area light source or a spotlight could be facing towards a completely different direction and may not have any effect on the point. Or the light source could be potentially too far away to have a noticeable effect on the point. This sampling technique asserts a faster sampling speed but can lead to very noisy images that we are trying to avoid.

In this bachelor's thesis we discuss a light sampling technique that improves the rendering speed in scenes with many emitters while still maintaining an comparable quality.

1.2 Content

We will introduce the light bounding volume hierarchy as an acceleration data structure to render scenes with complex illumination. In chapter 2, we will talk about some concepts of probability theory and path tracing that are required for the algorithm discussed in this thesis. Then, in chapter 3, we will give an in-depth introduction of the algorithm. In chapter 4, we will evaluate the algorithm. In chapter 5, we will reference related work and in chapter 6 we will come to a conclusion.

2 Preliminaries

2.1 Probability Theory Basics

In this section we will be discussing basic ideas and define certain terms from the probability theory. We will assume that the reader is already familiar with most of the concepts and therefore will only give a short introduction. A thorough introduction into statistics is out of the scope of this thesis and we refer the reader to read E. T. Jaynes *Probability Theory: The Logic of Science* for introducing literature. [Jay03]

2.1.1 Random Variable

A random variable X is a variable whose values are numerical outcomes chosen by a random process. There are discrete random variables, which can only take a countable set of possible outcomes and continuous random variables with an uncountable number of possible results. For instance, flipping a coin would be a random variable drawn from a discrete domain which can only result to heads or tails, while sampling a random direction over a unit sphere can produce infinite different directions. In rendering and particularly in path tracing, we often sample certain directions or light sources in order to illuminate the scene, therefore we will be handling both discrete and continuous random variables, albeit with the latter in the most cases.

The so-called canonical uniform random variable ξ is a special continuous random variable that is especially important for us. Every interval in its domain $[0, 1)$ with equal length are assigned the same probability. This random variable makes it very easy to generate samples from arbitrary distributions. For example, if we would need to sample a direction to estimate the incident lighting on a point, we could draw two samples from ξ and scale these two values with appropriate transformations so they reflect the polar coordinates of the direction to sample.

2.1.2 Probability Density Function

For continuous random variables, probability density functions (PDF) illustrate how the possible outcomes of the random experiment are distributed across the domain. They

must be nonnegative and integrate to 1 over the domain. $p : D \longrightarrow \mathbb{R}$ is a PDF when

$$\int_D p(x)\mathrm{d}x = 1. \tag{2.1}$$

Integrating over a certain interval $[a, b]$ gives the possibility that the random experiment returns a result that lies inside of given interval:

$$\int_a^b p(x)\mathrm{d}x = P(x \in [a, b]). \tag{2.2}$$

It is evident, that $P(x \in [a, a]) = 0$ which reflects the fundamental idea of continuous random variables: The possibility of getting a sample that exactly equals a certain number is zero. Therefore, PDFs are only meaningful when regarded over a interval and not over a single point.

2.1.3 Expected Values and Variance

As the name already indicates, the expected value $E_p[f(x)]$ of a function f and a distribution p specifies the average value of the function after getting a large amount of samples according to the distribution function $p(x)$. Over a certain domain D, the expected value is defined as

$$E[f(x)] = \int_D f(x)p(x)\mathrm{d}x. \tag{2.3}$$

The variance defines a measure that illustrates the distance between the actual sample values and their average value. Formally, it is defined by the expectation of the squared deviation of the function from its expected value:

$$V[f(x)] = E\Big[(f(x) - E[f(x)])^2\Big]. \tag{2.4}$$

When we talk about Monto Carlo Intergration later, the variance is a strong indicator of the quality of the PDF we chose. The main part of this thesis will be to minimize the variance of light sampling methods.

2.1.4 Error and Bias

For an estimator F_N, the parameter to be estimated I and a sample x the error $e(x)$ is defined as

$$e(x) = F_N(x) - I. \tag{2.5}$$

Since the error is dependent on the certain sample we took, we will also introduce bias. The bias b of an estimator is the expected value of the error:

$$b(F_N) = E[F_N - I]. \tag{2.6}$$

An estimator F_N is unbiased, if $b(F_N) = 0$ for all sample sizes N. Informally, it means that the estimator is going to return the correct value on average. In the next section, we will introduce an unbiased estimator, the Monte Carlo estimator.

2.2 Monte Carlo Integration

When generating an image using path tracing, we will be dealing with integrals and our main task will be to estimate the values of these integrals. Since they are almost never available in closed form, like the incident lighting of a certain point that theoretically requires infinite number of rays traced over infinite dimensions, analytical integration models rarely work. Instead, we have to use numerical integration methods give an appropriate estimation for these integrals. One of the most powerful tools we have in this regard is the Monto Carlo integration. We will be discussing the advantages of Monte Carlo integration, as well as its constraints and mechanisms how we can deal with these limits.

Using random sampling methods, we want to evaluate the integral

$$I = \int_a^b f(x)dx \tag{2.7}$$

with Monte Carlo integration. Different to Las Vegas algorithms, which are also randomized algorithms but always produce the correct results, Monte Carlo integration has a non-deterministic approach. Every iteration of the algorithm provides a different outcome and will only be an approximation of the actual integral. Imagine that we want to integrate a function $f : D \to \mathbb{R}$. The Monte Carlo estimator states that with samples of uniform random variables $Xi \in [a, b]$ and number of samples N the expected value $E[F_N]$ of the estimator

$$F_N = \frac{b-a}{N} \sum_{i=1}^{N} f(X_i) \tag{2.8}$$

is equal to the integral, since:

$$
\begin{aligned}
E[F_N] &= E\left[\frac{b-a}{N}\sum_{i=1}^{N} f(X_i)\right] \\
&= \frac{b-a}{N}\sum_{i=1}^{N} E[f(X_i)] \\
&= \frac{b-a}{N}\sum_{i=1}^{N} \int_a^b f(x)p(x)dx \\
&= \frac{1}{N}\sum_{i=1}^{N} \int_a^b f(x)dx \\
&= \int_a^b f(x)dx.
\end{aligned} \tag{2.9}
$$

If we use a PDF $p(x)$ instead of an uniform distribution, the estimator

$$
F_N = \frac{1}{N}\sum_{i=1}^{N} \frac{f(X_i)}{p(X_i)} \tag{2.10}
$$

is used to approximate the integral. Being able to use arbitrary PDFs is essential for solving the light transport problem and the importance of choosing a good PDF $p(x)$ will be explained in the section 2.3.

The Monto Carlo estimator is unbiased, because

$$
\begin{aligned}
b(F_N) &= E\left[\frac{1}{N}\sum_{i=1}^{N} \frac{f(X_i)}{p(X_i)} - I\right] \\
&= E\left[\frac{1}{N}\sum_{i=1}^{N} \frac{f(X_i)}{p(X_i)}\right] - E[I] \\
&= E\left[\frac{f(X)}{p(X)} - I\right] \\
&= I - I = 0.
\end{aligned} \tag{2.11}
$$

While standard quadrature techniques converge faster than the Monte Carlo integration in low dimensions, the Monte Carlo integration is the only integration method that allows us to deal with higher dimensions of the integrand, since it's convergence rate is independent from the dimension. In fact, standard numerical integration techniques do not work very well on high-dimensional domains since their performance becomes exponentially worse as the dimension of the integral increases. Later, we will explain why the light

transport problem of the path tracing algorithm is theoretically an infinite-dimensional problem and therefore we will be estimating the integrals in our light transport equations with Monte Carlo integration at a convergence rate of $O(\sqrt{N})$. [Vea97]

2.3 Importance Sampling

As we have mentioned earlier, the Monte Carlo estimator allows us to use any distribution $p(x)$ to get the samples from. Abusing the fact, that the Monte Carlo estimator

$$F_N = \frac{1}{N} \sum_{i=1}^{N} \frac{f(X_i)}{p(X_i)} \tag{2.12}$$

converges faster if we choose a sampling distribution $p(x)$ that is roughly proportional to the function $f(x)$, this will be our main variance reduction method. Suppose, we could use any distribution $p(x)$ to pick our samples from. We would then choose a distribution $p(x) = cf(x)$, basically a distribution proportional to the function of the integrand. Then, after each estimation, the value we would have calculated would be

$$\frac{f(X_i)}{p(X_i)} = \frac{1}{c} = \int f(x)dx. \tag{2.13}$$

We would be getting a constant value and since the Monte Carlo estimator is unbiased, every single of our estimates would exactly return the value of the integral and our variance would be zero. Obviously, that does not work, since it requires us to know the value of the integral $\int f(x)dx$ in beforehand, and then using Monte Carlo techniques to evaluate the integral would be pointless. Nonetheless, if we are able to find a distribution $p(x)$ that has a similar shape to $f(x)$, we would archive a lower expected variance with the number of samples. Therefore, when importance sampling, we are taking samples out of a distribution that is similar to the integral to estimate. To put it in relation to path tracing, assume we want to calculate the incident lighting for a given point on a diffuse surface. Imagine, the light sources of the scene are all focused on a certain direction of the point. It would make a lot of sense to mainly sample the directions that are similar to the vectors pointing to the light sources, since their contribution will be the biggest. In this case, we would want to use a distribution that is similar to the light source distribution in the scene given the position of the point.

2.4 Multiple Importance Sampling

We have been discussing how to deal with integrals of the form $\int f(x)dx$ with Monte Carlo techniques. While path tracing, we are mainly faced with situation where we have

to evaluate integrals that consists of a products of functions. In fact, the main function we need to estimate for the direct lighting given a certain point p, outgoing direction ω_o and the solid angle ω_i is in that form:

$$L_o(p, \omega_o, \omega_i) = \int_{\varsigma^2} f(p, \omega_o, \omega_i) L_d(p, \omega_i) |cos\theta_i| d\omega_i. \tag{2.14}$$

$f(p, \omega_o, \omega_i)$ describes the reflectance of the surface at given point p, and the ingoing and outgoing directions ω_i and ω_o. $L_d(p, \omega_i)$ specifies the incident lighting at given point and direction ω_i. It is apparent, that using a single distribution function in every situation will not yield the optimal results. A specular surface would only reflect the light in very specific angles and in these cases, having a distribution, that has a similar form of $f(p, \omega_o, \omega_i)$ would be preferable. On the other hand, if the surface was diffuse, we would obtain better results, when using a distribution that has a similar shape of the distribution of light sources over the unit sphere.

The solution to this dilemma is multiple importance sampling. When using multiple importance sampling, we will draw samples from multiple distributions to sample the given integral. The idea is that, although we do not know which PDF will have a similar shape to the integrand, we hope that one of the chosen distributions match it fairly well. If we want to evaluate the integral $\int f(x)g(x)dx$ and p_f and p_g are our distribution functions, then the Monte Carlo estimator with multiple importance sampling is

$$\frac{1}{n_f} \sum_{i=1}^{n_f} \frac{f(X_i)g(X_i)w_f(X_i)}{p_f(X_i)} + \frac{1}{n_g} \sum_{j=1}^{n_g} \frac{f(X_j)g(X_j)w_g(X_j)}{p_g(X_j)}, \tag{2.15}$$

where n_f and n_g are the number of samples taken from their respective distribution function and w_f and w_g are weighting functions so the expected value of the estimator still matches the value of the integral. Multiple importance sampling is able to significantly lower the variance because a good weighting function should dampen contributions with low PDFs. We will present two weighting heuristics, the *balance heuristic* and the *power heuristic*.

A good way to reduce variance is the balance heuristic:

$$w_b(x) = \frac{n_s p_s(x)}{\sum_i n_i p_i(x)}. \tag{2.16}$$

The power heuristic with exponent $\beta = 2$ is often a better heuristic to reduce variance:

$$w_p(x) = \frac{(n_s p_s(x))^2}{\sum_i (n_i p_i(x))^2}. \tag{2.17}$$

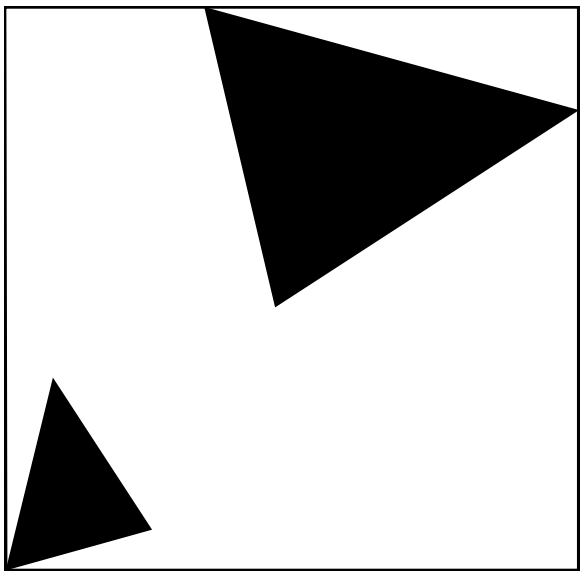


Figure 2.1: Two-dimensional bounding box for two triangles

For a more detailed explanation of the weighting heuristics, we recommend "Robust Monte Carlo Methods for Light Transport Simulation" by Veach to the reader. [Vea97]

2.5 Axis-Aligned Bounding Box

A data structure we will be using frequently is minimum bounding boxes. A minimum bounding box for a set of geometric primitives is defined as the box with a certain smallest measure, that contains every single primitive in the set. While path tracing, these boxes can be two- or three-dimensional, making their respective measures the area and the volume. Because many parts of the path-tracing algorithm operate on axis-aligned regions of space, it makes sense to use boxes, whose edges are parallel to the coordinate axes. These boxes are called axis-aligned bounding boxes (AABB) and we can see an example of it in Figure 2.1. Note, that in this case we have a two-dimensional bounding box but it is not hard to imagine, that AABBs work the same way in higher dimensions. To combine two AABBs, we just need to take the minimum and the maximum for each dimension of the two AABBs.

2.6 Bounding Volume Hierarchies

Acceleration data structures are mandatory for path tracers. They reduce the amount of ray-primitive intersection tests to logarithmic in the number of primitives. The two main approaches are to either split the room among the space or to choose a particular number

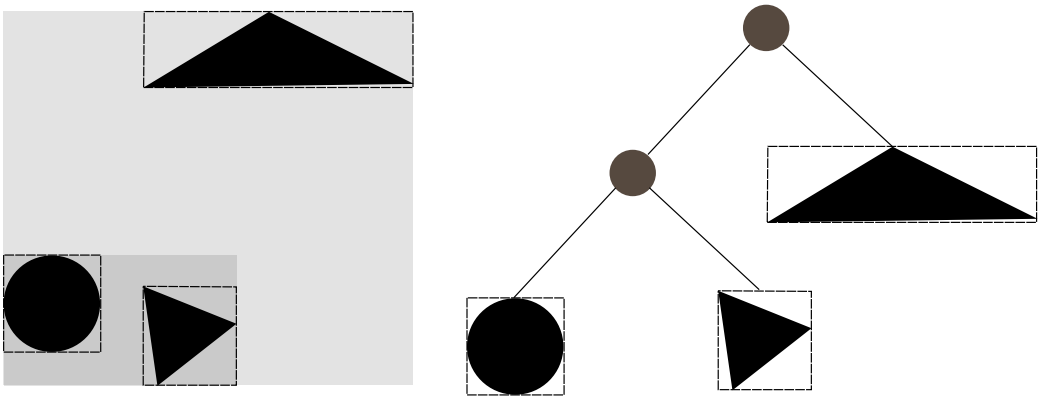


Figure 2.2: On the left side the geometric representation of the primitives in the room and on the right side the representation in the BVH

of primitives and wrap a bounding box around these primitives so when we intersect a ray with the scene only specific sets of primitives need to be tested for intersections. The acceleration data structure used in PBRT, the framework renderer we used, is the bounding volume hierarchy (BVH). It uses the latter approach. [PJH18]

An example to partition the space would be kd-trees that splits the space parallel to one of the axis in each step. Those two sets of primitives are split recursively again until every leaf has only a maximum amount of primitives. While kd-trees offer slightly faster ray intersection tests than BVH, their disadvantages outweigh. BVHs can be built much faster and are generally more numerically robust, so it is less likely to happen to miss intersection due to round-off errors than kd-trees are. Other desired traits of BVHs are that they require a maximum number of $2n - 1$ nodes for n primitives and, since we are partitioning the primitives, we will never have the same primitive in two different nodes which can happen in kd-trees. [PJH18]

2.6.1 BVH Construction

Obviously, the first step is the construction of the BVH data structure. The basic structure of BVH is a binary tree, with every node containing pointers to two children and every leaf holding a set maximum number of primitives. The construction technique that is most popular and also used in PBRT is the top-down construction. At the beginning of the construction we hold a set containing every single primitive of the scene. With each step, according to a split method we have chosen, we partition the primitives into two disjointed sets. Note that regardless of the split method we chose, we will always split the primitives among a certain axis to retain locality. Examples for popular splitting heuristics are "Middle", which just partitions the room among the middle of a chosen axis, "EqualCounts", which partitions the primitives among a chosen axis so the disjointed sets have the same number of primitives, and SAH (Figure 2.6.1.1). While the first two split

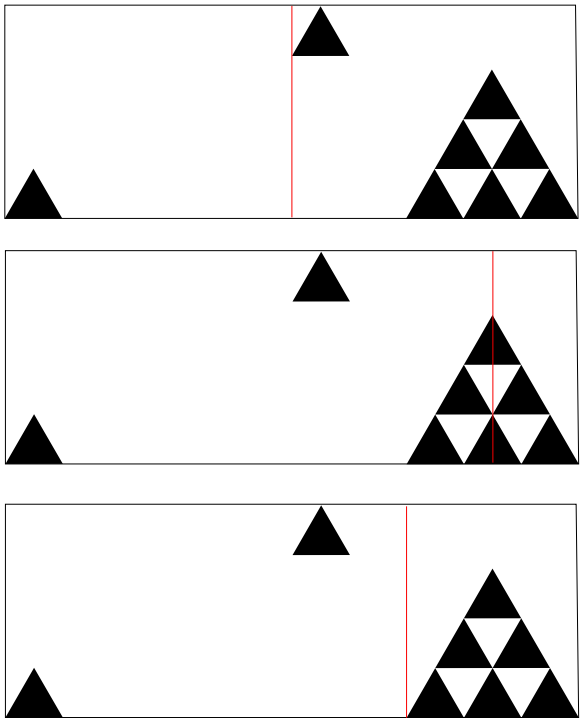


Figure 2.3: Three different split strategies: "Middle" in the top, "EqualCounts" in the middle and SAH in the bottom

methods allow a easy and fast construction, their quality is very lacking. Recursively splitting the root into two disjointed sets of primitives constructs the tree. When a certain node holds less than a defined number of primitives, we do not split again and instead call the node a leaf.

Every node needs to be holding onto some information in order to allow the intersection tests. Evidently, every inner node needs to have a reference to both of its children. As we have mentioned earlier, it also needs to store its bounding volume that wraps around all its primitives. When testing if we need to intersect any of the primitives of a node with a ray, we will be instead intersect the box with the ray. Typically, we will be using minimum fit axis-aligned bounding boxes for this task.

2.6.1.1 Surface Area Heuristics

We have presented two split methods in subsection 2.6.1. While they both do work well in some situations, they both have clear disadvantages, especially if the primitives are not evenly distributed over the scene. Looking at Figure 2.3, it is obvious, that both the "Middle" and "EqualCounts" split the scene in a way that can lead to very inefficient intersection tests. In the "Middle" split, although the triangle mesh on the right child node are focused on a very small space, the drawn ray will still make intersection test with

each of the triangles of the right child node. Similarly, in the "EqualCounts" split, many unnecessary intersection tests are made. A desirable split would be the third split and we will now introduce a split method that favors these kinds of splits.

The surface area heuristics (SAH) defines a model that assigns a quality to a given split. The idea behind the SAH cost model is actually very simple. When we want to decide the best possible split for given primitives, first, we have to decide if it is perhaps better not to split at all. That means, when a ray traverses through the bounding box of the node, we would have to make an ray-primitive intersection test with each primitive. That means the cost c_g is

$$\sum_{i=1}^{N} t_{isect}(i), \tag{2.18}$$

where N is the number of primitives and $t_{isect}(i)$ the time required for the intersection test with the i-th primitive. For simplicity, we will assume, that every ray-primitive intersection test takes the same amount of time.

Clearly, we can also split the primitives. The cost $c(A, B)$ would be

$$c(A, B) = t_{trav} + p_A \sum_{i=1}^{N_A} t_{isect}(a_i) + p_B \sum_{j=1}^{N_B} t_{isect}(b_j) \tag{2.19}$$

with t_{trav} being the additional overhead time to traverse through a child, p_A and p_B being the probabilities that the ray passes through the respective child node if we assume that the rays are evenly distributed in the scene. The probabilities p_A and p_B can be calculated using their respective surface areas:

$$p_C = \frac{s_C}{s_G}, \tag{2.20}$$

where s_G is the surface area of the node and s_C is the surface area of the child. This equation is the SAH.

Compared to the other two split strategies we presented, SAH actually adapts to the given geometry and tries to minimize the amount of ray-primitive intersection tests when traversing the tree. Therefore, SAH is much more versatile and performs better in most scenes.

2.6.2 BVH Traversal

Since the traversal of the bounding volume hierarchy is not relevant for this thesis, we will only give a short introduction of the general idea of it. If the reader is interested and would like to study additional readings about this subject, we would recommend the book "Physically Based Rendering" by Pharr, Jakob and Humphreys. [PJH16]

We have constructed our BVH as an acceleration data structure for ray-primitive intersection tests now. In the next step, we will be intersecting rays that are tracked in out path tracing algorithm, with the BVH. First, we will be intersecting the given ray with the root of our tree. If there is an intersection between the bounding volume of the current node and the ray, we will be further checking for intersections of the ray with the children of our current node. If we arrive at a leaf node that contain primitives, we will make ray-primitive intersection tests with every primitive in the node. Note, that we can stop after we have found the first actual interaction with a primitive and have asserted that any other intersection of the scene with the ray is further away than the intersection we already found. This way, we avoid many unneeded ray-primitive intersection tests and save a lot of time.

2.7 Path Tracing Basics

In this section we will discuss some of the basics of path tracing. It was first introduced in 1986 by James Kajiya in the proceedings of SIGGRAPH '86 [Kaj86]. The general idea is to generate paths along the scene that start at the camera and end at a light source of the scene. In the beginning, we will be spawning rays that start at the point of the camera. Each of these rays will gather the color at the respective pixel they represent. At the intersection of the ray with the scene, we will be evaluating the rendering equation we have mentioned earlier:

$$L_o(p, \omega_o, \omega_i) = \int_\Omega f(p, \omega_o, \omega_i) L_d(p, \omega_i) |cos\theta_i| d\omega_i. \tag{2.21}$$

We have noticed earlier, that this problem is a infinite-dimensional problem. That is because in order to calculate the color at the point we hit, we have to evaluate this equation for different directions ω_i according to the Monte Carlo integration. The reason for that is, as we have acknowledged earlier, it is impossible to calculate the result of this integral analytically. Instead, we will be sampling directions according to multiple distribution functions and estimate the contribution of these directions. The ore, we have to spawn rays that will hit another point in the scene. As the result, we will get more light transport equations that need to be evaluated. To render the perfect image with path tracing, we could not stop after a finite number of steps, but in practice, we will break after a certain number of bounces or after the contribution of the point would be too low to actually

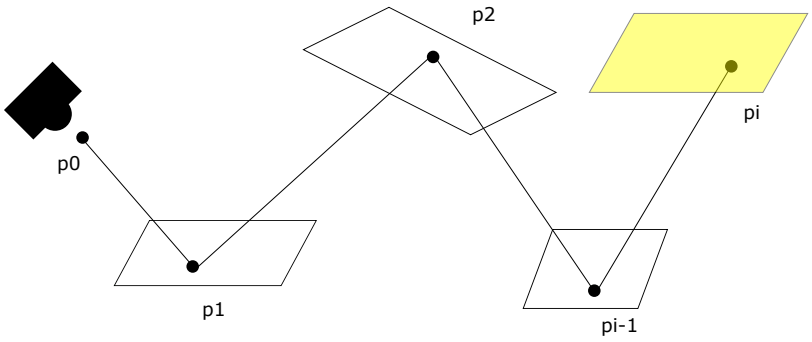


Figure 2.4: A typical path for the path tracer starting at the camera and sampling a light source as the last vertex.

change the image for the human eye. Instead, we will connect the current point with a sampled light source and finish the path. Such a path can be seen in Figure 2.4.

We have already acknowledged the importance of using a sampling distribution that has a similar form to the functions of the integrand, when using Monte Carlo integration. In our light transport equation, we have two different functions. $f(p, \omega_o, \omega_i)$ is defined by the bidirectional scattering distribution function (BSDF) of the material of the point, basically a function that defines how the material reflects incident lighting, and consequently, we can use a distribution function similar to it to sample the directions ω_i. The other function is the contribution of the light sources for the point $L_d(p, \omega_i)$. The sampling distribution needed to importance sample this function would be a distribution that reflects the amount of incident lighting from a given direction ω_i. Now, suppose we have a situation where the scene has a substantial number of light sources, say 10.000 or even one million light sources. In this case, it would take a great time to estimate this distribution. In this thesis, our goal is to deal with these kinds of scenes with a gigantic number of light sources by creating an accelerator data structure, the light bounding volume hierarchy. This data structure importance samples a light given the intersection point that can be then used to estimate the light transport equation.

3 The Algorithm

The algorithm and the underlying acceleration data structure we chose is inspired by the tree construction and traversal algorithm of the bounding volume hierarchy. Similarly to the BVH, we will construct a binary tree before rendering, and traverse through the tree to sample the appropriate light when we are given the vertex to be shaded in the scene. Analogous to most BVH algorithms, our light BVH construction runs on a single thread, while the traversal can run on concurrent threads. Since the construction of the tree takes only a small fraction of the rendering time, this does not pose a problem even on scenes with over one million light sources. The construction time of the scene rendered in Figure 4.8 took about 23 seconds on a computer with a Intel® Core™ i7-4770K processor and 16 gigabyte of main memory.

First, when we have access to all of the light sources of the scene, we want to create a binary tree data structure that includes every single light source of the scene. We call our tree the light bounding volume hierarchy (light BVH), because similarly to the BVH construction, we will also split the scene spatially parallel to coordinate axis. We will also use a heuristic to find the best split of all three dimensions, comparable to the surface area heuristic. Our heuristic, the surface area orientation heuristic (SAOH) has some crucial differences to the SAH. Firstly, instead of amounting the number of primitives, we will factor in the total emission power of the lights. Second, we have added an orientation factor. This orientation factor tries to keep light sources with similar orientations in the same node. This way, we split the branches in a way so different orientations most likely branch to different children of the tree and thus, finding the more likely child when traversing through the light BVH later will be clearer. When we have a node with only a single light source, this node is a leaf.

After we have constructed our light BVH, the next step would be traversing through the tree when we want to sample a light given a vertex to be shaded. Obviously, we don't just want to return a random light source which would make this algorithm pointless, but instead we want to importance sample a light that probably has a strong contribution to the incident radiance at the vertex. Therefore, we will define an importance measure for every node given the vertex to be shaded, that factors in the distance between the vertex and the node, the emission power of the node and an angle importance factor. We will start at the root of the light BVH and generate a uniform random number. Then, for every branch, a decision has to be made which child to take based on the importance measure and that random number. When we have arrived at a leaf node, that will be the light source we return.

```
struct Bounds_o {
    Vector3f axis;
    float theta_o;
    float theta_e;
}
```

Figure 3.1: Our internal representation of the Bounds_o struct

What we have just described was our algorithm when we only want to sample a single light source. Conty and Kulla have shown in their paper ([CK17b]), that there are situations, where it is desirable to sample multiple light sources for one intersection vertex. That is the reason why we allow the user to define a split threshold. Based on the split threshold and a score which we calculate for every node, instead of sampling only the left or the right child, it is possible to sample both nodes. At the end of our algorithm, we will return one or multiple sampled lights. Afterwards, the sampled lights are used for the path tracing algorithm.

Note that the general ideas presented in our algorithm are heavily influenced by the works of Conty and Kulla ([CK17a; CK17b]), a contribution by Sony Pictures in SIGGRAPH 2017. Later, they released another paper [CK18] which explains their algorithm in greater detail but that was after we have already finished developing our version of the algorithm. We will try to mark the spots where our version of the algorithm has received changes to the original algorithm.

3.1 Own Data Structures

In this section we will be discussing our own data structures. We will show our internal representation of our data structures now and reason why they are represented in the way they are.

3.1.1 Bounds_o

First, we will introduce the Bounds_o struct (Figure 3.1). This struct represents the orientation bounds of a certain light source or a whole node. For a single light, the axis defines the orientation direction of the light source, while for a node, the axis represents the interpolated axes of all light sources included in the node. Theta_o defines the maximum angle between the axis of the node and any light source included in this node. Theta_e defines the additional emission range added to theta_o for spotlights or area light sources. How we have pictured the geometric representation can be seen in Figure 3.2. In our im-

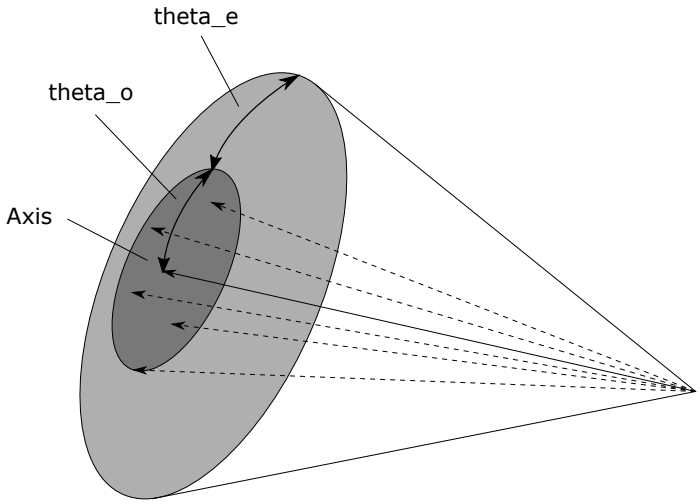


Figure 3.2: geometric idea of the Bounds_o struct

plementation, we accept point lights, spotlights and area light sources as light sources. This is how we initialize the Bounds_o struct for single lights:

- Point light
  - Axis = (1, 0, 0)
  - theta_o = π
  - theta_e = 0
- Spotlight
  - Axis = spot direction
  - theta_o = spot's aperture angle until falloff
  - theta_e = spot's falloff angle
- Area light
  - Axis = normal of the geometric representation
  - theta_o = 0
  - theta_e = $\pi/2$

In general, we defined the axes, orientation and emission ranges in the same way Conty and Kulla did ([CK17b]). We have made two small changes: First, we set theta_e for point lights to zero because the angle over a sphere is completely covered by $2 * \pi$ which is already comprised by theta_o. The second change we made are changes to the orientation bounds of spotlights. Conty and Kulla did not make a difference between a spotlights

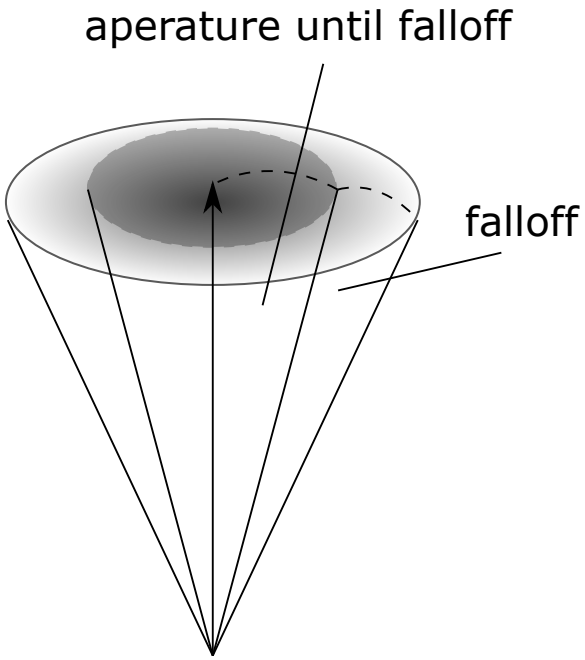


Figure 3.3: Our spotlight orientation bounds

aperture angle and their falloff angle. We make this distinction due to how PBRT was implemented which can be seen in Figure 3.3.

3.1.2 Node representation

We used two different implementations to represent a node in our light BVH tree (Figure 3.4 and 3.5). Both implementations are very similar, containing the two bounding members *bounds_w* for the world space bounds and *bounds_o* for the orientation bounds, as well as other other information that we will need for the tree traversal later. *Energy* defines the combined energy of the lights under this node, *nLights* describes the number of lights under this node, *centroid* stores the centroid of the world space bounds, and *splitAxis* stores the coordinate axis that splits the two children of this node.

The main difference between the two implementations is the way to access the children of the nodes. In our *LightBVHNode*, we have explicit pointers to the two children, while we define the children implicitly in out *LinearLightBVHNode*. Since all of our *LinearLightBVHNodes* are stored in aligned memory, the way we access the left child is just by incrementing the pointer of the current *LinearLightBVHNode* by 1. The index needed to access the right child is stored in $secondChildOffset$. To access the second child, we just add $secondChildOffset$ to the base pointer of the root of the tree. If the node is a leaf of the tree and thus only contains one light source, $lightNum$ stores the index of that light source in both implementations.

```
struct LightBVHNode {
    Bounds3f bounds_w;
    Bounds_o bounds_o;
    Point3f centroid;
    float energy;
    LightBVHNode *children[2];
    int splitAxis, nLights, lightNum;
};
```

Figure 3.4: Our internal representation of the LightBVHNode struct

```
struct LinearLightBVHNode {
    Bounds3f bounds_w;
    Bounds_o bounds_o;
    Point3f centroid;
    float energy;
    union {
        int lightNum;
        int secondChildOffset;
    };
    int nLights, splitAxis;
};
```

Figure 3.5: Our internal representation of the LinearLightBVHNode struct

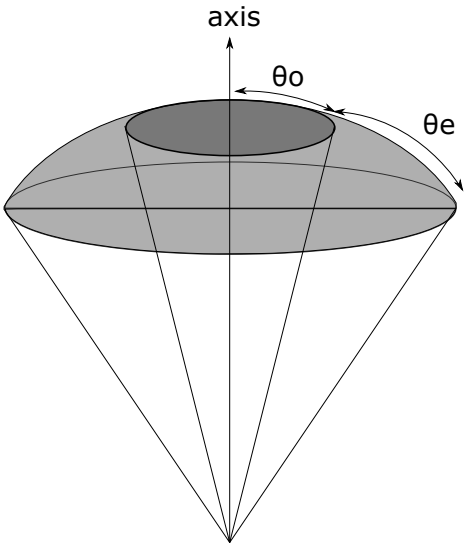


Figure 3.6: Bounding cone measure

3.2 Surface Area Orientation Heuristic

We have mentioned that we use a different splitting heuristic to build our tree as opposed to using the SAH in a BVH. But we do follow a similar main idea to split the primitives (in our case, the lights) parallel to a coordinate axis, which means that we split in world space. Comparable to splitting the BVH with SAH, where the goal is to minimize the cast of traversing a branch combined with the probability of hitting it, we want to minimize the probability of sampling a branch here. Also, instead of only regarding the number of primitives for each branch, we will rather take into account the combined energy of all the light sources of each branch. Equivalently to SAH, we will also include the surface area in our calculations. So while we do split in world space, we will use the orientation bounds, along with the surface area and the energy, to determine the quality of a certain split. In summary, we define the probability of sampling a given cluster C in the surface area orientation heuristic for as:

$$P(C) = M_A(C) \cdot M_\Omega(C) \cdot Energy(C), \tag{3.1}$$

$M_A(C)$ being the surface area of the cluster, $M_\Omega(C)$ being the bounding cone measure. The surface area of a clusters bounding box with length l, width w and height h is the sum of its six faces:

$$M_A(C) = 2(lw + wh + hl). \tag{3.2}$$

We used a weighted definition for our bounding cone measure. Remember that we have defined θ_o as the bounding angle that encloses the orientation vectors of all the light

sources in the cluster, while θ_e describes their emission range. Obviously, the cone enclosed with θ_o will most likely have a greater emission power and thus, we will weight the volume confined by θ_o with a cosine. A drawing of the individual cones can be seen in Figure 3.6. The measure we have used and has been described by Kulla and Conty ([CK17b]) is as follows:

$$M_\Omega(C) = 2\pi \cdot \left[(1 - \cos(\theta_o)) + \int_{\theta_o}^{\theta_o+\theta_e} \cos(\omega - \theta_o) * \sin(\omega) d\omega\right]. \tag{3.3}$$

Then, we describe the quality of a split as:

$$c(C_{left}, C_{right}) = \frac{P(C_{left}) + P(C_{right})}{P(C)}. \tag{3.4}$$

Consequently, the quality of a split is inverse to its cost:

$$q(C_{left}, C_{right}) = \frac{1}{c(C_{left}, C_{right})}. \tag{3.5}$$

It is notable that our heuristic tries to find out the possible split in all three dimensions with the given cluster as opposed to the SAH which only considers one alternating dimension. Therefore, we will try to find the split with the highest quality and split the cluster there.

Now, what does our heuristic exactly do for specific clusters? The general idea is that we try find splits in a way that the orientation cones of both children are as focused as possible. Suppose we have a great number of small triangle emitters that shape up together as two spheres like modelled in Figure 3.7. Obviously, the normals of the triangle emitters of a single sphere point to all possible directions in the scene. If we would have a cluster containing every single triangle emitter of the two spheres, our θ_o value of that cluster would be π; the normals over the unit sphere of 2π or 360°. Making the bottom split which splits the two spheres would result in not much change: The axis of the emitters would still cover every possible direction and the θ_o values of both children would still be π. In this case, we would definitely prefer the top split that splits both spheres in half, resulting in the normals covering half of the unit sphere. We reduced θ_o to $\pi/2$. Note that the upper split may still emit in all directions even though the normals only cover only a hemisphere. SAH alone would have not told the difference between those two splits and that's why the bounding cone measure is a very integral part of our cost calculation algorithm.

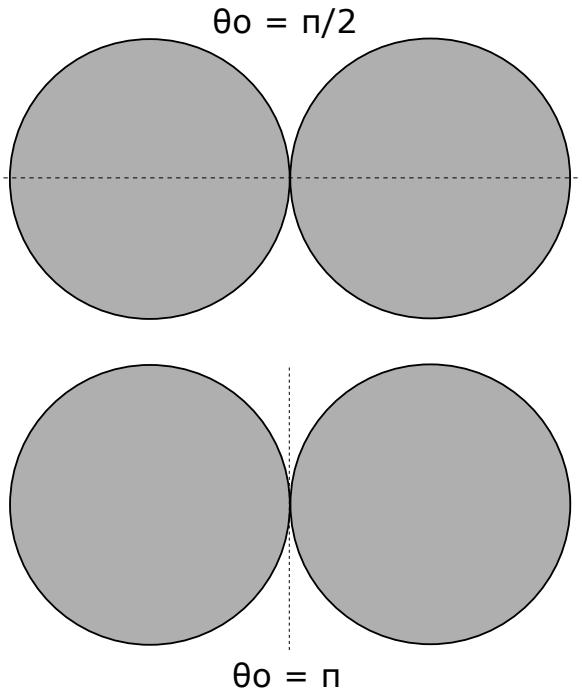


Figure 3.7: Preferred split example

3.3 Light Bounding Volume Hierarchy Construction

In this section we will be talking about our in-depth implementation of the tree construction. The tree constructor method takes two parameters. First, a vector containing all the light sources in the scene. Second, we will take a parameter defining the split threshold that describes if we want to sample multiple light sources when traversing the light BVH later (subsection 3.5.2). The split threshold is a normalized value between 0.0 and 1.0. A split threshold of 1.0 never splits, we would only shade one light per vertex and a split threshold of 0.0 always splits, which means all lights of the scene are shaded. Note that our algorithm is implemented to work with any combination of point light sources, area light sources and spotlights.

Algorithm 1 LightBVHAccelerator constructor

```
1: procedure LightBVHAccel(vector<Light> &lights, float splitThreshold)
2:     LightBVHNode * root ← recursiveBuild(lights, 0, lights.size());
3:     LinearLightBVHNode * nodes ← AllocAligned(totalNodes);
4:     flattenLightBVHTree(nodes, root, 0);
5: end procedure
```

The *lightBVHAccel* constructor initializes the tree construction. First, we will be using a recursive approach to build the light BVH. We will pass the light sources of the scene in an array and the starting and ending indices of the current node, obviously for the

root of the light BVH we will pass the whole range of our vector. Then, after we have constructed our tree represented by *LightBVHNode* objects, we will flatten the tree in a more compact form represented by *LinearLightBVHNode* objects to ensure that our tree requires as little memory as possible and to improve cache locality. The code for this function was mostly taken from the PBRT BVH flattening implementation and we only adapted it for our needs. *Nodes* will later be our object to run our tree traversal on.

Algorithm 2 LightBVHAccelerator recursive build

```
1:  procedure LightBVHNode* recursiveBuild(vector<Light> &lights, int start, int
    end)
2:      LightBVHNode *node;
3:      if end - start == 1 then
4:          *node = initLeaf();
5:          return node;
6:      end if
7:      for each dimension do
8:          <calculate axis and thetas for the whole node for current dimension>
9:          <calculate all split costs for current dimension>
10:     end for
11:     <find out best split>
12:     <initialize child nodes and make reference as children>
13:     return node;
14: end procedure
```

Our recursive implementation of the light BVH construction takes a vector including all lights of the scene, as well as start and end indices for the current node. The implementation can be split into multiple sections. Obviously, we want to cover the base case first, when we only have a single light source under the node. In this case, we cannot split further, so we will just initialize a leaf node with the one light source and return this leaf node. Next, we will be calculating the axis and θ values for the whole node for all three dimensions. This is required for the split cost calculations for every split in the current dimension we are working on. After we have computed the split qualities for every individual split in all dimensions, we will find out the best possible split among these we have calculated and initialize the left and right child, which we will reference as children of the current node.

The reader may have already realized that our implementation is very close to popular implementations of BVH constructions; that makes sense since our implementation was heavily influenced by BVH constructions, the structure of our tree is analogical to that of BVHs. It should be also noted that we count the number of nodes we create. This number is required for the flattening of our tree later (subsection 3.3.7).

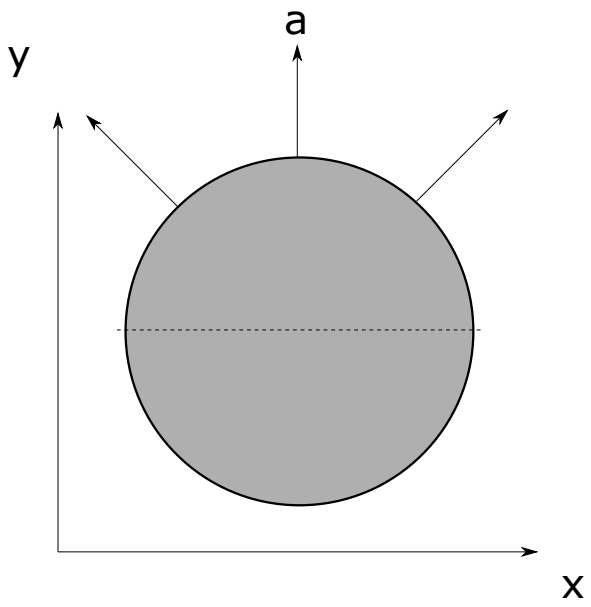


Figure 3.8: Axis calculation

3.3.1 Axis Calculation

Next, we will do axis calculations for the whole cluster for a specific dimension. We have tried two different approaches for the axis calculations. Our first approach used the median of the individual axes of all lights included in the cluster sorted by the specific dimension we are working on. So, if we wanted to find out the qualities of the individual splits, splitting the lights at a particular x-coordinate, we would take the axis of the median light if we sorted them according the x-coordinate of their axes. This approach leads to a major problem. Since our geometric representation is three-dimensional, taking the median axis sorted by a single dimension does not always represent the ideal axis that we are looking for. Suppose we have a scene as modelled in Figure 3.8 and suppose we have chose to split the sphere in half along the axis y. Like previously discussed in Figure 3.7, the axis of our hemisphere should be a or at least close to a. Calculating the cluster axis like just described would result to a very different and unexpected result. Instead of finding a as the axis, instead, we would find one of the other two axis plotted in the image. Those are the two median axis if you sort the axis of the individual emitters according to their y-value. It is evident, that this approach does not achieve the desired results.

Using the median axis sorted by another coordinate value works in this example, but does not yield the correct results in general either. Suppose we already have hemispheres like in Figure 3.9 and the split you can see in the drawing. In this situation, we would actually want to use the median axis sorted by the x-coordinate after having split along the axis x. Sorting by the y-values would lead to unwanted results. As you can see, only considering the value of one of the dimensions of the axes is not enough; instead we have to regard all the three dimensions. Our second approach that we are using now deals with this

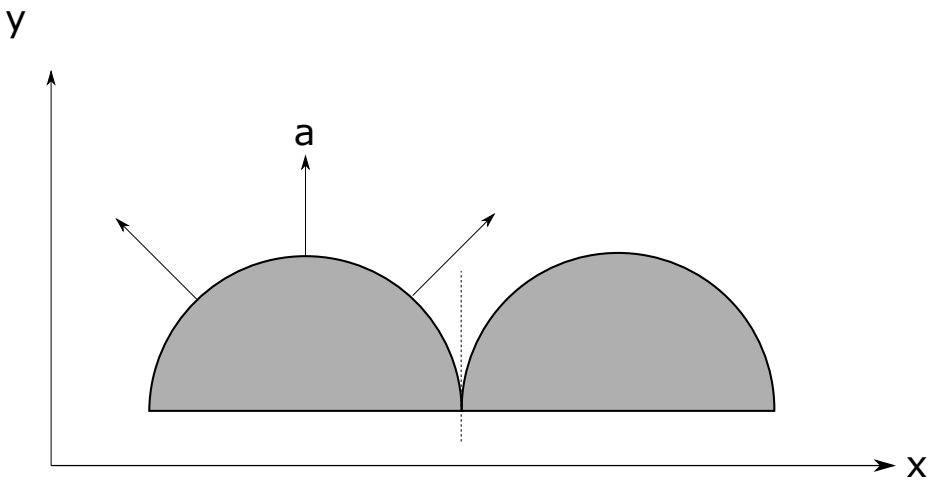


Figure 3.9: Axis calculation

problem properly. Instead of using the median axis, we use the average or the normalized sum of the individual axes of all light sources in the cluster:

$$C_{axis} = \frac{\sum_{i=0} a_i}{|\sum_{i=0} a_i|}, \tag{3.6}$$

with a_i being the axis of the i-th light source in the cluster. This axis definition for a cluster eliminates the problem presented in Figure 3.8. Both median and average calculations have $O(n)$ time complexity, but especially in scenes with symmetric objects that represent light sources, our current approach leads to much better results.

Obviously, this approach is not perfect either. If you could divide the axes in the cluster in two groups so for every emitter in one group there is another emitter in the other group with a complementary axis, the cluster axis we calculate with our method would result in a vector that is zero in all three dimensions. Clearly, we cannot work with this vector, so in these situations, we just choose an arbitrary light source axis as the cluster axis. Another problem is that we will not compute the "ideal" axis in general. With the way we calculate the importance of a cluster for a certain sampling point later, the ideal axis for our algorithm would be an axis that minimizes $\theta_e + \theta_o$. Is it not hard to think of an example where our calculated axis is not the ideal axis, but on average, our approach computes a decent axis to work with.

3.3.2 Theta Calculations

We have listed our calculations for θ_o and θ_e in Algorithm 3. The procedure takes the base address of a vector containing the lights that define the specific cluster, as well as

Algorithm 3 Theta calculations

```
1:  procedure CALCULATETHETAS(vector<Light> $lights, 3DVector axis, float *theta_o,
    float *theta_e)
2:      for int i = 0; i < vector.size(); i++ do
3:          Light l ← lights[i];
4:          *theta_o ← max(*theta_o, radianAngle(axis, l.axis) + l.theta_o);
5:          *theta_e ← max(*theta_e, l.theta_o + l.theta_e);
6:      end for
7:      *theta_e -= *theta_o;
8:      *theta_e ← clamp(*theta_e, 0, Pi - *theta_o);
9:      *theta_o ← min(*theta_o, Pi);
10: end procedure
```

the axis of the cluster. Also, pointer to two floats are passed to return the θ-values. The calculations made are very simple; we iterate through the lights that were passed to us and calculate the individual θ_o and θ_e values of the given axis with the axis of the whole cluster. If any of the θ values are greater than the current maximum θ values we have stored, we update our maximum θ values. At the end, when we have processed all light sources, we have to first subtract θ_o from θ_e as we only store the additional angle to θ_o in θ_e. Next, we will also need to clamp θ_e with 0 and $\pi - \theta_o$, since a negative angle does not make sense and otherwise it is possible to get a negative integral in our bounding cone measure Equation 3.3. This is something we want to avoid because the measured volume of the bounding cone should always be non-negative. θ_o will also be limited by π for the same reasons.

3.3.3 Buckets

Before we jump into our split cost implementation, we want to talk about a design decision we made. The first approach we tried was to compute costs for splits after every single light source, which means, if we had 1.000 lights, we would have 999 different splits in each dimension. We realized that this approach does not scale well for very large amounts of light sources. We measured the time for building the light BVH on different scenes with varying amounts of emitters on a computer with a Intel® Core™ i7-4770K processor and 16 gigabyte of main memory. For instance, a scene with 10.000 point light sources works just fine with a building time of 21 seconds. On the other hand, the tree construction time of another scene with one million light sources takes up to 3 hours. Obviously, this construction time is way too long and not useful for practical uses.

Therefore, we decided to use specific amounts of buckets that determine the position where we compute the costs of splits. At the start, we will define a maximum amount of splits that we want to calculate for any cluster. What has worked well for us is to calculate a maximum number of 1.000 splits. This way, we have an acceptable building time of 43 seconds for the scene with one million light sources, while the split quality stays roughly

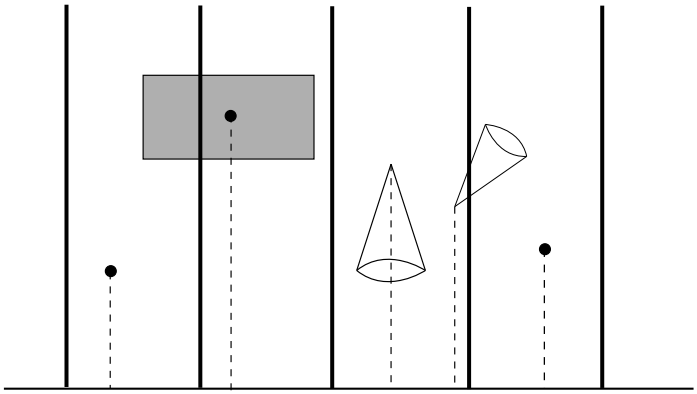


Figure 3.10: Bucket splits for area light sources, point light sources and spotlights

the same as if we would calculate the costs for every possible split. When we have settled on b, the amount of buckets to use, we will define the number of light sources in each bucket as:

$$n = \frac{b}{N}, \tag{3.7}$$

with N being the total number of emitters of the cluster. Then, we will split the lights of the cluster at the indices that are multiples of n, resulting to b different splits.

Another approach to split the scene in multiple buckets is to split the world space coordinates for a specific dimension into equally big areas and projecting the centroid value of the light sources on the coordinate plane of that dimension. The general idea can be seen in Figure 3.10. So, instead of computing the costs for splits after a certain amount of emitters, we compute the costs for splits after a definite chunk of world space. The difference between these two ideas was not huge, but we decided to stick to computing the costs of splits after a fixed number of emitters because especially in scenes, where the emitters are focused in a small area, this approach seemed to make more sense.

3.3.4 Split Cost Calculation

First, we have to sort the light sources by their coordinate value of the current dimension we are working on, since we split in world space. Then, we use the previously discussed buckets to determine positions where to split the emitters. For each individual split, we will be doing axis, theta and AABB as described in sections 3.3.1, 3.3.2 and 2.5. The total energy of a cluster is simply the sum of the energy values of the individual light sources. Afterwards, we will do the cost calculations for the left part, the right part and the total cluster according to section 3.2. The split cost of this bucket is then stored in the respective entry of cost. After we have iterated through all the buckets for this particular dimension, we will return the array containing all split values of this dimension.

Algorithm 4 Split Cost Calculation

```
1: procedure INT[] CALCULATECOST(vector<Light> $lights, int buckets)
2:     float[buckets] cost;
3:     for each split i do
4:         <calculate axis, thetas, AABB and energy for the left part>
5:         <calculate axis, thetas, AABB and energy for the right part>
6:         <calculate leftCost, rightCost and totalCost>
7:         cost[i] ← (leftCost + rightCost) / totalCost;
8:     end for
9:     return cost;
10: end procedure
```

3.3.5 Choosing the best split

There is not much to talk in this subsection. We iterate through the arrays containing the cost values of each individual split and choose the split with the lowest cost value, i.e. the split that has the highest quality. Depending on the implementation and if we use buckets, we have to calculate the index where we split our cluster and perhaps the dimension of the split.

3.3.6 Creating the children nodes

Algorithm 5 Children creation

```
1: procedure LIGHTBVHNODE* CREATECHILDREN(vector<Light> lights, int splitIndex,
   int splitDimension)
2:     partialSort($lights, splitIndex, splitDimension);
3:     LightBVHNode *leftNode ← recursiveBuild(lights, 0, splitIndex + 1);
4:     LightBVHNode *rightNode ← recursiveBuild(lights, splitIndex + 1, lights.size());
5:     node ← initInterior(leftNode, rightNode);
6:     return node;
7: end procedure
```

At this point, we have found out the dimension of our optimal split, as well as the index where to split our cluster. Our first step is to make a partial sort at the split index with individual lights of the cluster. We have to sort according to the centroids world space coordinates of the given dimension. That makes sense, since our primary goal was the split the cluster in world space and use the orientation solely for split cost calculations while constructing the tree. Afterwards, we recursively call *recursiveBuild* with the indices for the left and the right child and make references of the children in our parent node.

3.3.7 Tree Flattening

Algorithm 6 Tree flattening

```
1:  procedure INT FLATTENLIGHTBVHTREE(LinearLightBVHNode *nodes, LightBVHN-
    ode *node, int *offset)
2:      LinearLightBVHNode *linearNode ← &nodes[*offset];
3:      linearNode.copy(node);
4:      int myOffset ← (*offset)++;
5:      if node->nLights == 1 then
6:          linearNode->lightNum ← node->lightNum;
7:      else
8:          linearNode->splitAxis ← node->splitAxis;
9:          flattenLightBVHTree(nodes, node->children[0], offset);
10:         linearNode->secondOffset ←
11:             flattenLightBVHTree(nodes, node->children[1], offset);
12:     end if
13:     return myOffset;
14: end procedure
```

We have a representation of all light sources of the scene in a binary tree now and at this point, we say that we are finished with our tree construction and use this tree as the acceleration data structure to do the sampling on. But when traversing the tree, we have realized that the nodes were not ordered in a linear way in our memory. That leads to more cache misses, worse memory usage and just worse performance overall, which is why decided to flatten our representation to change this aspect of our light BVH representation. The implementation was mostly taken from PBRT's implementation of BVH flattening and we only adjusted it for our needs. How we modelled a single node in our code can be seen in Figure 3.5. The general idea is that the left child node is implicitly referenced because it is stored as the next child in the memory. Therefore, we only need to store the offset to the right child of each interior node explicitly. The pseudo code of our implementation can be seen in Figure 6. Remember that we call this method on the root of our light BVH tree before flattening with the base address of our pre-allocated memory passed in $*nodes$ (Algorithm 1) The offset passed is obviously 0 at the beginning.

The function returns the offset for the current node we are working on. First, we will creating a $LinearLightBVHNode$ at the address given by $\&nodes + offset$ and will be copying the data from $*node$ that are used for all nodes, regardless if the node is an interior node or a leaf node. That includes:

- World space bounds
- Orientation bound
- Energy of the node
- Centroid of the node

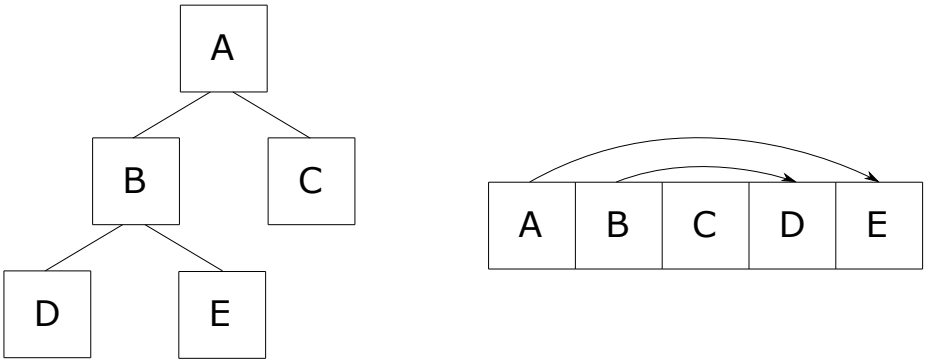


Figure 3.11: Tree representation before and after flattening

- Number of emitters under this node

After incrementing the offset, so $\&nodes + offset$ points to the address where the next *LinearLightBVHNode* should start, we will be dealing with two cases: Either the currently regarded node is a leaf node or an interior node. If the node is a leaf node, all we need to do is set the light number are finished. If the node is an interior node, the splitAxis need to be set and we will have to call the method recursively for the left and the right child, while storing the offset returned by the second child as the $secondChildOffset$. That will be our explicit reference to the second child when traversing the tree later.

The effect of calling this method on the root of the light BVH tree can be seen in Figure 3.11. Suppose we have a very simple tree with only five total nodes consisting of two interior nodes (A and B) and three leaf nodes (C, D and E) like in the drawing. In our pre-flattened version of our light BVH, the nodes do not necessarily need to be in a contiguous chunk of memory. Instead, every interior node would have references to two child nodes. After the flattening, the nodes are ordered in a linear way in a coherent chunk of memory. The left child is implicitly referenced by the next node in the memory and the right child can be referenced via an offset. In our example, the left child of A is implicitly referenced by the next node in memory, as well as the left child of B. The right child can be found with the sum of the offset and the base address. Obviously, with the non-linear implementation, our cache, memory and overall system performance could be extremely lacking in worst cases and will definitely be worse in average cases than our flattened version.

3.4 Importance from a vertex to be shaded

3.4.1 General idea

In this section we will define an importance measure for a given node and vertex to be shaded. We will need this importance measure when traversing the tree later, to be able to make decisions whether to sample the left or right child.

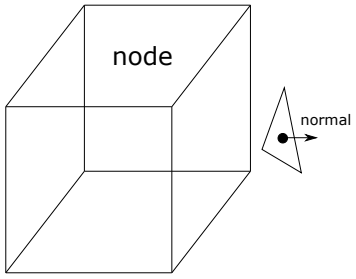


Figure 3.12: Our node lies behind the vertex to be shaded

Before we talk about our defined importance measure, we should point our that an emitter can only have a direct contribution to the incident lighting of the vertex to be shaded if there are no opaque objects blocking the view from the light source position and the vertex. Obviously, we cannot know if any arbitrary object is blocking this view without doing ray-intersection tests which would drastically worsen the runtime of our algorithm. But there is one implicit intersection test can be done very easily: whether the object of the vertex is blocking the view itself. Suppose we have a situation where the normal of the primitive at the vertex points away from the node that we are regarding right now like in Figure 3.12. If we know that the primitive is opaque, and therefore there will be no light transmissions, the contribution of any emitter in the node will be zero for the vertex because the geometric representation of the primitive is blocking the incident lighting of the emitter itself. The check for this case is very trivial. We simply test if all of the 8 corners of the nodes bounding box are behind the vertex. In this situation, the importance value of this node for the given vertex is zero. This visibility test was not used in Conty and Kulla's implementation explained in [CK17a; CK17b].

We want to traverse the light BVH making random decisions at every branch. These random decisions come from a random uniform number we drew earlier in our implementation, but we will also need an importance measure to complete our importance sampling. Our importance measure for a given node and vertex to be shaded has to take multiple factors into consideration. The three factors are:

- Contained energy of the node
- Inverse square distance from node centroid
- Cosine factor to the orientation bounds I_θ

The first two factors are pretty self-explanatory. In a situation where two nodes are represented by the same world space bounds and orientation bounds, the contained energy of the node is proportional to the contribution of the nodes. The same logic can be used for our second factor. The further away a node is from our sampling point, the lower will its contribution be. Note, that we will have to use the centroid of the node as the replacement of the position of the sampled light source since we do not yet know the actual emitter to sample now.

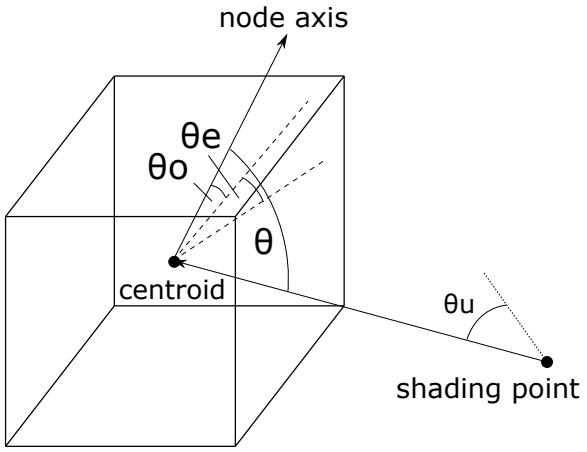


Figure 3.13: Importance of a node for a given vertex to be shaded

Our cosine factor to the orientation bounds I_θ requires us to look at our node representation more closely. We will define I_θ as follows:

$$I_\theta = \begin{cases} \cos(\theta - \theta_o - \theta_u) & \theta - \theta_o - \theta_u \leq \theta_e \\ 0 & else \end{cases}, \tag{3.8}$$

with θ being the angle between the axis of the node and a vector starting at the centroid of the node and the vertex and θ_u being the uncertainty angle that we will define in the next paragraph. An illustration of the individual angles can be seen in Figure 3.13. This is a point, where our algorithm vastly differs from the algorithm presented by Conty and Kulla [CK17a; CK17b]. They defined I_θ as:

$$I_\theta = cos(clamp(\theta - \theta_o - \theta_u, 0, \theta_e)). \tag{3.9}$$

We chose our version because elsewise a smaller emission range θ_e could lead to a bigger importance value I_θ. We rendered different scenes with our and their algorithm and the results can be seen in Figure **??**. Our version of the algorithm yields better results for the scenes we have tested it on.

First, notice that a single node can cover a big area in world space, which means that the actual position of the emitter to be sampled later and the centroid of the node can be very far away dependent on the size of the node. This is why we need a so-called "uncertainty factor" that considers this fact; otherwise it might be possible that we totally disregard nodes where emitters have a high contribution but where the emitters are very distant from the centroid of the node. But this uncertainty factor cannot be solely based on the size of the node, since this uncertainty factor will have a bigger input in nodes that are close to the sampling point. We decided to use an uncertainty angle θ_u. The idea behind θ_u is that we try to find the biggest angle between a vector starting from the vertex and

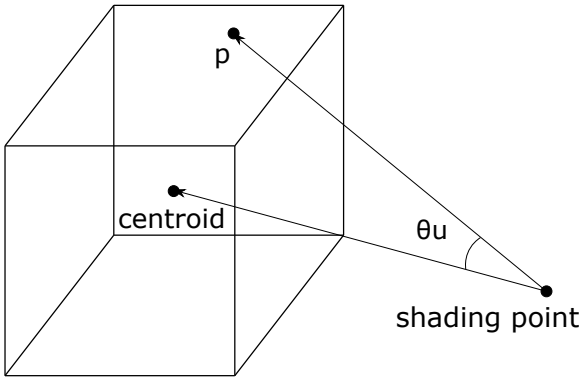


Figure 3.14: General idea of the uncertainty angle

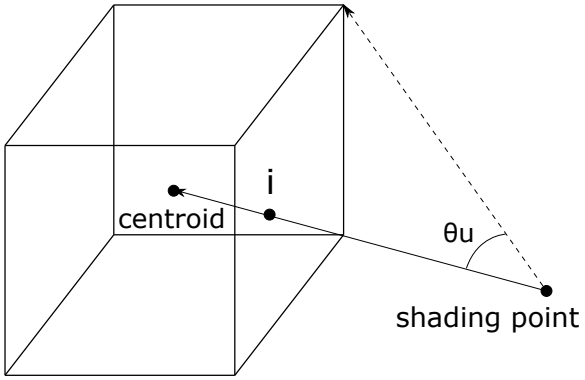


Figure 3.15: Uncertainty angle calculation by calculating the angle between a vector starting at the vertex to be shaded and pointing to the centroid and the 4 corners on the intersected side of the cuboid

pointing to the centroid and a vector starting from the vertex and pointing to any point p that lies inside of the node. A drawing of the uncertainty angle can be seen in Figure 3.14. Kulla and Conty have used a similar implementation, as we can see in [CK18] but at the time of our development, the referenced paper was not released yet.

We could use the positions of the emitters to determine the uncertainty angle, since that would return exactly what we want: The maximum angle difference from the centroid and a light source in the node. But especially in nodes with very large amounts of emitters, so in nodes that are close to the root of our tree representation, that seems to be not very practical. Instead, we can estimate the uncertainty angle by defining the corners of our nodes world space bounds as the ending points of our vector. While this method does not yield perfect results, it is much faster than calculating the angle of potentially thousands of lights in every step. At this point, we have limited the giant number of angle calculations to only 8, one for each corner of the node. In practice, we can even further limit these angle calculations: The corner that yields us the biggest uncertainty angle will always be on the side of the cuboid that was intersected when we drew a vector from the

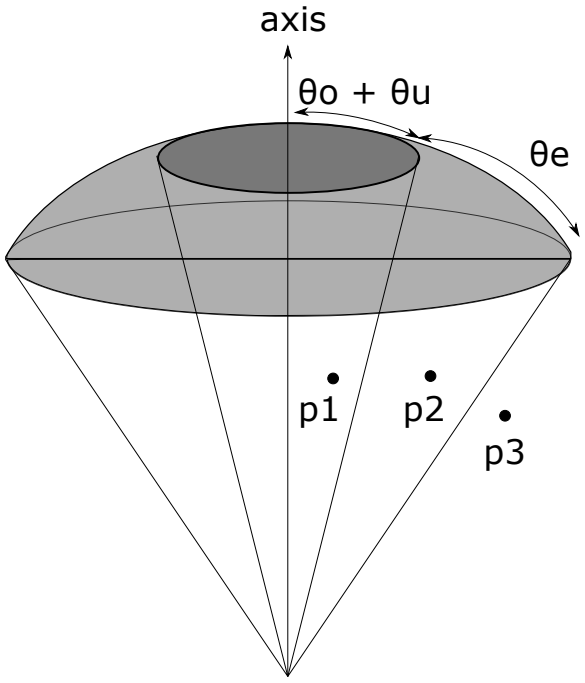


Figure 3.16: Different importance values for vertices

shading point to the centroid. That means, we can reduce the angle calculations to only 4, one for each corner on the intersected side of the cuboid. A graphic illustration can be seen in Figure 3.15. In some situations, the shading point will lie inside of the world space bounds of the node. In these cases, we will just set the uncertainty value as π, which will result to I_θ becoming 1.

As we can see in Equation 3.8, θ_u is only one of our factors for our cosine factor to the orientation bounds. Our primary goal was it to give nodes a higher importance that have light sources directly emitting to the vertex. That means, we have to regard the axis, θ_o and θ_e of the node. In total, we prefer nodes with θ_o cones that comprise the vertex after we have defined a certain tolerance with our uncertainty angle. We will also set I_θ to a non-zero value if the vertex is included in the emission range θ_e of the orientation bounds of our node. In a case where $\theta - \theta_o - \theta_u > \theta_e$, it would mean that even after having a tolerance value defined by θ_u, the vertex is not encircled in the orientation bounds of the node and thus the contribution of any light source in the node will be zero for the vertex. Suppose we have three different shading vertices and the orientation bounds of a node as illustrated in Rigure 3.16. The vertex $p1$ lies in the cone comprised by $\theta_o + \theta_u$ (cone that includes axes of all emitters in node plus tolerance) and thus its I_θ value will be 1. The world space coordinates of $p2$ are placed in the cone comprised by the emission range θ_e of the node and therefore its I_θ value will be between 0 and 1. Lastly, our vertex $p3$ is outside of the orientation bounds of the node even with the tolerance added to it, resulting to a I_θ value of 0.

In summary, the importance value for a given node and vertex is defined as:

$$I = \frac{E * I_\theta}{d^2}, \tag{3.10}$$

where E is the energy of the node, d is the distance between the node and the vertex and I_θ as in Equation 3.8.

3.4.2 Implementation details

Algorithm 7 Calculating the importance of a vertex to be shaded with a given node

```
1:  procedure FLOAT CALCULATEIMPORTANCE(Intersection &it, LinearLightBVHNode* node)
2:      <check if the bounding box of node is behind the vertex>
3:      <calculate the first side of the node we hit if we draw a vector from the shading point to the centroid of the node>
4:      <calculate the uncertainty angle>
5:      if θ − θ_o − θ_u − θ_e > 0 then
6:          I_theta ← cos(max(0, θ − θ_o − θ_u));
7:      else
8:          return 0;
9:      end if
10:     return node->energy * I_theta / (distance * distance);
11: end procedure
```

Our implementation of the importance calculation for a given node and vertex to be shaded (Algorithm 7) basically follows the steps we have mentioned above. First, we will be checking if the node lies behind the vertex. If that's the case, we will simply return 0 for the current node. Next, we will be doing calculations for our uncertainty angle starting by intersecting the vertex to centroid vector with the bounding box of the node. After we have found the intersection, we will define the uncertainty angle θ_u as the maximum angle between the vertex to centroid vector and the vertex to corner vector. At the end, we will do the final I_θ and importance calculations that we have defined in Figures 3.8 and 3.10.

3.4.2.1 Checking if the node is behind the vertex to be shaded

We want to avoid boring the reader with implementation details if possible but there are some things to note while checking if the current node is behind the vertex be shaded. The general idea is to calculate the angle the normal of the intersection n with the vectors starting at the vertex and pointing to the corners of the bounding box of the nodes. The cosine of these angles become negative for an angle bigger than 180°, basically when the individual corners are behind the vertex (compare Figure 3.17). Note that it is not required

Algorithm 8 Checking if the node is behind the vertex to be shaded

```
1:  bool behind ← false;
2:  if it.isOpaque() then
3:      for int i ← 0; i < 8; i++ do
4:          if dot(n, node->corner[i] - o) > 0 then
5:              behind ← (false);
6:              break;
7:          end if
8:      end for
9:      if behind then
10:         return 0;
11:     end if
12: end if
```

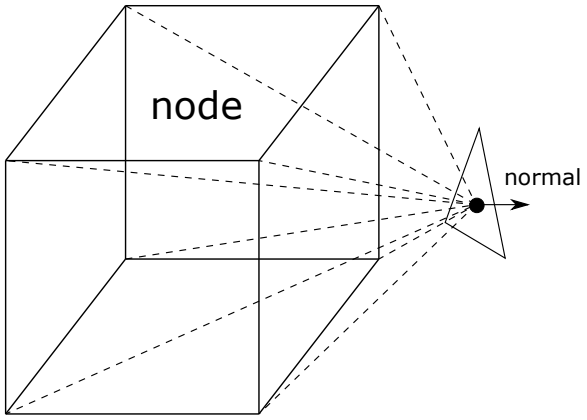


Figure 3.17: Checking if the node lies behind the vertex to be shaded

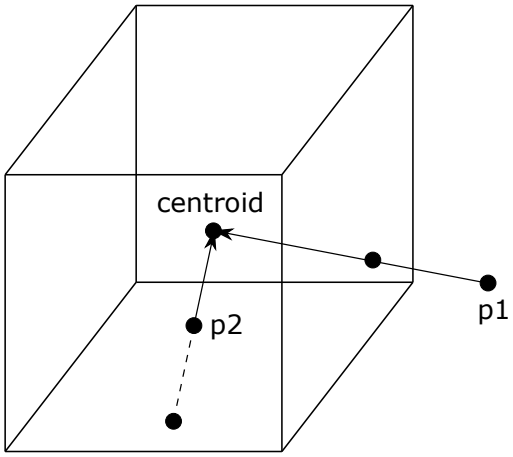


Figure 3.18: Two different cases of ray-bounding box intersection

to normalize the vectors pointing to the corners since the signing does not change even for non-normalized vectors. If all of the corners are behind the vertex, we return an importance value of 0, otherwise we move on with our algorithm.

3.4.2.2 Calculating the uncertainty angle

Algorithm 9 Ray-bounding box intersection test

```
1: Vector3D d = normalize(node->centroid - p);
2: float t ← −infinity;
3: for int i ← 0; i < 3; i++ do
4:     float invRayDir ← d[i];
5:     float tNear ← min((node->pMin[i] - p[i] * invRayDir), (node->pMax[i] - p[i] *
   invRayDir));
6:     t ← max(t0, tNear);
7: end for
```

We have mentioned earlier, that our first step for our uncertainty angle calculation will be the intersecting the vector starting from the vertex and pointing to the centroid with the world space bounding box of the node. This test is similar to the ray-bounding box intersection tests as the one that has been implemented in PBRT or has been described in [Bar11; Bar15]. As you can see in Algorithm 9, we use a very simplified version of the ray-bounding box intersection test since we only need the first intersection. Now, we have the distance t to our first intersection point and there are two possible cases. Either, t has a positive value, which means that the intersection point lies between the vertex and the centroid of the node or t has a negative value meaning that the intersection point is behind the vertex. In this case, the vertex lies inside of the bounding box and we will set I_θ to 1. An illustration of the two different cases can be seen in Figure 3.18. $p1$ has lies

outside of the box and thus has a positive t value, while $p2$ is inside the box and returns a positive t value.

Now, we can calculate the actual intersection point is:

$$is = p + d \cdot t, \tag{3.11}$$

and find out the 4 corners that belong to this side of the bounding box. For these 4 corners, we calculate the maximum angle as described in Figure 3.15. We will not show our implementation here because most of it just consists of a trivial switch-case statement.

3.5 Light Bounding Volume Hierarchy Traversal

We have successfully constructed an acceleration data structure for our light sampling problem. The next step will be obviously doing the tree traversal. If we think back to section 2.7, in our light transport equation we have different functions in the integrand.

$$L_o(p, \omega_o, \omega_i) = \int_{S^2} f(p, \omega_o, \omega_i) L_d(p, \omega_i) |cos\theta_i| d\omega_i \tag{3.12}$$

Both function can be sampled with Monte Carlo integration and while $f(p, \omega_o, \omega_i)$ depends on the BSDF of the material of the vertex to be shaded, $L_d(p, \omega_i)$ can be importance sampled when we choose an emitter that has a big contribution to the vertex. Our goal, in summary, is to return light sources that have a high contribution to the vertex more often. In most cases that would be done by creating a distribution that has a similar form to that of $L_d(p, \omega_i)$, but in our situation, that is not practicable. Calculating a distribution for every single new vertex would not make a lot of sense since these distribution cannot be reused. In special scenes, we might sample light sources for a single point multiple times, but in general that will not be the case. Instead of generating distributions, we will make a decision at every branch of our tree to go left or right based on a random number.

Algorithm 10 Sampling a single light source

```
1: procedure int sampleOneLight(Intersection &it, float *pdf)
2:     float sample1D ← UniformFloat();
3:     *pdf ← 1;
4:     return TraverseNodeForOneLight(nodes, sample1D, it, pdf);
5: end procedure
```

Our implementation offers two functions to call to sample a light source when passing an intersection point. The two functions are *SampleOneLight* when not splitting and *SampleMultipleLights* when splitting. The main reason for two different functions for

Algorithm 11 Sampling multiple light source

```
1: procedure VECTOR<PAIR<INT, FLOAT» SAMPLEMULTIPLELIGHTS(Intersection &it)
2:     vector<pair<int, float» lightVector;
3:     TraverseNodeForMultipleLights(nodes, it, &lightVector);
4:     return lightVector;
5: end procedure
```

a functionality that could be provided by a single function is that we want to avoid unnecessary work if possible. When sampling multiple lights, it is required to return a data structure that stores a collection of references to the emitters. That is not mandatory for sampling a single light source. Since this method is called every time we want to sample a light source, the additional overhead can add up to a substantial amount of extra rendering time. Our two implementations of the sampling functions can be seen in Algorithms 11 and 12. There are a few things to point out in our implementation. First, note that *nodes* is a pointer to our root of our flattened tree and is a private member of our tree representation. Second, the *Intersection* to be passed stores information about the vertex, including its world space position, as well as its normal and BSDF representation. That is required for splitting and optimization later. Third, additionally to the number of the light source, we return a value of the probability density function (PDF) in both sampling functions. As we have explained in earlier sections, this is an integral part of importance sampling that cannot be skipped. When sampling a single light source, we generate a uniform random number between 0 and 1 and make branch decisions based on that number. This step is postponed when sampling multiple light sources for two different reasons. It is not required at this point and we want to draw a uniform random number for each light source to sample instead of using a single uniform random number. Instead, we will initialize a vector that will later be filled with the index numbers of the sampled light sources and their respective PDFs. Lastly, we have to mention an important detail about the return of these functions. In some cases, when traversing through the light BVH, it will be already evident, that not a single emitter in the branch we are current traversing though will have a contribution to the intersection point. In these situations, we will return -1 to signal that we did not sample a light.

3.5.1 Sampling a single light source

We will be talking about how we traverse through the light BVH when sampling a single light source in this subsection. First, we cover the base case when we already arrived at a leaf node of our tree representation. In this case, we have reached a node that contains only a single light source which represents the sampled light source. We simply return the index of the emitter stored in the current node, as well as the PDF that has been calculated while traversing through the tree.

If we are currently in an interior node, we will have to calculate the importance values for both children. The left child is simply the next node in memory, while we will reference

Algorithm 12 Sampling a single light source

```
1: procedure int TraverseNodeForOneLight(LinearLightBVHNode *node, float
   sample1D Intersection &it, float *pdf)
2:     if node->nLights == 1 then
3:         return node->lightNum;
4:     end if
5:     float left ← calculateImportance(it, &node[1]);
6:     float right ← calculateImportance(it, nodes[node->secondChildOffset]);
7:     float total ← left + right;
8:     if total == 0 then
9:         return -1;
10:    end if
11:    left /= total;
12:    right /= total;
13:    <branch according to importance and sample1D>
14: end procedure
```

the right child explicitly with the $secondChildOffset$ of the current node. Then, we will normalize these importance values and check if their sum is zero. In this case, the current node we are working on, will have no contribution to the vertex and we will return -1. If the total value is non-zero, we will make a decision to go left or right based on the importance values and the uniform random number we drew earlier.

Algorithm 13 Choosing left or right child traversal

```
1: if sample1D < left then
2:     *pdf *= first;
3:     return TraverseNodeForOneLight(&node[1], sample1D / first, it, *pdf);
4: else
5:     *pdf *= second;
6:     return TraverseNodeForOneLight(&nodes[node->secondChildOffset], (sam-
   ple1D - first) / second, it, *pdf);
7: end if
```

A few things are to note when choosing the left or right child node for the next node to traverse through. First, since we do not have an explicit distribution function for importance sampling, we do not have a probability density function either. A PDF value is required to get a reasonable result when using importance sampling. In our case, we will calculate the PDF value of the sampled light by cumulatively multiplicating with the importance values of the nodes on the path to the final sampled light. Just like the PDF of a distribution, this technique yields us the possibility of sampling a certain emitter. An example of our PDF calculation can be seen in Figure 3.19.

The next thing to talk about is the way we stretch the random number $sample1D$ in the course of the traversal. Remember that $sample1D$ is a uniform random number between

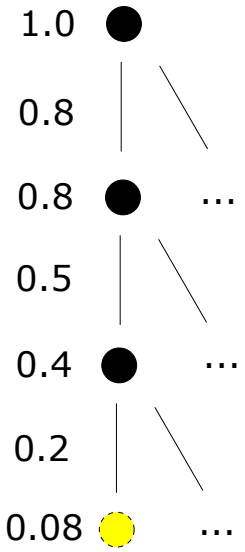


Figure 3.19: PDF calculation. Values next to nodes are the current PDF values and values next to vertices are the importance values.

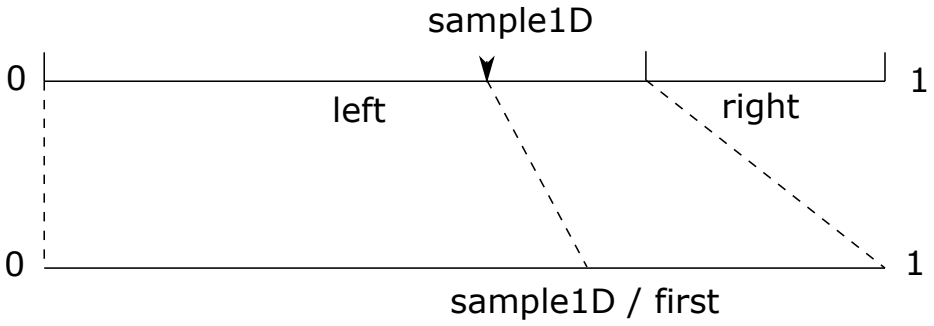


Figure 3.20: Sample1D stretching

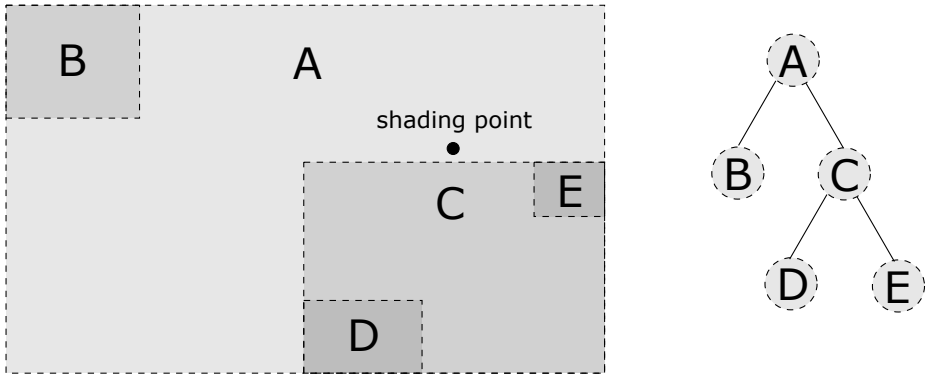


Figure 3.21: Splitting example

0 and 1. When traversing, we want to keep stretching sample1D by projecting the importance of the child node that we take to [0; 1]. An illustration of that can be seen in Figure 3.20 for the left child traversal and is analogous for the right child traversal. In that way, it is possible for us to keep *sample1D* pointing at the same node of the tree at all times until we have arrived at our leaf node with a single emitter and return the index of that light source.

3.5.2 Sampling multiple light sources

A traversal strategy that we have mentioned earlier was to sample multiple light sources by splitting. In some situations, the cluster we are regarding right now might be too close to the vertex or too big. In these situations, it might be desirable to sample light sources from both children of the node instead of just to sample a single light source.

Suppose we have a vertex and a tree as illustrated in Figure 3.21. First, we will choose the root node A for splitting because it is so big that it contains the vertex. B is not selected for splitting afterwards, which means we just sample a single light source from B. We will then again sample multiple light sources from C, because it is too close to the vertex. A single emitter is then returned by D and E respectively, which results to 3 sampled light sources instead of just 1, if we didn't split during the traversal.

There are multiple things to note when splitting. First, once we have chosen not to split for a certain node, we will only sample a single light source for that node and thus, we will not split one of the child nodes of that parent node. Second, as you can see in our introductory example, splitting is applied recursively and when we split the first time, that means we will at least sample two different light sources. Due to it's recursive nature, if we chose to split at every interior node, it would also be possible to sample every light stored in our tree.

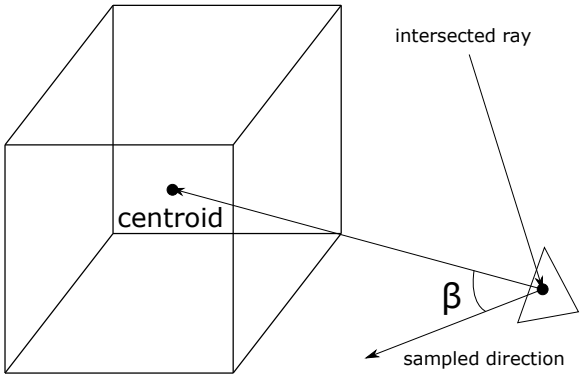


Figure 3.22: Sampling a direction for the BSDF peak

3.5.2.1 Split Heuristic

In this subsection we will define the split heuristic that returns for a certain node if we split at that node or not. We acknowledged earlier, that we will split if the current cluster is either too close to the vertex or too big. An easy check if the cluster is too close would be obviously checking if the bounding box of the cluster comprises the shading vertex. In these situations, we will always split. We decided to use an approximation of the solid angle of the cluster's bounding box as one of the factors for our splitting heuristic. This basically is a combination of how close and big the cluster is relative to the vertex. Another factor we regarded was the BSDF peak. It is computed as follows. First, we sample a random direction from the BSDF and calculate the cosine with the vector from the vertex to the cluster's center (Figure 3.22). Then, we will evaluate the PDF value of the sampled direction for the BSDF of the vertex.

In total, the split value of a cluster with solid angle α and a randomly sampled direction with conservative maximum cosine with the vector to the cluster's center β and the PDF value c is:

$$I_{split} = normalize(\alpha \cdot \beta \cdot c), \tag{3.13}$$

when normalize is a function that normalizes the value to between 0.0 and 1.0. A split value of 0.0 means that we split at every node and a value of 1.0 means that we will never split, only one light will be shaded per vertex.

3.5.2.2 Implementation details

Algorithm 14 Sampling multiple lights

```
1: procedure TraverseNodeForMultipleLights(LinearLightBVHNode *node, Intersection &it, vector<pair<int,float» *lightVector)
2:     if node->nLights == 1 then
3:         lightVector->push(pair(node->lightNum, 1.f));
4:         return ;
5:     end if
6:     bool split ← false;
7:     if vertex is in node then
8:         split ← true;
9:     else
10:        <approximate solid angle>
11:        <sample random direction and calculate BSDF peak>
12:        <calculate normalized split value>
13:        if splitValue > splitThreshold then
14:            split ← true;
15:        end if
16:    end if
17:    if split then
18:        <recursively traverse left and right child>
19:    else
20:        <call TraverseNodeForOneLight for current node and store return in vector>
21:    end if
22: end procedure
```

Pseudo code of our implementation of *TraverseNodeForMultipleLights* can be seen in Algorithm 14. The procedure is pretty self-explanatory. First, we cover the base case when we are already in a leaf node by simply pushing the index of the emitter in our vector. Note that in this case, the PDF of the light is 1.0 because we haven't made a branch decision at any node in the path to the leaf node. Then, if the bounding box of the node contains the vertex, we will split, otherwise we will calculate the split value and compare it to the split threshold. Note that when we decide not to split, we will call *TraverseNodeForOneLight* for the current node to make a decision to go left or right.

Since most of our code for splitting is pretty trivial, we decided to exclude most of it. The only section we wanted to talk about is the section where we approximate the solid angle of the node. Van Oosterom and Strackee [VS83] showed us analytical ways to calculate the exact solid angle of triangles. By defining the sides of the bounding box as triangles, we could use this approach to compute the precise solid angle of a given node. We decided that these calculations take too much time and the difference between an approximation and an exact calculation would not make a worthwhile difference. Instead, we approximated the solid angle by the maximum pairwise angle of two vectors starting at the vertex

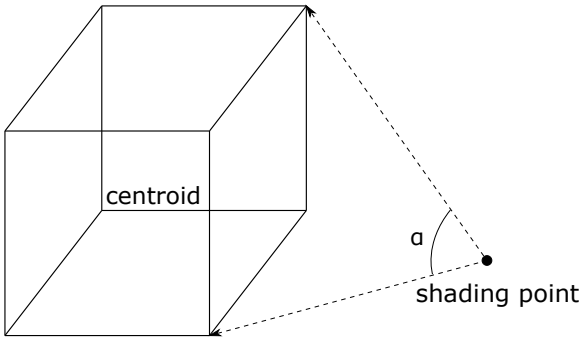


Figure 3.23: Solid angle approximation

and pointing to a corner of the node. Obviously, this is not an accurate representation of the solid angle but it is a good approximation (Figure 3.23).

4 Evaluation

In this chapter we will be comparing our algorithm without splitting with the typical light sampling strategy power sampling that creates a light distribution according to the energy of the emitters on different scenes. We are including scenes with point light sources, spotlights and area light sources and also combinations of these light sources. All scenes were rendered on a computer with a Intel® Core™ i7-4770K processor and 16 gigabyte of main memory. The framework used was PBRT version 3 by Pharr, Jakob and Humphreys. [PJH18]

4.1 20 spot lights

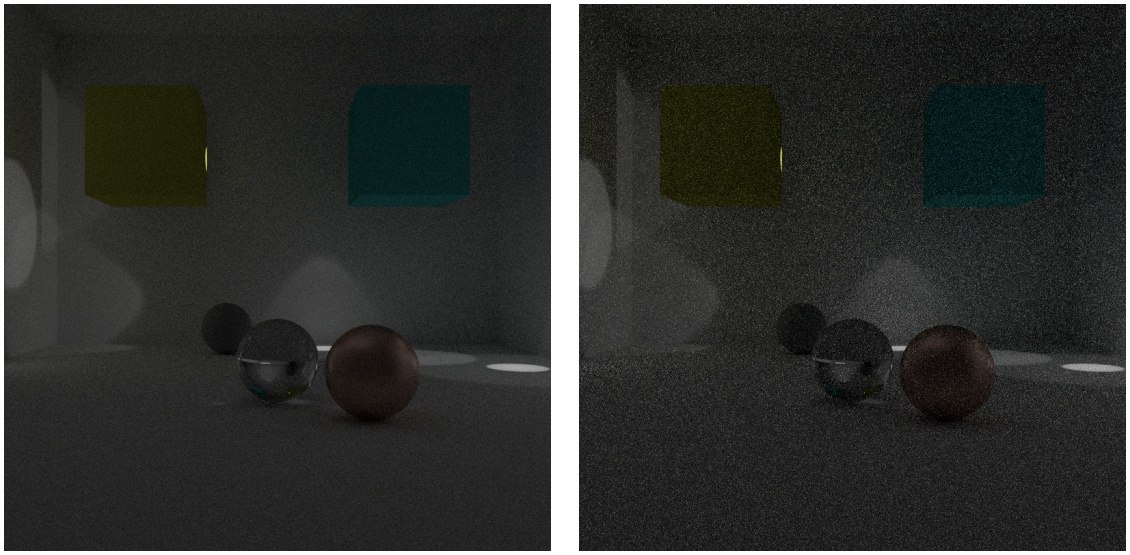
First, we wanted to show a scene with very few light sources (Figure 4.1). This scene is only lighted by 20 spotlights that we randomly placed in the scene. Both were rendered in about 150 seconds. As we can see, even with very small numbers of emitters, our algorithm leads to a much better image quality.

4.2 1.6 million point light sources on the ceiling

This scene (Figure 4.2) uses about 1.6 million point light sources that are offset by a bit at the ceiling of the room and covering the complete surface of the ceiling. The point lights are ordered in a 2D-raster with identical distance. The rendering time of this scene was 230 seconds, while the same scene with a single area light source that covered the whole ceiling is about 3 times faster with a rendering time of 73 seconds. That proves that our algorithm does indeed scale very well with many emitters.

4.3 12.000 point light sources in a 3D-raster covering the whole room

This scene (Figure 4.3) is lighted by 12.000 point light sources that are organized in a 3D-raster. Both images Figure 4.3a and Figure 4.3b are rendered with similar rendering times of about 170 seconds. The raster covers the whole room with only a bit of space left on



(a) Our algorithm, 64 samples (b) Power sampling, 100 samples

Figure 4.1: 20 spotlights

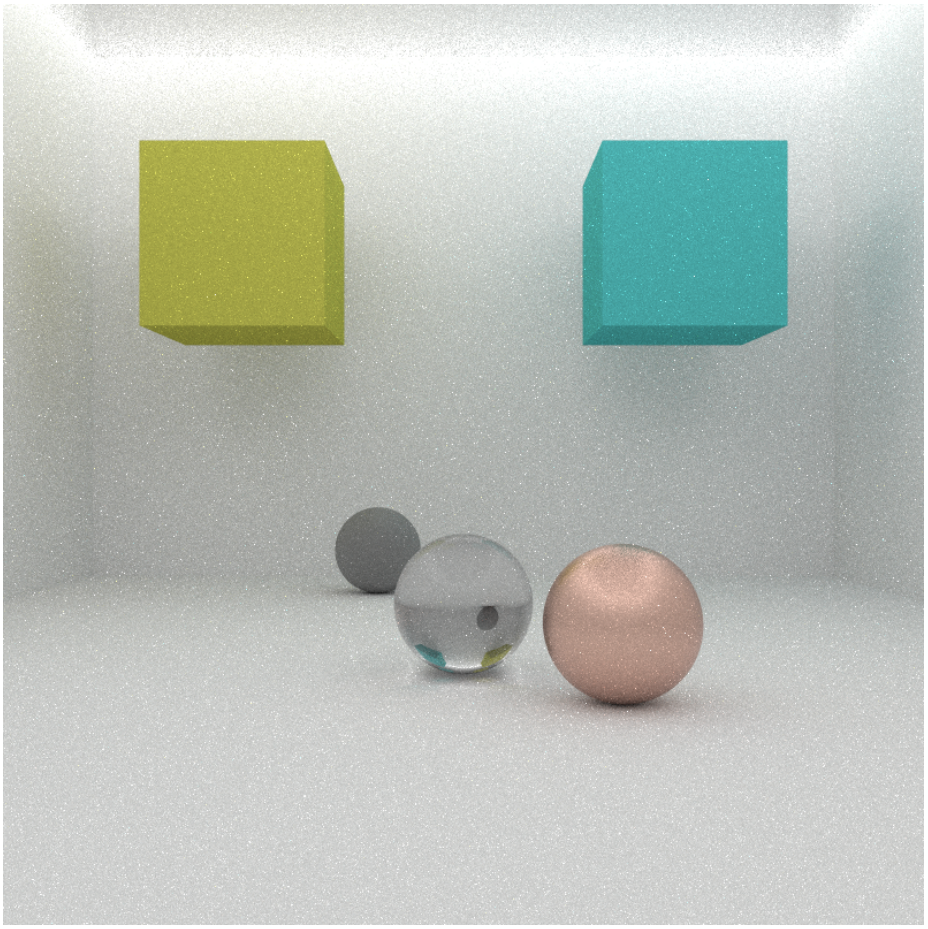
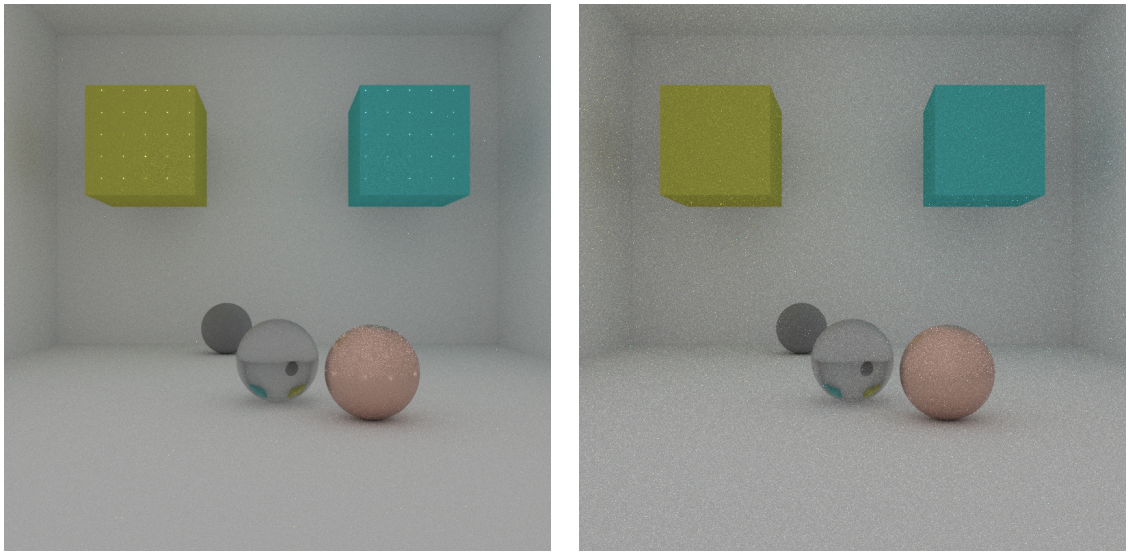


Figure 4.2: 1.6 million point lights on the ceiling



(a) Our algorithm, 32 samples

(b) Power sampling, 90 samples

Figure 4.3: 12.000 point lights in a 3D-raster

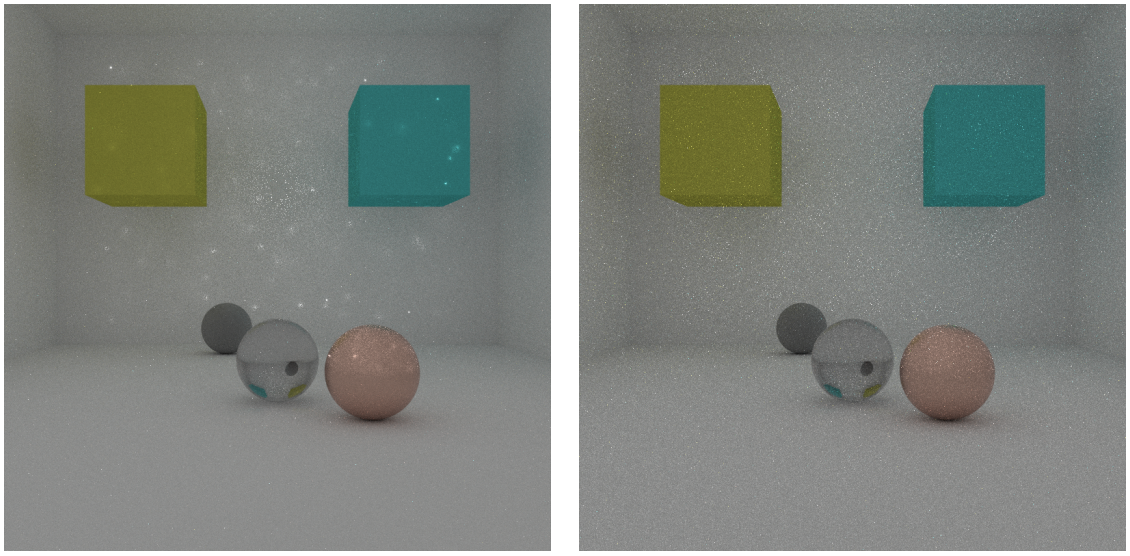
the edge of the room. Note, that the small illuminated spots you can see on the boxes in Figure 4.3a come from point lights that are very close to the surfaces of the boxes. Using power sampling, the probability of sampling that single light that has a high distribution to the surface is very unlikely and thus, you cannot see these dots while using power sampling. Also, our algorithm produces a less noisy image.

4.4 12.000 point light sources randomly distributed

This scene (Figure 4.4) is lighted by 12.000 point light sources that were randomly distributed over the whole room. Both images Figure 4.3a and Figure 4.3b are rendered with similar rendering times of about 170 seconds. Similar effects like in the scene before can be seen: Our algorithm leads to images with illuminated dots in some places where the light sources were very close to the surface, while the power sampling technique does not. Again, our algorithm produces a less noisy image with a better overall image quality.

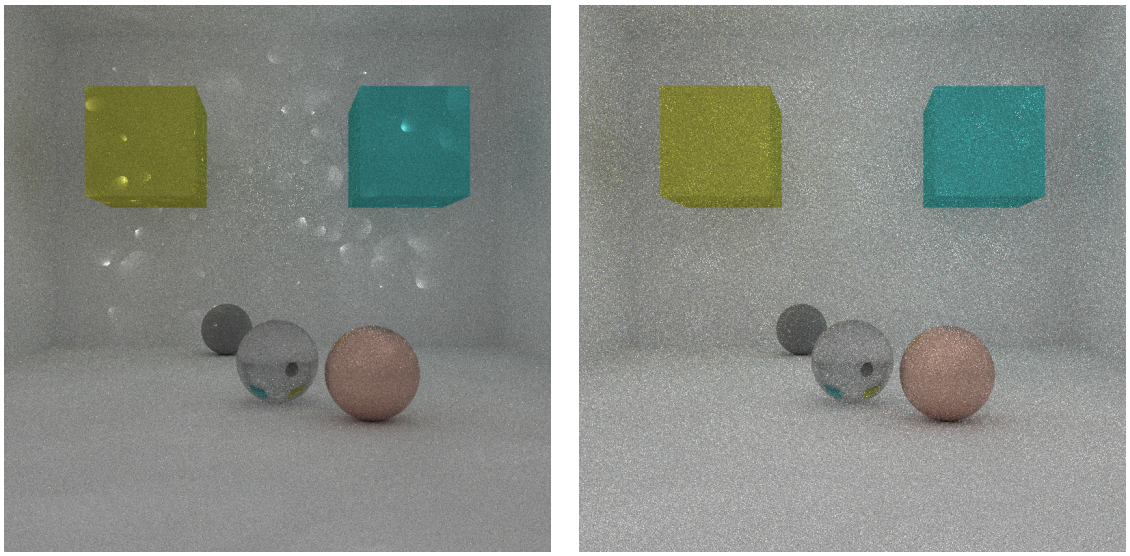
4.5 12.000 spotlights randomly distributed

This scene (Figure 4.5) is lighted by 12.000 spotlights with random position, orientation and aperture angle. Noticeable again are the effects we mentioned earlier, when a light source is close to the surface. Our algorithm again renders an image with a better quality.



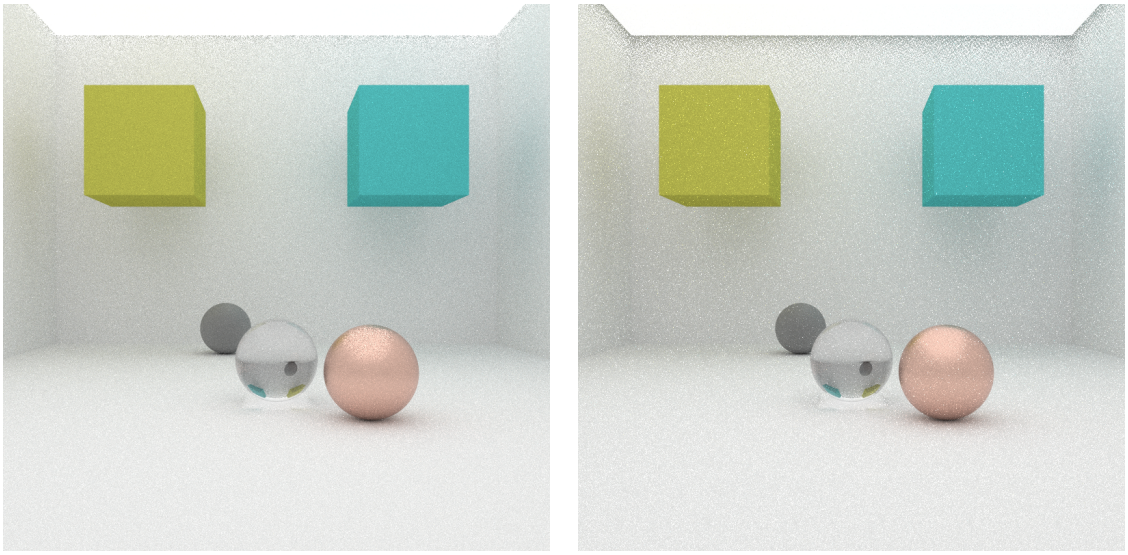
(a) Our algorithm, 32 samples (b) Power sampling, 90 samples

Figure 4.4: 12.000 point lights randomly distributed



(a) Our algorithm, 32 samples (b) Power sampling, 90 samples

Figure 4.5: 12.000 spotlights randomly distributed



(a) Our algorithm, 64 samples

(b) Power sampling, 160 samples

Figure 4.6: 2.000 area lights covering the ceiling

4.6 2.000 area lights covering the ceiling

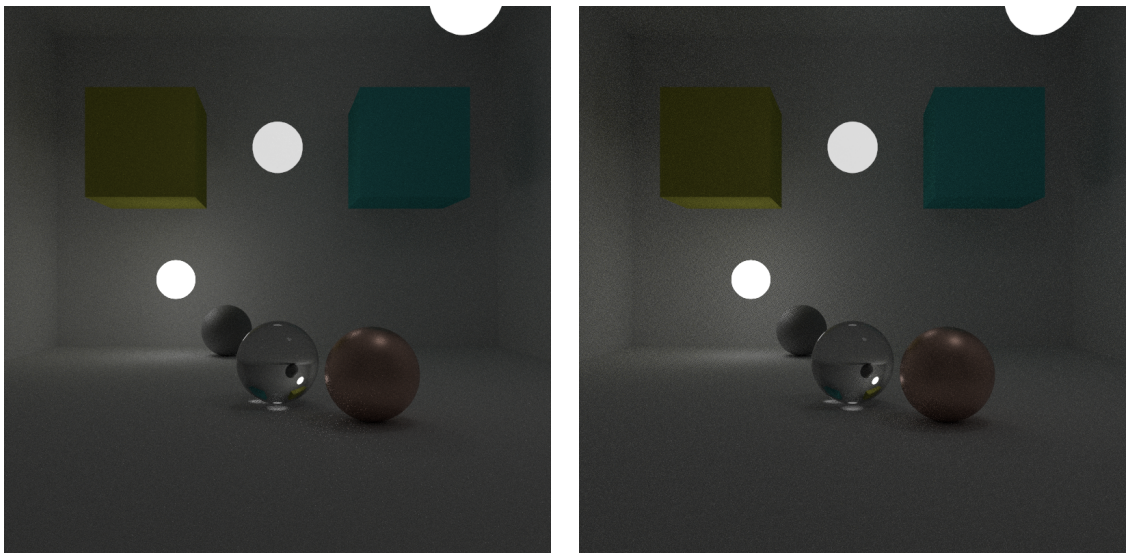
This scene (Figure 4.6) is lighted by about 2.000 triangle emitters on the ceiling of the room. Especially at the edges around the ceiling, our algorithms yields much better, less noisy results.

4.7 6.000 triangle emitters shaping 3 spheres

This scene (Figure 4.7) is lighted by 3 spheres that each consists of 2.000 triangle emitters. Again, you can see the difference in overall image quality. But, in the image we rendered with our algorithm, you can see small blight dots behind the glass sphere. The reason for that is that transmissions of mediums like glass is not covered by our algorithm. If we were to use a higher number of samples, these dots would disappear.

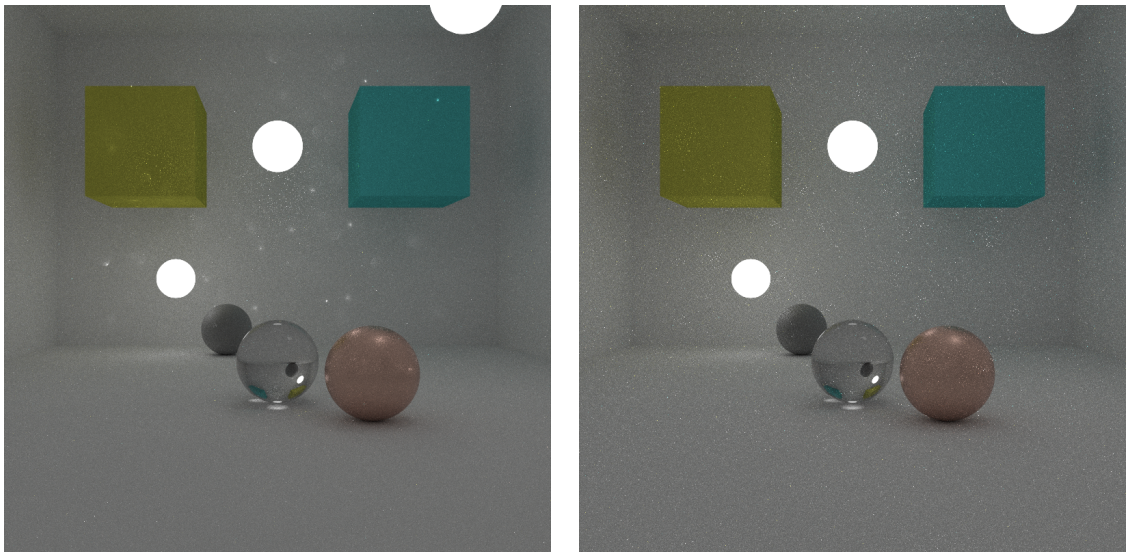
4.8 Mixed scene with 6.000 triangle emitters, 4.000 spotlights and 4.000 point lights

This scene consists of the 3 spheres from the previous scene, as well as 4.000 spotlights and point lights randomly distributed over the room. Again, there is not much to say other than that our algorithm creates a less noisy image.



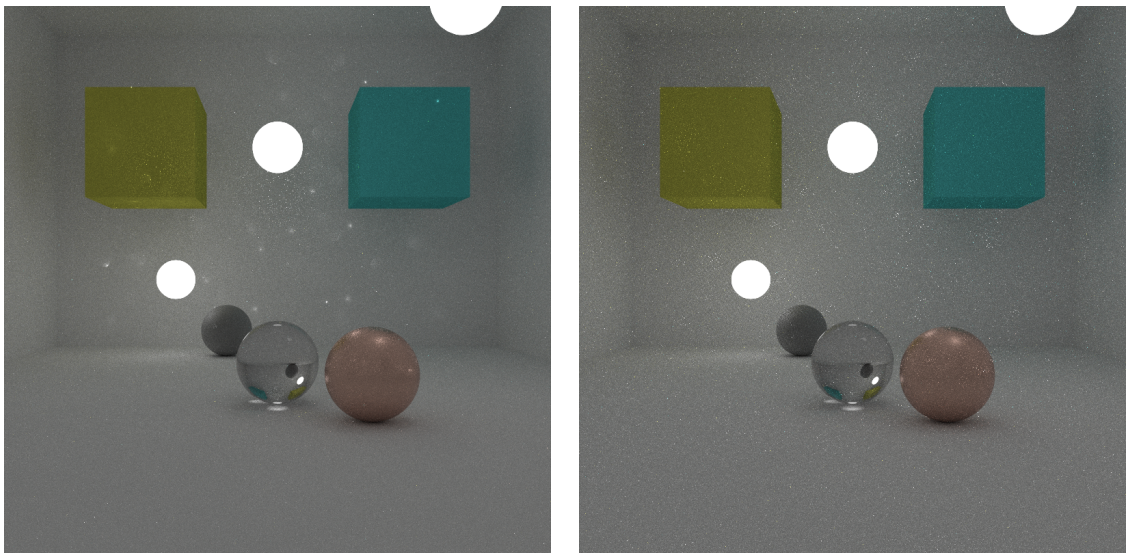
(a) Our algorithm, 32 samples (b) Power sampling, 75 samples

Figure 4.7: 2.000 area lights covering the ceiling



(a) Our algorithm, 64 samples (b) Power sampling, 150 samples

Figure 4.8: Mixed scene



(a) No split, 64 samples

(b) Split, 32 samples

Figure 4.9: Mixed scene

4.9 Splitting versus not splitting

Kulla and Conty [CK17b] showed us that splitting can improve the quality of the rendered image in some scenes drastically. Sadly, we were not able to reproduce the effects on the overall image quality that has been described in their presentation.

These two images (Figure 4.9) were rendered with the previous scenes with and without splitting. The image quality is pretty close but overall a bit better when using no splits. We think that our algorithms still has some issues that we need to address before our splitting algorithm will yield the desirable quality.

5 Related Work and Future Work

5.1 Lightcuts

Lightcuts [Wal+05] tried to deal with the same problem as we did very early on in 2005. In scenes with complex illumination with many light sources, Lightcuts tries to approximate the direct lighting by substituting a cluster of lights by a single representational light source. So, in situations, where we would want to illuminate an intersection point with every light source in the scene, we can instead illuminate the point with a smaller subset of light sources. That leads to a model that scales much better with many light sources in the scene.

The basic algorithm follows a simple concept that can be divided into certain steps:

- Convert illumination to point lights
- Build light tree
- For each eye ray, choose a cuts to approximate the illumination

We will break down the single steps for the reader. First, the complex illumination of the scene is substituted by point lights with a similar lighting. The reason for that is that the algorithm required point lights to run. Next, a binary tree that includes every emitter of the scene is built, with the interior nodes representing light clusters and the leafs representing single emitters. When rendering, for each eye ray, the algorithm calculates a cut that minimizes the number of lights sampled, while guaranteeing a certain quality threshold. A cut of the binary tree is a substitution of the total number of emitters in the scene. Either, the cut includes the actual light source or a cluster as a substitution of said light source. Therefore, when we light a intersection point with a cut, we limit the number of sampled lights.

To find out a suitable cut for an eye ray, the algorithm will start with the cluster that substitutes the whole lighting of the scene with a certain light. Gradually, the algorithm will increase the number of representative lights, as long as the quality of the lighting would increase over a certain threshold. If none of these substitutions will yield us a noticeable increase in quality, we will stop the algorithm and work with the cut calculated.

The paper further introduces reconstruction cuts, that calculates the color of certain positions sparsely over the image and then interpolates their illumination to shade the rest of the image.

While lightcuts are a interesting approach to dealing with scenes with complex illumination, while path tracing, we would have to calculate much more cuts to illuminate the scene. Also, due to it's approximating nature, the rendered image will never converge to the ideal image.

5.2 Importance Sampling of Many Lights with Adaptive Tree Splitting

The contribution of Conty and Kulla [CK17a; CK17b] was basically the source of the idea for this algorithm. Many of the things we have explained has been implementated the same way or similarly in their works. We have marked the spots in this thesis where we drastically changed our algorithm compared to their ideas. In the scenes we have tested our version of the algorithm, the images appear less noisy. Sadly, our splitting algorithm could not meet the qualities of the quality they have described in their presentation.

5.3 Future Work

As we have mentioned in the previous section, our splitting algorithms does not have the desirable quality. Obviously, that is the first thing we want to improve. Being able to archive a higher quality with a shorter rendering time is just what this thesis is about.

Next, we would like to be able to use our algorithm in scenes that are not only lighted by point lights, spotlights and area light sources. For instance, we would like to implement distant light sources like the illumination of the sun or projection light sources that model the lighting of a slide projector.

Finally, at some points we have tried optimized our algorithm, for instance by creating a more compact data structure for a faster tree traversal. On other points, we have favored keeping the code in a more readable state instead of optimizing it for development reasons. When we are sure that we will not change too much in our implementation anymore, it would clearly be desirable to optimize the code for a faster rendering time.

6 Conclusion

We have introduced the light BVH as an acceleration data structure for scenes with many light sources. In images that have been rendered with similar rendering times, our algorithm outperforms typical sampling strategies by a good margin. The differences of the image quality can be seen directly. Surprisingly, compared to simple light sampling strategies like power sampling, our algorithm already performs better in scenes with as little as 20 light sources. In addition to that, our algorithm scales very well in gigantic numbers of light sources with over 1.5 million emitters.

Our algorithm works on scenes that include point light sources, spotlights and area light sources. We showed scenes where we only had a single type of light sources or where we have combined multiple types. Additionally, we ordered these emitters in symmetrical and random ways to showcase that it works well under arbitrary conditions.

Sadly, we were not archive a better imagine quality when using splitting. As we have mentioned when we talked about future works, that is a spot we would prioritize to improve in the future.

Bibliography

[18] *NVIDIA developer website*. 2018. URL: https://developer.nvidia.com/rtx (visited on 08/21/2018).

[Bar11] T. Barnes. *Fast, branchless Ray/Bounding Box Intersections*. 2011. URL: https://tavianator.com/fast-branchless-raybounding-box-intersections/ (visited on 08/09/2018).

[Bar15] T. Barnes. *Fast, branchless Ray/Bounding Box Intersections, Part 2: Nans*. 2015. URL: https://tavianator.com/fast-branchless-raybounding-box-intersections-part-2-nans/ (visited on 08/09/2018).

[CK17a] A. Conty and C. Kulla. "Importance Sampling of many Lights with Adaptive Tree Splitting". In: *SIGGRAPH '17 Talks*. 2017. URL: https://docs.google.com/viewer?a=v&pid=sites&srcid=ZGVmYXVsdGRvbWFpbnxja3VsbGF8Z3g6NjNmMDkxZmRjZDQ5N

[CK17b] A. Conty and C. Kulla. *Importance Sampling of many Lights with Adaptive Tree Splitting Slides*. Presentation in SIGGRAPH '17. 2017. URL: https://docs.google.com/viewer?a=v&pid=sites&srcid=ZGVmYXVsdGRvbWFpbnxja3VsbGF8Z3g6NWM0NmU2YW

[CK18] *Importance Sampling of Many Lights with Adaptive Tree Splitting*. 2018. URL: http://www.aconty.com/pdf/many-lights-hpg2018.pdf.

[Jay03] E. T. Jaynes. *Probability Theory: The Logic of Science*. 1st ed. Cambridge University Press, 2003.

[Kaj86] *The rendering question*. ACM SIGGRAPH 1986 conference proceedings, 1986, pp. 143–150.

[PJH16] M. Pharr, W. Jakob, and G. Humphreys. *Physically Based Rendering, From Theory to Implementation*. 3rd ed. Morgan Kaufmann Publishers, 2016.

[PJH18] M. Pharr, W. Jakob, and G. Humphreys. *PBRT source code*. 2018. URL: https://www.pbrt.org/resources.html (visited on 08/13/2018).

[Vea97] E. Veach. "Robust Monte Carlo Methods for Light Transport Simulation". 1997.

[Vol14] J. Volpe. *Disney rendered its new animated film on a 55,000-core supercomputer*. 2014. URL: https://www.engadget.com/2014/10/18/disney-big-hero-6/ (visited on 08/13/2018).

[VS83] A. Van Oosterom and J. Strackee. *The Solid Angle of a Plane Triangle*. published in IEEE Transactions on Biomedical Engineering, Volume BME-30. 1983.

[Wal+05] *Lightcuts: a scalable approach to illumination*. ACM SIGGRAPH 2005 conference proceedings, 2005.