LUND UNIVERSITY

MASTER'S THESIS

Two Hybrid Methods of Volumetric Lighting

Author:
Anders NILSSON
Master of Science in Engineering Mathematics
Lund University

Advisor:
Petrik CLARBERG
Department of Computer Science
Lund University

ABSTRACT

This thesis presents two new viable approaches to accelerating the visualization of light shafts from a point light source. A coarse representation of the three-dimensional scene is used to identify regions of space that are fully lit, fully in shadow and those that can not be determined from the coarse representation alone. The two methods then visualize the different regions as efficiently as possibly. The first of the methods fully relies on sampling but ignores many of the evaluations required by previous sampling-based methods. The other method uses an analytical solution in the fully lit region, drastically reducing the required fillrate of the algorithm.

March 20, 2008

Table of Contents

Chapter 1

Introduction

Visualization of environments in computer games and computer generated films can benefit from taking participating media into account. Usually only the effects of light being reflected at object surfaces are visualized but light also interacts with microscopic particles in the air giving rise to effects such as fog and haze. Light also interacts with the molecules inside within objects resulting in subsurface scattering.

Photons from the sun that enters a cloud interact with the water molecules. Some are absorbed and some are scattered in new directions. In the space between two clouds we can see a ray of light known as a god ray. The same effect can be seen in a foggy environment behind a character being lit by a light source; in front of the character we can see the lit air, behind him shadows in the air.

1.1 Previous Work

There are many different problems related to visualizing participating media. Different restrictions result in different methods. The main differences are in how the density of the media are modeled. Either it is homogeneous (constant density) or inhomogeneous (varying density). If it is inhomogeneous it can either be arbitrary (defined through a function or a texture) or expressed as a sum of radial function (such as Gaussians) centered around 3D points. Another restriction is in the type of scattering that is taken into consideration. Single-scattering means that photons are only scattered once. In multiple-scattering photons can be scattered an arbitrary number of times. In [single or multiple] forward scattering, photons can only be scattered in the forward direction so that they always travel away from the light source. See Section 2.1 for a discussion on scattering. Another restriction is how advanced the scattering is modeled. Depending on the type of the media, different models are suitable from a physical standpoint. If, however, a simpler scattering model is chosen over the correct, model performance can be improved.

Jensen and Christensen, show how photon mapping [2] can be extended to include volumetric effects [1]. The effects of multiple scattering of photons in inhomogeneous air and inside objects are taken into account. The method is divided into two phases. In the first phase, photons are emitted from light sources. They are stored at surfaces they hit and also in the air where they interact with the medium. In the second phase an image is rendered from the camera using ray marching. Incoming light at a surface point or in the air can be approximated using a photon density estimation. The result from the first phase is stored in a kd-tree for efficient density evaluation. Unlike rendering using a standard photon map (without the volumetric effects), ray marching must be used instead of ray tracing. This increases the rendering time. The resulting images are realistic but the rendering times are long and the method is not suited for real-time rendering.

A real-time sampling based method was introduced by Dobashi et al. [6]. It supports visualization of light shafts in a single scattering homogeneous media. We review their technique in Section 3.1 in detail, since it forms the basis for our own work.

Harris et al. investigate multiple forward scattering in clouds composed of Gaussian particles with varying density [14, 15, 16]. They focus on finding the correct amount if incoming light for each such particle. Each particle that the clouds are composed of, is rendered as a billboard. Air without particles, is not rendered so light shafts cannot be captured by this method. It archives real-time performance by caching the images of the individual clouds in dynamically updated impostors. A blend between this method and the sampling method introduces cloud rendering with light shafts [7]. This method, however, is currently not running in realtime.

Y. Zhu et al. uses depth peeling to extract an interval per pixel that is inside a light cone [4]. The light inside the cone is then sampled using a shadow map. With hardware that supports dynamic branching we can optimize the sampling phase, using fewer shadow map lookups in uncomplicated intervals.

Kun Zhou et al. introduce a method for visualizing a single scattering inhomogeneous participating media [3]. The density of the media is modeled as the sum of radial basis functions, each a Gaussian centered around a three-dimensional point. Simplifying approximations allow the method to run in realtime, but makes it unable to capture high frequency effects, such as, light shafts.

Eva Cerezo et al. reviews many volumetric methods [5]. Mitchell has written a review, that focus on realtime methods of light shafts visualization [9].

1.2 Outline

In Chapter 2 we look at the model used to describe the light sources and the participating media. In Chapter 3 we look at two methods for visualizing the participating media. One of the methods introduces and needs the *visible set* in order to function. In Chapter 4 we describe a tree structure suitable for finding the visible set of simple triangular scenes. Finally in Chapter 5 we utilize the tree structure and develop two hybrid schemes for visualizing the participating media. The results are presented in Chapter 6 and our conclusions in Chapter 7. Ideas for further improvements on the two methods are discussed in Chapter 8.

1.3 Contributions

Our first contribution is a data method that partitions the volume of a point light source into three different regions; fully lit, uncertain and fully unlit. The result can be used to accelerate methods that visualizes the effect of point light sources in a homogeneous participating media. Our second contribution is the development of two such methods that utilize this data structure to accelerate the rendering process. The improved methods are based on the idea of moving some of the GPU-heavy calculations from the GPU to the CPU. While this goes against the current trend in realtime graphics, it might make sense in the context of rendering volumetric effects that are fill rate bound.

Chapter 2

Participating Media and Light Model

Much of the work presented here is based on Dobashi et al's work [6] using Nishita and Nakamae's light model [12] as a basis for the light transfer inside the participating medium. In this chapter this model is introduced.

2.1 Absorption and Scattering

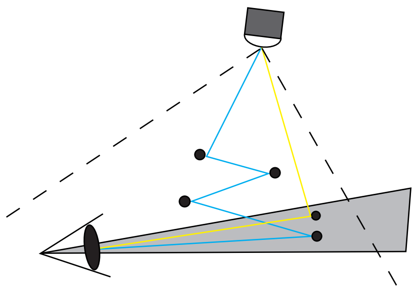


Figure 2.1: Multiple and Single Scattering of Light Rays

When photons leaving a light source collide with small microscopic particles, they will either be absorbed or scattered in a new direction. In Figure 2.1, we can see two different paths of photons reaching the eye. The photons traveling the yellow path are scattered only once while the photons traveling the blue path are scattered multiple times. We denote the two cases single and multiple scattering. If we take the effects of absorption and scattering into account we say that we have a participating media that influence the rendering.

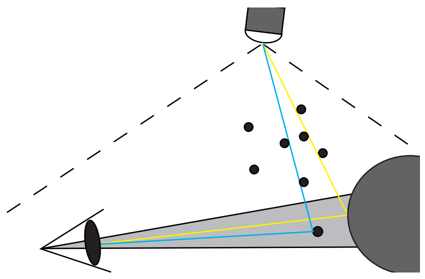


Figure 2.2: Scattering of Photons

In realtime rendering the influence of a participating media is often limited to quite simplistic fog. If we study the two paths in Figure 2.2, the scattering from the yellow path is sometimes taken into account while contribution from the blue path is ignored. Objects that are far away will correctly be darker and objects that are far from light sources will get less incoming light. However, the fog itself will not be visualized.

We denote the intensity of the light from a light source $L(\mathbf{x}, \mathbf{w})$ where $\mathbf{x}$ is a point in space and $\mathbf{w}$ the direction[1]. If we denote the rate of absorption σ_a and the rate of scattering, σ_s we can define the rate of extinction as $\sigma_t = \sigma_a + \sigma_s$. If the medium is homogeneous σ_t is constant and we get

$$\begin{aligned} \frac{dL(\mathbf{x}, \mathbf{w})}{ds} &= -\sigma_t L(\mathbf{x}, \mathbf{w}) && (2.1) \\ &\Rightarrow \\ L(\mathbf{x} + s\mathbf{w}, \mathbf{w}) &= e^{-\sigma_t s} L(\mathbf{x}, \mathbf{w}) = T(s) L(\mathbf{x}, \mathbf{w}). && (2.2) \end{aligned}$$

Here $T(s)$ denotes transmittance, the amount of intensity of light that is left after traveling s length units. With the choice of $T(s) = e^{-\sigma_t s}$ we get the well-known Beer's law. If the initial energy leaving the light was I_{obj} then the light reaching the eye s units away is $I_{obj} T(s)$.

2.2 Phase Function

When a photon with the direction $\mathbf{w}'$ collides with an air particle it is scattered with a probability σ_s. It will then continue in a possibly new direction according to a phase function $p(\mathbf{w}' \to \mathbf{w})$ where $\mathbf{w}$ is the new direction. Independent of $\mathbf{w}$, all scattered photons will continue in some direction so the total probability is:

$$\int_{S^2} p(\mathbf{w}' \to \mathbf{w}) d\mathbf{w}' = 1. \qquad (2.3)$$

[1]Notice that this is intensity per time unit. Since light in computer graphics is considered to travel with infinite speed this functions only describes the instantaneous light distribution in space. Normally this can easily be defined for any point but if we take scattering into account it is not so easy.

Often the air particles are spherical and the probability function is spherically symmetric so the phase function is often a function of one variable $\alpha = cos^{-1}(\mathbf{w} \cdot \mathbf{w'})$ denoting the angular difference between the incoming direction and the scattered direction. A phase function suitable for a hazy atmosphere [12] is given by:

$$p(\alpha) \quad = \quad K(1 + 9\cos^{16}(\frac{\alpha}{2})), \tag{2.4}$$

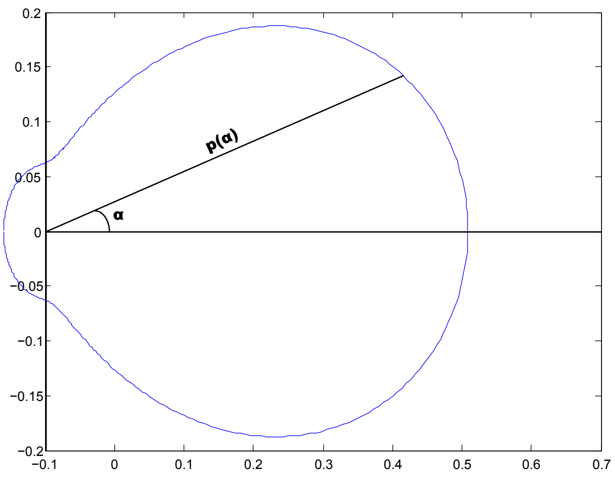


Figure 2.3: A Phase Function suitable for a hazy atmosphere

where K is a normalization constant[2]. In Figure 2.3 we see that there is a high probability of scattering in the forward direction while there is little scattering in the backward direction.

2.3 Light Model

Each pixel on the screen corresponds to an eye ray that starts at the point $\mathbf{p_0}$ and has a direction $\mathbf{\Omega_e}$ that depends on the pixel[3]. The points on a ray are then given by:

$$\mathbf{P}(z) = \mathbf{p_0} + z\mathbf{\Omega_e}, \ z \geq 0, \ |\mathbf{\Omega_e}| = 1. \tag{2.5}$$

In order to render a scene we need to calculate the intensity of the light that arrives at $\mathbf{p_0}$ from the direction $-\mathbf{\Omega_e}$. We denote this intensity I_{eye}. Usually this is a vector that has three intensities; red, green and blue.

In Figure 2.4, we can see the principals of our lighting model. There are two types of light contributions. The light source illuminates the object resulting in the intensity I_{obj} being reflected toward the eye. It is then attenuated due

[2]See appendix A for details on K.
[3]Here we assume a pin-hole camera.

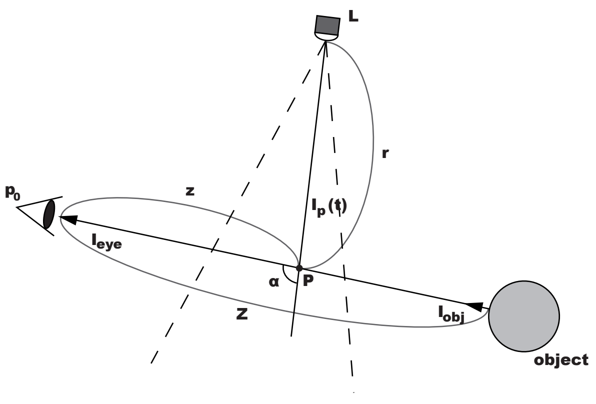


Figure 2.4: Light Model

to absorption and scattering. The contribution to I_{eye} is $I_{obj}T(Z)$. The other contribution comes from single-scattering toward the eye, from each point on the eye ray. The amount of the scattering is governed by σ_s and the phase function. In general, we have:

$$\begin{aligned} e(z) &= T(z)I_p(z)\sigma_s p(\alpha), && (2.6) \\ I_{eye} &= I_{obj}T(Z) + \int_0^Z e(z)H(z)\mathrm{dz}, && (2.7) \end{aligned}$$

where $p(\alpha)$ is the phase function, $T(z)$ the transmittance, $H(t)$ the boolean visibility at P and $I_p(t)$ the light intensity at P[4]. The function $e(z)$ is the point wise contribution to $I_s(Z)$ assuming $H(z) = 1$.

Our goal in this paper is to improve the previous methods for evaluating $I_s(z)$. We use the phase function (2.4) and the transmittance function (2.2).

A point light is modeled by:

$$I_p(t) = \frac{I(\mathbf{\Omega})T(r)}{r^2}, \qquad (2.8)$$

where Ω is the direction from the light source to the point $\mathbf{P}$. A simple spotlight can be modeled by using a point light and varying the intensity $I(\Omega)$ depending on the angle between Ω and the forward direction of the spotlight.

$$I(\mathbf{\Omega}) = \begin{cases} 1, & cos^{-1}\left(\mathbf{L}_{\text{forward}} \cdot \vec{\mathbf{LP}}\right) \leq \mathbf{L}_{\text{fov}} \\ 0, & \text{otherwise} \end{cases} \qquad (2.9)$$

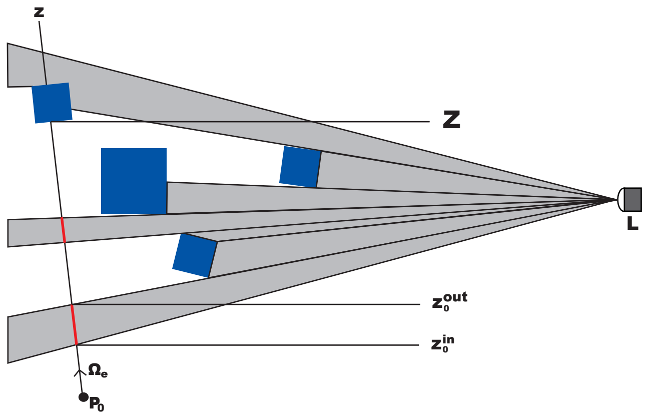


Figure 2.5: Lit intervals on an eye ray.

2.4 Reformulations

In Figure 2.5, we can see an eye ray decomposed into the fully lit segments, drawn in red, in front of the scene geometry. If we denote the intervals of these segments $[z_i^{in}, z_i^{out}]$, we can use the integral of $e(z)$:

$$E(z_0, z_1) \quad = \quad \int_{z_0}^{z_1} e(z)dz, \tag{2.10}$$

to obtain $I_s(Z)$ using summation as

$$I_s(Z) = \sum_i E(z_i^{in}, z_i^{out}) = \sum_i E(0, z_i^{out}) - \sum_i E(0, z_i^{in}). \tag{2.11}$$

This will turn out to be useful in Section 3.2.

2.5 Simplified Lighting Model

For evaluation of our method, that requires $E(z)$, we have developed a simplified version of $e(z)$ called $\hat{e}(z)$. It has a form that is easier to integrate, yielding $\hat{E}(z)$. We will now describe this model.

This simplified model ignores the effect of transmission of light. Light is only attenuated by the squared distance from the light source. The terms a_0 and a_1 can be used to keep the light from being to bright near the light position, and to fake some transmission from the light source to the point. A simple isotropic phase function $p(\alpha)$ is used. The terms in the light model is the same as in (2.6)

[4]Obviously $I_p(t)$ influences I_{obj} as well but we are not interested in I_{obj} and we don't want to burden this presentation with the BRDF of the objects material.

with:

$$\begin{aligned}
\hat{e}(z) &= T(z)I_p(z)\sigma_s p(\alpha), & (2.12)\\
I_p(z) &= \frac{I(\mathbf{\Omega})}{a_1 r(z)^2 + a_0}, & (2.13)\\
p(\alpha) &= \frac{1}{4\pi}, & (2.14)\\
T(s) &= 1, & (2.15)
\end{aligned}$$

and $I(\Omega)$ as in (2.9).

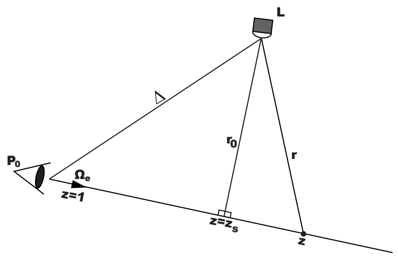


Figure 2.6: Finding $r(z)$.

First let us find an expression for $r(z)$. From the Figure 2.6 we can see that there must be a point on the eye ray that is closest to the light source. We denote the z-value of this point z_s and the distance from the light source to the point r_0. We see that z_s is obtained as the orthogonal projection of Δ on $\mathbf{\Omega_e}$. Pythagora's Theorem gives us an expression for r_0 and of the exact distance from the light source to any point on the eye. The sought identities are:

$$\begin{aligned}
\Delta &= \mathbf{L} - \mathbf{p_0}, \\
z_s &= \Delta \cdot \mathbf{\Omega_e}, \\
r_0 &= \sqrt{|\Delta|^2 - z_s^2}, \\
r(z) &= \sqrt{r_0^2 + (z - z_s)^2}. & (2.16)
\end{aligned}$$

Now we want to find the integral of $\hat{e}(z)$. First consider only segments within the fully lit cone such that $I(\mathbf{\Omega}) = 1$. Using (2.16) we get a function of z to integrate. The result is

$$\begin{aligned}
\hat{E}(z_0, z_1) &= \int_{z_0}^{z_1} \hat{e}(z)dz = \\
&= \frac{\sigma_s}{C4\pi}\left[\tan^{-1}\left(\frac{a_1(z_s - z_0)}{C}\right) + \tan^{-1}\left(\frac{a_1(z_1 - z_s)}{C}\right)\right] & (2.17)
\end{aligned}$$

with

$$C = \sqrt{a_1(a_1 r_0^2 + a_0)}.$$

Now consider a segment that is not fully inside the cone such that $I(\mathbf{\Omega})$ varies. If we want to calculate $\hat{E}(z_0, z_1)$ we can use a cone intersection test [22] to determine a new smaller interval on the ray that lies fully inside the lit cone. First we determine the interval on the eye ray that is inside the cone. We then take the intersection of that interval and $[z_0, z_1]$ as our new interval. The key here is that $I(\mathbf{\Omega})$ is either 0 (outside a cone) or 1 (inside a cone).

While this model is not physically correct it matches some of the lighting methods used in realtime graphics. Sun et al. has presented an approximation to the full integral of $e(z)$ [8]. Combined with the cone intersection above it could be used for more realistic results for spotlights as well. We have chosen to use the simpler model in order to limit the scope of this thesis.

Chapter 3

Visualization of Media

This chapter investigates two methods to visualize the spotlight cone in participating media using a GPU. As a light source shines on the particles in the participating media, light is scattered from each visible point toward the viewer according to an intensity function $e(z)$ (for a given eye ray). The function $e(z)$ does not take occlusion between the light source and the point $p(z)$ into account. We denote the volume that is actually illuminated $\mathbf{B} = \{p(z) : H(z) = 1, z \in [0, Z]\}$.

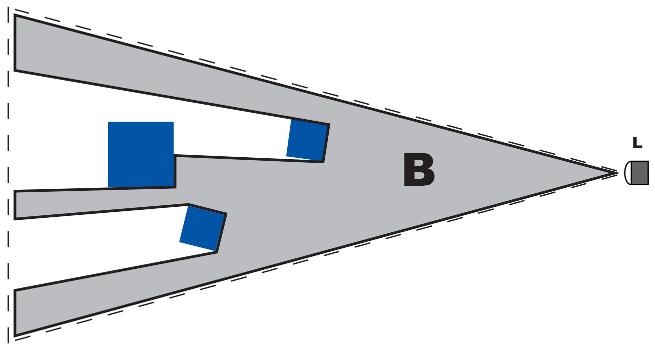


Figure 3.1: The illuminated volume **B**.

For each ray from the eye through the spotlight cone we shall calculate the total amount of incoming light on that ray. First, the surfaces of the scene is rendered using the graphics card. The term $I_{obj}T(Z)$ is written to the frame buffer and the z-value of that fragment is written into the z-buffer. This corresponds with standard rendering with fog of constant density, with $T(Z)$ being the effect of the fog. Second, the effect of the participating media, $I_s(Z)$, must be added. Together they form the full solution. The two following sections will discuss two methods to find $I_s(Z)$.

3.1 Sampling based approach

Dobashi et al. approximates $I_s(Z)$ using sampling of the integral in (2.7) [6, 7]. The integrand is assumed to be constant over a small depth. Several virtual planes parallel with the image plane is rendered at different z-values, dz steps apart, as in Figure 3.2[1].

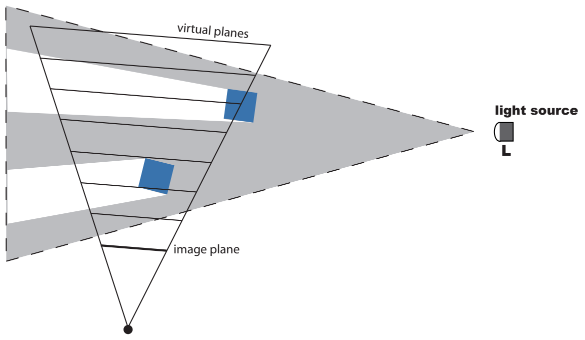


Figure 3.2: Virtual sampling planes aligned with the image plane.

As we can see in Figure 3.3, portions of the sampling planes that are behind geometry in the z-buffer, here drawn in red, will not be drawn, resulting in contributions only from visible portions of the eye rays.

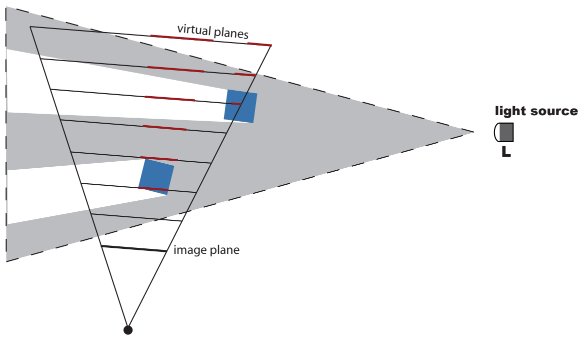


Figure 3.3: Red portions of sampling planes behind scene geometry.

While most of the terms of (2.7) can be be easily evaluated at the center point (or by using a closed form solution to the full integral) the term $H(z)$ is more troublesome. In the sampling based approaches it is evaluated using shadow mapping [27]. Shadow mapping enables evaluation of the visibility at an arbitrary point in space by projection of the shadow map. This makes it suitable

[1]Notice that if you place planes dz apart then the distance between the intersections of a ray and those planes will depend on what ray you chose. For eye rays only the ray that looks along the forward vector will intersect the planes every dz length unit, to the sides the intersections will be further apart. Take care that most of the times we use z to refer to a distance along a ray, not to the z-value of a point on that ray (unless the z-buffer is involved in the argument).

for evaluating $H(z)$ in mid air. In Figure 3.4 we can see the portions of the sampling planes that receive no light due to occlusion drawn in red.

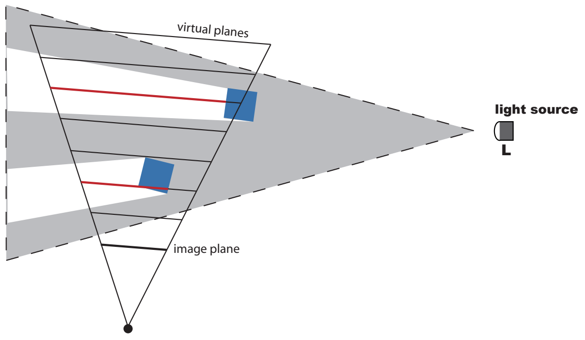
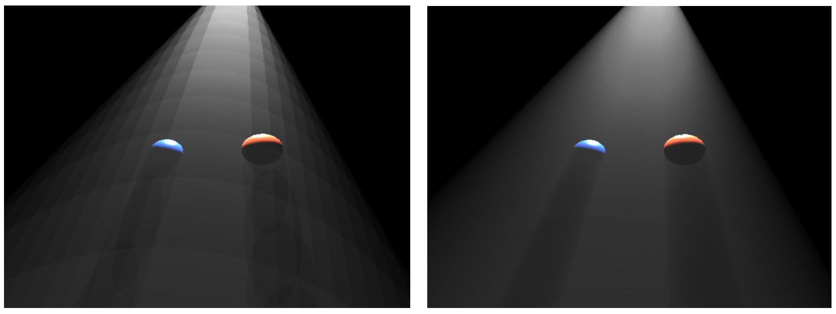


Figure 3.4: Red portions of the sampling planes in shadow.



(a) Too few virtual planes. (b) Enough virtual planes.

Figure 3.5: Effect of under-sampling.

There are two main problems of this algorithm. First, there is aliasing both due the sampling by virtual planes, and due to shadow mapping. In Figure 3.5 we can see what happens when too few planes are used. The other problem is that the algorithm requires rendering a lot of planes covering large portions of the screen which results in a lot of fragments being rendered by the GPU. We say that the algorithm is fill-rate bound[2].

Mitchell suggests increasing the efficiency by clipping the planes to the volume that bounds the influence of the light source to reduce the fill-rate [9]. A ground plane is also proposed such that portions below the floor will not be needlessly rendered. He also suggest more aggressive trimming of the sampling planes so that portions that are known to be lit does not perform a possibly costly shadow map computation, and that portions that are known to be in shadow could be ignored altogether.

A third problem with the sampling method is due to low precision in the texture formats, accumulating quantization errors as the number of planes increases. This can be seen in Figure 3.5(b).

[2]The limiting factor is not the number of vertices but the numbers of fragments.

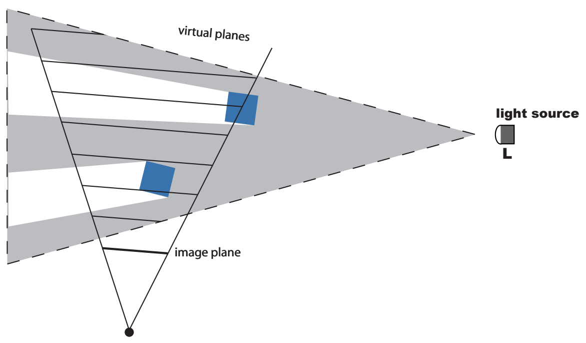


Figure 3.6: Sampling Planes trimmed by the light frustum.

3.2 Volume-based Integration

In this section, we will present a new approach to light shaft visualization that is inspired by existing algorithms of fog volume visualization.

When programmable graphics cards was new several simple methods to visualize volumes of fog was developed. Boyd and Baker describes one such method [10]. The basic concept is simple; the distance a ray travels through the fog volume affects how much off the fog colors it picks up. The distance is measured by comparing the two z-buffers, one that contains the backside of the triangles in the fog volume, one that contain the front-sides. Given that the fog volume is convex the distance can be calculated with a simple subtraction. Complications arise when the fog volume intersects geometry or when the camera is inside the fog volume. One last issue is how to store the z-buffer with enough precision, given the low number of bits per pixel typically for textures found on older graphics cards.

In Figure 3.7, we can see a light source and the scene geometry. We can also see the volume **B** where the light source has some influence. Had there not been any geometry in the world **B** would be as large as the full dashed influence volume of the light source. However since there is blocking geometry not all points will be illuminated. If we place ourself at the light position and look around, we see the portions of the polygons that will be lit. The air that will be illuminated is the air in front of those lit polygon portions, but inside the dashed influence volume.

3.2.1 The Visible Set

From now on we assume that the influence volume of a light source is a volume that fully encloses the space that the light source affects in a significant way (regions where the intensity of the light is impossible to perceive can be left outside this volume). For a spotlight, we use a pyramid with the apex at the light source. This is important because we want to be able to express the surface of the influence volume using triangles. Had we used the true cone

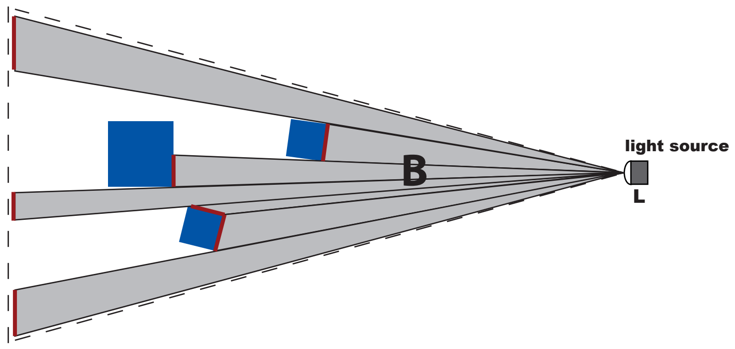


Figure 3.7: The Visible Set

as our influence volume the surface of that volume would be curved, requiring many triangles in order be express properly.

We denote the polygons that the are visible from the light source, with occluded portions trimmed away, as *the visible set*. We include the backplate of the light source influence volume. In Figure 3.7, these are the polygons drawn in red. As we can see in the figure each such polygon contributes with a pyramid to the volume **B**, with the polygon as the base and L as the apex. All the individual pyramids taken together spans **B**. Since no polygons are overlapping as seen from the light source the individual pyramids do not overlap, and also there can be no geometry inside them. This yields the full volume **B**. The surface of **B** is not as simple as it could be since it has a lot of interior polygons as shown in Figure 3.8.

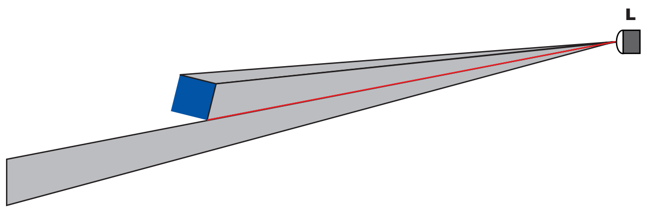


Figure 3.8: Two pyramids that forms **B** with duplicated polygons in red.

In Chapter 4, we describe a data structure called the Beam Tree that can be used to construct the visible set. An alternate approximate algorithm based on image analysis methods to find the surface of **B**, without any interior polygons such as the on in Figure 3.8, is proposed by McCool [17].

3.2.2 Visualization using the GPU

Our aim is to calculate $I_s(Z)$ for each pixel simultaneously. We do this by exploiting equation (2.11). By rasterization of the surface of **B** (the surfaces of all the pyramids) the pixels that are written to are those that will receive scattered light from **B**. For a front-side face then the length to the rasterized position for the written fragment, z_{frag}, is the depth of an entry into **B** for a given pixel[3]. In the same manner a backside face represent and exit out from **B**.

Each in/out-pair represents a fully lit segment of the corresponding eye ray for a given pixel. See Figure 2.5. Since no geometry can intersect **B**, the z-buffer will for an interval either do nothing (if the scene geometry is behind and not blocking the scattered light) or remove both in/out (if the scene geometry blocks the scattered light).

Thus we can find the term $\sum_i E(0, b_i)$ in (2.11) by rendering the backsides of **B** using additive blending, writing the term $E(0, z_{frag})$ in the fragment shader. The term $-\sum_i E(0, f_i)$ can be found in the same manner by rendering the front-sides, using subtractive blending and writing $E(0, z_{frag})$.

If the eye point is inside **B** there will be an exit from **B** but no entry into it. The backside will be rendered but not the front side. This, however, poses no problem since $E(0, b_i)$ represents the full integral from 0 to b_i which is what we wanted.

3.3 Discussion

The two approaches work in two very different ways and they have different benefits and drawbacks.

The performance of the sampling method is highly dependent on the number of virtual sampling planes and the visible size of them. The number of planes can be somewhat reduces by introducing noise but even then it is an expensive method. In the distance however the size of the sampling planes shrink and the number of planes can be also be reduced. So while sampling is slow it is easy to adapt the quality of it depending on how large the spotlight cone is on the screen.

The performance of the volume-based integration method is dependent on a high number of factors. First we need to determine the visible set and the performance of this step is dependent on the geometry of the scene. The Beam Tree is expensive to construct making this approach impossible for highly tessellated scenes. Once the visible set is found it is rendered by the GPU. The number of vertices rendered is dependent on the total number of pyramids. If the number of pyramids that a given ray can see are fewer than the number of sampling planes the same ray would see this method will use a lower amount of fill-rate. The work per fragment work however could be higher if $E(z)$ is expensive to calculate. It is not possible to reduce the work of this algorithm when the light source is far away, it might even be better to switch to a sampling based approach. A drawback is that the light cone might seem too smooth since it is hard to introduce a noise factor. Optimizations that limits the number of

[3]Notice that z_{frag} is not the z-value written to the z-buffer but rather the length from the camera to the position of the point being rasterized.

polygons in the visible set could be used to reduce the amount of work needed for shadow casters not visible to the observer (not all shadow casters need to be included in the visible set since they are not visible themselves and can not cast shadows onto anything visible) [11].

Depending on the precision of the destination buffer we are rendering to we might have to render each pyramid individually, adding the back side contribution and then subtracting the front side contribution. This is to avoid the problem when the sum of all backside contributions are larger than the maximum value in that buffer. If the buffer can not handle negative values then care must be taken so we do not underflow. The method is also sensitive to very bright lights since the back side contributions can again overflow. Overflowing is in usual rendering not a problem but here we need to subtract from the value so clamping to a maximum value is not possible. Since blending cannot be set on a per triangle/fragment basis on the GPU we need so submit smaller batches. By defining a value *numOverflow* we can use the intensity numOverflow^{-1} instead of 1, that is we divide all contributions by numOverflow^{-1}. This allow us to overflow the buffer a certain number of times, but at the same time we are reducing the precision of the buffer. If we use a destination buffer with 8 bits per color channel then a value of $\text{numOverflow} = 8$ will only give us 32 gray tones. This might appear as banding in the image, especially if many thin pyramids are present. The same problem applies to the sampling-based approach but there we never have to subtract so clamping to a maximum value is a valid option.

Chapter 4

Beam Tree

In this chapter, we describe a special subclass of the Solid BSP Tree called the Beam Tree that can be used to determine the visible set from a point **L**. The term was coined by Abrash [18]. It is a degenerate variant of the Solid BSP Tree which is often used for solid modeling [19]. An alternate way to construct the Beam Tree is through Beam Tracing introduced by Heckbert [20]. It should be possible even for complex scenes with the new methods by Overbeck et al. [21]. Pseudo code for the beam tree can be found in appendix B.

4.1 Nodes and Leafs

We begin by introducing the building blocks of the Beam Tree.

The Beam Tree is a binary tree that recursively partitions the space R^3 using splitting planes. The algorithm start with the entire space. Each node in the tree then represents a division of the parent space into two parts, a front side and a back side. Each node holds the splitting plane and references to two child nodes describing the space in front and behind respectively. The two children spaces can then be further subdivided, resulting in a tree structure. Since each partition splits a convex volume by a plane, the resulting volumes will also be convex. Thus each node represents a convex subspace of the original space.

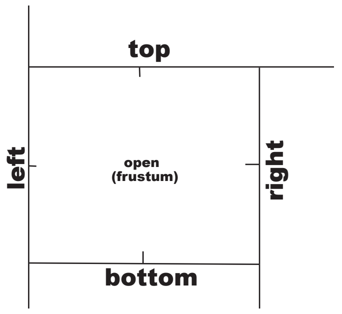
The leafs (the nodes without children) represent convex subspaces of R^3. They are classified as being either open or closed. An open leaf means that the space represented by that leaf is considered to be unoccupied. A solid leaf is considered to be occupied.

In a Beam Tree all the splitting planes passes through a single point **L**. This makes all the convex subspaces into beams, or pyramids. The pyramids only have walls and are infinitely long.

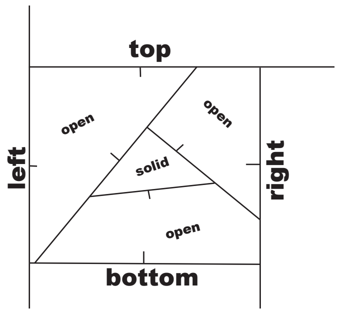
Initially the tree contain one open leaf, representing a beam as large as the frustum of the camera or the light source[1]. We can see a Beam Tree in Figure

[1]Even if a light source might have a circular frustum (a cone) we use the four bounding

4.1. While this figure is in two dimensions the Beam Tree itself is in three dimension with each line in the figure being a plane passing through the point **L**.



(a) Initial Tree.



(b) One solid beam inserted.

Figure 4.1: A Beam Tree as seen from the light source.

We can see the Beam Tree from Figure 4.1(a) in tree form in Figure 4.2.

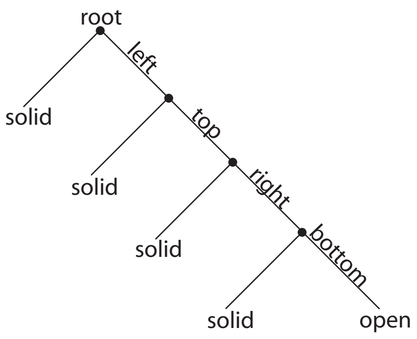


Figure 4.2: Initial Beam Tree in Tree Form.

4.2 Insertion

When we insert a polygon we first construct that which we want, the portions of that polygon which are part of the visible set. For each such portion then we take its polygon and insert the corresponding beam into the beam tree.

To insert, we traverse the tree with the primitive, splitting it against the splitting planes. Convex portions of the original primitive will end up in an open or a closed leaf. If it ends up in a closed leaf we ignore it since that space is already represented. If it ends up in open space we insert a new beam, marking that space as being occupied. We also add the portion of the polygon that created the beam to the visible set (along with the plane of that polygon).

walls of that cone as our frustum.

In order to get not only the correct beams but also the correct visible set, we need to insert the polygons of the scene in a front-to-back order (as seen from the light), and they must not intersect each other. If they do intersect each other this must be resolved before constructing the Beam Tree.

In our application we use the Beam Tree to obtain the visible set from a light source. For a light source, we are not interested in the full visible set but rather only the portion that is in front of the far plane of the light source. Thus we clip all polygons to this plane prior to insertion. After the full frustum is created, we can traverse the tree with a polygon with the shape of the far clipping plane. The portions that ends up in open leafs are fully visible and are added to the visible along with the far plane as its plane.

4.3 Example: Efficient Point Test

Program 1 Beam Tree point classification.

```
def isInside(node, point):
  while (isLeaf(node)==false)
    if (node.plane.isInFront(point)) node=node.frontNode;
    else node=node.backNode;
  return isLeafSolid(node)
```

While constructing the tree can be expensive it can efficiently be used to answer queries. To determine if a point is in the solid subspaces, we simply traverse the tree and choose child depending on if the points lies in front or behind the splitting plane. Once we end up in a leaf, we classify the points depending on if that leaf is solid or not.

Points lying exactly on a splitting plane traverses the front node. Once it reaches the leaf it will be classified as being in open space if it is exactly on a boundary between solid and open space. However, unless exact arithmetic is used, due to precision issues, it will be randomly classified. The alleviate this splitting planes are given a thickness ϵ. Program 1 would then be rewritten as:

Program 2 Beam Tree point classification with thick splitting planes.

```
def isInside(node, point):
  while (isLeaf(node)==false)
    distance=node.plane.calculateDistance(point)
    if (distance<-epsilon) node=node.backNode
    else node=node.frontNode
  return isLeafSolid(node)
```

Now the point will always be classified as being in front of the plane if the distance from the plane is less than ϵ. For the same reason the splitting planes are given a thickness during construction of the (S)BSP Tree as well. Tolerances such as ϵ should always be carefully chosen [25, 26]. If the world is modeled in SI units then $\epsilon = 1mm$ might be reasonable, unless the user can zoom in on small things.

4.4 Partitioning a polygon

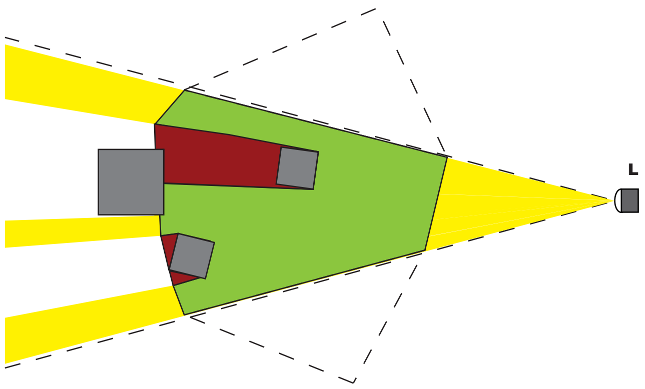


Figure 4.3: Partitioning a polygon into green parts that are in front and red parts that are behind.

If the Beam Tree is created with the position of the light source as **L** then the portion in front of the visible set is the lit parts of the polygon and the portions behind the visible set is the occluded parts. One operation we then might want to perform with a Beam Tree is to partition a polygon into two parts, one that is in front of the visible set and one that is behind it. The situation is depicted in Figure 4.3 with the green regions being the parts that are in front of the visible set and the red regions being the parts that are behind the visible set.

We can first partition our polygon into small parts, one for each leaf in the Beam Tree that the polygon passes through. We do this by traversing the tree. We take our initial polygon and clip it with the splitting planes of the nodes. If a part ends up in an open leaf then that whole part is in front of the visible set. For each such part ending up in a solid leaf, we can split it with the plane of the polygon that was used when creating that leaf. This will yield two parts, one that is in front and one that is that behind the visible set.

Chapter 5

New Hybrid Methods

We have seen how the two method developed in chapter 3 can be used to visualize the full light cone. However, they both have their drawbacks. The first one requires a lot of fill-rate, while the second can only handle scenes with few faces or the CPU work from constructing the Beam Tree is too demanding. Here, we first review the use of bounding volumes and how they relate to volumetric lighting. We then develop two methods to take advantage of the bounding volumes to reduce the fill-rate requirements of the sampling method.

5.1 Bounding Volumes

Bounding volumes can be used to perform conservative but efficient evaluations of properties of objects. An *outer bounding volume* fully encloses an object. If we want to determine if an object is inside the influence of a spotlight cone we could first determine if the outer bounding volume is inside the cone. If it is not, we know that the object receives no light. If, however, the outer bounding volume is partially inside the cone, the actual object might still be receive no light. This is what makes the test conservative and not exact.

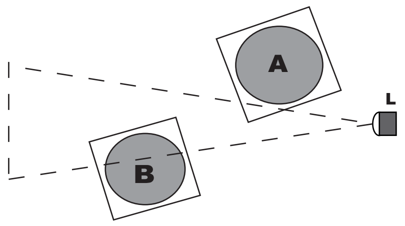


Figure 5.1: Outer bounding volumes can be used to conservatively test if an object is inside a cone.

Furthermore, a *inner bounding volume* can be used. It is fully enclosed by an object. It can be used if we want to find out if an object A receives any light from a light source when another object B is potentially blocking the light. If the outer bounding volume of A is fully inside the volume behind the inner bounding volume of B as seen from the light source, no light is received by A.

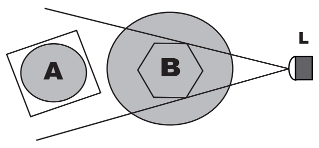


Figure 5.2: The use of inner bounding volumes. Object B blocks the light from arriving at A.

It is also possible to have several inner bounding volumes for a single object. Usually bounding volumes are assumed to be convex to further speed up the test carried out with them.

5.2 Beam Tree of Bounding Volumes

For both our methods, we construct a Beam Tree of the polygons of the bounding volumes instead of the actual scene geometry. This is less expensive since the number of triangles in the bounding volumes is much smaller than number of triangles in the full scene. We first construct a Beam Tree using the polygons of the *outer* bounding volumes. This gives us the visible set of the outer bounding volumes (and the visible polygons of the backplate).

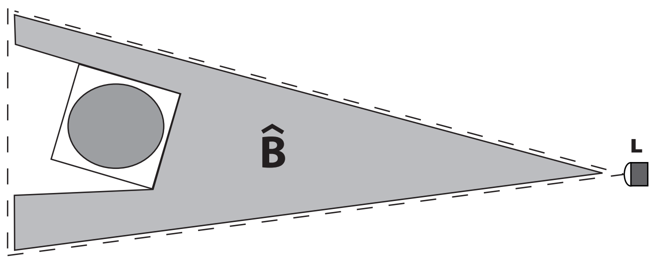


Figure 5.3: By using the visible set of the outer bounding volumes we get $\hat{\mathbf{B}}$.

Constructing pyramids using this visible set gives us a subset $\hat{\mathbf{B}}$ of the volume $\mathbf{B}$ introduced in Section 3.2.1. If the outer bounding volume closely resembles the actual object the difference of $\hat{\mathbf{B}}$ and $\mathbf{B}$ will be small.

At this point we could choose to use the volume-based approach to visualize the influence inside the volume $\hat{\mathbf{B}}$. If we have sufficiently good bounding volumes this approach might be good enough, especially if the light source is far away from the viewer.

5.2.1 Partitioning a beam

Using the Beam Tree of bounding volumes, we can divide a beam into three types of regions as seen in Figure 5.4. First, we have the fully lit yellow region in front of the visible set. Then we have the uncertain gray region where the status of being lit or unlit depends on the complex object that is enclosed in the outer bounding volume. The third region in black is unlit since the inner bounding volume tells us that the gray object is blocking the light. Some beams might not have a fully unlit region (because the beam does not see a inner bounding volume). Some beams are also fully lit since the uncertain region starts at the backplate of the light source influence volume.

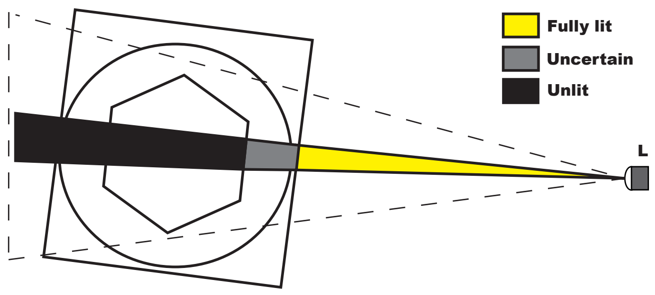
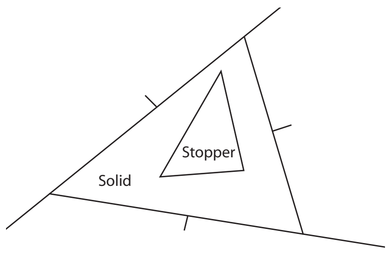


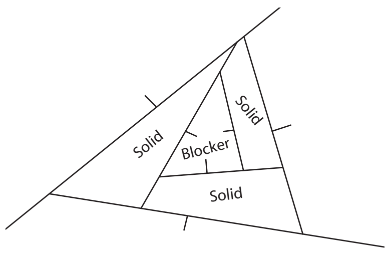
Figure 5.4: We divide a beam into three types of regions.

After the construction of the Beam Tree, we have many beams, one for each polygon in the visible set. In order to partition the beams as in Figure 5.4 we need the information from the inner bounding volume in addition to the information from the outer bounding volumes.

We want to partition the beams using planes. Let us assume that we have a Beam Tree constructed from the surfaces of the inner bounding volumes as well. If these two Beam Trees had the exact same leafs we could match the beams by matching the leafs. We would then have two planes for each beam; one from each leaf. Taken together they would form the partitioning planes between the regions of that beam. The plane from the leaf in original Beam Tree would be the partitioning plane between fully lit and uncertain. The plane from the leaf from the Beam Tree of the inner bounding volumes would be the partitioning plane between uncertain and fully unlit. In reality however the two Beam Trees will have different leafs, and we would have to subdivide the leafs of both trees until perfectly matching leafs could be formed.

We have chosen to construct only one Beam Tree, the one composed of the surfaces from the outer bounding volumes. We then augment this tree with the information from the inner polygons. We do this like in Figure 5.5. We start by clipping the polygons from the inner bounding volumes and keep the parts that end up in solid leafs (that represents a beam that does not see the backplate). We call these clipped polygons *stopper polygons.* The situation is depicted in Figure 5.5(a). We then subdivide the leaf and adds new leafs. All but one of these are solid just as the original leaf, but the last is a *blocker leaf.* The blocker leaf is assigned the plane of the stopper polygon, called the *stopper plane*.





(a) A solid leaf. (b) Result after augmenting with inner bounding volume.

Figure 5.5: Subdividing a solid leaf.

Now we have three types of leafs; open, solid and blocker. Open leafs represents beams that only intersect the backplate. Solid leafs represents beams that intersects an outer bounding volume, but not an inner. A blocker leaf represents a beam that intersects both outer and inner bounding volumes. This gives us a modified partitioning algorithm. If we have a polygon that we want to partition we clip it (as in Section 4.4) so we get the portion that ends up in leafs. We can then use the partitioning planes of the corresponding beam to select only the part that is fully lit or uncertain. Parts that are fully unlit can be discarded all together.

5.3 Method 1: Sampling with Aggressive Clipping

We construct a Beam Tree from the bounding volumes of the objects as described in the previous sections. We then use the sampling method as described in Section 3.1. However, we partition the sampling polygons using the method in Section 4.4 so that portions that are fully lit and portions that are partially lit are separated. Portions that are unlit are discarded. The two lists of polygons are then rendered with different shaders, one that performs shadow lookup using the shadow map and one that does not. The real gain is from the reduction in fragment rasterized due to discarding fragments that are unlit. This is quite close to the suggestion for future improvements made by Mitchell [9].

A good side effect of using this method is that we can utilize all the sampling tricks developed by Dobashi et. al [6] and Mithcell [9]. We introduce no additional errors or restrictions compared to the original method.

5.4 Method 2: Hybrid Method

We construct a Beam Tree from the outer bounding volumes of the objects as described in the previous section. In order to visualize the fully lit portions we use the integration-based method described in Section 3.2.

The Beam Tree is then augmented with the information from the inner bounding volumes as described in Section 5.2.1. The sampling method (Section 3.1) is then used to visualize the uncertain portions of the beams. The augmented Beam Tree gives us the extra information needed to clip away everything but the uncertain parts of the sampling planes.

Taken together we get the full contribution and a correct result. This reduces fill-rate drastically when the bounding volumes are placed such that a majority of the illuminated volume is either fully lit or fully unlit. When most of the region is uncertain we need to fully rely on sampling and since we can not use alias hiding techniques (such as noise) we might need more virtual planes compared to using a plain sampling based approach.

In lack of a good estimate of $E(0, Z)$ we have not been able to obtain the same visual results as the method that fully rely on sampling. In testing we have used the simplified lighting model from Section 2.5 to get an estimate of how beneficial this would be performance wise in comparison to the other method.

Sampling does not yield the exact solution and if the number of virtual planes is too small there will be visible differences between the two solutions. Also if the precision of the destination buffer is too low small contributions from the sampling might be ignored, yielding an incorrect sum. This is handled better in the integration-based approach since it operates on longer intervals. This is another source of difference between the solutions.

Chapter 6

Results

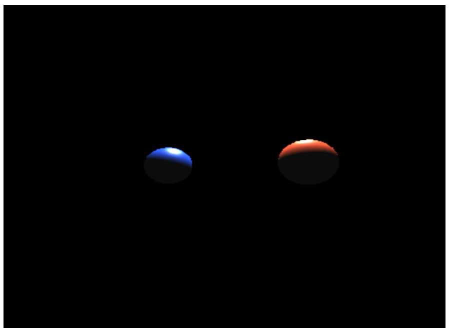
We have implemented the original sampling based method [6][1] and the two new hybrid methods for comparison. We have not implemented the methods by Mitchell [9] to reduce banding due to a low number of sampling planes since those would not work together with the second hybrid method. We use variance shadow mapping [23] as our shadow mapping solution. It is a fairly cheap shadow method compared to PCF-filtering [24]. This reduces the difference in rendering sampling planes with and without shadow mapping, something that should be remembered when we compare the original method with our first hybrid method.

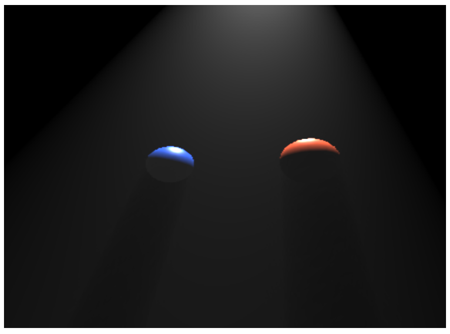
A restriction of the implementation is that we do not support objects whose outer bounding boxes overlap. This is a serious limitation in this particular implementation but it can easily be lifted by building a BSP-tree for the outer bounding volumes and then inserting them front-to-back into the Beam Tree (the inner bounding volumes are convex and if they can be assumed not to intersect they can be added front-to-back into the Beam Tree after a simple sort on their center point).

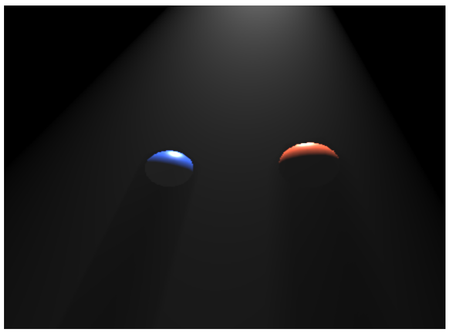
We want the measure performance and quality of the two methods. Our quality concerns are both inside the two different regions and at the boundary between them. Speed wise we have three principal cases that are of interest. When the scene contains few objects with simple bounding volumes the beam tree will be uncomplicated. When the number of objects goes up so will the complexity of the beam tree, making both the construction and the visualization of the pyramids slower. The performance of the sampling depends solely on the drawn area of the sampling planes. The number of virtual planes goes up as the light source gets more parallel to the viewing direction. The other performance/quality factor here is the spacing factor dz between the virtual planes.

The volume of the regions rendered using the different methods depends not only on where the objects are located relative the light source but also on how well the objects are described by the inner and outer bounding volumes. If we assume all bounding volumes to be convex then highly non-convex objects will

[1]We have however chosen not to subdivide the sampling planes into a grid and calculate slowly varying information per vertex. This is the same choice as was done by Mitchell [9].


(a) Without light-shafts.


(b) With light-shafts.


(c) Contribution from Beam Tree.

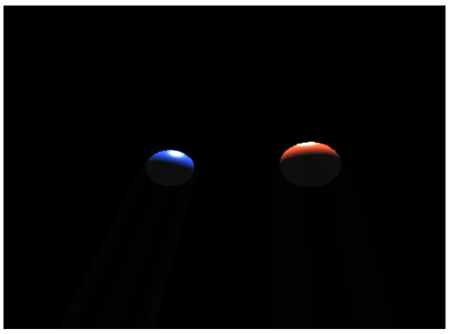

(d) Contribution from Sampling.

Figure 6.1: Scene: Toplight.

be poorly bound. This will increase the volume rendered by the slower sampling method.

To evaluate performance, we have constructed three different scenes:

1. Parallel: Parallel light source
2. Toplight: Orthogonal light source
3. Multiple: Many objects

Toplight can be seen in Figure 6.1. *Parallel* and *Multiple* can be seen in Figure 6.2.

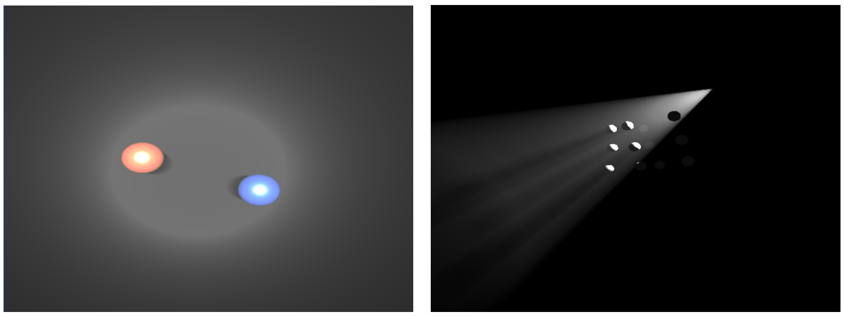

(a) Scene: Parallel. (b) Scene: Multiple.

Figure 6.2: Test scenes.

We will take performance measures using different values for dz. Notice that the Beam Tree column only includes a partial solution so the results in that column will be lower. Also since the primary bottleneck is the fill rate required by both the algorithms, especially in simpler scenes, we measure at different resolutions.

We have not made any special benchmark to tests the image quality. This is primarily because not enough time has been spent on optimizing the quality of the sampling method. Hence, this is left for future work. Since the sampling method using many small contributions added together truncation is a severe problem, especially with small values for dz. But with larger values of dz there are more pronounced boundaries between the sampling regions and the beam tree regions. Another factor here are that since they are rendered into the same buffer, we must deal with both overflow and truncation issues. A solution here would be to render the contributions into different buffers, allowing different quality settings.

dz	Scene	Size	Beam Tree	Sampling	Method 1	Method 2
0.5	Parallel	64	850	924	204	174
	(71)	256	311	325	204	180
		512	102	100	100	146
		1024	28	34	27	44
	Toplight	64	879	985	563	515
	(17)	256	342	567	339	515
		512	117	240	116	269
		1024	32	73	32	85
	Multiple	64	79	440	223	206
	(47)	256	79	440	224	200
		512	79	257	175	200
		1024	57	80	54	89
0.1	Parallel	64	850	556	53	45
	(356)	256	311	80	50	45
		512	102	24	53	21
		1024	28	7	27	6
	Toplight	64	879	719	288	229
	(88)	256	342	172	288	150
		512	117	50	111	47
		1024	32	13	31	13
	Multiple	64	79	407	100	83
	(239)	256	79	164	100	83
		512	79	64	102	50
		1024	57	13	47	15

Table 6.1: Performance for the different methods (in frames per second). Number of parenthesis is number of virtual planes used for sampling.

Chapter 7

Conclusions

We can see that our two different methods perform better than the original sampling method for some setups. Furthermore, the portions that are rendered by the Beam Tree approach suffer far less from banding, yielding smoother looking light shafts. Some applications that have a light setup similar to those cases where the new methods perform well could use them. It is, however, not simple to implement the methods. If any of the methods should become popular it must be made easier to implement.

The Beam Tree based methods are most benficial for simple scenes rendered in high resolutions. In order to handle scenes with more objects the Beam Tree implementation must be better than the one we implemented. We do not feel that we have made the fastest solution possible. On the other our implementation for the reference sampling method is not perfect either.

A disadvantage of the hybrid method with the Beam Tree visualization is the alias-reducing methods available for the sampling based method. That is while we can sometimes render the regions that are fully lit faster, the regions that are in doubt and thus clipped might be slower than the corresponding regions would be in a pure sampling based method. This is because we might have to increase the number of virtual planes a lot for the Beam Tree solution and the sampling solution to match, and since we cannot use anti-aliasing techniques/hacks, such a noise, since they are not present in the Beam Tree solution.

Another negative thing with the Beam Tree visualization is that we must render only a few pyramids at the time or we will overflow our buffer. Were it possible to use additive blending for both front sides and back sides but giving the front side contribution a negative color then we could render all the pyramids at once using a vertex buffer. Negative colors are however not a part of the OpenGL standard (although some cards seems to support it).

Although it is hard to compare methods that have many different quality parameters, we can see that our methods is not always better, and sometimes worse. This could be due to a poor implementation, but it could also very well be that the simplicity of the sampling algorithm; in some cases the Beam Tree does not add anything.

We see very little benefit from using *Method 1.* This might be due to our choice of VSM as our shadow mapping algorithm; it is much cheaper than percentage closest filtering so an additional shadow map lookup is not that expensive[1].

[1]VSM only does one texture lookup per evaluation and supports mip-mapping of the shadow map.

Chapter 8

Future Work

It should be possible to insert polygons into a Beam Tree in a random order (not front-to-back as seen from the light). This would allow overlapping outer bounding boxes. The trick would be to check when a polygon is inserted into a solid leaf. It should then be clipped by the plane of that leaf. If a portion of it is in front we subdivide the original leaf and fix things up. It might hamper the performance/quality of the tree a lot but might be worth it. The alternative is to first build a BSP-tree and then traverse the resulting BSP-tree front-to-back.

Instead of partitioning polygons to the Beam Tree, we can just generate them in a geometry shader. Given a pyramid and two planes (where uncertainty begins and stops) in space, we want to generate the intersection with the virtual planes $z = n * dz$. In some cases, this corresponds to the projection of either one of the two planes onto the planes $z = n*dz$. Sometimes clipping needs to be performed. Either way the result will be a convex planar polygon. This algorithm should be possible to perform in a geometry shader.

It might also be possible to use the Beam Tree alone, and not do any sampling at all if the bounding volumes are good enough. An adaptive version of the sceen could be used, giving adaptive control of the complexity of the Beam Tree and the number of pyramids that needs to be visualized.

Acknowledgments

Thanks to Edward Kmett for discussion on the Beam Tree. Thanks to Albert for GUI-work. Thanks to Gustaf, David and Stefan for tips and suggestions. Thanks to Fredrik for 3d models. Thanks to Linnéa for patience.

References

[1] Henrik Wann Jensen and Per H. Christensen, "Efficient simulation of light transport in scences with participating media using photon maps" SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques. (1998)

[2] Henrik Wann Jensen , "Realistic Image Synthesis Using Photon Mapping". AK Peters, Ltd.; 1st edition (July 15, 2001).

[3] Kun Zhou, Qiming Hou, MinMin Gong, John Snyder, Baining Guo, Heung-Yeung Shum "Fogshop: Real-Time Design and Rendering of Inhomogeneous, Single-Scattering Media" Microsoft Research Technical Report, MSR-TR-2007-37, March 2007.

[4] Y. Zhu, G.S. Owen, F. Liu, and A. Aquilio, "GPU-based Volumetric Lighting Simulation".

[5] Eva Cerezo, Frederic Perez, Xavier Pueyo, Francisco J. Seron, François X. Sillion, "A Survey on Participating Media Rendering Techniques" The Visual Computer (2005).
`http://artis.imag.fr/Publications/2005/CPPSS05/`

[6] Y.Dobashi, T. Yamamoto , T.Nishita, "Interactive Rendering Method for Displaying Shafts of Light", Proc. Pacific Graphics 2000, pp. 31-37 (2000).

[7] Y. Dobashi, T. Nishita , T. Yamamoto , "Interactive Rendering of Atmospheric Scattering Effects Using Graphics Hardware," Proc. Graphics Hardware 2002, pp. 99-108 (2002).

[8] Bo Sun, Ravi Ramamoorthi, Srinivasa G. Narasimhan, Shree K. Nayar, "A practical analytic single scattering model for real time rendering", Proc. ACM SIGGRAPH 2005, pp 1040 - 1049.

[9] Jason Mitchell "Light Shafts. Rendering Shadows in Participating Media", ATI Research Inc. Presented at GDC 2004.

[10] Charles Boyd and Dan Baker, "Volumetric Rendering in Realtime" Gamasutra (2001)
`http://www.gamasutra.com/features/20011003/boyd_pfv.htm`

[11] Luke Hutchinson, "Shadow Caster Volumes For The Culling Of Potential Shadow Casters", published online 2006. `http://www.gamedev.net/reference/articles/article2330.asp`

[12] T. Nishita and E. Nakamae, "A shading model for atmospheric scattering considering luminous intensity distribution of light sources," Computer Graphics, Vol.21, No.3, 1987-7, pp.303-310.

[13] T. Nishita,Y.Dobashi, E.Nakamae,"Display of Clouds Taking into Account Multiple Anisotropic Scattering and Sky Light," Proc. SIGGRAPH'96, 1996-8, pp.379-386.

[14] Mark J. Harris and Anselmo Lastra, "Real-Time Cloud Rendering". Computer Graphics Forum (Eurographics 2001 Proceedings), 20(3):76-84, September 2001.

[15] Mark J. Harris "Real-Time Cloud Rendering for Games". Proceedings of Game Developers Conference 2002. March 2002.

[16] Mark J. Harris, William V. Baxter III, Thorsten Scheuermann, Anselmo Lastra. "Simulation of Cloud Dynamics on Graphics Hardware." Proceedings of Graphics Hardware 2003.

[17] Michael McCool, "Shadow Volume Reconstruction from Depth Maps". ACM Transactions on Graphics, 2000.

[18] Michael Abrash, "Michael Abrash's Graphics Programming Black Book Special Edition". 1997.

[19] Bruce Naylor, John Amanatides, and William Thibault, "Merging BSP trees yields polyhedral set operations". 1990.

[20] Heckbert P. S., Hanrahan P, "Beam tracing polygonal objects". SIGGRAPH 84 (1984) pp. 119-127.

[21] Ryan Overbeck, Ravi Ramamoorthi, and William R. Mark, "A Real-time Beam Tracer with Application to Exact Soft Shadows" Eurographics Symposium on Rendering, 2007.

[22] David Eberly, "Intersection of a Line and a Cone" `http://www.geometrictools.com/Documentation/IntersectionLineCone.pdf`

[23] William Donnelly, Andrew Lauritzen, "Variance Shadow Maps". I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games `http://www.punkuser.net/vsm/vsm_paper.pdf`

[24] Reeves, W., Salesin, D., and Cook, R. 1987. "Rendering antialiased shadows with depth maps." In Proc. SIGGRAPH, vol. 21, 283Ũ291.

[25] Schirra, Stefan. "Robustness and precision issues in geometric computation." Research Report MPI-I-98-004, Max Planck Institute for Computer Science, 1998.

[26] Hoffmann, Christoph. "Robustness in Geometric Computations." JCISE 1, 2001, pp. 143-156.

[27] Williams, L. "Casting curved shadows on curved surfaces." Computer Graphics (Proceedings of SIGGRAPH 78), vol. 12, 270Ũ274.

Appendix A

Conservative Phase Function

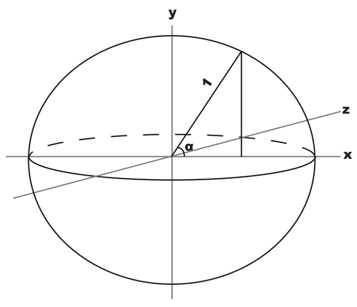


Figure A.1: Rewriting Integral over Sphere

For a phase function that only depends on the angle α, we can rewrite the conservative condition as:

$$\int_{-1}^{1} p(\alpha) 2\pi h dx = 2\pi \int_{-1}^{1} p(\cos^{-1} x)\sqrt{1-x^2} dx = 1. \tag{A.1}$$

For the phase function in (2.4)

$$p(\alpha) = K\left(1 + 9\cos^{16}\frac{\alpha}{2}\right) = K\left(1 + 9\left(\frac{1+\cos\alpha}{2}\right)^8\right),$$

this means that

$$2\pi \int_{-1}^{1} p(\cos^{-1} x)\sqrt{1-x^2} dx = K\frac{54647\pi^2}{2^{15}},$$

which gives us the normalization factor $K \approx 16.45948^{-1}$.

Appendix B

Beam Tree

Program 3 Pseudo code for Inserting a polygon in a Beam Tree

```
insert(root, polygon.vertices)

def insert(node, vertices):
  if len(vertices)==0: return

  if node.mode==SOLID: // Fully occluded, ignore
    return

  if node.mode==OPEN:  // Fully visible, insert
    attachPolygon(node, vertices)
    return

  [backVert,frontVert]=split(node.plane, vertices)

  insert(node.backNode,  backVert)
  insert(node.frontNode, frontVert)
```

Program 4 Pseudo code for creating new nodes for a newly found solid leaf in a Beam Tree

```
def attachPolygon(node, vertices):
  for each edge v0,v1 in vertices:
    prevNode=node

    node.frontNode=new Node(OPEN)
    node.backNode =new Node(SOLID)

    node.mode=SPLIT
    node.plane.construct(L, v0, v1)

    node=n.back

  prevNode.solidPlane=Plane(vertices)
```

Program 5 Pseudo code for partition a polygon with a Beam Tree

```
partition(root, polygon.vertices)

def partition(node, vertices):
  if len(vertices)==0: return

  if node.mode==SOLID: // We ended up in fully occluded space
    [backVert,frontVert]=split(node.solidPlane, vertices)

    behindPolygons.add(Polygon(backVert))
    frontPolygons.add(Polygon(frontVert))
    return

  if node.mode==OPEN:  // Fully visible, insert
    frontPolygons.add(Polygon(vertices))
    return

  [backVert,frontVert]=split(node.plane, vertices)

  partition(node.backNode,  backVert)
  partition(node.frontNode, frontVert)
```