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

English abstract.

Zusammenfassung

Deutsche Zusammenfassung

Contents

List of Figures

List of Tables

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 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. Then again, ray tracing is not yet useful for real-time applications like video games where the rendering speed is critical. But even regarding ray tracing, we cannot completely ignore the rendering time. Too long rendering times are becoming a problem in scenes with many lights. For instance, a scene that consists of a big city with skyscrapers at night could have hundreds or thousands of lights that could potentially all affect a single point in the scene. Lighting methods that calculate the incident lighting of a point for every single light in the scene would be too slow to deal with these kinds of scenes since every ray to the camera could potentially trace multiple points that all need to be lighted.

There are 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 fast sampling speed but can lead to very noisy images that we are trying to avoid.

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

1.2 Content

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. If the reader struggles following the key parts of this section, he is heavily advised to read more extensive literature about this subject. We suggest E. T. Jaynes *Probability Theory: The Logic of Science* for this matter. [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 are often sampling 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 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:

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

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 methods do not 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

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

with Monte Carlo integration. Different to Las Vegas algorithms, 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 $f : D \rightarrow \mathbb{R}$. 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:

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

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 next section.

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 of 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 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 $\int f(x)dx$ before, 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 be archive a faster convergence rate. 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 product of two functions. In fact, the main function

we need to estimate for the direct lighting given a certain point and the outgoing direction is in that form:

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

$f(p, \omega_o, \omega_i)$ 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 that of the incident lighting.

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 estimateor 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:

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

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

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

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]

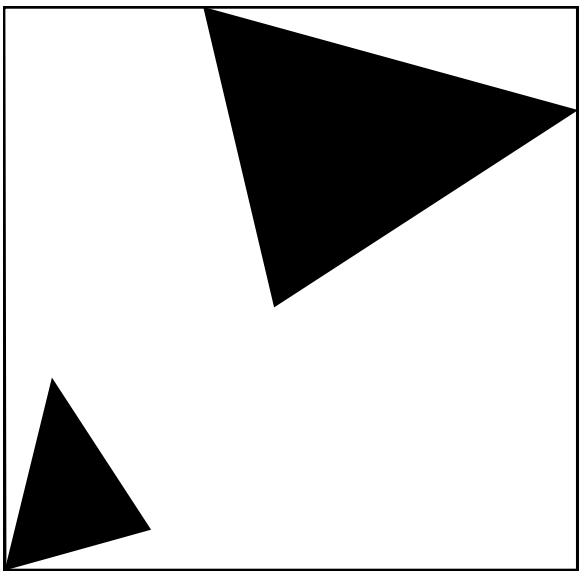

Figure 2.1: Two-dimensional bounding box for two triangles

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 define use boxes, where the 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 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 can be reduced to logarithmic in the number of object. The basic idea of those data structures is to partition the primitives into sets and order those in a hierarchy, so only specific sets of primitives need to be tested for intersections. The two main approaches, when dividing the primitives, is to either split the room among the space or to choose a particular number of primitives and wrap a bounding box around

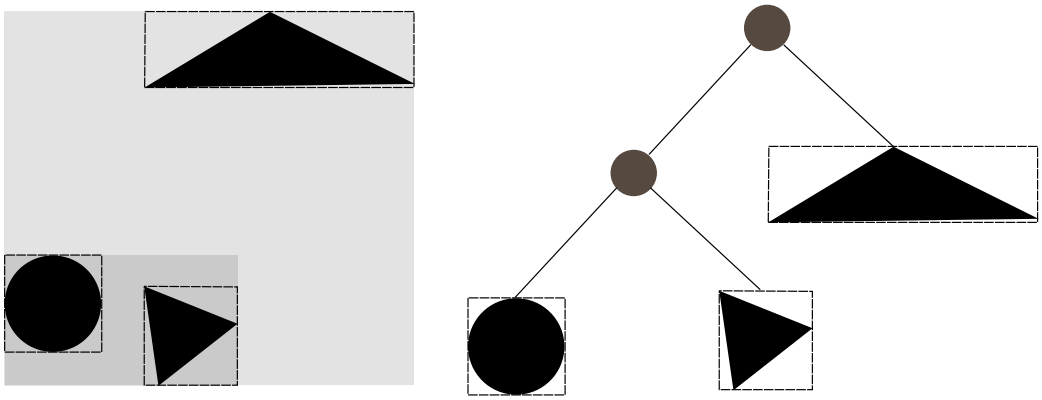

Figure 2.2: On the left side the geometric representation of the primitives in the room and on the right side the representation in the BVH

these primitives. The acceleration data structure used in PBRT is the bounding volume hierarchy (BVH). It uses the latter approach.

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 duo to round-off errors than kd-trees are. Other desired traits of BVHs is that they require a maximum number of $2n - 1$ nodes for n primitives and, since we are splitting the primitives after a particular amount of primitives, we will never have the same primitive in two different nodes.

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 up 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 2.6.1.1. While the first two split methods allow a easy and fast construction, their quality is very lacking. Recursively splitting a nodes into two disjointed sets of primitives constructs the tree. When a certain node holds

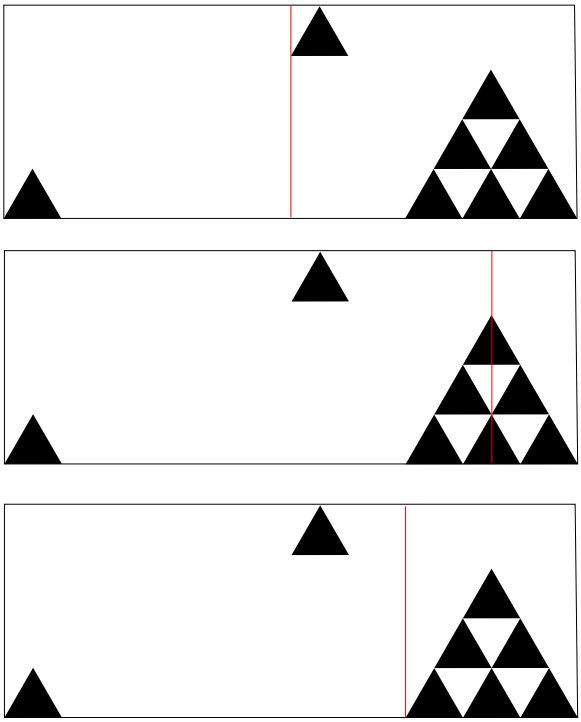

Figure 2.3: Three different split strategies: "Middle" in the top, "EqualCounts" in the middle and SAH in the bottom

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 is a bounding volume that wraps around all its primitives. When testing, if we will need to intersect any of the primitives of a node with a ray, we will be instead, intersecting 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 earlier in this paper. While they both do work well in some situations, they both have clear advantages, especially if the primitives are not evenly distributed over the scene. Looking at 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

$$\sum_{i=1}^{N} t_i sect(i), \tag{2.18}$$

where N is the number of primitives and $t_i sect(i)$ 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

$$c(A, B) = t_t rav + p_A \sum_{i=1}^{N_A} t_i sect(a_i) + p_B \sum_{i=1}^{N_B} t_i sect(b_i) \tag{2.19}$$

with $t_t rav$ 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. The probabilities p_A and p_B can be calculated using their respective surface areas:

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

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

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 interaction 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. Although most readers probably already have a good understanding of path tracing, it does not hurt to refresh the reader's mind and also helps the reader to understand what this thesis is exactly about. Still, we will try to keep this section as short as possible to avoid boring the reader.

Path tracing was first introduced in 1986 by James Kajiya. 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 light transport equation we have mentioned earlier:

$$L_o(p, \omega_o) = \int_{\varsigma^2} f(p, \omega_o, \omega_i) L_d(p, \omega_i) |cos\theta_i| d\omega_i \tag{2.21}$$

We have mentioned 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 mentioned 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. Therefore, we have to spawn rays that will hit another point in the scene. Obviously, 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 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 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 BSDF of the material of the point 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

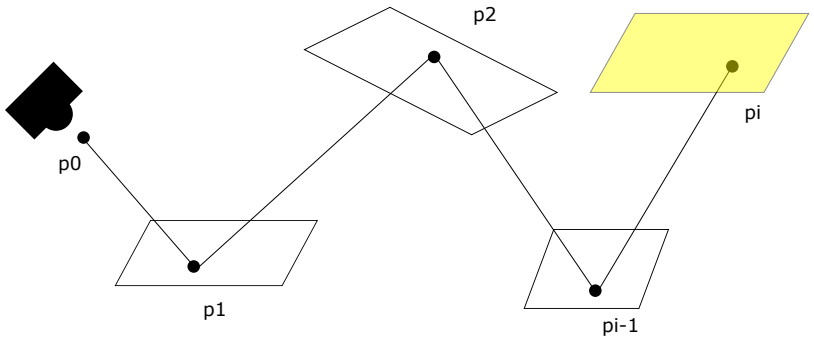

Figure 2.4: A typical path for the path tracer starting at the camera and sampling a light source as the last vertex.

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 Our 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 find the appropriate light when we are given the point of the intersection 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 only a small fraction of the rendering time, this does not pose a problem even on scenes with over one million light sources.

3.1 Informal Description of the Algorithm

In this section we will give rough overview of the ideas of our algorithm. We have thought about jumping straight into the in-depth implementation part and skipping the informal description of the algorithm but some parts of the implementation decisions are hard to understand without having at least a rudimentary overview of the algorithm. For instance, when we talk about the information every node of the tree needs to store, it is much easier for the reader to comprehend our train of thought when he has a general idea how our tree traversal algorithm works.

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 difference to the SAH. First, 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 have split the branches 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 an intersection point. Obviously, we don't just

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

Figure 3.1: Our internal representation of the Bounds_o struct

want to return a random light source which would make this algorithm pointless, but instead we want to importance sample a light that has most likely a strong contribution to the point. Therefore, we will define an importance measure for every node given the intersection point, that factors in the distance between the point 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.

What I have just described was our algorithm when we only want to sample a single light source. Conty and Kulla have shown in [CK17], that there are situations, where it is desirable to sample multiple light sources for one intersection point. 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 branch, instead of sampling only the left or the right child, we will sample both nodes. At the end, return one or multiple sampled lights. Afterwards, the sampled lights are used for the path tracing algorithm.

3.2 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 like this.

3.2.1 Bounds_o

First we will introduce the Bounds_o struct (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 3.2. In our implementation,

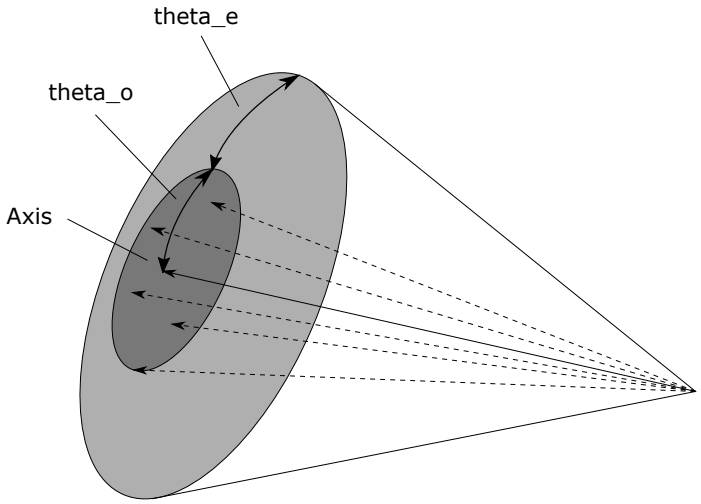

Figure 3.2: geometric idea of the Bounds_o struct

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 = $\pi/2$
- Spot light
  - Axis = spot direction
  - theta_o = 0
  - theta_e = spot's apperture
- Area light
  - Axis = normal of the geometric representation
  - theta_o = 0
  - theta_e = $\pi/2$

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

Figure 3.3: 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 3.4: Our internal representation of the LinearLightBVHNode struct

3.2.2 Node representation

We used two different implementations to represent a node in our light BVH tree (3.3 and 3.4). Both implementations are very similar, containing the two bounding bounds $bounds_w$ 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.

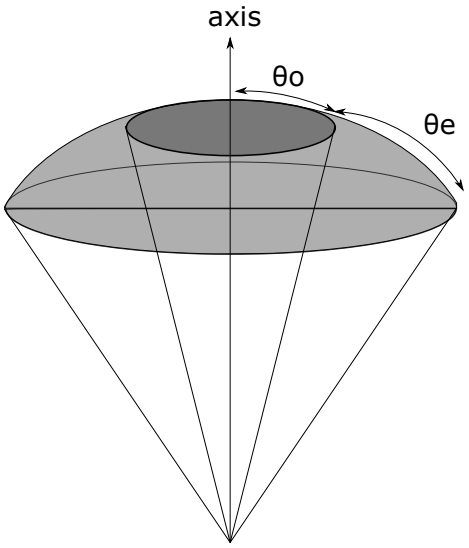

Figure 3.5: Bounding cone measure

3.3 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:

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

$M_A(C)$ 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

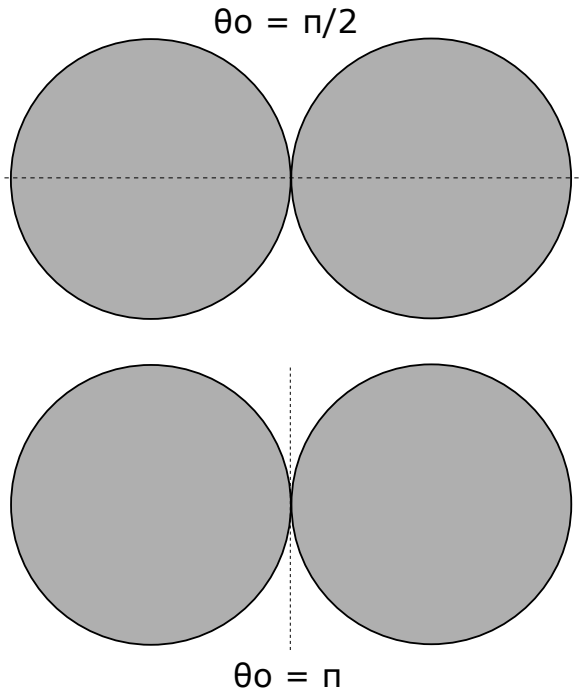

Figure 3.6: Preferred split example

sources in the cluster, while θ_e describes their emission range. Obviously, the cone enclosed with θ_o will most likely have a greater in 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 3.5. The measure we have used and has been described in [CK17] is as follows:

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

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. 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 3.6. 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 over covering half of the unit sphere. We reduced θ_o to $\pi/2$. 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

3.4 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 (3.5.3). 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 point 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 compacter form represented by *LinearLightBVHNode* objects to ensure that our tree requires as little memory as possible and to improve cache locality. *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 take 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 (3.4.7).

3.4.1 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 if we split 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

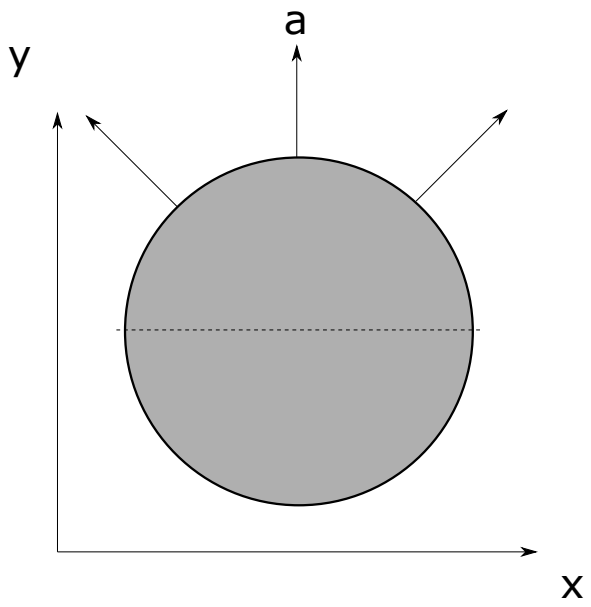

Figure 3.7: Axis calculation

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 3.7 and suppose we have chose to split the sphere in half along the axis y. Like previously discussed in 3.6, 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 archive 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 3.8 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 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 3.7. Both median and average calculations

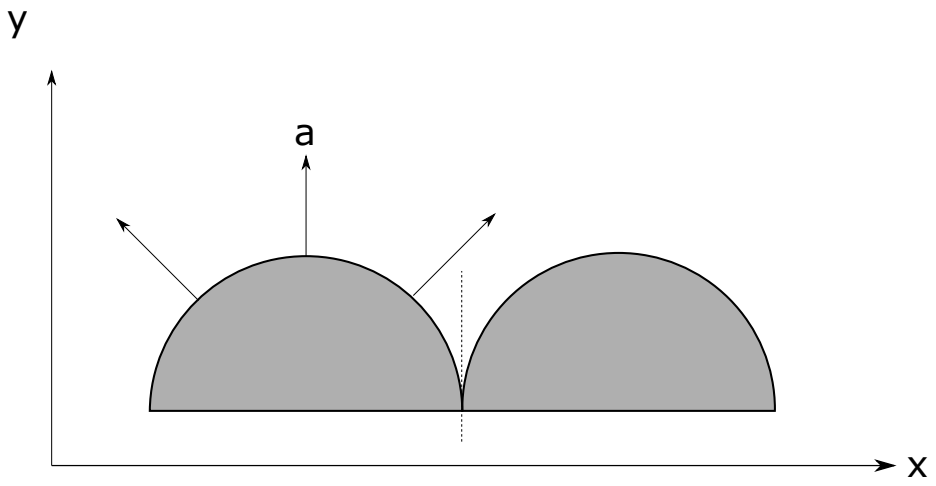

Figure 3.8: Axis calculation

have $O(n)$ time complexity, but especially in scenes with symmetric objects that represent light sources, our current approach leads to much better results.

3.4.2 Theta Calculations

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:          float current_o ← 0.f;
5:          float current_e ← 0.f;
6:          *theta_o ← max(*theta_o, radianAngle(axis, l.axis) + l.theta_o);
7:          *theta_e ← max(*theta_e, o + l.theta_e);
8:      end for
9:      *theta_e -= *theta_o;
10:     *theta_e ← min(*theta_e, π - *theta_o);
11: end procedure
```

We have listed our the calculations for θ_o and θ_e in 3. The procedure takes the base address of a vector containing of the lights that define the specific cluster, as well as 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 limit θ_e with $\pi - \theta_o$, otherwise it is possible to get a negative integral in our bounding cone measure 3.3. This is something we want to avoid because the measured volume of the bounding cone should always be non-negative.

3.4.3 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 split the light sources 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 emitters works just fine with a rendering time of TODO. On the other hand, the tree construction time of another scene with one million light sources takes TODO. 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 split the light sources. 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 10.000 splits. This way, we have an acceptable rendering time of TODO for the scene with one million light sources, while the split quality stays roughly the same like if we would calculate 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 3.9. So, instead of splitting after a certain amount of emitters, we split after a definite chunk of world space. The difference between these two ideas was not huge, but we decided to stick to splitting 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.

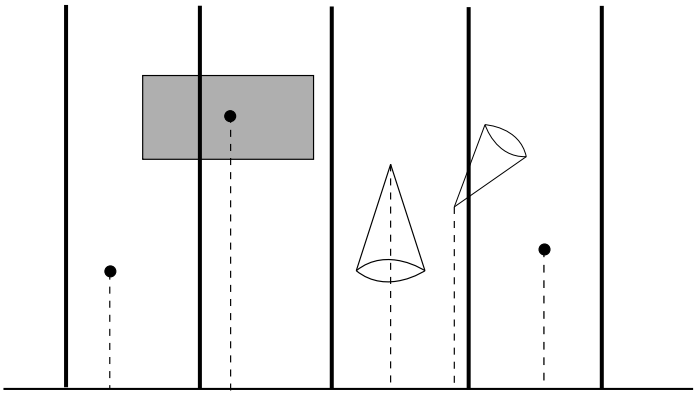

Figure 3.9: Bucket splits for area light sources, point light sources and spotlights

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
```

3.4.4 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 3.4.1, 3.4.2 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 3.3. 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.

3.4.5 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.

3.4.6 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.

3.4.7 Tree Flattening

Algorithm 6 Tree flattening

```
1: procedure int flattenLightBVHTree(LinearLightBVHNode *nodes, LightBVHNode *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. At first, we did exactly that and did not bother to flatten the tree. 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 we looked for a way to change this aspect of our light BVH representation. We chose to use a implementation that uses a contiguous chunk of memory. How we modelled a single node in our code can be seen in 3.4. 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 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 $*nodes$ (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

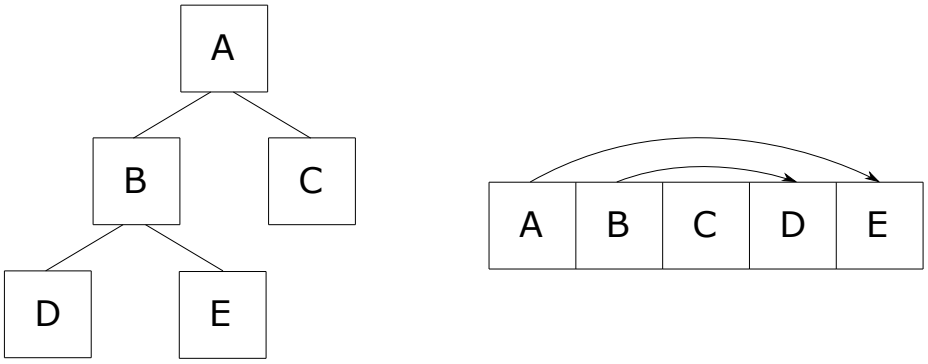

Figure 3.10: Tree representation before and after flattening

- 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 3.10. 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.

3.5 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 2.7, in our light transport equation we have two different functions in the integrand.

$$L_o(p, \omega_o) = \int_{S^2} f(p, \omega_o, \omega_i) L_d(p, \omega_i) |cos\theta_i| d\omega_i \tag{3.8}$$

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 intersection point, $L_d(p, \omega_i)$ can be importance sampled when we choose an emitter that has a big contribution to the intersection point. Our goal, in summary, is to return light sources that have a high contribution to the intersection point 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 intersection point 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 7 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
```

Algorithm 8 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
```

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 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 9 and 8. 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 intersection point, 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

Algorithm 9 Sampling a single light source

```
1:  procedure int TraverseNodeForOneLight(LinearLightBVHNode *node, 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
```

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 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 intersection point 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.

3.5.2 Importance from a shading point

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 intersection point 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

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.

Our cosine factor to the orientation bounds requires us to look at our node representation more closely. 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 shading point and pointing to the centroid and a vector starting from the shading point and pointing to any point p that lies inside of the node. A drawing of the uncertainty angle can be seen in 3.11.

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

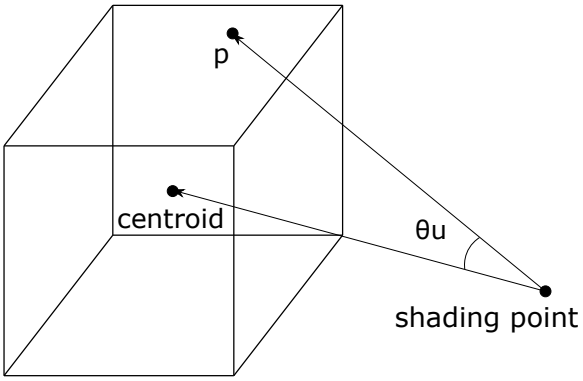

Figure 3.11: General idea of the uncertainty angle

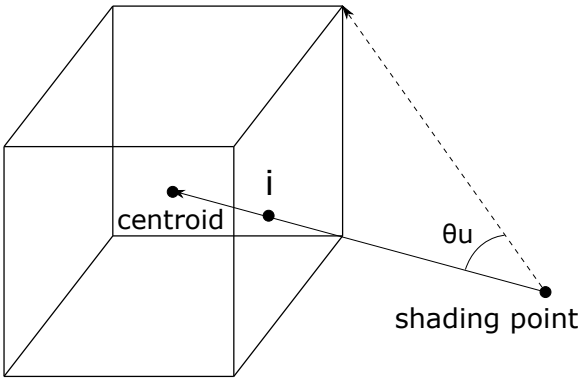

Figure 3.12: Uncertainty angle calculation by calculating the angle between a vector starting at the shading point and pointing to the centroid and the 4 corners on the intersected side of the cuboid

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 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 3.12. 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 our cosine factor to the orientation bounds becoming 1.

3.5.3 Splitting

Algorithm 10 Calculating the importance of a sampling point with a given node

```
1: procedure FLOAT CALCULATEIMPORTANCE(Intersection &it, LinearLightBVHNode* node)
2:     <check if the bounding box of node is behind the shaded point>
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:     <calculate the importance value of the node>
6: end procedure
```

4 Evaluation

5 Conclusion

…

Bibliography

[CK17] 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=ZGVmYXVsdGRvbWFpbnxja3VsbGF8Z3g6NWM0NmU2YWV

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

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

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