Real-time Rendering of Refractive Objects in Participating Media

Masterarbeit

der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Michael Pfeuti

2009

Leiter der Arbeit:
Prof. Dr. Matthias Zwicker
Institut für Informatik und angewandte Mathematik

Abstract

In this master's thesis we propose a rendering pipeline that is capable of rendering images with refractive objects in participating media in real-time. Real-time refresh rates are achieved because most parts of the pipeline are executed on the graphics processing unit. The pipeline computes the lighting by photon tracing and storing the illumination data in a 3D array. Our pipeline performs an importance sampling in order to use as few photons as possible. Due to this adaptive photon sampling a specialized filtering technique needs to be applied. In comparison with similar rendering pipelines, we were able to reduce the number of required photon by more than 50%. Yet, because of hardware limitations on current GPUs, our pipeline is slower in common environments.

Contents

Chapter 1

Introduction

A large research area in computer graphics is finding techniques to synthesize images physically correctly. In addition, the computation should be fast. Depending on the application area of these techniques, the available time span to compute an image is only a few milliseconds. This allows the construction of interactive systems such as computer games.

Many optical effects are hard to compute efficiently. One of the most prominent example is global illumination. Global illumination is the illumination of a scene when all possible light paths are considered and is, therefore, physically correct. Global illumination is often split into direct and indirect illumination. Direct illumination is when we only consider light paths than are not reflected. This leads to images in which only surfaces are illuminated that are visible from a light source. Indirect illumination, on the other hand, is the light that reached an area via a reflection of an object in the scene. The computation of global illumination is not trivial because there are infinitely many paths to consider. Hence, it is necessary to find a way to obtain a close approximation while computing as few paths as possible.

Another difficult behavior of light is when it passes through so called participating media. Participating media is a material that consists of many small particles with which the light might interact. An example would be a cloud or fog. In both cases the small particles are tiny water droplets. When light hits such a droplet the light might change direction. This process is called scattering. Assuming the particles are for instance smut then the light might be absorbed by the particle. This process is called absorption. In addition, there are also particles that emit light. Since participating media consists of many such particles, light might interact with these many times before it leaves the medium. Therefore, computing the physically correct illumination inside participating media is not a trivial task.

Global illumination and light transport in participating media can be formulated mathematically. These formulations can be used to compute images with global illumination and participating media. However, the computation may take a long time. As a result, to compute these images quickly it is necessary to use the processors of a computer as efficiently as possible. For the computation of images, the graphics processor is a suitable processor. With the graphics processor it is possible to compute images with global illumination and participating media in a fraction of a second where the CPU requires several seconds.

1.1 Goal and Achievements

The goal of this master's project is to compute images of scenes with refractive objects in participating media in real-time. An example, which was computed in 380 ms, is shown in Figure 1.1. The glass sphere resides in a foggy surrounding. Thus, we can see how the light is refracted and, therefore, focused by the glass sphere.

Figure 1.1: A glass sphere focusing light in participating media.

To reach our goal we implemented a modified version of the pipeline proposed by Sun et al. [SZS+08] in a first part of the master's project. It is a pipeline based on photon tracing and is executed entirely on the graphics card. This allows rendering images with refractive objects in participating media in real-time. Sun et al. propose to use many photons to obtain noise free images. In the second part of the master's project we attempted to reduce the photon count with an adaptive photon sampling. This resulted in the adaptive rendering pipeline. We hoped that with an adaptive sampling we could reduce the rendering time. In fact, we were able to reduce the photon count by more than 50%. Unfortunately, this did not lead to the expected performance increase. The fact that we use an adaptive photon sampling forces us to perform additional work. In most scenes, this additional work requires more time than we could save by using less photons. Our new adaptive pipeline yields a better performance only when several hundred thousand photons are used.

1.2 Related Work

The mathematical formulation for the light transport in participating media that is used in computer graphics has its origins in physics. It is a derivation from the physical process called thermal radiation heat transfer [SH01]. In 1982 James Blinn was one of the pioneers who introduced light scattering to computer graphics [Bli82]. Many algorithms today include all effects in participating media, namely absorption, emission and scattering. Jensen and

Christensen [JC98] propose a photon mapping approach, which is also the basis for many recent rendering algorithms. Eventually, due to the high performance processors available today it became even possible to render images with participating media in real-time.

In 2008, Sun et al. [SZS+08] proposed a new technique for rendering refractive object in participating media in real-time. The main idea is to represent a scene and its illumination in 3D arrays. These 3D arrays are positioned over the participating media and the refractive object. Each array element represents a small part of the scene. We call these 3D array voxel grids. A voxel is a volume pixel, which is how we use the elements of a 3D array. Since we also represent the model in a voxel grid, it is required to transform a description of a refractive object into a representation. Sun et al. follow the algorithm of Crane et al. [CLT07]. In 2008, Eisemann and Décoret presented a new technique that allows this voxelization process to be executed quickly [ED08]. This is to date the fastest voxelization technique. Sun et al. did not use this algorithm because Eisemann and Décoret published their work also in 2008.

In order to compute the illumination in the voxel grid Sun et al. make use of photon tracing. They shoot many photons into the grid in order to obtain a smooth image. There are, however, ways to reduce the photon count which should lead to faster rendering. For example, in successive frames of an animation most of the scene remains the same. As a result, the illumination hardly changes. Dmitriev et al. [DBMS02] suggested probing the scene for changes with so called pilot photons. Only when a change is discovered has this region to be updated. We change the pipeline proposed by Sun et al. such that we adopt this idea of using pilot photons to determine how many photons are needed for which regions. This leads to an adaptive photon sampling similar to what Wyman and Nichols proposed [WN09]. In contrast to the idea of Dmitriev et al. we do not rely on successive frames. Therefore, our technique can be applied to render a single image.

1.3 Outline

This master's thesis is structured in four chapters. The first chapter supplies the reader with all the necessary theoretical background to understand the physical and computational issues that arise when building a system for rendering participating media with refractive objects in real-time. The second chapter gives in-depth information on a possible implementation for such a system. We implemented two rendering pipelines; the reference pipeline which is based on the pipeline proposed by Sun et al. and our new adaptive pipeline. In the third chapter we analyze and compare the two pipelines. We take a closer look at the performance and quality differences. Finally, the last chapter gives the conclusion and an outlook to possible future work.

1.4 Acknowledgments

I would like to thank Prof. Dr. Matthias Zwicker for offering me this topic as a master's thesis and for providing me always with new ideas. Furthermore, I would like to thank Claude Knaus for his support and very helpful advises and consulting hours without which this master's thesis would not have been possible. Lastly, I would like to thank Brigitte Hulliger for her support and valuable feedback on my thesis.

Chapter 2

Theory

In this chapter we present the theoretical background that is required for rendering refractive objects in participating media. We start by explaining how light interacts with participating media. For this, we present an integral equation that captures the effects that may occur when light passes through participating media. In order to get a better understanding of these effects and, therefore, also of the integral equation itself, we analyze each of the effects separately. Eventually, we assemble the integral equation, called volume rendering equation, from the separate effects.

In Section 2.2 is explained how the volume rendering equation can be solved with Monte-Carlo integration. We first describe Monte-Carlo integration in general and why it is a suitable tool for solving the volume rendering equation. We show that this technique provides us with an approximation and relies solely on sampling a given function. Depending on the sampling strategy that we apply, we obtain a better or worse approximation. Therefore, we additionally analyze several sampling strategies.

One possibility to apply Monte-Carlo integration is to use photon mapping [JC95]. As the name suggests, we compute how photons move through space and interact with the environment. Photon mapping can be used in basically any environment. The classic case is where we do not have participating media. Therefore, we first explain how photon mapping can be used in this case. Then, we look at how photon mapping can be used with participating media.

An important effect in optics is refraction. Light that passes through refractive object can produce caustics, i.e. spots where light is focused and is, therefore, brighter. We are going to investigate how refraction can be formulated mathematically and how we can include refraction in photon mapping.

Finally, a brief presentation is given on programming on recent graphics processing units (in the following called GPUs). Here we restrict ourselves to GPUs by NVIDIA because we use CUDA (Compute Unified Device Architecture) by NVIDIA. CUDA is a technology that allows programmers to execute C code directly on the graphics card.

In this thesis we use the notation from Table 2.1.

2.1 Participating Media

When formulating light transport in computer graphics, we usually make some simplifying assumptions because we are especially interested in formulations which allow a fast compu-

$x, x', \ldots$	Locations in the environment
$\omega, \omega_i, \omega_o, \ldots$	Directions
$L_i(x, \omega)$	Incident radiance at location x from direction ω
$L_o(x, \omega), L_{ve}(x, \omega), L_e(x, \omega), L(x, \omega)$	Excitant radiance at location x in direction ω
Ω	Upper hemisphere of a surface
S	Entire sphere
$\vec{n}$	Normal of a surface
$n_1, n_2, \ldots$	Refraction indices
$f(x, \omega \to \omega')$	Bidirectional reflectance distribution function
$\theta, \theta_1, \ldots$	Angles
σ_a, σ_s	absorption and scattering coefficients
$p(x, \omega \to \omega')$	Phase function

Table 2.1: Notation

tation. One of the earliest and best known formulation of light transport is the rendering equation,

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega f(x, \omega_i \to \omega_o) L_i(x, \omega_i) \cos(\theta)\, d\omega_i\,, \tag{2.1}$$

which was proposed by Kajiya in 1986 [Kaj86]. It describes the global illumination in a scene. Here we make the assumptions that light only interacts with surfaces and propagates along straight lines through space. This is a correct assumption if all surfaces reside in a vacuum.

The rendering equation describes how much light $L_o(x, \omega_o)$ leaves a surface at position x in direction ω_o. Obviously, it is necessary to know how much light arrives at x in order to compute the outgoing radiance $L_o(x, \omega_o)$. Since light below the horizon cannot contribute anything, it suffices to consider the upper hemisphere (Figure 2.1). The bidirectional reflectance distribution function (BRDF) $f(x, \omega_i \to \omega_o)$ describes how the surface at x reflects light arriving from ω_i and leaving to ω_o. Thus, to compute $L_o(x, \omega_o)$ an integration over the incoming direction ω_i over the upper hemisphere Ω is necessary. For every incoming direction we evaluate the BRDF which gives us the amount of light that contributes to $L_o(x, \omega_o)$. The cosine is a scaling factor that accounts for the projection of the incoming light onto the surface. Since a surface may emit light we also have a light emission term $L_e(x, \omega_o)$ in the rendering equation. The light emission term describes how much light a surface at x emits in direction ω_o.

Under the assumption that light travels unhindered between surfaces, it is obvious that the radiance $L_i(x, \omega_i)$ arriving at x from direction ω_i is equal to the radiance leaving from x' in direction $-\omega_i$ where x' is the closest intersection when leaving x in direction ω_i. Thus, we have $L_i(x, \omega_i) = L_o(x', -\omega_i)$ (Figure 2.1). Obviously, to compute $L_o(x', -\omega_i)$ we have to apply the rendering equation once more.

The rendering equation provides us with a recursive description of how to compute images with global illumination. A common approach to compute this is recursive ray tracing which was introduces by Whitted in 1979 [Whi79].

The assumption we made in the rendering equation might be valid in vacuum but in reality this is rarely this case. Mostly, there is air or other gases between surfaces. Although the interaction between air and light is usually hardly visible, there are situations where the air affects the light noticeably. An example would be smoke or fog. Here, small particles interact

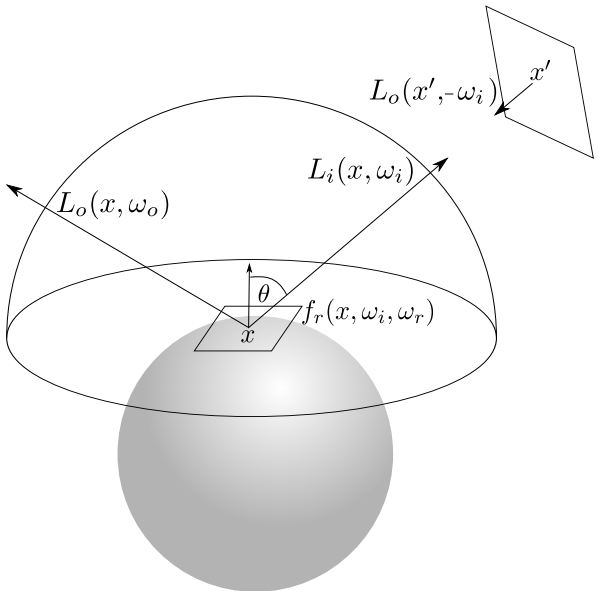


Figure 2.1: The rendering equation describes how light arriving from ω_i at x is reflected of a surface in direction ω_o.

with the light. As a result, the assumption that light propagates without being modified between surfaces has to be abandoned. The rendering equation has to be modified in such a way that it includes effects which occur in such environments, called participating media. The emission and reflectance on surfaces can still be described in the same way as in the rendering equation. The difference, however, is that we cannot assume $L_i(x, \omega_i) = L_o(x', -\omega_i)$ anymore, but we have to compute $L_i(x, \omega_i)$ by taking effects into account that might occur between surfaces. The so called volume rendering equation describes how the radiance changes when light passes through participating media.

The effects which may occur are absorption, emission and scattering [SH01]. In the following, each of these effects are described and then we compose the volume rendering equation from the separate parts.

2.1.1 Absorption

Absorption is the process when light interacts with a particle and is transformed into a different form of energy, for example thermal energy. As a result, the radiance decreases (Figure 2.2).

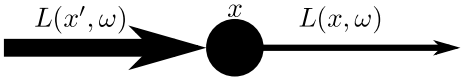


Figure 2.2: When an absorption event occurs the radiance is smaller after the absorption.

Absorption is described with the absorption coefficient σ_a. σ_a defines how much light is absorbed over a unit distance. Thus, to compute the energy loss due to absorption over some

distance Δt we have (assuming that the absorption coefficient is constant over Δt)

$$\begin{aligned} L(x+\Delta t,\omega) - L(x,\omega) &= -\sigma_a(x+\Delta t)\cdot L(x,\omega)\cdot \Delta t \\ \frac{L(x+\Delta t,\omega) - L(x,\omega)}{\Delta t} &= -\sigma_a(x+\Delta t)\cdot L(x,\omega)\,. \end{aligned}$$

If we now let $\Delta t \to 0$ then

$$\frac{dL(x,\omega)}{dt} = -\sigma_a(x)\cdot L(x,\omega)\,.$$

This describes the rate of radiance change along ω over an infinitesimal distance. This is a standard differential equation that can be solved by separation of variables. This yields

$$L(d,\omega) = e^{-\int_0^d \sigma_a(t)\,dt}\,.$$

With the boundary condition that $L(0,\omega) = L(x,\omega)$ we end up with

$$L(x+d\cdot\omega,\omega) = e^{-\int_0^d \sigma_a(t)\,dt}\cdot L(x,\omega)\,. \tag{2.2}$$

The intuitive interpretation of this equation is that when light travels through an absorbing volume from x in direction ω to $x+d\cdot\omega$ the light has less energy, namely $e^{-\int_0^d \sigma_a(t)\,dt}\cdot L(x,\omega)$.

To simplify the notation we denote $T_a(x,x') = e^{-\int_0^d \sigma_a(t)\,dt}$ where $x + d\cdot\omega = x'$. We call $T_a(x,x')$ the absorption transmittance. This is reasonable because $T_a(x,x')$ regulates how much light passes through the volume, in other words, how much light it transmits. The absorption transmittance has the handy property of being multiplicative, meaning, when $x + d\cdot\omega = x'$ and $x' + s\cdot\omega' = x''$ then

$$T_a(x,x'') = T_a(x,x')T_a(x',x'')\,.$$

Figure 2.3 illustrates the multiplicative property of the absorption transmittance. The multiplicative property follows immediately from the fact that the absorption transmittance is an exponential function. This allows us to rewrite Equation 2.2, which becomes

$$L(x,\omega) = T_a(x',x)L(x',\omega)\,.$$

To include absorption in the rendering equation we have to substitute $L_i(x,\omega_i)$ in Equation 2.1 by $T_a(x',x)L_o(x',-\omega_i)$ where x' is the closest intersection when leaving x in direction ω_i (Figure 2.1). This needs to be done because when including absorption $L_i(x,\omega_i) = L_o(x',-\omega_i)$ does not hold anymore but it is $L_i(x,\omega_i) = T_a(x',x)L_o(x',-\omega_i)$.

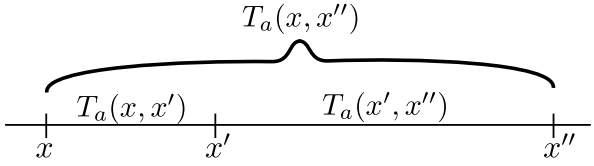


Figure 2.3: To compute the absorption transmittance $T_a(x,x'')$ between x and x'' we can multiply the absorption transmittance $T_a(x,x')$ with $T_a(x',x'')$

2.1.2 Emission

It is not only possible for participating media to absorb light but also to emit light. There are different physical processes, for instance thermal or nuclear, which cause a volume to emit light. This means that at every location in the medium there is a certain amount of light added in a direction ω (Figure 2.4). We can formulate emission with a volume emission function $L_{ve}(x, \omega)$ that tells us how much light is emitted at x in direction ω.

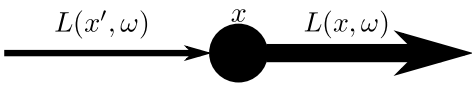


Figure 2.4: When a participating medium emits light at x in direction ω the radiance along this ray is increased.

In the rendering equation we are interested in the incident light $L_i(x, \omega_i)$. When there is emitting media in direction ω_i, it is necessary to integrate along direction ω_i to the next surface intersection x' (Figure 2.1),

$$L_i(x, \omega_i) = L_o(x', -\omega_i) + \int_0^d L_{ve}(x + s \cdot \omega_i, \omega_i)\, ds$$

where $d = \|x - x'\|$.

2.1.3 Scattering

A further and probably the most interesting effect occurring in participating media is called scattering. Scattering is when light interacts with a tiny particle in the participating medium and changes its direction because of this interaction. These tiny particles may have different shapes and behaviors. Therefore, different theories exist to describe these different behaviors, for example Rayleigh scattering describes the light behavior for particles much smaller than the wavelength of the light or Mie theory describe the light behavior for particles similar to or larger than its wavelength [Hul81].

In order to describe scattering we use the scattering coefficient $\sigma_s(x)$ and the phase-function $p(x, \omega_i \to \omega_o)$. The scattering coefficient is used to define how much light is scattered per unit distance whereas the phase-function specifies the probability for a scattering event at the location x for light coming from ω_i and changing direction to ω_o (Figure 2.5).

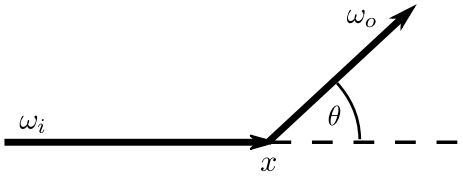


Figure 2.5: The phase function $p(x, \omega_i \to \omega_o)$ describes the probability for light changing direction from ω_i to ω_o at x.

A phase function has the property

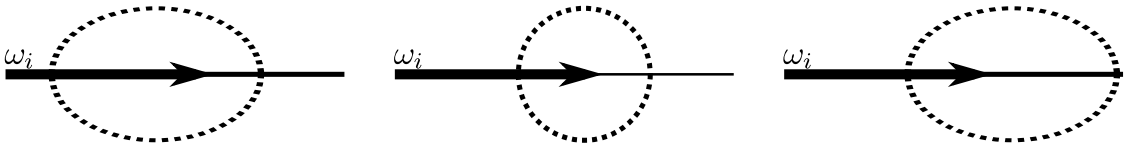
$$\int_S p(x, \omega_i \to \omega_o)\, d\omega_o = 1$$

because it can be interpreted as a probability distribution over the whole sphere S.

The simplest phase function is the function $p(x, \omega_i \to \omega_o) = \frac{1}{4\pi}$. This is the phase function for isotropic scattering. This means that light is scattered equally, or more precisely with the same probability, in all directions. However, there is not only isotropic scattering. In the following we distinguish only between two additional scattering types. One is forward and one is backward scattering. The first means that it is more probable for light to be scattered in a similar direction as the light had before the scattering event. The latter means that light is likely to change direction by more that ninety degrees. Figure 2.6 shows forward and backward scattering. There are many different phase functions, also for the previously mentioned Mie and Rayleigh scattering, but in this master's project we use only the Schlick phase function

$$p_{\text{schlick}}(x, \omega_i \to \omega_o) = \frac{1}{4\pi} \frac{1 - k^2}{(1 - k\cos(\theta))^2}$$

where $k \in (-1, 1)$ and θ is the angle between ω_i and ω_o as indicated in Figure 2.5. We use the Schlick phase function because depending on k the phase function is isotropic ($k = 0$), forward ($k > 0$) or backward ($k < 0$) scattering. Thus, this phase function covers several types of scattering and is easy to implement.



(a) $k < 0$ (b) $k = 0$ (c) $k > 0$

Figure 2.6: Depending on k the Schlick phase function is backward (a) scattering, isotropic (b) or forward scattering (c).

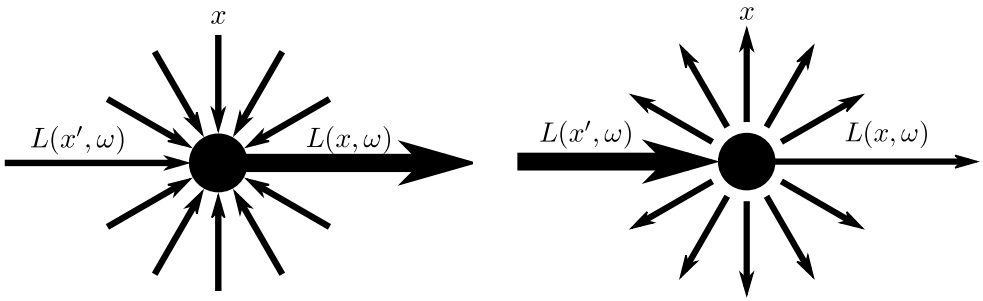
We separate scattering in two parts. The reason being that there are two things that can happen along a line from x to x'. Either light which does not move along this line is scattered such that in propagates along this line afterwards (Figure 2.7a) or that light expanding along this direction and scatters away from this line (Figure 2.7b). The first is in-scattering, which increases the radiance, and the latter is called out-scattering, which reduces the radiance.

In-Scattering

In-scattering is the process when the radiance $L(x, \omega)$ is increased because light propagating through x but in a direction ω' which differs from ω is scattered in the direction ω. Therefore, $L(x, \omega)$ is larger after in-scattering (Figure 2.7a). As already mentioned, the phase-function is used to describe the probability of such an event and the scattering coefficient is used to model how distinctive the scattering is. To determine how much light is scattered in direction ω at x we have to consider all possible incoming directions. Thus, we have to integrate over the whole sphere at location x,

$$L_o(x, \omega) = \sigma_s(x) \cdot \int_S p(x, \omega' \to \omega) L_i(x, \omega')\, d\omega'$$

where $\sigma_s(x)$ determines the amount of light being in-scattered per unit distance. As with emission, in-scattering might occur anywhere along the incident direction ω_i in the rendering



(a) In-scattering (b) Out-scattering

Figure 2.7: Scattering is split into two events. (a) In-scattering increases the radiance $L(x, \omega)$ because light arriving from different directions is scattered towards ω. (b) Out-scattering reduces the radiance because parts of $L(x, \omega)$ is scattered into different directions.

equation. Thus, it is necessary to integrate from x to x' (Figure 2.1),

$$\begin{aligned} L_i(x, \omega_i) &= \int_0^d \sigma_s(x + s \cdot \omega_i) \cdot \int_S p(x + s \cdot \omega_i, \omega' \to \omega_i) L_i(x + s\omega_i, \omega')\, d\omega'\, ds \\ &\quad + L_o(x', -\omega_i)\,. \end{aligned}$$

Out-Scattering

Out-scattering is the opposite process of in-scattering. When we look at the radiance $L(x, \omega)$ and an out-scattering event occurs then the radiance $L(x, \omega)$ after the out-scattering is lower because a fraction of the radiance was scattered into another direction (Figure 2.7b). This means that out-scattering reduces the radiance. This is the same behavior as absorption. Hence, it can be treated the same way as absorption. The only difference is that we replace the absorption coefficient $\sigma_a(x)$ with the scattering coefficient. This yields

$$T_s(x, x') = e^{-\int_0^d \sigma_s(t)\, dt}$$

where $x + d \cdot \omega = x'$.

2.1.4 Extinction

There is a the similarity between absorption and out-scattering. Both reduce the radiance and have the same mathematical formulation. Thus, the two processes can be combined into one which we call extinction,

$$L(x, \omega) = T_a(x, x') T_s(x, x') L(x', \omega)\,.$$

The multiplication makes intuitively sense because there are two processes that hinder the light propagation. Since T_s and T_a are exponential functions it is easy to combine them,

$$\begin{aligned} T_a(x,x')T_s(x,x') &= e^{-\int_0^d \sigma_a(t)\,dt} \cdot e^{-\int_0^d \sigma_s(t)\,dt} \\ &= e^{-\int_0^d \sigma_a(t)\,dt - \int_0^d \sigma_s(t)\,dt} \\ &= e^{-\int_0^d \sigma_a(t)+\sigma_s(t)\,dt} \\ &= e^{-\int_0^d \tau(t)\,dt} \\ &= T(x,x') \end{aligned}$$

where $\tau(t)$ is known as the extinction coefficient. As opposed to the absorption transmittance $T_a(x,x')$ we simply call $T(x,x')$ the transmittance from x to x'. With the transmittance the incident radiance in the rendering equation respecting absorption and out-scattering can be expressed as

$$L_i(x,\omega_i) = T(x',x)L_o(x',-\omega_i)\,.$$

2.1.5 Volume Rendering Equation

Emission, absorption and scattering are all effects in participating media [SH01]. With the previous equations, the volume rendering equation can be assembled. The goal is to compute the excitant radiance $L_o(x,\omega)$. The volume rendering equation describes how light propagating from x to x' changes. The distance from x to x' is said to be d and $L_i(x',-\omega)$ is the incident radiance at x' (Figure 2.8). Here the locations x and x' do not need to be on a surface.

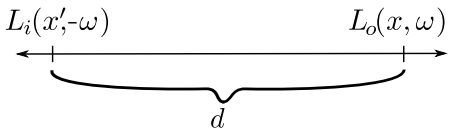


Figure 2.8: Setting for the volume rendering equation. $L_o(x,\omega)$ is the excitant radiance we want to compute and $L_i(x',-\omega)$ is the incident radiance.

First, we include the effects that might increase the radiance, namely in-scattering and emission. It is necessary to integrate along direction ω from location x' to x because at each position between x and x' the radiance might be increased. Thus, we have

$$\begin{aligned} L_o(x,\omega) &= \int_0^d L_{ve}(x'+t\cdot\omega,\omega) \\ &\quad +\sigma_s(x'+t\cdot\omega)\int_S p(x'+t\cdot\omega,\omega'\to\omega)L_i(x'+t\cdot\omega,\omega')\,d\omega'\,dt\,. \end{aligned} \tag{2.3}$$

If the medium decreases the radiance along the direction ω, for instance because it scatters or absorbs light, then we have to correctly multiply with the transmittance. Now, when integrating along the direction ω the transmittance changes with the distance from x. The further away from x, in other words the larger t in Equation 2.3, the more light is attenuated.

Thus $T(x' + dt \cdot \omega, \omega)$ has to be included as follows

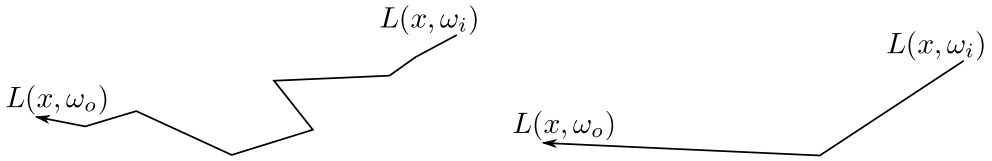
$$L_o(x,\omega) = \int_0^d T(x, x' + t\cdot\omega)\big[L_{ve}(x' + t\cdot\omega, \omega) + \sigma_s(x' + t\cdot\omega)\int_S p(x' + t\cdot\omega, \omega' \to \omega) L_i(x' + t\cdot\omega, \omega')\, d\omega'\big]\, dt \tag{2.4}$$

Finally, the incident $L_i(x', -\omega)$ radiance at x' needs to be added. It is only affected by the extinction which brings us to the final volume rendering equation

$$\begin{aligned} L_o(x,\omega) = & \int_0^d T(x, x' + t\cdot\omega)\big[L_{ve}(x' + t\cdot\omega, \omega) \\ & + \sigma_s(x' + t\cdot\omega)\int_S p(x' + t\cdot\omega, \omega' \to \omega) L_i(x' + t\cdot\omega, \omega')\, d\omega'\big]\, dt \\ & + T(x, x') L_i(x', -\omega)\,. \end{aligned} \tag{2.5}$$

To include the volume rendering equation in the rendering equation we simply have to realize that $L_o(x,\omega)$ from Equation 2.5 equals $L_i(x,\omega_i)$ of the rendering equation when $\omega = -\omega_i$. Thus we only have to replace ω with $-\omega_i$ in Equation 2.5 to compute $L_i(x,\omega_i)$.

Equation 2.5 is computationally expensive to evaluate. The reason is the inner integral which yields a theoretically infinite recursion, similar to the rendering equation. To evaluate the integral we need to know $L_i(x' + t\cdot\omega, \omega')$ which again is evaluated with Equation 2.5. This needs to be done if one wants the correct value for $L_i(x' + t\cdot\omega, \omega')$. Intuitively, what this recursion describes is multiple scattering. This means that light which entered the medium changes direction multiple times due to scattering events. Figure 2.9a illustrates multiple scattering of a light ray.



(a) Multiple scattering (b) Single scattering

Figure 2.9: (a) Multiple scattering is when a light path changes direction more than once due to scattering events. (b) Single scattering is when only one scattering event is allowed.

A simplification that is often made is to limit the number of scattering events to one (Figure 2.9b). The result is that we approximate $L_i(x' + t\cdot\omega, \omega')$ in the inner integral by computing the attenuated incident light from a light source or from a surface.

Considering only single scattering is a reasonable simplification because the visually important features are produced by single scattering and not multiple scattering. Multiple scattering appears just as diffuse lighting whereas single scattering is responsible for the so called "god rays", which are very distinctive in participating media (Figure 2.10). Thus, in real-time rendering it is sensible only to take single scattering into account because multiple scattering is not worth spending computation time. Even with the simplification of including only single scattering we still need to evaluate the inner integral in Equation 2.5 but without the recursion. The inner integral is not trivial to evaluate. In the upcoming section we present a well-known technique to compute such integrals. The outer integral, on the other hand, can be easily approximated with a finite Riemann sum.

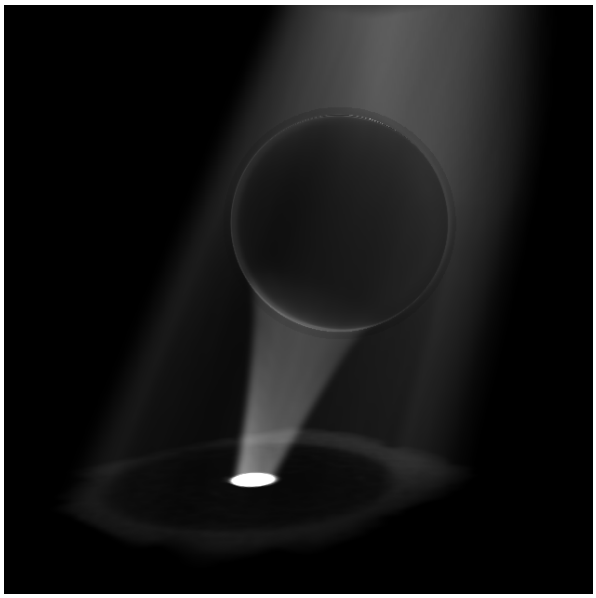


Figure 2.10: God ray: A visible light cone due the single scattering.

2.2 Monte-Carlo Integration

In this section we are concerned with evaluating the volume rendering equation, especially the inner integral which integrates over the whole sphere S. In fact, the Monte-Carlo integration, which is presented here, can be used to evaluate any integral. As we have seen, to compute the radiance with participating media volume we have to evaluate the inner integral

$$\int_S p(x' + t \cdot \omega, \omega' \to \omega) L_i(x' + t \cdot \omega, \omega') \, d\omega' \, .$$

and the outer integral along a line. The outer integral can be easily approximated as a finite Riemann sum. The inner integral is more complex to evaluate. A common technique to solve problems like this is Monte-Carlo integration.

Monte-Carlo integration is a result from probability theory. In the following, a simplified mathematical derivation for functions $f(x) : A \to \mathbb{R}$ where $A \subseteq \mathbb{R}$ is presented. It is possible to generalize the derivation to higher-dimensional functions [GS92]. It is assumed that the reader is familiar with basic stochastic terms like probability distribution, random number and the like. As a reminder, we are interested in solving an integral $\int_A f(x) \, dx$. First we formulate the general definition of the expected value.

Definition 1. *Let X be a random number over $A \subseteq \mathbb{R}$, p the probability density function for X and $f(x) : A \to \mathbb{R}$. The expected value of $f(X)$ is*

$$E(f(X)) = \int_A p(t) f(t) \, dt$$

Whenever this integral is not analytically solvable then the sample mean is generally used instead.

Definition 2. *Let $\{X_i\}$ for $i = 1, \ldots, n$ and X be independent and identically-distributed random variables over $A \subseteq \mathbb{R}$. The sample mean of a function $f(x) : A \to \mathbb{R}$ is*

$$\overline{f(X)} = \frac{1}{n} \sum_{i=1}^{n} f(X_i)$$

The law of large number describes how the sample mean and the expected value relate to one another.

Proposition 1 (Law of Large Numbers)**.** *Let $\{X_i\}$ for $i = 1, \ldots, n$ and X be independent and identically-distributed random variables over $A \subseteq \mathbb{R}$ with finite expected values then*

$$\lim_{n \to \infty} \frac{1}{n} \sum_{i=1}^{n} f(X_i) = E(f(X))$$

with probability one

Proof. Can be found in any standard stochastic book [GS92].

This lets us then approximate the integral that we want to compute,

$$\begin{aligned} \int_A f(x)\, dx &= \int_A \frac{f(x)}{p(x)} p(x)\, dx \\ &= E(\frac{f(x)}{p(x)}) \\ &= \lim_{n \to \infty} \frac{1}{n} \sum_{i=0}^{n} \frac{f(x_i)}{p(x_i)} \\ &\approx \frac{1}{n} \sum_{i=0}^{n} \frac{f(x_i)}{p(x_i)} \,. \end{aligned}$$

Therefore, it is possible to evaluate any integral $\int_A f(x)\, dx$ by sampling the function $f(x)$ at random locations, weight the values with the inverse of the probability of these locations and build the sum over all the function values and scale it by $\frac{1}{n}$. This is called Monte-Carlo integration. Depending on the sample count and the sample distribution we obtain a more or less accurate approximation of the integral. The issue of sampling is addressed in the next section.

Coming back to the initial task of evaluating

$$\int_S p(x' + t \cdot \omega, \omega' \to \omega) L_i(x' + t \cdot \omega, \omega')\, d\omega \,.$$

As mentioned, Monte-Carlo integration can be generalize and is, therefore, applicable to our spherical integral too. Since we integrate over directions of the sphere we have to sample directions instead of real numbers according to some distribution. The rest, however, remains the same.

2.2.1 Sampling

As it was pointed out earlier, the distribution of the samples that is used for Monte-Carlo integration influences the quality of the approximation.

The most obvious sampling strategy is certainly a uniform random sampling. This sampling is not optimal though. The reason is that purely random sampling may lead to clumping. It is possible that certain regions are sampled tightly where other regions are sampled too sparsely. Depending on the function and the sampling this can lead to a very bad approximation (Figure 2.11). Another obvious sampling strategy is the grid sampling. Here the samples

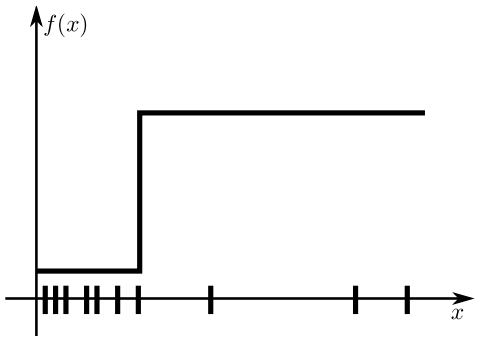


Figure 2.11: With purely uniform random sampling it may happen that a region that has a high contribution to the integral is sampled too sparsely whereas regions with low contribution are sample too tightly. This leads to a bad approximation.

have a fixed distance from one another. In fact this is not a random sampling. Yet, it can still be used because it is one possible outcome of uniform random sampling. Again this sampling is not optimal. This time the reason is the regular pattern. When the function exhibits also a regular pattern then depending on the alignment the approximation might be bad (Figure 2.12). Thus, a sampling strategy needs to be found that does not clump but is still appears

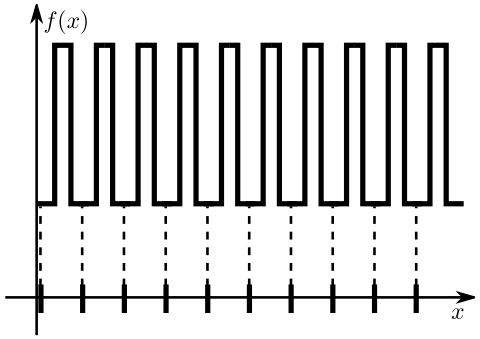


Figure 2.12: The gird sampling fails to sample the spikes thus leading to a bad approximation.

random. This requirement leads to quasi-random distributions. The most famous of these are the Halton and the Hammersley sequence [WLH97]. Quasi-random distributions have several advantages. The first is that they are not random, they only appear to be random thus the name quasi-random. This property is desirable because this way the approximation is the same for all computation runs. This is not the case with random sampling, which is with computers usually only pseudo random. The second is that these sequences lead to a

fast convergence. With the same number of samples the approximation with quasi-random numbers is in general better than with most other sampling techniques. This can be shown mathematically [Nie92].

Since quasi-random Hammersley and Halton sequence are easy to generate and have the nice property of a leading to a good approximation it might be advantageous to apply them for sampling tasks.

2.3 Photon Tracing

We have seen how global light transport with participating media can be formalized mathematically. However, the question that remains is how can we compute light transport practically. Many different techniques have been developed over the years, for example radiosity [GTGB84] or path tracing [Kaj86]. One method in particular suits our purpose of computing images with refractive objects in participating media in real-time. The method of tracing photons which was first proposed by Jensen and Christensen [JC95]. Photon tracing consists basically of two phases. In phase one we trace photons through a scene and build a so called photon map. The photon map stores the illumination of the scene which is then needed in phase two. In phase two the final image is rendered with ray tracing.

Photon Tracing and Photon Map

For each light source a certain amount of photons is created and shot into the scene (Figure 2.13a). As in physics, each photon carries energy which it deposits all over a scene while moving through the scene. Assuming there is no participating medium then the photons lose energy only when they are reflected of surfaces. Jensen proposed to trace photons and every time a photon interacts with a surface the location, incident direction and energy of the photon is stored in a map (Figure 2.13b). Thus, the technique is called photon mapping. When looking at the rendering equation we realize that each entry in the photon map corresponds to a scaled incident radiance $L_i(x, \omega_i)$ where the photon's location is x, the direction is ω_i and $L_i(x, \omega_i)$ is the energy. The reason that the entries in the photon map are scaled incident radiances comes from the fact that photon mapping is based on Monte-Carlo integration. In terms of Monte-Carlo integration each entry in the photon map corresponds to one summand

$$\frac{f(x_i)}{p(x_i)}$$

of the sum

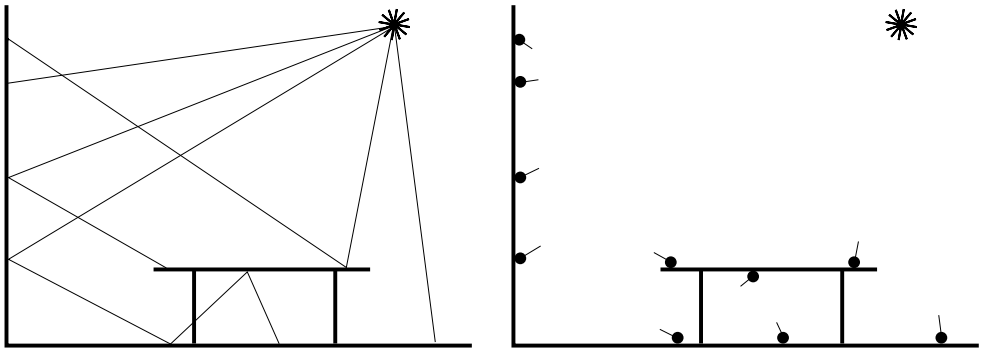
$$\frac{1}{n}\sum_{i=0}^{n}\frac{f(x_i)}{p(x_i)}$$

where $f(x_i)$ corresponds to the incident radiance and $p(x_i)$ the scaling.

The photon map represents the lighting in the scene we want to render and is, therefore, all the lighting information that is needed to compute the final image.

Radiance Estimation

For computing the final image with ray tracing we shoot a ray for each pixel of the final image into the scene. At each intersection of a ray with a surface we perform a look up in

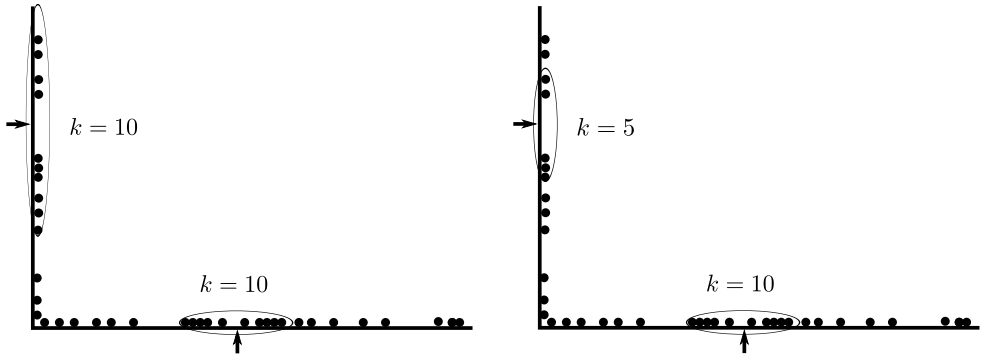


(a) Photon tracing (b) Photon map

Figure 2.13: (a) A light source emits photons which are traced through the scene. (b) Every time a photon is reflected of a surface the location, direction and energy is stored in a photon map.

the photon map to determine the incoming radiance at this location. Unfortunately, it is very unlikely that a photon was recorded at this exact intersection. Hence, it is necessary to estimate the radiance at this location from the entries in the photon map. There are two standard estimators for this task.

The first is the k-nearest-neighbor estimator (Figure 2.14a). It estimates the radiance at the intersection to be a weighted sum of the k-closest entries to the intersection. The second standard estimator uses a constant radius (Figure 2.14b). This means that we build a weighted sum of all the photons which are closer to the intersection than a given distance. Both estimators estimate the local outgoing radiance as



(a) k-NN estimator (b) Constant radius estimator

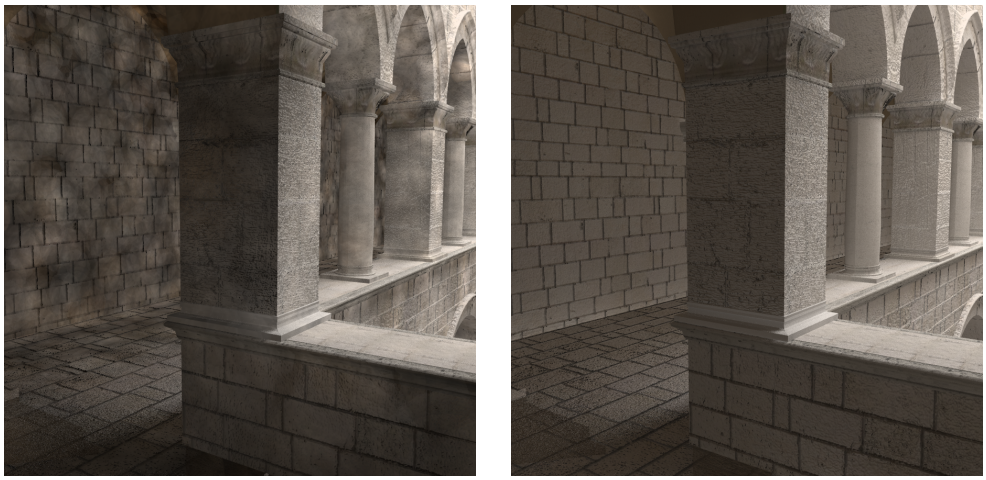
Figure 2.14: In both images the ellipse indicates the estimator radius and the arrows point the the locations where an estimation is required. (a) This example shows the k-nearest-neighbor estimator with $k = 10$. Thus the estimator radius is expanded until it includes at least ten photons. This results in different filter radii for each location. (b) The filter radii are constant. Thus, a different number of photons is used for the estimation depending on the location.

$$L_o(x, \omega_o) \approx \frac{1}{r^2\pi} \sum_k f(x, -\omega_i \rightarrow \omega_o) photon_i$$

where ω_i is the incident direction of photon i, $photon_i$ its scaled incident radiance and $f(x, -\omega_i \rightarrow \omega_o)$ the BRDF. When the k-nearest-neighbor is used, k is constant for all locations x and the radius r of the circle that contains k photons changes for each location x (Figure 2.14a). When the constant radius estimator is used, the radius r is constant for all locations x but the photon count k in the circle varies (Figure 2.14b).

When looking at the rendering equation we realize that the estimation is in fact an approximation of the integral over the upper hemisphere. However, this is not a proper Monte-Carlo integration because according to Monte-Carlo integration the incident light $L_i(x, \omega_i)$ must be computed for each sample direction ω_i at the fixed location x. In photon mapping we assume that $L_i(x, \omega_i) \approx L_i(x', \omega_i)$ for some close by location x'. Still, it is obvious that with infinite photons (i.e. the photon map holds the incident radiance for each location and each direction) photon mapping fulfills the rendering equation. Therefore, the more photons are used the better an approximation can be obtained.

This technique produces images with much low frequency noise because of the sloppy implementation of the Monte-Carlo integration. An improvement is the idea of splitting the photon map into a caustic and a diffuse map and using a technique final gathering for the diffuse map [Jen09]. This reduces the noise noticeably which can be seen in Figure 2.15.



(a) Without Final Gathering (b) With Final Gathering

Figure 2.15: (a) Photon mapping without final gathering shows low frequency noise. (b) With final gathering the noise disappears. Source: [PH04b]

2.3.1 Participating Media

Jensen and Christensen extend photon mapping to include participating media [JC98]. To include scattering, absorption and emission it is necessary to store the photons not only on the surfaces but in three dimensional space. The reason being that during the final image computation, which is usually done by ray tracing, it is necessary to know the illumination inside the volume containing the participating media.

In this master's thesis we follow the idea of storing the photons directly in a three dimensional array [SZS+08]. This approach is reasonable for real-time photon mapping because the look up for neighboring photons in the photon map is expensive even with accelerating data structures. With a 3D array each cell of the array covers a small subspace of the region that contains participating media (Figure 2.16). These cells are called voxels i.e. volumetric pixels.

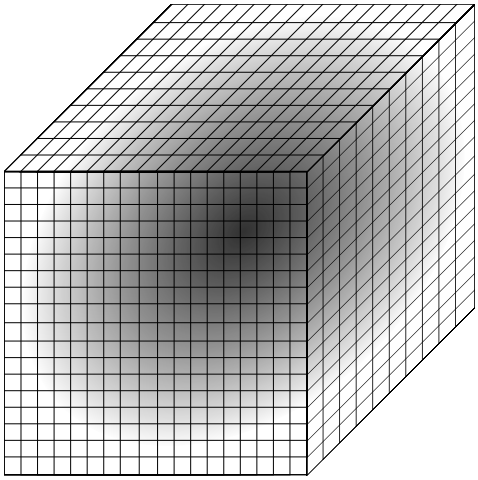


Figure 2.16: The participating media is covered by a voxel grid, i.e. three dimensional array.

When tracing the photons we simply add the energy a photon carries to the voxels it passed through. This obviously implies an immediate smoothing filtering with a constant radius of approximately half the size of a voxel and with the estimation location at the center of a voxel. This makes it necessary to interpolate between the voxel centers when computing the final image, otherwise the estimation locations are misplaced.

This allows a fast rendering of scenes with participating media with photon tracing because the expensive nearest-neighbor search can be avoided.

2.4 Refraction

The goal of this master's project is to render images with refractive object in a participating volume. Hence, a formulation for light refraction is required. There are two standard formulations which are used in computer graphics. One is Snell's law,

$$\frac{\sin(\theta_1)}{\sin(\theta_2)} = \frac{n_2}{n_1}\,, \tag{2.6}$$

which describes the change in direction when light passes from one medium into another. In Equation 2.6 θ_1 is the angle between the direction of the incident light and the surface normal, θ_2 is the angle between the direction of the refracted light and the surface normal, n_1 is the refraction index of material where the light came from and n_2 is the refraction index of the material the light enters (Figure 2.17). It can be seen that the refraction indices, which are material properties, determine how light changes direction when it switches material. Snell's law is valid if the materials are homogeneous, meaning they have a constant refraction index.

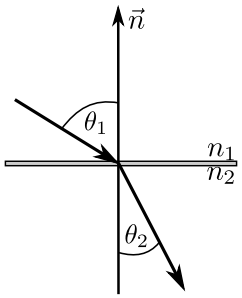


Figure 2.17: An illustration of the parameters in Snell's law. $\vec{n}$ is the normal of the surface, n_1, n_2 are the refraction indices and θ_1, θ_2 are the angles between the incoming direction, the refracted direction and the normal.

In a photon or ray tracing algorithm, Snell's law makes it necessary to compute the intersection of a photon or ray with the surface of a refractive object. This intersection test can be rather expensive. Thus, Snell's law is not ideal for our purpose.

An other formulation for refraction is a differential equation,

$$\frac{d}{ds}\left(n\frac{dx}{ds}\right) = \nabla n, \tag{2.7}$$

which is derived from the eikonal equation and the transport equation [Bor99]. ds is an infinitesimal step in the direction of the tangent of the path of the light ray. n is the refraction index. Here it does not have to be constant. In other words, this formulation respects inhomogeneous materials, which have a varying refraction indices. An additional difference to Snell's law is that it includes total internal reflection.

Equation 2.7 can be rewritten in such a way that it can easily be included in photon tracing. Let $v = n\frac{dx}{ds}$, which is the scaled velocity along the tangent of the photon or light ray, then

$$\frac{v}{n} = \frac{dx}{ds} \tag{2.8}$$

$$\nabla n = \frac{dv}{ds} \tag{2.9}$$

The next step is to discretize both (2.8) and (2.9) which yields

$$\begin{aligned} \frac{v}{n} &= \frac{dx}{ds} \approx \frac{x_{i+1} - x_i}{\Delta s} \\ \nabla n &= \frac{dv}{ds} \approx \frac{v_{i+1} - v_i}{\Delta s}\,. \end{aligned}$$

This formulation allows us to easily compute the path of photons or light rays that pass through inhomogeneous materials by advancing the photons stepwise. Assuming that the step size $\Delta s > 0$ and that x_0 is the initial position and v_0 the initial propagation direction of a photon. The next position and direction for the photon can be computed with

$$x_{i+1} = x_i + \frac{\Delta s}{n} v_i \tag{2.10}$$

$$v_{i+1} = v_i + \Delta s \nabla n\,. \tag{2.11}$$

Obviously, this produces only a piecewise linear approximation to the true path but the smaller Δs is the better the approximation becomes.

The latter formulation of refraction suits our needs well because it respects total internal reflection and intersection testing can be avoided. This is beneficial in order to achieve real-time computation because intersection testing with arbitrary geometry is computationally expensive. Thus, we follow the idea of Sun et al.. They propose to store most of the scene data, including the refraction indices, in a three dimensional array that represents the scene. This is the same idea as the one that was explained in Section 2.3.1. Again each cell of the array, which we call voxel, covers a certain part of the scene. Hence, a voxel stores the refraction index of the space it covers. This allows the integration of refractive objects in a scene without computing any intersections.

2.5 GPU Computing

The last requirement for this master's project is that the rendering is performed in real-time. Hence, it is sensible to exploit the most powerful processing unit available in a common personal computer. This would be the graphics processing unit (GPU). Over the last decade GPUs have become by far more powerful than CPUs. Figure 2.18 shows that the NVIDIA GT200 Series is able to process more than four times the number of floating point operations per second (FLOPS). Additionally, the memory bandwidth is more than five times higher (Figure 2.18).

The main reason for this discrepancy is that the GPU is designed for highly parallel computations. Current GPU are equipped with hundreds of processors whereas a CPU consists of usually less than eight. For example, the NVIDIA GTX 260 has 192 stream processors and the Intel i7-975 has four cores. Figure 2.19 show a schematic CPU and GPU. As a result a GPU is able to process a lot more data per time unit. The reason why a GPU can be equipped with hundreds of processors is because they have some limitations compared to CPU cores. A GPU is primarily designed for computing images. Since the work to determine the color of a pixel in an image is for all pixels the same, the computation can be easily parallelized. Additionally, the data access is usually simpler than in general CPU applications. Therefore, much less data caching is needed. Furthermore, the instructions that are required for image computations are more limited. This allows GPU manufacturers to produce highly specialized processors which are very powerful and because hardly any caching is needed there is room for more stream processors (Figure 2.19).

These circumstances opened a new field of research called GPGPU, general purpose computation on graphics processing unit. The goal of this research area is to find ways how to compute general problems efficiently on the GPU; data sorting, fluid simulation, medical-image reconstruction to give just a few examples.

2.5.1 CUDA

Unfortunately, for a long time the GPU was only accessible through graphics application programming interfaces (API's) such as OpenGL or DirectX. This made the implementation of general problems tedious because these interfaces were primarily designed for image rendering. Generally, a program is implemented as a shader which is a small part of the rendering pipeline of OpenGL or DirectX. This often enforces the programmer to arrange the data in

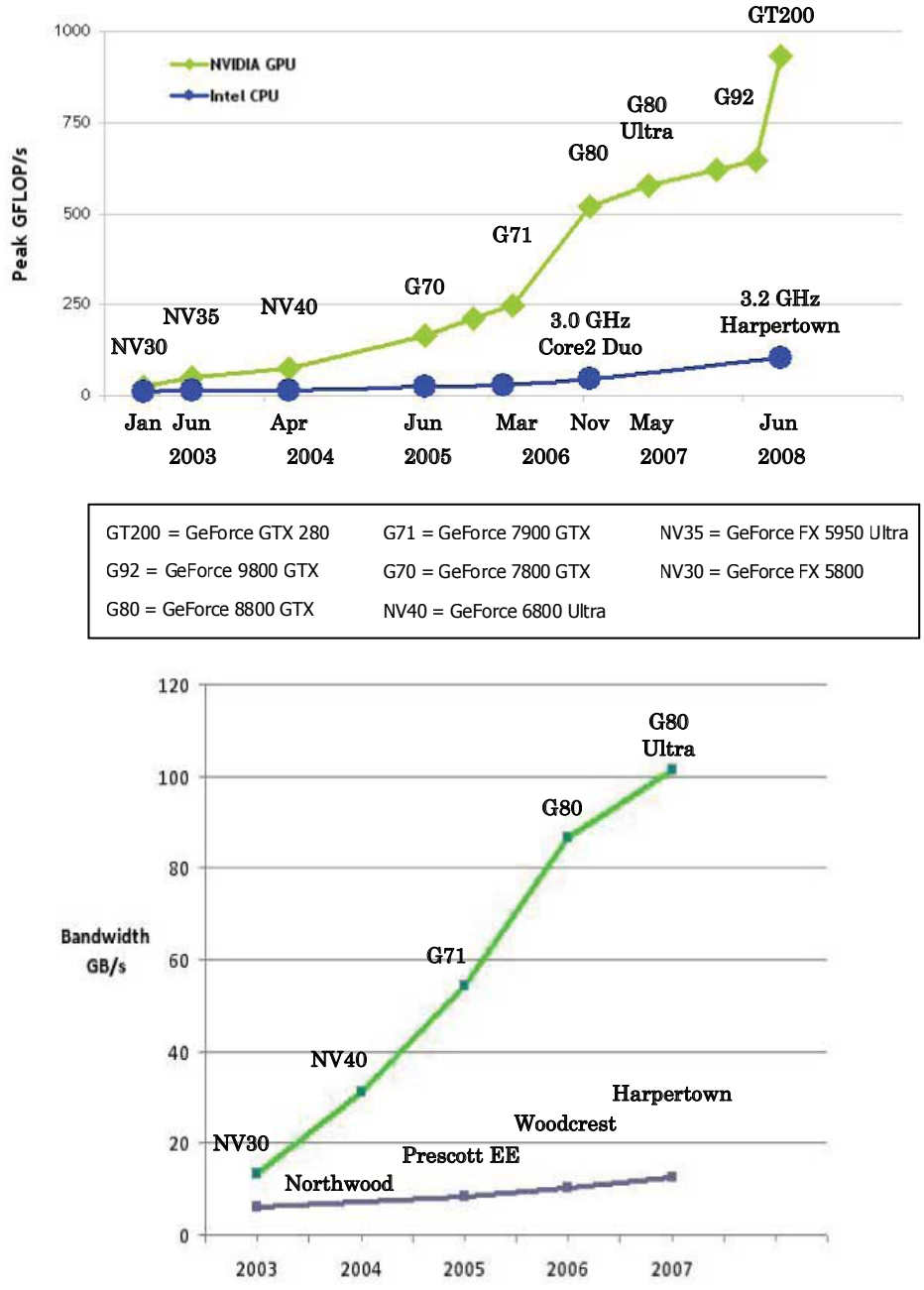


Figure 2.18: The superiority of GPUs compared to CPU. Source: [NVI09b]

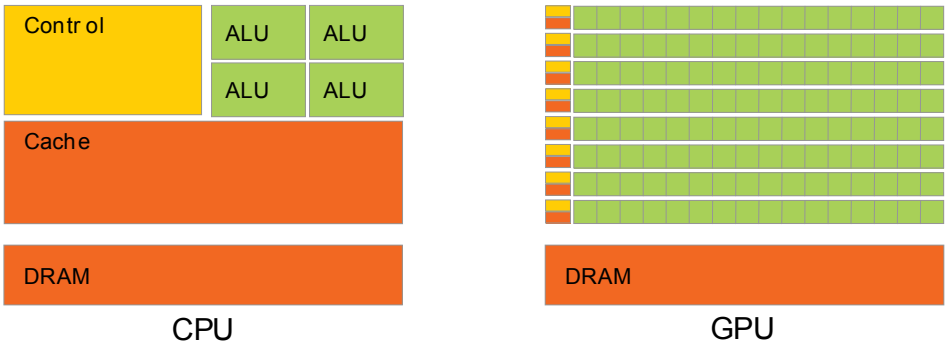


Figure 2.19: A contemporary GPU consists of many processors whereas a CPU is equipped with less than eight. Source: [NVI09b]

an unnatural way in order to make the data available to the shader. Furthermore, a shader cannot write output versatilely because the output is linked with the framebuffers. A thread can write data at most to one pixel per framebuffer. Additionally, the number of threads that are generated is controlled via scene geometry. GPGPU research is interested in finding smart ways how to deal with these restrictions.

The GPU manufacturers are also aware of these problems and have started addressing these issues by providing novel interfaces and architectures to access the GPU directly. ATI introduced the ATI Stream SDK and NVIDIA introduced CUDA (Compute Unified Device Architecture). The Khronos Group developed a specification for GPGPU computing called OpenCL which should standardize GPGPU computing in the future.

In this master's project we focus on NVIDIA's CUDA because OpenCL is not yet well-tried and we had only NVIDIA hardware at our disposal.

CUDA has two APIs which must not be mixed; the low-level and the high-level driver API. Here we focus on the high-level API only but first we discuss the device architecture of a CUDA supporting GPU.

Architecture

As previously mentioned, the computation of an image can be efficiently parallelized because for each pixel similar instructions are executed. Normally, only the data for each pixel differs, for example each pixel accesses a different texture coordinate. As a result, it is assumed that all threads have the same list of instructions, or instruction stream. It is not required that all threads on a GPU have the same instruction stream but if they do the performance is better. An NVIDIA GPU consists of multiple, so called, Streaming Multiprocessors (SMs) and global device memory that is shared among all SMs. Each SM consists of eight Scalar Processor (SP) cores. Figure 2.20 illustrates the architecture of an NVIDIA GPU.

In general CUDA threads are as versatile as CPU threads. As mentioned however, if threads behave differently the performance might drop. Threads are organized in two levels. The first, controlled by the programmer, is that threads are grouped in blocks. At runtime, each SM processes the threads of a block. The second level, is that the threads of a block are split into warps, 32 threads. An SM processes one instruction per warp that must be the same for all threads of the warp. This is done in four cycles. Thus all eight SP execute the same instruction four times. When threads have different instruction they are processed serially.

This is where the assumption of all threads having the same instruction stream enters. This explains also why there may be a performance penalty when not all threads have the same instruction stream.

It was mentioned that on GPUs there is hardly any caching necessary. On NVIDIA GPU there is only caching for constant and texture memory. The reason why only constant and texture memory is cached is because both are immutable and cause, therefore, no cache inconsistency. In CUDA texture memory is in fact regular global device memory that is constant. However, memory access through textures may have performance benefits over regular global device memory access because of caching. Additionally, with textures one can use built-in data interpolation. Apart from the caches, each SM has so called shared memory. Shared memory is managed by the programmer and might, therefore, be used for manual caching.

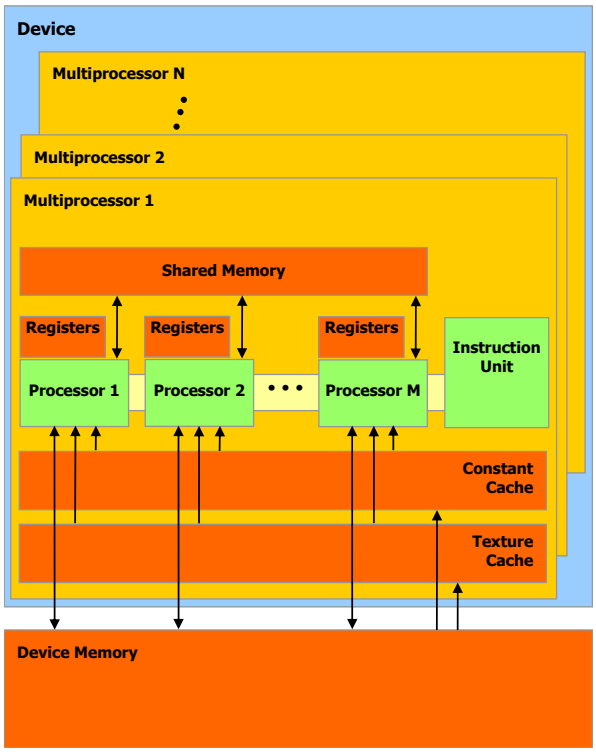


Figure 2.20: NVIDIA GPU architecture. Source: [NVI09b]

API

CUDA is a technology which allows the GPU to be used as co-processors to the CPU. Programmers can access these processors through the CUDA-API and the programming language "C for CUDA", which is basically standard C with several new keywords. The basic concept of CUDA is that GPU code is arranged in special functions, called kernels. Kernels are regular C functions which must have `void` as their return type and must be marked with the keyword `__global__`.

In order to execute a kernel, NVIDIA introduced new syntax. Since the GPU is designed for highly parallel computation, it is expected that a kernel has many instances. Each instance

of a kernel is a separate thread which is executed in parallel with other threads. It is necessary to specify how many threads should be generated of a kernel. This is done with the new <<< >>> syntax. In order to ease the mapping of threads to multidimensional grid coordinates, CUDA requires that the number of threads to be launched is specified in blocks and grids (see Figure 2.21). The grouping is as follows. Threads are arranged three dimensionally in blocks and blocks are grouped in a two dimensional grid.

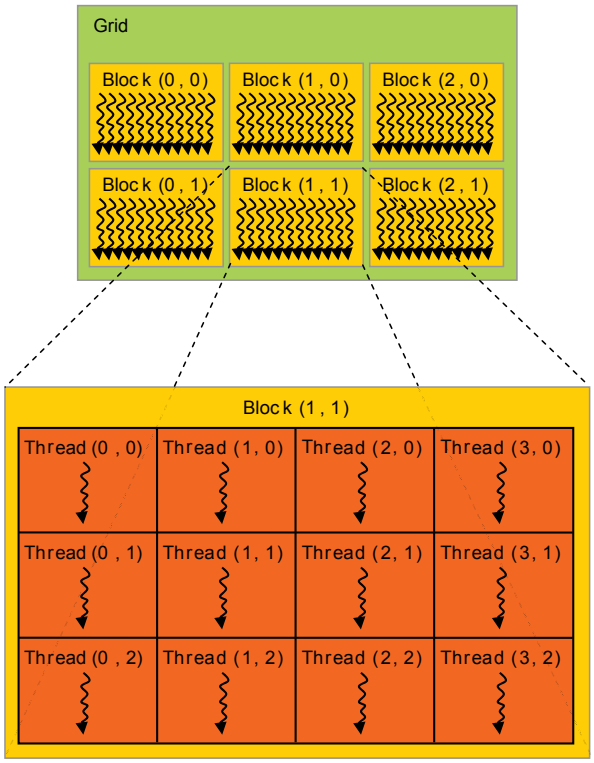


Figure 2.21: CUDA threads are grouped in blocks and blocks are aligned in grids. Source: [NVI09b]

In a kernel we can obtain the size of a block and the grid through the built-in variables `gridDim, blockDim`. The coordinate of a block in a grid and the coordinate of the thread in the block can be accessed through `blockIdx, threadIdx`.

To give a demonstration, we launch a kernel `foo(float *data)` with a two by two grid, thus four blocks, with 512 threads per block which are arranged as a three dimensional 8x8x8 block (Listing 2.1). Furthermore, we allocate GPU memory, for each thread one float in an array named `data` which is then passed to the kernel. Allocating memory on the GPU is done with the `cudaMalloc(void** ptr, size_t size)` method, which behaves the same way as the standard C `malloc`. With the built-in variables for the grid and block sizes and coordinates we can compute the array index for each thread.

Listing 2.1: CUDA kernel launch and memory allocation

```
__global__ void foo(float *data)
{
   int dataIdx = threadIdx.x +
      threadIdx.y*blockDim.x +
```

```
        threadIdx.z*blockDim.x*blockDim.y +
        blockIdx.x *blockDim.x*blockDim.y*blockDim.z +
        blockIdx.x *blockDim.x*blockDim.y*blockDim.z*gridDim.x;
    ...
    // some interesting work
    ...
}

void kernelLauncher()
{
    // allocate memory on the GPU
    float *data;
    cudaMalloc((void**) &data, sizeof(float)*2*2*8*8*8);

    // missing dimension is set to 1
    dim3 grid = dim3(2,2);
    dim3 block = dim3(8,8,8);
    // launch 8*8*8*2*2 threads of the foo kernel
    foo<<<grid, block>>>(data);
    ...
}
```

It is also possible to write methods which are not standalone kernels but helper functions which can be called from kernels. These function are called device functions and are annotated with the keyword `__device__` in the same way we annotated a kernel with the `__global__` keyword. We distinguish between host, the CPU side, and the device, the GPU side. This distinction has to be made for functions and for memory. There are several restrictions in the way the device and host interact with each other.

- `__device__` functions may only be called directly or indirectly from a kernel but not from the host.
- `__global__` functions (kernels) may only be called from the host but not from the device.
- Host memory, memory allocated with `malloc` must not be accessed within a kernel.
- Device memory, memory allocated with `cudaMalloc` or other CUDA memory allocation methods, must not be accessed from the host.

Thus, it would not be acceptable to access the `data` array in `kernelLauncher` or to allocate the memory for `data` with `malloc` instead of `cudaMalloc`.
A convenient feature of CUDA is that it provides interoperability with OpenGL. Buffer Objects from OpenGL can be used directly as linear memory just as it were allocated by `cudaMalloc`. This allows us to implement a rendering pipeline in almost regular C and then display the result with OpenGL without a hassle. Similar interoperability is guaranteed for DirectX.

A final noticeable feature is that CUDA offers atomic manipulations on memory. This allows a safe interaction of threads. Unfortunately, these atomic operations are costly because it introduces a dependency between threads and forces the thread scheduler to serialize the execution of threads.

This is only a brief introduction to CUDA, more detailed information can be found in the NVIDIA CUDA Programming Guide [NVI09b] and the NVIDIA CUDA C Programming Best Practices Guide [NVI09a].

Chapter 3

Implementation

In this chapter we present two rendering pipelines. The first pipeline is based on the rendering pipeline proposed by Sun et al. [SZS+08]. It is a photon marching based pipeline that runs entirely on the GPU guaranteeing interactive frame rates. The pipeline uses voxel grids for storing scene and illumination data to allow parallelization. The effects that can be rendered with this pipeline are refraction, reflection, single scattering and absorption. The second pipeline extends the first. We introduce a replacement for the photon generation and the photon marching. The replacement performs an adaptive photon sampling of the scene. Thus, the second pipeline is called the adaptive rendering pipeline. Due to the adaptive sampling less photons are required. We hoped that using less photons would accelerate the image computation. Unfortunately, this was not the case as we show in Section 4.1.

3.1 Reference Rendering Pipeline

In the year 2008, Sun et al. proposed a rendering pipeline for rendering dynamic refractive objects in participating media in real-time [SZS+08]. Their rendering pipeline is built from five passes. We follow the main idea of Sun et al., but our implementation differs in several aspects. We implemented a different voxelization method and the photon generation and photon marching are entirely written in CUDA, Sun et al. used OpenGL. Our implementation also runs entirely on the GPU, thus, guaranteeing very fast computation such that interactive frame rates are possible. In the following we first discuss the pipeline on a high level (Section 3.1.1) then we take a look at the implementation details of each pass separately (Section 3.1.2 - Section 3.1.6).

3.1.1 Overview

Since the GPU consists of many processing units it is important that all passes can be executed in parallel with as little synchronization between threads as possible. Sun et al. propose using three dimensional arrays to store the scene information. All 3D arrays cover the same area which is where participating media is. This means that each cell of the 3D arrays covers a certain region of the participating media. In the 3D arrays we store the refraction indices, extinction and scattering coefficients of the region covered by the 3D array. In addition, the radiance distribution in the covered region is also stored in a 3D array. A cell of such an array is henceforth referred to as a voxel. This is sensible because a voxel represents a small

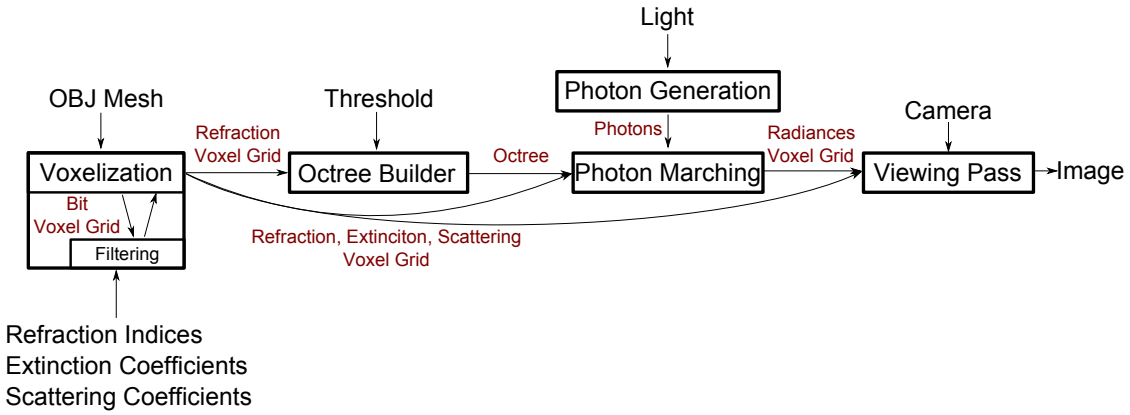


Figure 3.1: The rectangles are the five passes in the pipeline. The data passed between the passes is displayed next to the arrows. The only data not being a 3D array is the output of the photon generation pass which is a list.

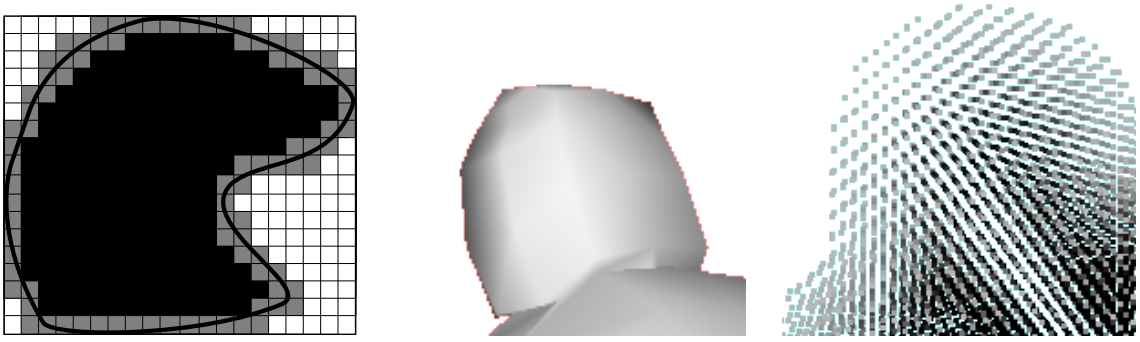
cubic volumetric piece of the scene. These voxel grids are the basic data structures the whole rendering pipeline operates on. Every pass except the photon generation uses these grids in one or the other way.

As already mentioned, there are five passes which make up the whole pipeline. The passes are assembled as shown in Figure 3.1.

Voxelization

In the first pass the voxel grids are built. We have voxel grid for the refraction indices, scattering and extinction coefficients. In our implementation we restrict ourself that a scene consists of two parts, an object, for example a sphere, and surrounding air. Each has a separate refraction index, scattering and extinction coefficient. In the voxelization pass we determine which voxels are inside the object, which in the air and which are on the border (Figure 3.2). Border voxels should build a smooth transition between the inside and the outside of the model. Then, we can assign each voxel grid the values of the data it is supposed to represent. In the case of refraction indices, we assign the voxel which are inside the model the refraction index of the model, for the voxel outside the model the index of the air, and for border voxel the appropriate interpolated index. This is reasonable because the border voxels contain partly the interior of the object and partly the exterior. Hence, the refraction index of border voxels should lay somewhere in between. Furthermore, smoothly varying refraction indices are required in order to apply the differential refraction equation (Equation 2.7). For scattering and extinction coefficients we proceed in exactly the same way as for the refraction indices.

An important fact is that by assigning a refraction index to the model and to the air it is no longer necessary to have the model available for the successive passes. The model is implicitly represented through the refraction indices. This implicit representation is used for the reflection in the viewing pass.



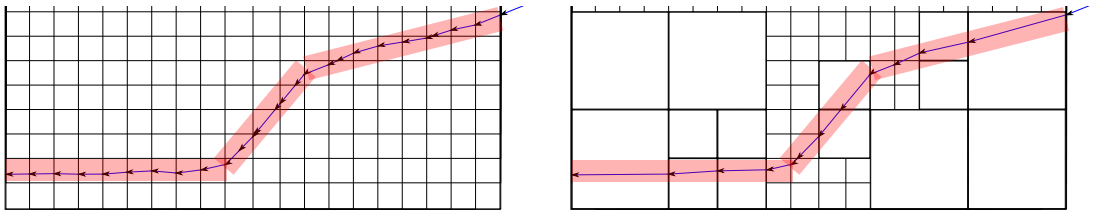
(a) 2D Example (b) Model (c) Voxel Grid

Figure 3.2: The voxelization pass determines which voxels are inside the model (black), in border regions (gray) and outside the model (white). (a) A 2D example of the voxelization pass. The black continuous shape is the border of the model. (b) The model is given through a watertight mesh. (c) The resulting voxel grid of the model in (b). Each dot represents the center of a voxel.

Octree

The next step is to build an acceleration structure for the photon marching pass that is based on the refraction indices. We observe that photons moving through an area of nearly constant refraction indices advance approximately on a straight line (Figure 3.3). So, it is not necessary to advance a photon from one voxel to the next to obtain the path of the photon. It is possible to move a photon on a straight line through a whole region of nearly constant refractive indices. This results in fewer steps which in return accelerates the pipeline.

For our acceleration structure we use an octree that groups regions of nearly constant refraction indices.



(a) Voxel to voxel path (b) Octree node to octree node path

Figure 3.3: (a) Apparently, the path through the refractive volume is approximately piecewise linear. This is because the refraction indices in these regions are nearly constant. (b) The path remains very similar when taking larger steps in these regions (b).

Photon Generation

Before we can trace photons we obviously need to generate the photons first. This is done in the photon generation pass. The current implementation allows only one point light source. The photon generation pass can be easily extended to include multiple and different kinds

of light sources. We make sure that we only generate photons that are very likely to pass through the voxel grid.

Photon Marching

With the list of photons, the grid of refraction indices, extinction and scattering coefficients we march each photon stepwise through the voxel grid. We use the octree to determine the step size which reduces the number of steps a photon needs to traverse the grid. In addition to the voxel grids that represent the scene we also have a voxel grid that is used for storing the illumination. This illumination voxel grid also covers the same region as the voxel grids for the refraction indices, scattering and extinction coefficients. While marching a photon through the voxel grid, we update the radiance in each voxel a photon passes. After the photon marching pass we computed a voxel grid that holds the illumination of the region covered by the voxel grid.

When marching the photons through the gird we have to attenuate the energy of each photon after each step due to absorption and out-scattering. Since we are only interested in rendering single scattering we do not need to change the direction of the photons because of scattering events.

Viewing Pass

The last pass is the viewing pass. This is basically a ray marcher that is used to visualize the voxel grid holding the illumination data that was computed in the photon marching pass. We shoot rays from the viewer towards the grid and march the rays stepwise through the voxel grid. This is done in the same way as we marched the photons in the previous pass except that the octree is not used for determining the step size. We move the ray with a step size of one voxel because we do not want to miss the contribution of any voxel.

In each step we look up the radiance of the voxel in which the ray currently is and evaluate the scattering process. This leads to an image showing single scattering because in the photon marching pass we move the photons through the voxel grids but do not change direction due to scattering. The same thing happens in the viewing pass. The evaluation of the scattering process after each step in the viewing pass connects the light paths through exactly one scattering event, which leads to single scattering. Figure 3.4 illustrates this. Apart from evaluating the scattering process, we also have to update the transmittance for the ray in each step in order to respect extinction. Once the ray leaves the grid, we perform a simple environment map look up.

Additionally, there are two cases where we want reflections. One is the floor and the other is the surface of the model. It was mentioned that the model is implicitly represented through the refraction indices. To define the surface of the model the user chooses a refraction index. We assume that the surface of the model is where the refraction index in the voxel grid equals the user give index. The reflective surface of the object is henceforth called the isosurface because the refraction indices on the surface are constant.

In the whole rendering pipeline, whenever we read data from a voxel grid we apply trilinear interpolation to obtain smooth results. This interpolation is provided by CUDA when we access a voxel grid as a 3D texture. Accessing the voxel grids as textures is wise because CUDA has also a caching mechanism for textures which makes the access faster than a

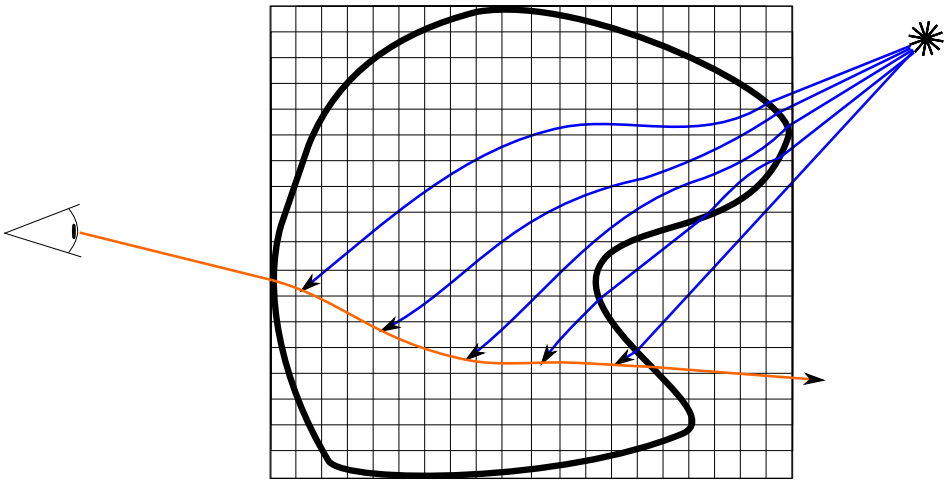


Figure 3.4: The black shape is the border of the refractive object. Blue are the paths of the photons and orange is the path of one viewing ray. At every intersection a scattering event is evaluate. Thus leading to an image that shows single-scattering.

regular global memory lookup.

A convenient feature of this rendering pipeline is that it is not necessary to execute the whole pipeline for every frame. Depending on how the user changes the settings, only parts on the pipeline have to be executed. In Figure 3.1 it can be seen where the different user inputs are inserted into the pipeline. Only when a parameter changes is it necessary to execute the passes afterwards. For example, we only need to execute the viewing pass when the camera settings are changed. The voxel data can be reused because the lighting did not change. However, when the light is modified then the photon generation, photon marching and the viewing pass need to be executed. When the model, refraction indices, extinction or scattering coefficients are changed the whole pipeline needs to be executed.

This concludes the high level overview of the reference pipeline. In the following we look at each pass in more detail and with more technical aspects.

3.1.2 Voxelization

The first step in the voxelization pass is to determine which voxels are inside, on the border and outside of the refractive object. In a second step we assign each kind of voxel different refraction indices, scattering and extinction coefficients.

The first step of the voxelization pass is based on the voxelization technique introduced by Eisemann and Décort [ED08]. The basic idea of their algorithm is to interpret each bit of an RGBA framebuffer, where each color channel is a 32 bit integer, as a separate slice of a voxel grid (Figure 3.5). When a bit is set to one then the corresponding voxel is interpreted to be inside the object we want to voxelize. This allows us to voxelize an object into a voxel grid with the size of 4096x4096x1024 in one rendering pass. The reason being that the maximal resolution of a framebuffer is currently 4096x4096 and the maximal render targets is limited to eight framebuffers while each framebuffer holds 128 bits RGBA values, 32 bit for each color. This resolution is more than enough for our purpose. Usually, a voxel grid of 128x128x128 is sufficient. Besides, larger voxel grids are problematic for our rendering pipeline because of memory limitations.

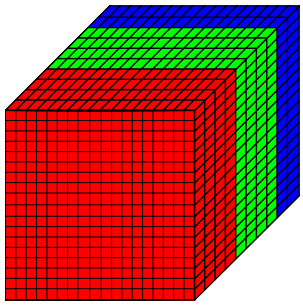


Figure 3.5: This illustrates the idea of mapping bits of the color channels to slices of a voxel grid. In the illustration each color channel has only four bits. In reality each channel has 32 bits. Thus representing 32 slices of a voxel grid. In OpenGL a texture may hold RGBA values, therefore, representing 128 slices.

The algorithm voxelizes a watertight 3D mesh with the use of a single fragment shader. Therefore, to voxelize an object, the object only needs to be rendered with the voxelizer shader attached to the OpenGL pipeline. The width and height of the voxel grid is defined by size of the framebuffer used for rendering. The depth of the grid can be passed to the shader as an argument.

The shader uses the depth coordinate `gl_FragDepth` of a fragment to determine in which slice the fragments resides. More precisely, it determines the bit that represents the slice. When the shader sets these bits to one for all fragments of an object we would voxelize only the surface of the model. However, the goal is to voxelize the interior of the model. This can be done by a simple modification. The shader not only changes the bit of the current fragment but every bit that represents a slice closer to the camera. We compute "1 XOR bit in the framebuffer" for every bit that is closer to the camera. This can be done with `glLogicOp(GL_XOR)`. Since we have a watertight model, this technique sets all bits to one that are inside the object. Figure 3.6 illustrates this with a simple example.

Back to the voxelization pass, the first step of the voxelization pass is to use the described algorithm to compute the interior of the model. The algorithm sets a bit to one when the corresponding voxel is inside the object and zero otherwise. However, in Section 3.1.1 we said that we want to have a smooth transition between the interior and the exterior. Here, we follow the idea of Sun et al.. We look at a 3x3x3 neighborhood around each voxel. When all voxels are either on the inside or the outside we keep the value. If the 3x3x3 neighborhood has varying values, which means it is a border voxel, then we make use of a higher resolved voxelization. This means we not only voxelize the object with the resolution of the voxel grid we are going to use in the later passes but also with a four times higher resolution. Sun et al. figured that a four times higher resolved grid is optimal. Now, if we discover a border voxel then we build the average value of the bit data representing this region from the high resolution voxelization. This gives us fractional value for border voxels. Figure 3.7 illustrates this process.

The next step is to assign each voxel a refraction index, scattering and absorption coefficient. This is done by assigning voxels with the value zero the data for the air, voxels with value one the data for the object and for the border voxels interpolated data according to the voxel value. The refraction index voxel grid is then filtered by a 9x9x9 gauss filter that has a standard deviation of nine sixth to obtain smoothly varying refraction indices. Eventually,

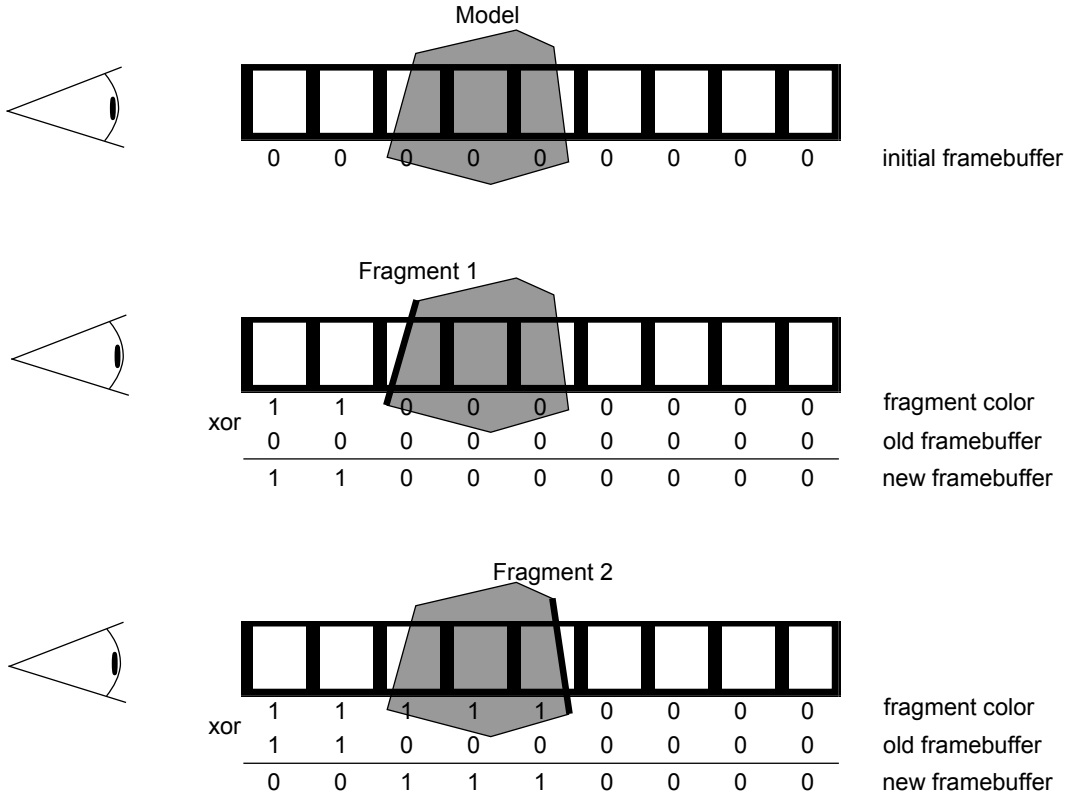
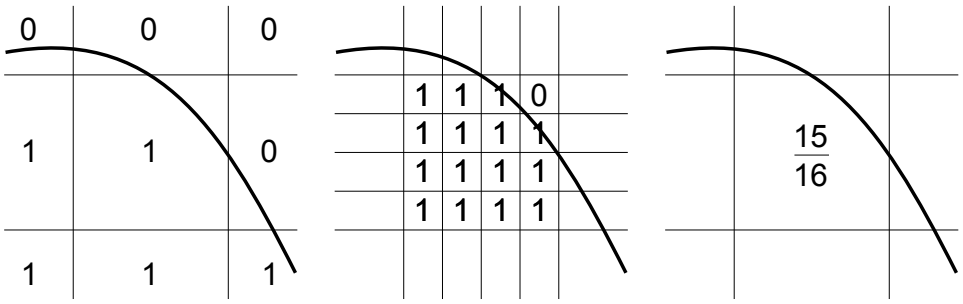


Figure 3.6: This illustrates how the XOR operation leads to the voxelization of the interior of the gray model. For simplicity, only one pixel of the whole framebuffer is displayed here.



(a) Low resolution neighborhood

(b) High resolution lookup

(c) Average of high resolution

Figure 3.7: It is desired that border voxels have fractional coverage values. The value should tell how much of the voxel is on the inside of the object and how much on the outside. (a) First, the border voxels need to be found. When not all of the neighborhood of a voxel have the same value then it is a border voxel. (b) Then a lookup in the high resolution voxelization data is performed to compute a more accurate coverage value (c).

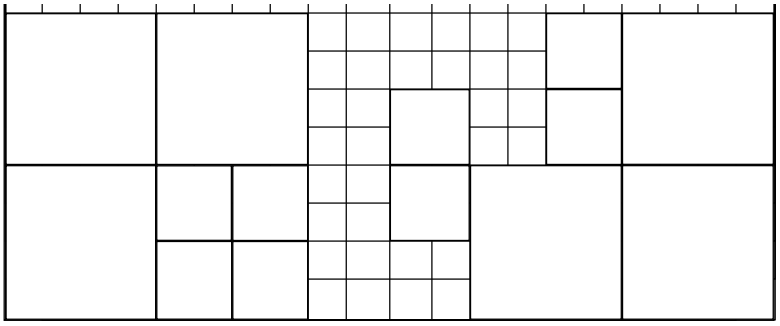
we build the gradient array from the refraction indices. The gradients are needed for the computation of the photon and ray paths (see Equation 2.11).

Technically, the voxelization pass is partly OpenGL and partly CUDA. The voxelization according to Eisemann and Décort is purely OpenGL. The fragment shader (Listing A.1) renders into an OpenGL texture whose data is then copied to a Buffer Object in order to make it available to the CUDA kernels . This has to be done twice because we need a low and high resolution voxelization to compute the border voxels. The remaining tasks are realized as CUDA kernels that operate on the Buffer Objects and linear CUDA memory. Eventually, all data is copied to `cudaArrays` so that the successive passes can access the data as textures. This way we can use the trilinear interpolation that is provided by CUDA.

As for the thread arrangement, we launch the kernels such that each thread handles a single voxel.

3.1.3 Octree

For the octree construction we use the same technique as Sun et al.. Usually, an octree is implemented using a structure that represents a node and has pointers to its children. This sparse representation of an octree is not optimal of our purpose because this way it is difficult to parallelize the octree construction. Instead, we simply use a three dimensional array with the same size as the voxel grid for storing the octree level to which a voxel belongs. Figure 3.8 illustrates the interpretation of the octree array.



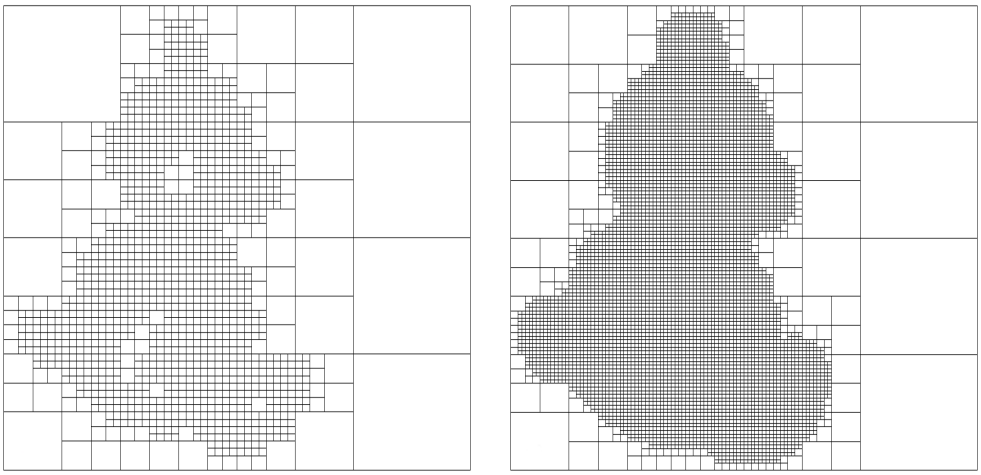
(a) Sparse octree

2	2	2	2	2	2	2	2	0	0	0	0	0	0	1	1	2	2	2	2
2	2	2	2	2	2	2	2	0	0	0	0	0	0	1	1	2	2	2	2
2	2	2	2	2	2	2	2	0	0	1	1	0	0	1	1	2	2	2	2
2	2	2	2	2	2	2	2	0	0	1	1	0	0	1	1	2	2	2	2
2	2	2	2	1	1	1	1	0	0	1	1	2	2	2	2	2	2	2	2
2	2	2	2	1	1	1	1	0	0	1	1	2	2	2	2	2	2	2	2
2	2	2	2	1	1	1	1	0	0	0	0	2	2	2	2	2	2	2	2
2	2	2	2	1	1	1	1	0	0	0	0	2	2	2	2	2	2	2	2

(b) Octree array

Figure 3.8: Two different representations of the same octree section. (a) Shows the sparse representation of an octree. This is not suitable for parallel construction. (b) By using an array that stores to which level a cell belongs to it is possible to parallelize the construction.

Before we can start with the octree construction, the user has to define when we regard refraction indices to be nearly constant. This is realized with a threshold ϵ. When the difference of the refraction indices is smaller than the threshold we say they are nearly constant. This way the user can influence the size of the octree nodes. Figure 3.9 shows two octrees of the same scene for different thresholds.



(a) $\epsilon = 0.1$ (b) $\epsilon = 0.001$

Figure 3.9: The figure shows a projection of the 3D octree based on the refraction indices once for a threshold $\epsilon = 0.1$ and once for $\epsilon = 0.001$. It can be seen that the larger the threshold the larger the regions that are regarded as having nearly constant refractive indices.

Listing 3.1: Octree Pseudocode

```
// refraction pyramid
FOR i = 0 TO maxLevel
    // executed for each cell of level i
    buildMinMaxRefrLevel(i)

// octree level pyramid
FOR i = maxLevel TO 0
    // executed for each cell level i
    buildOctLevel(i)
```

The construction uses a pyramid approach to build the octree. The pyramid is similar to a mipmap pyramid. We use two pyramids, one storing information about the refraction indices and one holding the octree levels. We first build the refraction pyramid bottom up and then the octree level pyramid top down (Listing 3.1). Since the octree should group regions with nearly constant refraction indices, we store in each pyramid level the maximal and minimal refraction index covered by a cell. The minimum and maximum on the lowest level, the input, is always the same because there is no lower level. This way we can build the pyramid iteratively and highly parallel from bottom to top (Figure 3.10).

Listing 3.2: Refraction Pyramid Pseudocode

```
buildMinMaxRefrLevel(i)
{
```

```
    maxIdx = -Infinity
    minIdx = Infinity
    FOR all chlid nodes DO
      minIdx = min(chlid minIdx, minIdx)
      maxIdx = max(chlid maxIdx, maxIdx)

    store minIdx and maxIdx on level i
}
```

For building the refraction pyramid, we create a thread for each cell of the level we want to build. Each thread has to lookup the eight values of the previous level and find the maximum and minimum and store them in the cell of the level to build. The pseudocode can be found in Listing 3.2. This is done until there is only one cell in the level, which is the highest level or the tip of the pyramid.

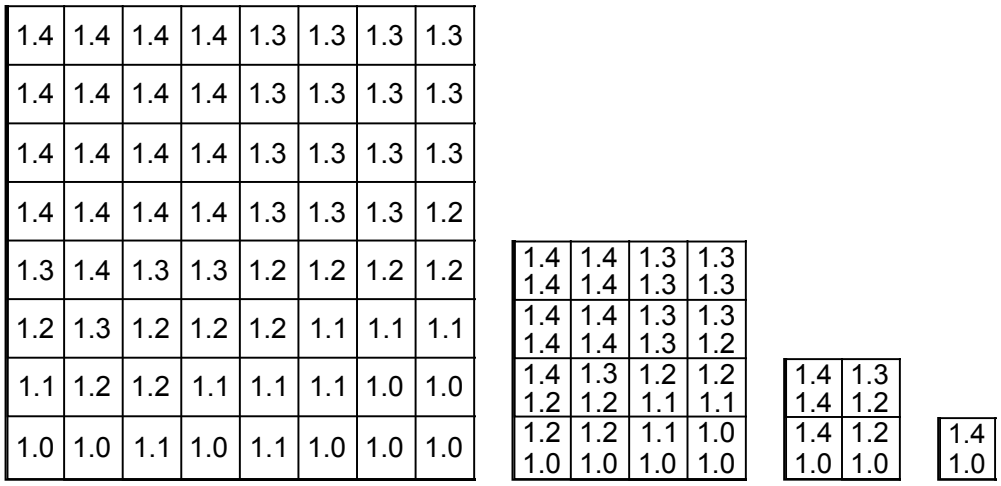


Figure 3.10: The refraction pyramid stores the minimal and maximal refraction index of the previous levels. The construction is bottom up. Thus, it begins with the array on the left where the maximum and minimum are the same. Iteratively the construction stores in each cell the maximum and minimum of the covered region.

Listing 3.3: Octree Level Pyramid Pseudocode

```
buildOctLevel(i)
{
   IF parent octree level value = 0
      IF parent maxIdx - parent minIdx < epsilon
         store i on level i
      ELSE
         store 0 on level i
   ELSE
      store parent octree level value on level i
}
```

Then we use the second pyramid that stores which cell belong to which octree node. This pyramid is constructed top down. For each cell we determine the octree level to which it belongs to by comparing the refraction indices from top to bottom. To build the tip we simply compare the maximum and minimum refraction index of the highest level of the

refraction pyramid. If the difference is smaller than the user provided threshold ϵ we write the number of the largest level into the cell of the octree data. Otherwise we write zero. Now the iteration starts. To build the next lower level we create a thread for each cell of this level. If the value of the parent of a cell is not zero we write the same value as the parent into this cell. This guarantees that in all child nodes keep the octree node of their parents. When the value of the parent is zero, we compare the maximum and minimum refraction index of the corresponding cell in the refraction pyramid on the same level as the one we want to build. As before, if the difference is smaller than the user provided threshold ϵ we write the number of the current level into the cell. Otherwise we write zero. The pseudo code is in Listing 3.3. We repeat this until we reach the lowest level which is the octree we use in the photon marching. Figure 3.11 shows a sample pyramid.

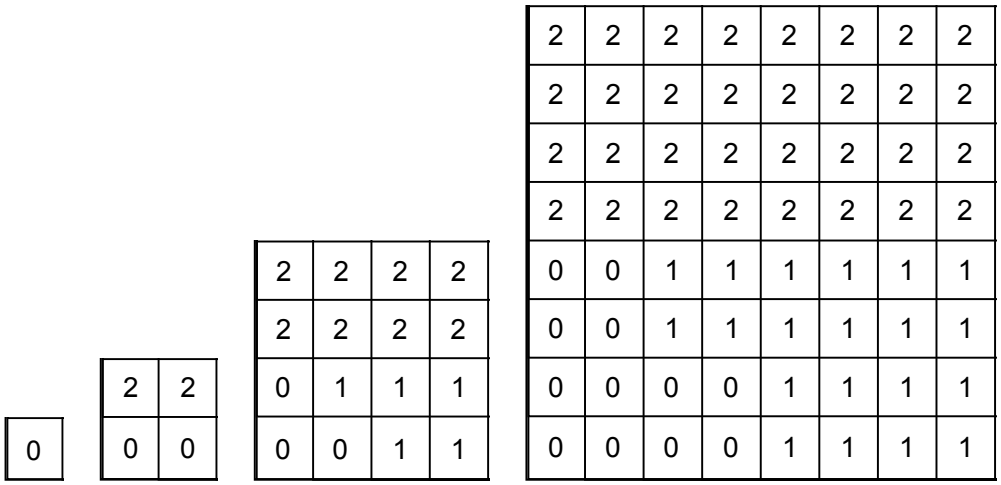


Figure 3.11: To build the octree array, the construction starts at the tip. In this example the data from Figure 3.10 is used with a user threshold $\epsilon = 0.15$. First the tip of the refraction pyramid is used to see is if the maximum and minimum differ by less than ϵ. Since it does not the tip is 0. In the iteration the parent node is checked if it holds a non zero value. If so, store this value in the cell. If not, check in the corresponding cell in the refraction pyramid if the maximum and minimum differ more or less than the threshold. If less then write the current level to the cell, else zero.

Technically, the whole octree construction is implemented in two CUDA kernels. One for building the refraction pyramid bottom up and one for building the octree level pyramid top down. As for the memory layout, we simply use two arrays of arrays. One for each pyramid and each array contains arrays of linear memory for each level. See Listing A.2 for CUDA code excerpts.

3.1.4 Photon Generation

The goal of this process is simply to generate a list of photons. The data that is required by the photon marcher is the initial position, direction and energy of the photons. To generate this list we place a square between the light source and the scene such that the square covers the whole voxel grid when looking from the light source towards the grid. We call this square

photon plane. We create the photons on this photon plane. This guarantees that we only create photons that potentially hit the grid.

We distribute a user given number of photons (`photon_count`) on the photon plane. In order to distribute and, therefore, compute the initial position of the photons we use a 2D Hammersley sequence. Basically, any uniform sampling technique could be used here but the Hammersley sequence turned out to produce the best visual and performance results (Section 4.1). Wong et al. discuss the Hammersley sequences and give a reference implementation [WLH97]. When a photon has a position the direction can also be computed. The energy of each photon is

$$\frac{light\_power}{photon\_plane\_area \cdot photon\_count}.$$

Depending on the type of light we have to adjust the color slightly. For example, for a point light source we have a multiplication with a cosine term because of the projection onto the sphere.

The next step is to compute the intersection of the photons and the voxel grid. This is done because it is assumed that nothing is between the light and the grid. Hence, when a photon intersects with the grid we store the intersection location as the photons initial position. If a photon and the grid have no intersection we discard the photon. The direction and energy is unchanged because nothing is between the square and the voxel gird. Figure 3.12 illustrates the photon generation.

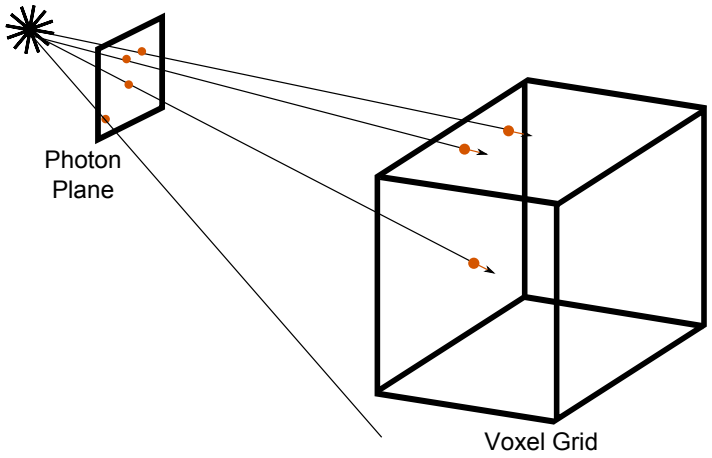


Figure 3.12: The photons are represented as ocher dots. First the photons are distributed on the photon plane. Then the photons are move forward such that their initial position is on the voxel grid.

Technically, the list is in fact three CUDA arrays. One array represents the position, one the propagation direction and one the color of the photons. Each array consists of as many elements as there are photons to be generated. This way we can parallelize the generation of the photons easily. We create a thread for each photon. Each thread has its own unique index. This index is used as the index in the photon list. Thus, each thread has a reserved position in the photon list where it writes the initial position, direction and energy of its photon. Listing A.3 shows the CUDA kernel of the photon generation pass.

3.1.5 Photon Marching

The goal of the photon marching pass is to compute the illumination in the participating media. This illumination is stored in a voxel grid that stores the radiance distribution for each voxel. How this data is stored is explained in the "Radiance Representation" paragraph. To compute the illumination we march photons stepwise through the voxel grid. How this is done is described in the "Photon Path Computation" paragraph. As mentioned earlier, we use the octree to advance photon further than a voxel at a time. Thus, we need to determine which voxels a photon jumped over in order to make sure that all voxels a photon passed receive energy. To solve this problem we implemented a 3D rasterizer which is discussed in the "3D Rasterizer" paragraph.

Technically, an important issue is the thread synchronization for writing data into the voxel grid. We launch the CUDA kernel for photon marching such that there is a thread for each photon. This means that it might happen that two threads try to write to the same voxel at the same moment. Thus, we use the atomic operations provided by CUDA for writing into the arrays. The problem with these atomic operations is that they noticeably slow down the kernel execution. The reason is that when there is a conflict, threads have to wait for each other. The more atomic operations there are the more the thread execution is serialized. Therefore, it is wise to try minimizing the number of atomic operations. Hence, we use a temporary 64 bit integer array to pack the color and direction data. This way we can reduce the number of atomic operations to one 64 bit compare and swap operation. The trick is to pack the data as described in the "Data Packing" paragraph.

As Sun et al. propose, we apply a 3x3x3 gauss filtering for smoothing the voxel grid holding the illumination data after marching the photons through the grid.

Radiance Representation

The photon marching pass generates a voxel grid that stores the radiance distribution for each voxel. Every time a photon passes a voxel it deposits its energy and updates the radiance distribution accordingly. Sun et al. argue that the radiance distribution in a voxel can be approximated by two values. The first, we store the sum of the energy of all photons that passed through the voxel. The second, we store a luminance weighted average of all the directions of all photons that passed through the voxel (Figure 3.13). This is obviously a very coarse approximation of the true radiance distribution in a voxel. Instead of storing an energy for each direction we only have the sum of all energies and one average direction. Yet, Sun et al. argue that this approximation is sufficient for our rendering purposes because it does not exhibit artifacts.

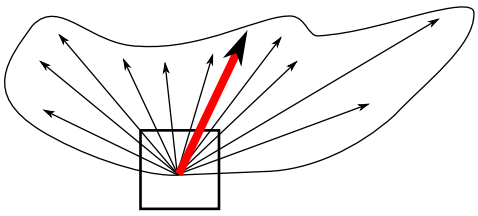


Figure 3.13: Instead of storing the whole radiance distribution in a voxel, only a weighted average direction (red bold arrow) and the summed energy is stored.

Photon Path Computation

For the computation of the photon paths we need the voxel grids holding the refraction indices, their gradients, extinction coefficients and the list of photons. The paths are then give by the Equation 2.10 and Equation 2.11 where x_0 is the initial photon position and v_0 the initial photon direction. The only undefined variable is the step size Δs. As already mentioned, we use the octree to determine Δs. We would like to move a photon in one step over a region of nearly constant refraction indices, which corresponds to an octree node. In each step we lookup the current octree level and compute the step size such that a photon reaches to border of this octree node (Figure 3.3b). In fact we move the photon a little bit further into the next octree node, right over the border, to avoid additional iterations due to floating point imprecision.

An issue that needs to be addressed is the attenuation. The energy of a photon needs to be reduced due to extinction in each step. This is done by assuming the extinction coefficient is constant over the region that the photon moved. This assumption, however, leads to an error because the step size is based on the refraction indices. Thus, it is possible that we ignore fast local changes in the extinction coefficients. It would not be hard to include the extinction coefficient in the octree construction but Sun et al. argue that the visual impact of computing the extinction more precisely is not worth the performance loss

3D Rasterization

Our 3D rasterizer is similar to the Bresenham algorithm. We are given two 3D positions in the voxel grid, two color values and two directions; always one for the `start` and one for the `end` point. The rasterization is determined by the two positions. The algorithm works in the following steps:

1. Determine the `axis` of the largest change.
2. Determine the `stepSize` along the line between `start` and `end` such that we move exactly one voxel in the direction of `axis`.
3. Determine how many `steps` we have to take without rasterizing the end voxel (for example when `axis` is the x-axis `steps = floor(end.x) - floor(start.x)`).
4. Move `start` and `end` along the line between `start` and `end` such that they are in the middle of the voxel along `axis`.
5. If we moved the `start` point backward, i.e. away from `end`, then do not rasterize the first voxel, i.e. move the start position by `stepSize` along the line and reduce `steps` by one.
6. If we moved the `end` point backward, i.e. towards the `start`, then we rasterize last voxel. Thus `steps` is increased by one.
7. Move `steps` times along the line with `stepSize`.

This gives us the voxels into which we have to update. Obviously, we have to interpolate the color and direction between the start and end values. Otherwise, we ignore the attenuation and the change of direction between the `start` and `end` position. Figure 3.14 illustrates the rasterization.

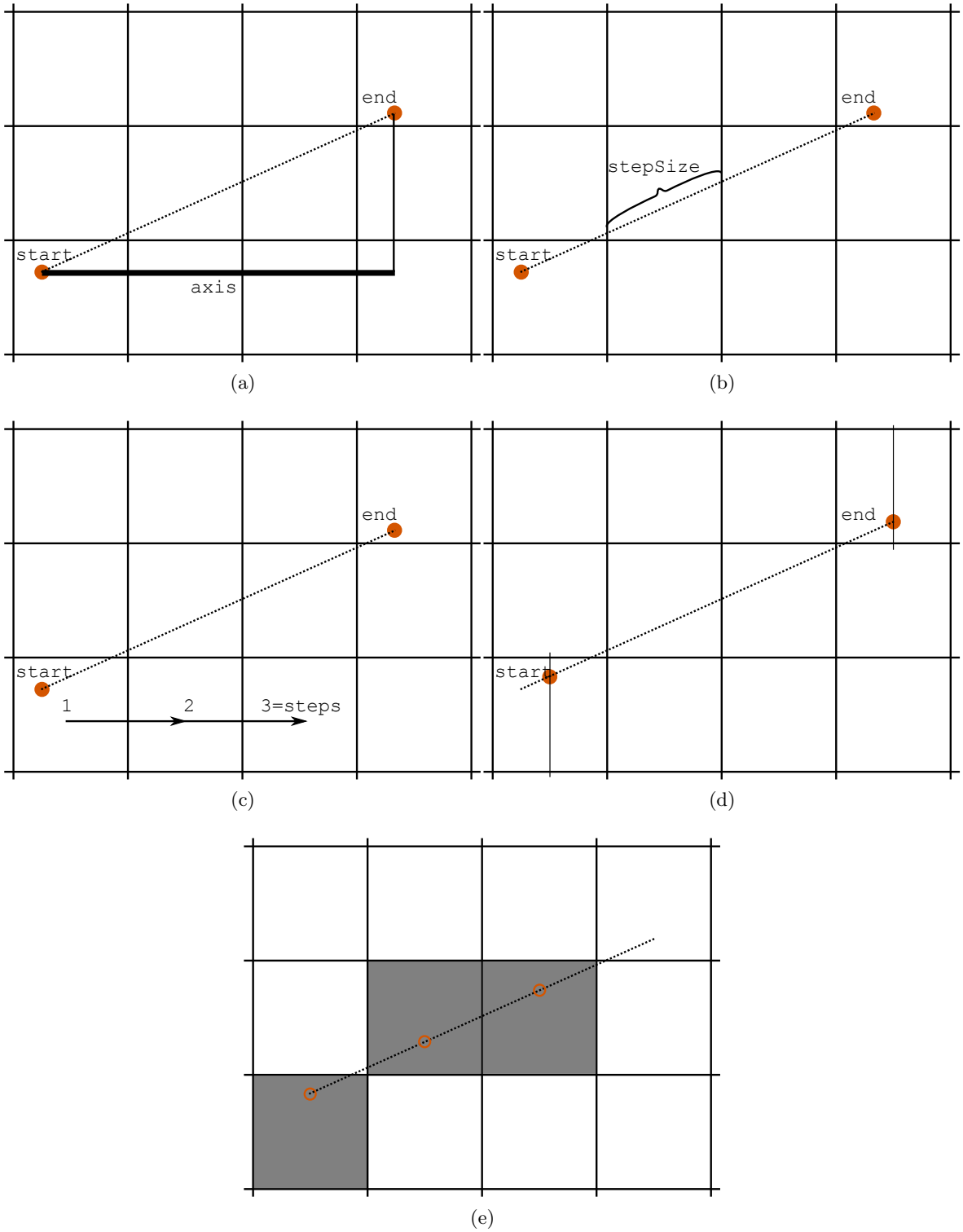


Figure 3.14: (a) - (c) illustrate `axis`, `stepSize` and `steps`. In (d) both `start` and `end` are moved forward to the middle of their voxel. Therefore, the end voxel is not rasterized but the start voxel is. The iteration (e) moves the start `stepSize` forward `steps` times. Each voxel that is touched is rasterized.

Data Packing

CUDA offers 32 and 64 bit atomic integer operations. Thus, to reduce the time consuming atomic operations we pack the color and direction data into one 64 bit integer. We use 32 bits for the three color values RGB and 32 bits for the three coordinates of the direction.

For the RGB data we use the RGBE data format that was invented by Greg Ward [War91]. It allows us to store precision floating point RGB data in 32 bits. RGBE is necessary because we cannot use a simple linear mapping to integers. It often happens that a single photon has very little energy, so little that it would not change the integer value of a voxel. Furthermore, when using linear mapping to integers we have to decide what the largest value is that we want to represent. Thus, saturation issues may occur. The RGBE format does not have these limitations.

For the three direction components a linear mapping to integers of 10 bit precision is suitable. This way we pack the six components into one 64 bit atomic write operation.

Unfortunately, we cannot use an atomic addition because of the nonlinearity of RGBE. This makes it necessary to unpack the data of a voxel in order to change it. Thus, we should be able to define a critical section where only one thread is active. This section would consist of reading the data, unpacking, modifying, repacking and writing it back. However, implementing locks is not advisable because of performance loss due to synchronization issues. As a result, we decided to follow an optimistic approach. Meaning that the data is read, unpacked, modified and repacked. Then we hope that the data in the voxel has not been changed by an other thread in the meantime. When we discover that the data has been manipulated we start all over again. This can be done with the atomic compare and swap operation. To prevent a thread from waiting a long time trying to write the data, we set a maximum number of retries. When a thread reaches this limit we simply give up trying to write the data into the voxel. This introduces a possible error but it turned out that with a maximum of ten attempts there are no more visual artifacts.

This packing of data is absolutely crucial to achieve real-time frame rates. We measured a five to ten fold performance increase with the packing technique compared to writing each of the six components in separate atomic operations.

3.1.6 Viewing Pass

The viewing pass is basically a regular ray marcher which displays the illumination voxel grid that was generated in the photons marching pass. To enhance the visual effect of the refraction we include an environment map. Besides the refraction and participating media we also include two reflections, once a reflective table and once a reflective surface of our refractive model. This lets the scene look more natural.

In our implementation we assume that we only have the voxel grid and an environment map as our scene. Therefore, we first perform an intersection test with the voxel grid to determine how to handle a ray. When a ray does not intersect with the grid, we execute an environment map lookup. If a ray intersects with the voxel grid, we move the ray to the intersection location. Then we start marching the ray with the step size of the size of a voxel. This way we do not miss any voxel and, therefore, any details. When a ray reaches the end of the grid, we again carry out an environment map lookup. The following paragraph discusses the ray marching in more detail.

Ray Marching

The ray marching is summarized in Listing 3.4.

Listing 3.4: Ray Marching Pseudocode

```
while ray inside voxel grid {
   // reflective table
   if ray entered bottom voxels {
      compute exact intersection
      set ray direction to reflected direction
   }

   // reflective isosurface
   else if ray crossed isosurface {
      compute exact intersection with bin search
      compute interpolated direction at intersection
      environment map lookup
   }

   update transmittance
   evaluate scattering
   update ray position and direction
}
environment map lookup
```

To determine the path of ray through the voxel grid we use Equation 2.10 and Equation 2.11, just as in the photon marching pass. It is important that we remember the previous position and direction because we require this information for the exact intersection computation for the reflections. The following list gives more details for the separate task that are performed in the ray marching.

Reflective Table Check if the ray hit a voxel at the bottom. If so, compute the exact intersection with the voxel border, move the photon to the intersection and change the direction of the ray to the reflected direction.

Reflective Isosurface Compare refraction index of the previous position with the refraction index of the current position. If the previous index was smaller and the current index larger than the user specified isosurface index then we know that the ray crossed the isosurface. To determine the exact location of the isosurface we perform a binary search between the previous and current position. Since the isosurface is given by the refraction indices, we can use the gradients of these to obtain the normal of the isosurface. The gradients were computed in the voxelization pass. Finally, we execute a lookup with the reflected ray in the environment map. It would also be possible to continue to march the reflected ray through the grid to include participating media and refraction in the reflection. However, it turned out that the visual impact in not worth the effort. If the object is partly reflective and partly refractive we execute the texture lookup for the reflection and continue marching the ray.

Transmittance Update In order to respect attenuation we update the transmittance in each step. We make the assumption that the extinction coefficient stays constant along the line we moved the ray. This is a simplification but eases the computation and does not produce any noticeable artifacts. To update the transmittance we can make use of the multiplicative property of the transmittance.

Scattering Evaluation Since we approximate the radiance with only one direction, we lookup this average direction for the current position. Then we evaluate the phase function for the average direction and the incident ray direction. The value of the phase function is then multiplied with the average color of the voxel. The result is an approximation of the inner integral of the volume rendering equation. Therefore, to account for single-scattering all that remains to be done is a multiplication of this value with the scattering coefficient.

Technically, one important issue is the CUDA launch configuration. First of all, for each ray one thread is created. As for the arrangement of the threads, we realized that the viewing pass shows a better performance when a CUDA thread block represents the rays of a more square-like subregion of the final image than when a block computes a row or a column of the image. Hence, we do not split the image row or column wise. We chose that a block represents a 8x16 pixel region of the final image. Our explanation for the performance difference is that rays of neighboring pixels follow a similar path. Therefore, the threads of a block take the same path in `if-then-else` statements. This leads to blocks where threads often execute the same instruction. Thus, less thread serialization occurs per Streaming Multiprocessor. Furthermore, since we use textures to lookup the data we benefit from texture caching. Such a behavior leads to faster kernel execution according to [NVI09b, NVI09a]. The kernel for the viewing pass can be found in Listing A.4.

3.2 Adaptive Rendering Pipeline

With the adaptive pipeline we tried to implement a rendering pipeline that produces images with the same quality as the reference but does this in shorter time. We figured that shooting less photons should accelerate the rendering pipeline. Therefore, we reduced the number of required photons. As our evaluation shows the adaptive pipeline has a better performance only when the the reference pipeline uses more than 200'000 photons (Section 4.1).

To reduce the photon count we make the following observation. In the reference pipeline, the distribution of the photons is not optimal in the sense that we sample regions too tightly where the radiance in the voxel grid ends up the be nearly constant. We call these regions homogeneous regions (Figure 3.15). In these regions only a few photons and a appropriate filtering should produce the same result as shooting many photons. Therefore, our goal is to sample these regions with less photons. This leads to an adaptive photon sampling.

In the remainder of the section we first present an overview of the adaptive rendering pipeline. After the overview we discuss the pipeline in more detail.

3.2.1 Overview

The adaptive pipeline reuses a large portion of the reference pipeline. The voxelization, octree and viewing pass are not affected by the adaptive photon sampling. As a result we substitute only the photon generation and photon marching pass from the reference pipeline with new passes.

The main difference between the reference and the adaptive pipeline is that we distribute the photons on the photon plane in the photon generation pass differently. In the reference pipeline we distributed them uniformly. In the adaptive pipeline we want a adaptive sampling such that we shoot many photons through inhomogeneous and few photons through

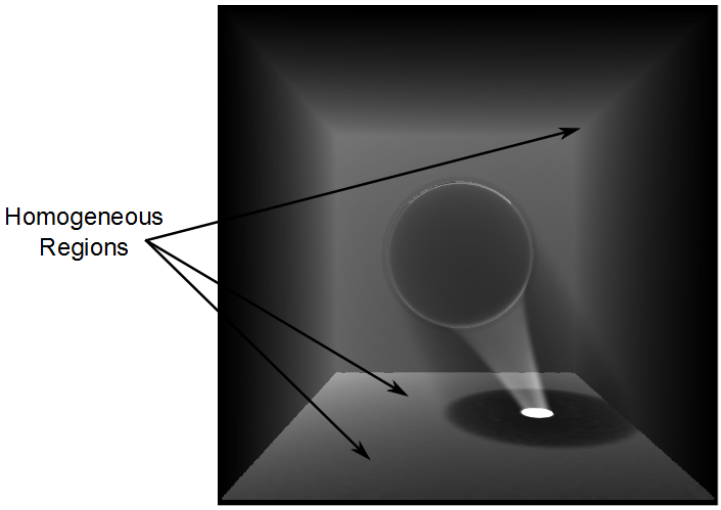


Figure 3.15: In homogeneous regions the sampling of the reference pipeline is very tight. It is possible to reduce the photon density in these regions.

homogeneous regions. Therefore, we first have to determine how to distribute the photons on the photon plane, then we compute the illumination voxel grid and, eventually, we need to reconstruct a smooth illumination distribution in the voxel grid because of the adaptive photon sampling. This leads to three phases.

Instead of the photon generation and photon marching pass we have three phases and each of which consists of several passes. Figure 3.16 shows the new passes and how we group them into the three phases.

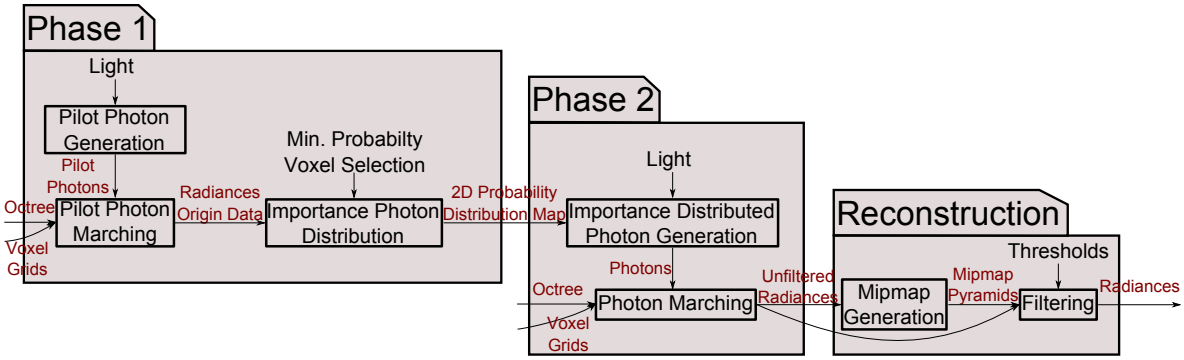
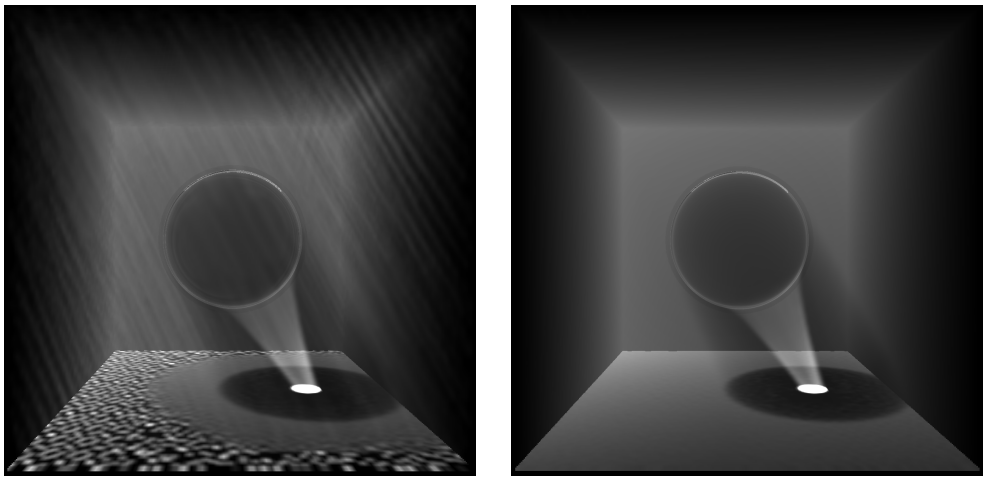


Figure 3.16: In the adaptive pipeline the photon generation and marching pass from the reference pipeline are substituted with the illustrated passes.

The goal of phase 1 is to detect where the homogeneous and inhomogeneous regions are and then build a 2D probability density function that describes how to distribute the photons on the photon plane. Phase 2 uses the 2D probability density function from phase 1 to generate photons accordingly and marches the photons to compute a illumination voxel grid that could be passed to the viewing pass. Due to the adaptive sampling we need to perform some form of reconstruction because the illumination voxel grid exhibits much high frequency noise (Figure 3.17). The reconstruction phase filters the illumination voxel grid from phase 2 such that the grid has no more noise. The following sections describe each phase in detail.



(a) Unfiltered (b) After reconstruction phase

Figure 3.17: (a) Due to the sparse sampling voxels end up empty in the final voxel grid. (b) The reconstruction phase removes the high frequency noise with a smoothing filter.

3.2.2 Phase 1

In phase 1 we are concerned with computing a 2D probability density function that describes how we ought to distribute the photons on the photon plane in the photon generation pass such that we shoot many photons through inhomogeneous and few photons through homogeneous regions. A priory we do not know where these regions are. Therefore, we generate few, so called, pilot photons that are uniformly distributed on the photon plane. We use these pilot photons to detect the inhomogeneous regions. It is crucial that we generate only few pilot photons, roughly 10% of the photons that we use in phase 2, otherwise the pilot photon marching consumes too much time. The pilot photon generation pass fills the photons into the list with an ordering such that we can derive where each photon came form on the photon plane. The pilot photon marching pass builds voxel grids that store this origin data along with the illumination. This way we know for each voxel where to place photons on the photon plane such that the photons will pass through this voxel. In the importance photon distribution pass the user selects the voxels that need to be sampled tightly with two thresholds. Since we know where to place the photons on the photon plane such that the photons pass through the selected voxels, we can build a 2D probability density function that leads to our desired sampling.

Pilot Photon Generation

The pilot photon generation pass is basically the same as the photon generation pass in the reference. The main difference comes from the fact that, in the pilot photon marching pass, we need to know from where the photons in a voxel came. This is done by arranging the photons in a squared grid and adding the photons to the list in order (Figure 3.19). Thus, the photon origin of a photon can be calculated from the photon's list index. Our implementation produces always a square number of photons which is given by the user through the photon count along one axis (`photon_count`). Since the pilot photons are only used for the region

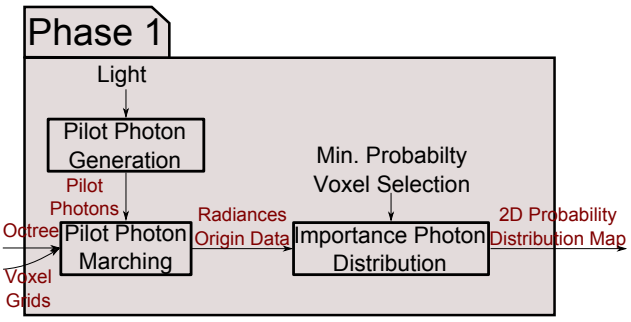


Figure 3.18: Passes of phase 1

detection, it suffices to generate only a few photons; typically around 10% of the photons the we use in the importance distributed photon generation.

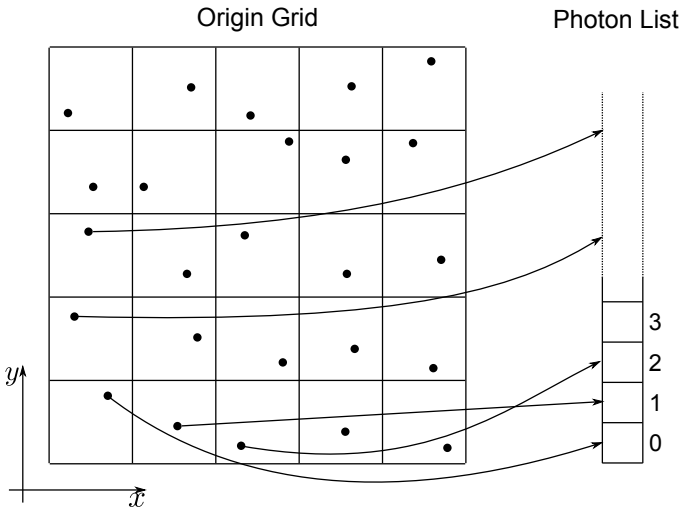


Figure 3.19: The jittered photons are filled into the list such that the coordinates in the grid can be calculated from their list index.

It often happens that photons arranged in a grid produce aliasing artifacts. These artifacts can disturb the region detection. To reduce the artifacts we jitter the photons slightly. However, for the origin coordinates in the pilot photon marching we still us the grid index and ignore the jittering. This is not a problem because the error that is introduced is not big enough to be noticeable.

Pilot Photon Marching

The pilot photon marching is essentially the same as the photon marching in the reference pipeline. The only difference is that we gather more information per voxel, namely the origin data. Therefore, in addition to the 64 bit integer array which we use to store the packed color and direction per voxel, we also have a 64 bit integer array that stores the following packed data:

1. n: The number of photons passing through a voxel (12 bit)

2. $\sum_n photon\_origin_x$,$\sum_n photon\_origin_y$: The sum of the origin grid coordinates of the photons passing through a voxel (15 bit for each the x and y axis)

3. $\sum_n photon\_origin_x{}^2$,$\sum_n photon\_origin_y{}^2$: The sum of the squared origin grid coordinates of the photons passing through a voxel (10 bit for each the x and y axis)

From 1) and 2) we can compute the mean origin coordinate on the photon plane and from 1) and 3) we can compute the variance of the origin coordinates. In the importance photon distribution pass we will use circles to build the 2D probability density function. The mean origin coordinate serves as the center of a circle and the variance and the radius. The reason why we use more bits for 2) than 3) is because it is more important that the circle has the correct center than radius.

The origin coordinates of the photons can be computed easily. As before we create a CUDA thread for each photon. Thus, `threadIdx.x + blockIdx.x * blockDim.x = idx` is the index of a photon when we launch the kernel with a one dimensional block grid. This lets us compute the origin coordinates as follows:

$$\begin{aligned} \texttt{photon\_origin.x} &= \texttt{idx} \bmod \texttt{photon\_count} \\ \texttt{photon\_origin.y} &= \left\lfloor \frac{\texttt{idx}}{\texttt{photon\_count}} \right\rfloor \end{aligned}$$

As one can see, we now have two atomic write operations, one for the packed origin data and one for the packed colors and directions. In both cases we make use of the optimistic approach described in Section 3.1.5.

For the region detection we do not use the voxel grid with the resolution of the final voxel grid because of high frequency noise. This is because we only send very few photons as pilot photons. To reduce the high frequency noise, we use a voxel grid with a lower resolution to store the result of marching the pilot photons. This is the same as rasterizing the photon paths into a voxel grid of higher resolution and applying a smoothing filter. Typically we use a 128^3 voxel grid and for the region detection we only use a 16^3 voxel grid. This makes the treatment of the pilot photon very fast.

Importance Photon Distribution

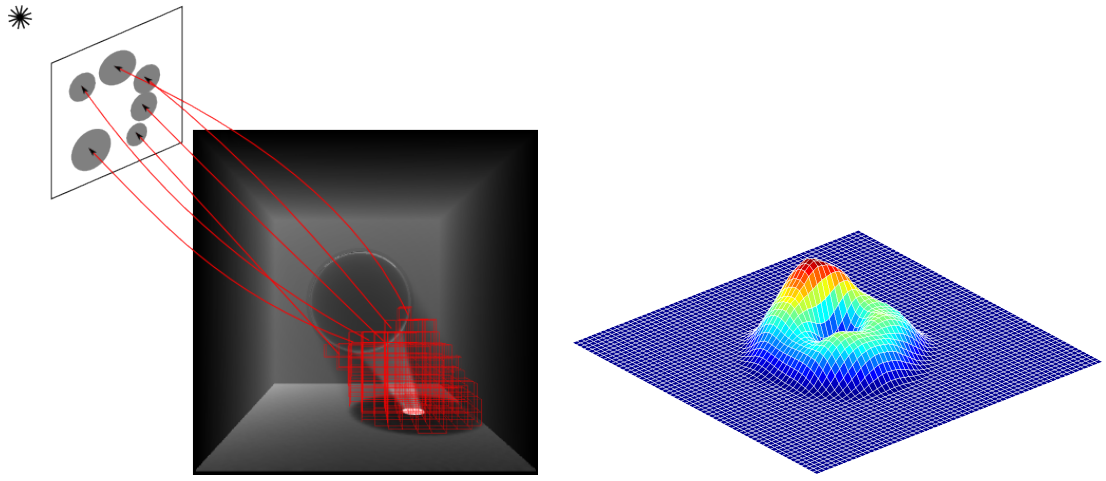
Our goal is to sample regions where the illumination ends up the be homogeneous more sparsely than where it ends up inhomogeneous, like around caustics. As already mentioned, we want to select voxels in inhomogeneous regions and sample these voxels more densely. In this pass we select these voxels by the application of two thresholds.

One threshold is for the color based gradient. We say a voxel needs a tight sampling when its gradient is higher than the threshold. This means that we try to shoot many photons through voxels on the border of caustics and shadows. This is in fact what is desired because we want sharp caustics and shadows.

The second threshold is used simply to select the shadow voxels. When the color luminance of a voxel is lower that the threshold, we select the voxel. This is also desirable because when we try to shoot photons into the shadow we sample regions where photons diverge more tightly. In other words, the borders of the shadows are sampled more tightly.

After selecting the voxels that we like to sample more tightly, we create the 2D probability density function. This function is represented with a 2D array of the same size as the number

of pilot photons. For example, if we use 64^2 pilot photons then the 2D array is of the size 64x64. This dependency is an arbitrary choice and can easily be changed. We start by initializing the grid with zero. Then we increase the probability for the origins of the voxels that were selected for a tight sampling. This is done by increasing the probability in the cells of the 2D grid covered by a circle with the center at the origin mean and a radius of the variance (Figure 3.20). The value by which the value of a cell is increased is arbitrary. We usually used 1. Finally, we add a background probability to each cell to guarantee that all regions are sampled with a certain minimum. In fact, this is the probability for the sparse sampling and has to be supplied by the user.



(a) (b)

Figure 3.20: (a) For each selected voxel, we increase the probability on the 2D square for all cells that are covered by a circle. (b) This leads to the illustrated probability density function.

To compute the content of the array efficiently we use a GLSL shader that draws the circles in a empty framebuffer by summing up the color of the fragments with `GL_FUNC_ADD` blend equation. The shader code can be found in Listing B.1. The shader produces the distribution without the background probabilities. We add the background probabilities to each cell in the 2D map in a CUDA kernel that also does a gauss filtering with the size of 5x5 and variance of two to obtain a smoothly varying distribution. Eventually, the 2D array is normalized in an additional kernel in order to obtain a 2D probability density function.

3.2.3 Phase 2

Phase 2 is very similar to the photon generation and photon marching pass of the reference pipeline. Instead of generating uniformly distributed photons the importance distributed photon generation pass distributes the photons according to the 2D probability density function that was computed in phase 1. The photon marching pass uses these photons and builds an illumination voxel grid. In contrast to the reference photon marching pass the adaptive photon marching pass collects also some additional data.

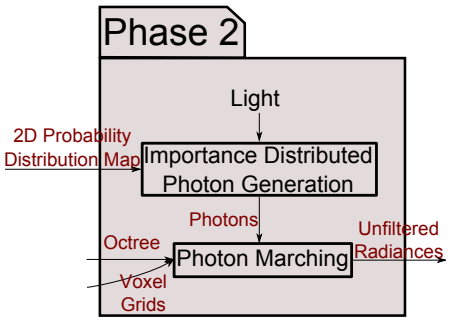
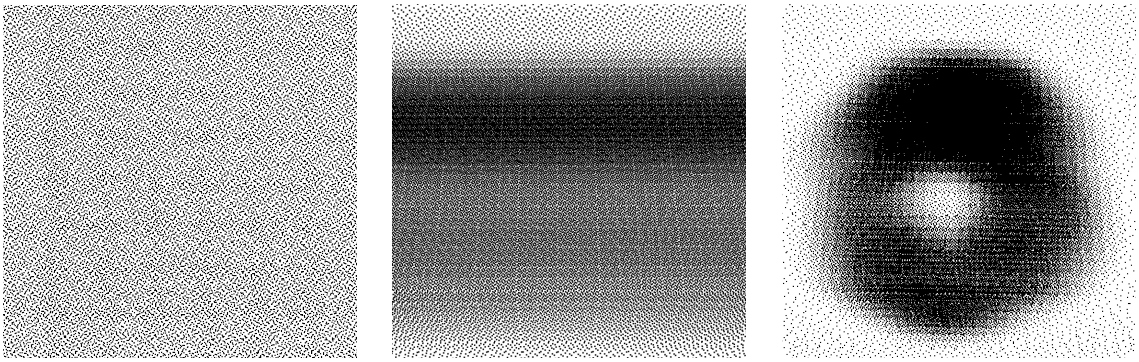


Figure 3.21: Passes of phase 2

Importance Distributed Photon Generation

The importance distributed photon generation pass generates a list of photons where the photons are distributed according to the 2D probability density function from phase 1. We use the technique proposed by Pharr and Humphreys [PH04a]. The main idea is that we modify a uniform sampling such that we obtain a sampling according to a given 2D probability density function. In the first step we stretch the uniform sampling along the Y-axis and then along the X-axis. Figure 3.22 illustrates this stretching process. For this we split the 2D distribution into multiple 1D distributions.

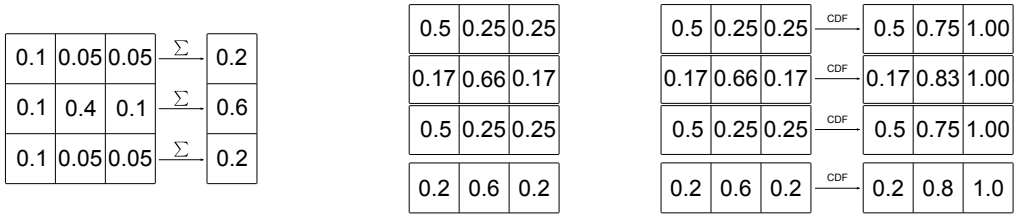


(a) Uniform (b) Y-axis stretched (c) X and Y-axis strechted

Figure 3.22: To compute a sampling for a given 2D probability density function, we transform a uniform sampling to the desired sampling by first stretching the uniform sampling along the Y-axis and the along the X-axis.

We have one 1D density function for the Y-axis and multiple 1D density functions for the X-axis. First, we project the 2D distribution along the X-axis onto one dimension (Figure 3.23a). This gives us a 1D distribution along the Y-axis. Then, we normalize each row of the 2D distribution. This gives us the multiple 1D distributions along the X-axis. Note that the projection is already normalized because the 2D probability density function is already normalized (Figure 3.23b). Then, we build the cumulative distribution function for each 1D distribution (Figure 3.23c).

To compute one component of the final coordinate of a sample according to a 1D density function, we require the inverse cumulative distribution. By sampling the inverse cumula-



(a) 2D to 1D projection (b) Normalization (c) Cumulative distribution function construction

Figure 3.23: To generated photon according to a given 2D distribution the distribution is split into 1D distributions. (a) The rows are summed to compute the 1D distribution along the Y-axis. (b) Each row is normalized, resulting in the required 1D distributions. (c) The last preparation step is to build the cumulative distributions of each 1D distribution.

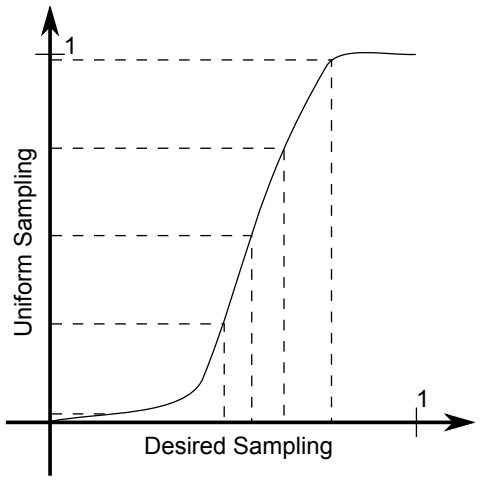


Figure 3.24: When uniformly sampling the cumulative distribution along the Y axis we obtain a distribution from which the cumulative distribution was built.

tive distribution function uniformly we obtain a distribution according the the underlying probability density function. Figure 3.24 illustrates this.

We used a Hammersley sequence to generate uniformly distributed 2D samples. Then, we used the second component of a sample to generate the Y-coordinate. This stretches the uniform sampling along the Y-axis. Then we do the same with the first component of the Hammersley samples and use the 1D cumulative density function along the X-axis that covers the stretched Y-components. This results in the desired photon distribution. Figure 3.25 illustrates this.

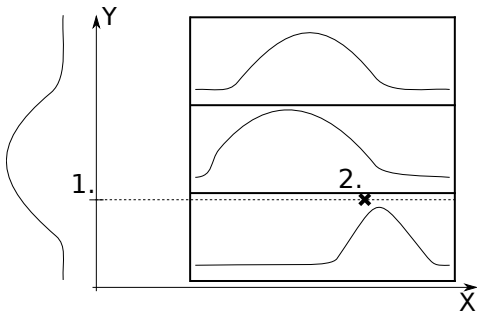


Figure 3.25: First, we generate the Y-coordinate according to the 1D projected density function. Secondly, we generate the X-coordinate according to the 1D density function of the row that covers the Y-coordinate.

In the end, the photons need to be weighted appropriately. For this, we multiply the probabilities for the coordinates of each of the two 1D functions that were used. Thus, the energy of a photon is

$$\frac{light\_power}{p_x(position_x) \cdot p_y(position_y) \cdot square\_area \cdot photon\_count}$$

where $p_x(position_x)$ is the probability for the X-coordinate ($position_x$) and $p_y(position_y)$ the probability for the Y-coordinate ($position_y$).

Technically, we do not build the inverse explicitly because of sampling imprecisions. To calculate the inverse value of a cumulative distribution we execute a binary search on the original cumulative distribution.

Photon Marching

The photon marching pass is essentially the same as the reference photon marching pass except that we collect additional data. In the successive filtering pass two quantities are required: the average photon weights and the photon count per voxel. The weight of a photon can be derived by a division of the energy a photon carries and the power of the light. Thus the weight corresponds to the part

$$\frac{1}{n} \cdot \frac{1}{p(x_i)}$$

of one summand from

$$\frac{1}{n} \sum_{i=0}^{n} \frac{f(x)}{p(x_i)}$$

of the Monte-Carlo integration.

Our implementation reuses the code from the pilot photon marching kernel. Therefore, we still use the 64 bit array from the origin data. This time we use 32 bits to store the photon count and 32 bits to store the average photon weight.

3.2.4 Reconstruction

The illumination voxel grid that was built in phase 2 needs filtering because in the sparsely sampled regions there might be empty voxels (Figure 3.17). In the reconstruction phase we try to fill these holes by estimating the radiance there. This corresponds to the estimation process in photon mapping. In order to execute this estimation or filtering quickly we build a mipmap pyramid of the illumination voxel grid. Each level of this pyramid corresponds to the original voxel grid after a certain filtering. The higher the mipmap level the larger the filter radius. Thus, applying different filter radii for different voxels results in choosing a mipmap level for each voxel of the final voxel grid (Figure 3.26).

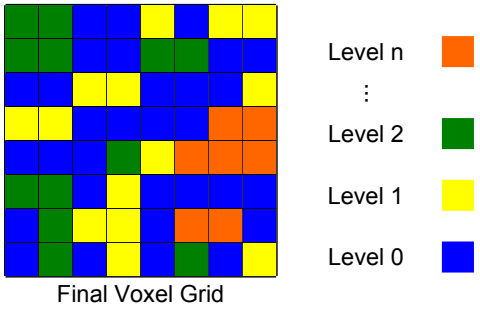


Figure 3.26: 2D illustration of the mipmap filtering. For each cell of the final voxel grid an appropriate mipmap level is used to determine the value (color, direction) of this cell.

Obviously, the reconstruction phase consists of two passes. The mipmap generation pass constructs the mipmap pyramids that are required by the filtering pass. The filtering pass selects an appropriate mipmap level for each voxel of the final voxel grid and assigns the corresponding radiance value to this voxel.

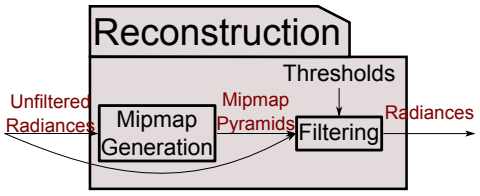


Figure 3.27: Passes of the reconstruction phase

Mipmap Generation

The mipmap generation pass is rather simple. We build the 3D mipmap pyramids for the color, direction, photon count and photon weight voxel grids from the photon marching pass. For the computation of the next higher level of the color and direction we use a gauss kernel of the size 4x4x4 with variance four. We compute each level through convolution separation

with CUDA kernels. For the photon count a box filter is used. A modified box filter is used for the photon weights. When a voxel has an average weight of zero we ignore this voxel entirely. It is also ignored for the normalization (see Listing B.2). We figured out that it is not necessary to build the whole mipmap pyramid because the levels at the tip are never used for filtering. Hence, we skip the construction of the last few levels.

Filtering

We saw that the photon marching pass produces illumination voxel grids with a lot of high frequency noise in the regions where few photons pass through (Figure 3.17). To reduce the noise we apply a smoothing filter. Since the magnitude of the noise varies over the voxel grid we use filters with different radii. In Figure 3.17 it can be seen that the caustic, the shadow and the border of the shadow show hardly any noise. Thus, we do not want a large filter radii there. In fact we would like to use the unfiltered values. On the other hand, in sparsely sampled regions we would like to use larger filter.

As mentioned, we work with mipmap pyramids that store prefiltered voxel grids. Therefore, filtering means that we have to assign each voxel an appropriate mipmap level. Unfortunately, we cannot use standard filters such as a constant radius filter or a k-nearest-neighbor filter (Section 4.2). Therefore, we use a filtering technique of our own. Our technique consists of a combination of two independent filters. One is the center surround filter and the other is the photon weight based filter:

Center Surround Filter The center surround filter chooses a filtering radius as large as possible such that no inhomogeneous parts, such as edges, are included. This means that it tries to reduce as much high frequency noise as possible without blurring edges. This is done by comparing different degrees of filtering. In our case this means different filter sizes and, therefore, different mipmap levels. Since our goal is to find a mipmap level that reduces local high frequency noise but preserves global edges we analyze how the filtered values change. In homogeneous regions, with increasing the filter size the resulting value approaches the average of this region. Therefore, the difference between two filtering levels decreases monotonously in homogeneous region. However, when the filter suddenly includes an inhomogeneous region, like caustic or shadow, the difference between two filtering levels increases because the inhomogeneous part drags the filtered value away from the average of the homogeneous region.

Hence, we increase the filter size (the mipmap level) as long as the difference between two levels decreases. When the difference increases we detected that the filter includes some inhomogeneity. This means we use the smaller level of the two we compared. Figure 3.28 illustrates this.

The center surround filter builds in fact the second derivative over the filter size.

Photon Weight Based Filter Each photon has a specific weight due to the results of Monte-Carlo integration. Photons with an origin of low probability have a high weight and vice versa. Therefore, the photons in the sparsely sampled regions have a higher weight than those in the tightly sampled regions. Now, we want the regions which are tightly sampled to have a small filter size, in other words a low mipmap level, and vice versa. As a result, we can use the weight of a photon to determine the mipmap level.

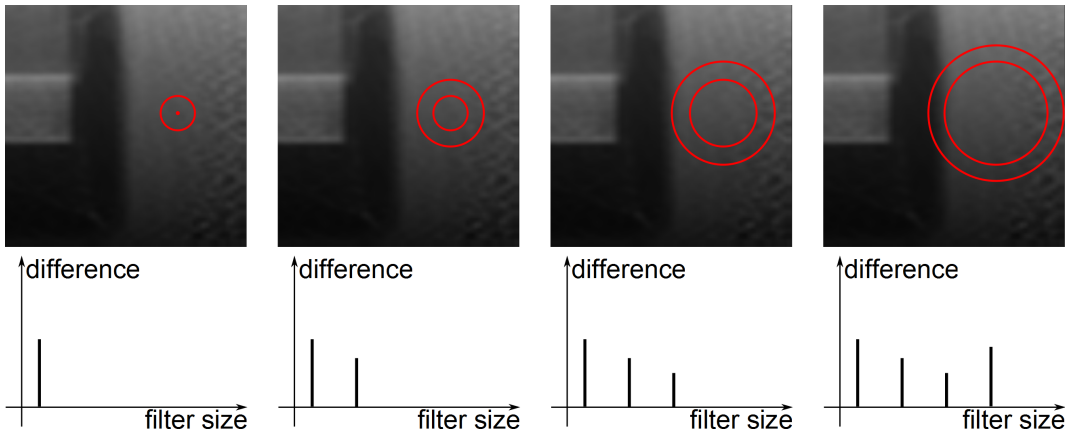


Figure 3.28: When we increase the filter radius and compare two successive filtering results, we see that the difference decreases as long as the filter only includes the homogeneous region. As soon as the filter starts to include inhomogeneous regions the difference increases. The radius before the difference starts to increase is the radius we choose for the reconstruction.

We compute the level by letting the user specify a $\Delta weight$. We then compute the level per voxel of the final grid as follows

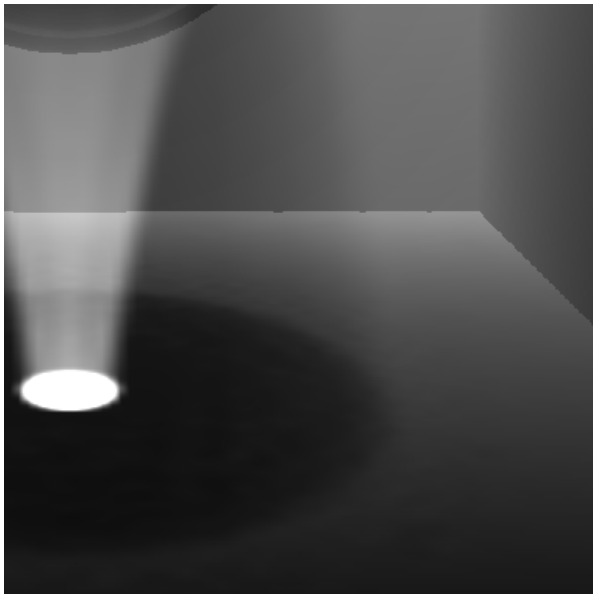
$$level = \left\lfloor \frac{photon\_weight}{\Delta weight} \right\rfloor$$

where $photon\_weight$ is the average weight of all photons that passed through the voxel we want to filter.

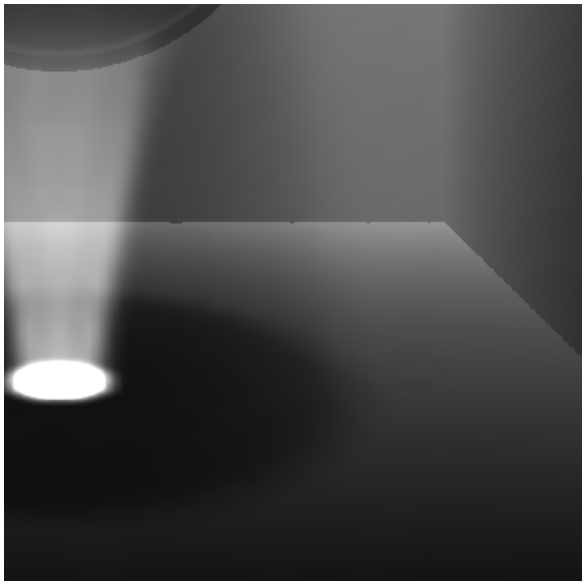
Of course, when no photons pass through a voxel we cannot rely on the photon weight because it is always zero. To avoid this problem, we simply use the photon weights of a higher mipmap level. We increase the level until at least one photon passed through this region. This can be done with the photon count mipmap.

Figure 3.29 shows that neither the center surround filter nor the photon weight based filter alone is sufficient for an appropriate reconstruction. The borders between tightly and sparsely sampled regions as well as the borders between caustic and shadow are blurred in case of the center surround filter because it chooses a level that is too high. However, the sparsely sampled regions are reconstructed nicely. The photon weight based filter tends to choose mipmap level in the sparse sampled regions that are too high. This leads to bright patches which are the values from the largest mipmap level. On the other hand, the photon weight based level selection keeps the caustics and borders sharp. Since both filter produce bad results when they choose a high level, we combine the two filter by taking the lower of the two proposed mipmap levels. This assigns each voxel an appropriate mipmap level for the reconstruction of the final color and direction voxel grid.

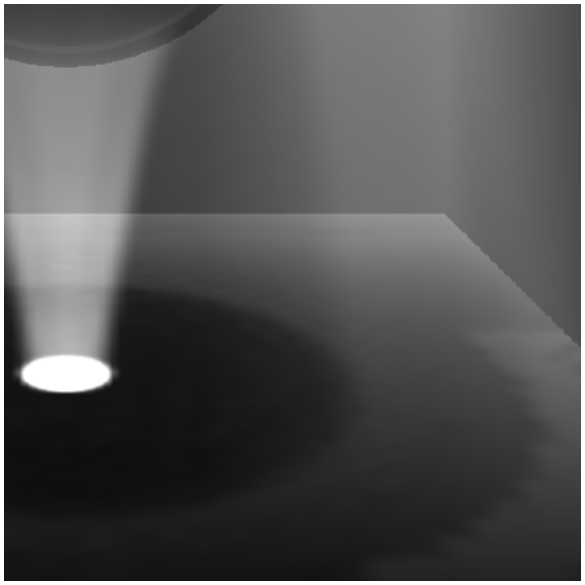
Technically, we implemented a CUDA kernel that launches a thread for each voxel in the final voxel grid, typically 128^3 threads. Thus, each thread tries to figure out which mipmap level to use and assigns the color and direction to its voxel of the final grid. For determining the mipmap level we use the combination of a center surround filter on the color values and a photon weight based level selection. The kernel code can be found in Listing B.3.



(a) Photon weight based and center surround filtering



(b) Only center surround filtering



(c) Only photon weight based filtering

Figure 3.29: Neither the center surround nor the photon weight based filtering alone are capable of reconstruction. The center surround filter does a good job in the sparsely sampled regions but not around borders. The border of the shadow is slightly blurry and the bright caustic spot is also blurry. The photon weight based filter performs well in shadows and near caustics but in the sparsely sampled regions it chooses always too highest mipmap levels. This is the reason for the bright patches on the right which originate from the highest mipmap level. The combination of the two works much better.

Chapter 4

Evaluation

The purpose of this chapter is threefold. For one, it shows that the sampling choice we made for distributing the photons on the photon plane is reasonable, meaning the Hammersley sequence. The second part of this chapter shows that the standard photon mapping estimators cannot be used in the adaptive pipeline for filtering. Thirdly, a comparison between the adaptive and the reference rendering pipeline is discussed. We show that the new rendering pipeline is in fact able to achieve a similar quality with less photons. In addition, we show that, unfortunately, the adaptive pipeline is not faster even though less photons are used.

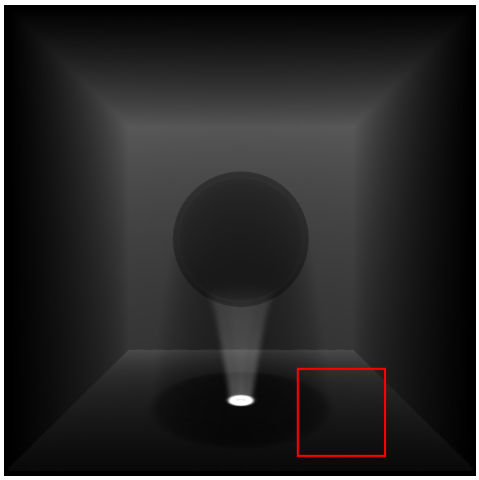
All timings in this chapter are an average over ten frames to provide a stable measurement. The timings are always for the execution of the whole pipeline, meaning that all passes are executed for each frame.

4.1 Sampling Strategies

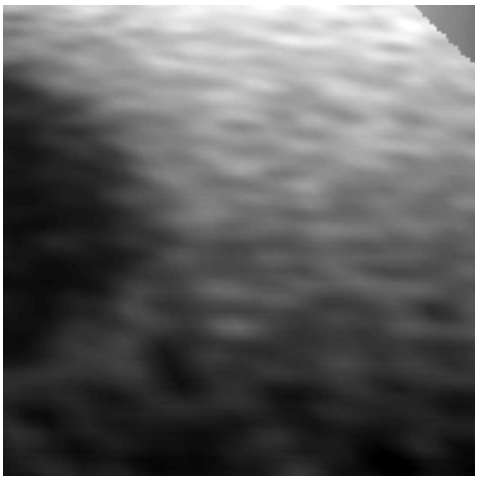
We saw in Section 2.2 that the sampling plays an important role in Monte-Carlo integration. Depending on the sampling a better or worse approximation is computed with the same number of samples. In our rendering pipelines, the only sampling that needs to be performed is in the photon generation passes. There we distribute photons on the photon plane. The pilot photon generation (Section 3.2.2) and photon generation in the reference pipeline (Section 3.1.4) require a uniform 2D sampling. For the importance distributed photon generation pass (Section 3.2.3) the photons are not distributed uniformly but according to the probability density function that was built in phase 1.

For the analysis of different sampling strategies, we restricted ourself to the most common samplings. These are grid aligned, purely random, jittered and quasi-random samplings (Halton and Hammersley sequence [WLH97]). We analyze only the uniform distribution but the arguments and conclusions are also valid for adaptive distributions from the importance distributed photon generation pass (Section 3.2.3). In this section, all renderings were made with the reference pipeline if not stated otherwise. For all measurements, the scene in Figure 4.1 was rendered.

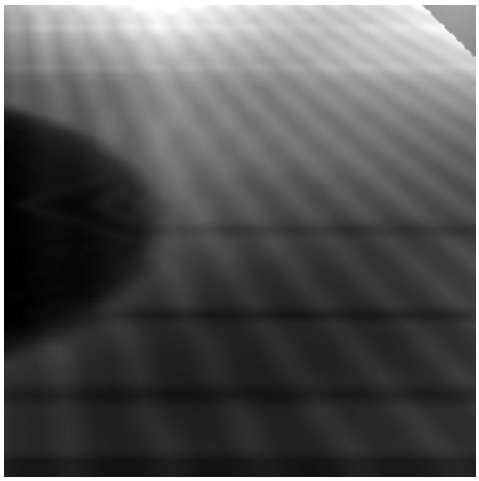
In Chapter 3 we chose to use the Hammersley sequence to distribute the photons on the photon plane. This is the best choice for two reasons, quality and performance.



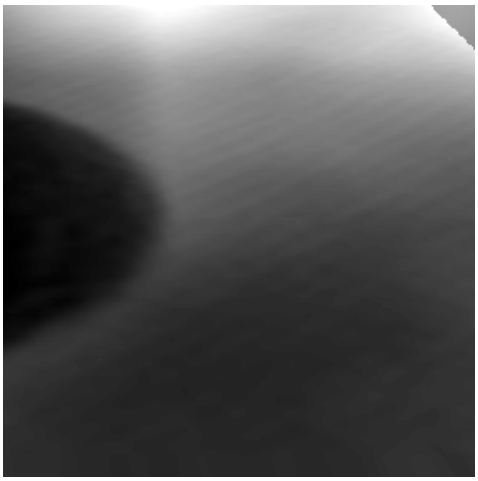
(a) Scene



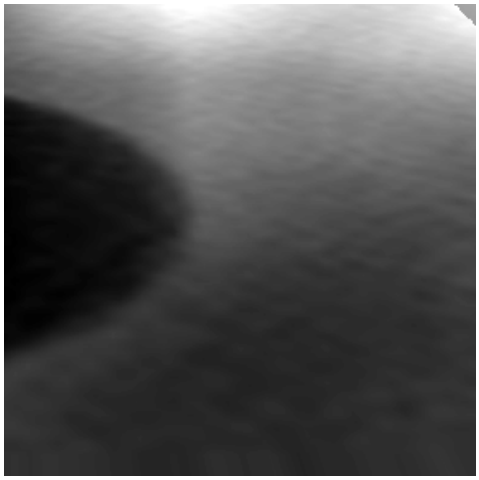
(b) Random



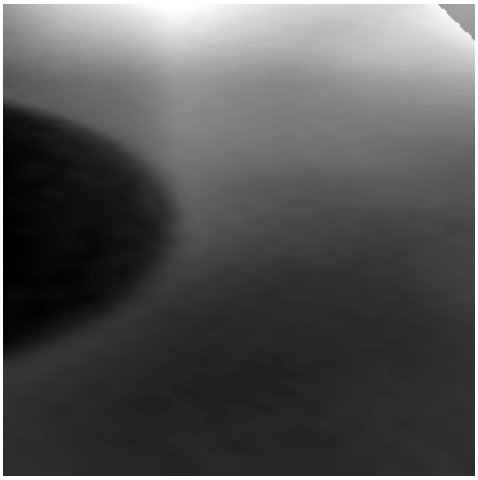
(c) Grid



(d) Jittering



(e) Halton



(f) Hammersley

Figure 4.1: The scene shown in (a) is rendered with roughly 122'000 photons. (b) - (f) are contrast enhanced close ups of the rectangle in (a) with different photon samplings. Random and the Halton sequence show high frequency noise whereas the grid and jittering sampling show moiré patterns. The Hammersley sequence produces a smooth result.

Quality

Figure 4.1 shows a visual comparison of the different sampling techniques with 122'000 photons. The grid sampling shows very distinctive moiré patterns and is, therefore, unusable. Jittering is much better but shows still some moiré patterns. The purely random sampling exhibits high frequency noise due to under sampling in certain regions. The Halton sequence also shows some high frequency noise. Without doubt the Hammersley sequence produces the best result. As a result our choice with respect to visual quality is to use the Hammersley sequence.

Performance

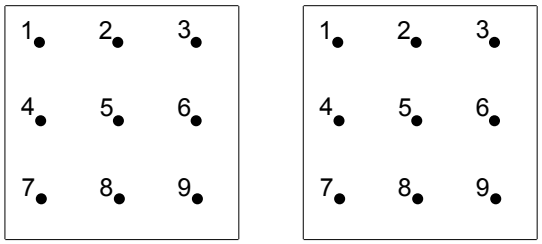
A further argument for choosing the Hammersley sequence is the performance. The photon generation pass is very fast with all samplings (< 1 ms). Therefore, we are not interested in the performance of the photon generation pass. The photon marching pass however is strongly affected by the sampling. Table 4.1 lists the execution time of the photon marching pass depending on the photon sampling strategy. The timings are obtained by rendering the scene from Figure 4.1 with 122'000 photons.

Sampling Strategy	Photon Marching Execution Time
Random	154 ms
Grid (Row First Ordering)	158 ms
Grid (Column First Ordering)	191 ms
Jittering (Row First Ordering)	148 ms
Jittering (Column First Ordering)	194 ms
Halton	163 ms
Hammersley	109 ms

Table 4.1: Execution time of the photon marching pass for 122'000 photons depending on the photon sampling. It can be seen that the Hammersley sequence results in a fast photon marching pass.

The reason for these performance discrepancies is not entirely clear. Although, our measurements indicate that the atomic compare and swap operation and memory access patterns are responsible. As mentioned earlier, when we encounter an atomic conflict then the GPU has to serialize the write operations. Since a GPU has only a limited number of processors, not all threads are executed in parallel. Therefore, it is theoretically possible to avoid all atomicity conflicts when conflicting threads are arranged such that they are never executed in parallel. The ordering of the photons in the list from the photon generation pass determines how the photons are grouped into threads blocks in the photon marching pass. The ordering of the photons in the list is different for each sampling because in the photon generation pass a thread is created for each photon and each thread has an index which is the index in the photon list. For example, with the grid sampling a typical ordering is the row first or column first ordering (Figure 4.2). As a result, it can be expected that each ordering leads to a different performance.

We expect that the grid sequences cause many conflicts because the resulting list ordering groups the photons that are close together in the same CUDA block. In contrast, a random or quasi-random ordering of the photons in the list should produce less atomic conflicts because



(a) Row first ordering (b) Column first ordering

Figure 4.2: Typical enumeration of grid elements.

the photons are grouped randomly.

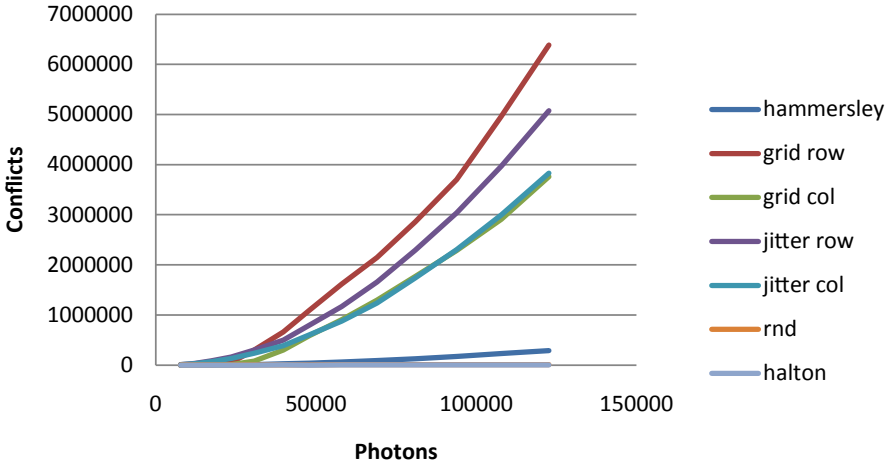
To analyze the behavior of the different sampling strategies with respect to conflicts, we rendered the scene in Figure 4.1 with the reference pipeline with different photon counts and different sampling strategies (Figure 4.3a). Since we follow an optimistic approach for writing into the voxel grid, we simply counted the number of retries that were made. The number of retires is exactly the number of conflicts. As expected, our experiments show that the grid and jitter arrangements have more than an order of magnitude the number of conflicts depending on the photon count. The Hammersley sequence has also more than an order of magnitude the number of conflicts compared to the Halton and random sampling. The random and the Halton sequence are more or less equal in the number of conflicts (Figure 4.3b).

We saw that random and quasi-random sampling lead to less conflicts than grid ordering. Unfortunately, when the ordering in the list is random the memory access in the photon marching pass is similarly random. An NVIDIA GPU is capable of coalesce data access. This means that accessing data (reading or writing) can be grouped into a single instruction for multiple threads of a block when these threads access neighboring memory (see NVIDIA CUDA Programming Guide [NVI09b] for more information). Thus, random samplings cannot benefit from this feature. On the other hand, the grid ordering can be expected to take advantage of coalesce data access. Therefore, we expect grid orderings to show better memory access performance.

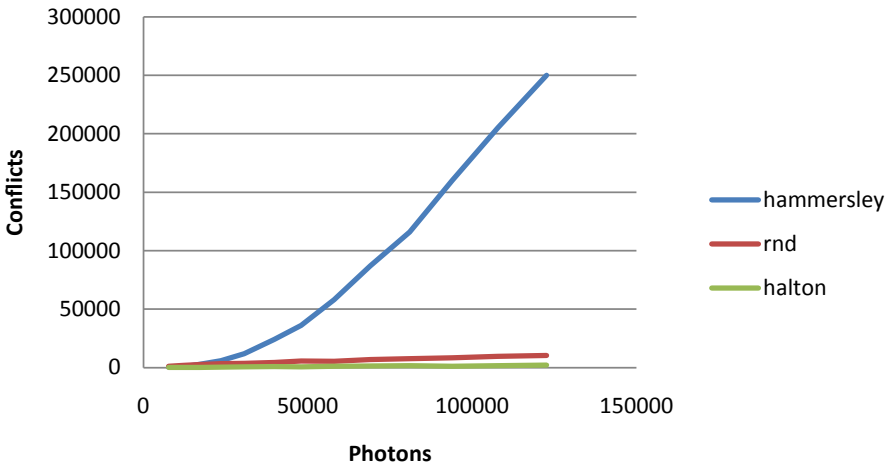
To measure the memory access performance we replaced the atomic write in the photon marching pass with a non atomic write. As a result there are no more conflict issues. The experiment has the same setup as the previous one except that we measure the execution time of the photon marcher instead of the conflicts. As expected, in Figure 4.4 it can be seen that the grid ordering in the photon list leads to a better performance than a random or quasi-random ordering.

Figure 4.3 and Figure 4.4 show that the Hammersley sequence is in both measurements in the center span. Thus, the Hammersley sequence is a good compromise between coalesce memory access and atomic conflicts. To show this we measured the execution time of the photon marching pass with atomic writes for different photon counts and samplings. The results are shown in Figure 4.5. It can be seen that the Hammersley is a good choice for sampling the photons with respect to performance.

In conclusion, the Hammersley sampling is by far the best choice among the analyzed sampling strategies with respect to performance as well as visual quality.



(a) Conflict graph



(b) Close up conflict graph

Figure 4.3: The number of conflicts equals the number of retries that were necessary to write into the voxel grid. Grid ordering in the photon list leads to many atomic conflicts. The more random the photons in the list are the less conflicts occur.

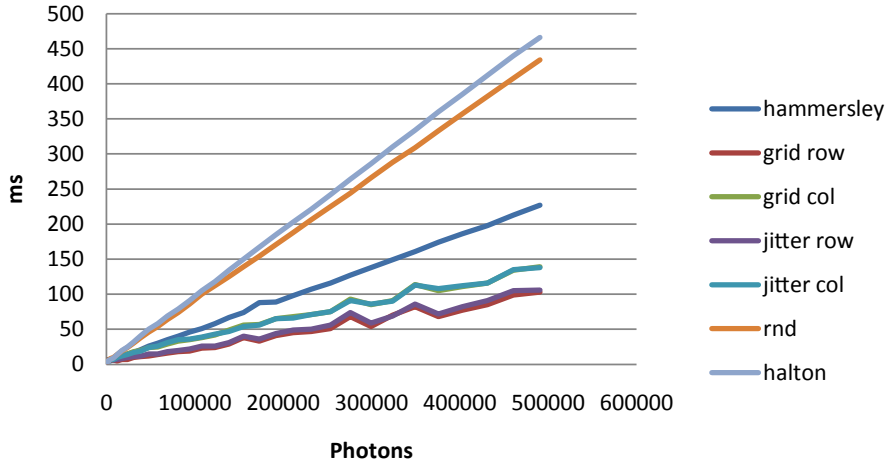


Figure 4.4: The execution time of the photon marching pass when the atomic write is replaced by a non atomic write. Because there are no more atomic conflicts this is a measurement of the memory access. The grid orderings show better performance the random orderings because of the coalesce memory access. This also explains the difference between the column and row orderings.

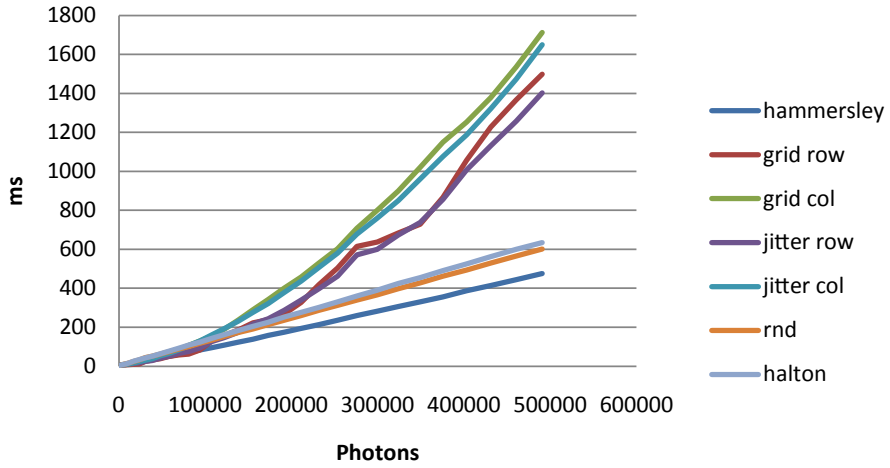


Figure 4.5: The Hammersley sequence leads to the fastest photon marching because is appears to be a good compromise between atomic conflicts and coalesce memory access.

4.2 Filtering

In photon mapping there are two standard estimators or filters, the constant radius filter and the k-nearest-neighbor. The goal of this section is to show that neither of the two can be use for the filtering in the adaptive pipeline.

Since a mipmap level corresponds to a filtering with a constant radius we could just visualize the levels to analyze the constant radius estimator. In Figure 4.6 it can be seen that when the radius is small then the caustic appears sharp but there is a lot of noise in the homogeneous regions (Figure 4.6c). On the other hand, when the radius is large the homogeneous region is smooth but the caustic and shadow border is completely blurred (Figure 4.6d). Thus, this filter is not applicable for our purpose.

The k-nearest-neighbor filter shows a similar behavior. Figure 4.6 shows the filtered results for $k = 1$ and $k = 20$. Visual artifacts appear in both cases in the shadow because the filter radius is enlarged to include at least k photons. This leads to a leakage near the caustic (Figure 4.6f and Figure 4.6e). Furthermore, when k is small then even the inhomogeneous region is poorly filtered (Figure 4.6e).

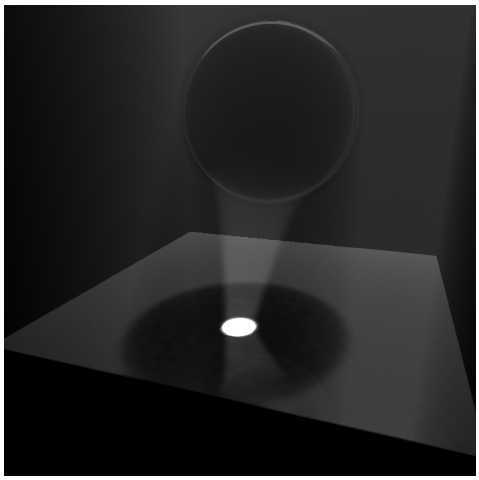
In conclusion it is not possible to use either of the standard estimators. This enforced us to find a more sophisticated filter with which the reconstruction is as good as the reference (Figure 4.6b).

4.3 Pipeline Comparison

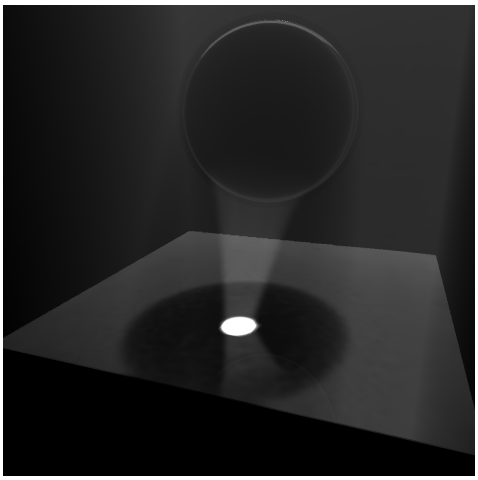
For the comparison of the two rendering pipelines, we rendered three different scenes. We compare the image quality first and then the performance. For all scenes the reference pipeline generates 262144 photons. However, approximately only half of the generated photons are actually marched through the voxel grid. This is because many photons do not hit the voxel grid and are, therefore, discarded. In the adaptive pipeline we always generate $64^2 = 4096$ pilot photons. Thus, we use an origin grid with a resolution of 64x64. Then we distribute 64'536 photons according to the importance photon distribution. Since we want to compare the two rendering pipelines we limit the maximal photon density in the importance distributed photon generation to the photon density of the reference pipeline. The voxel grid is in all cases 128^3. The 800x600 pixel images are rendered with an NVIDIA GTX 260.

Quality

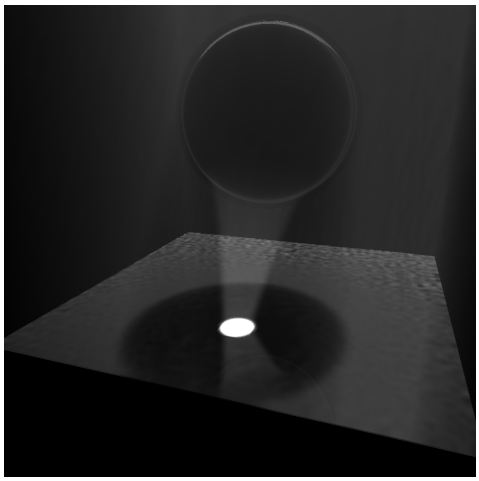
With an adequate setting of the thresholds we can show that the adaptive pipeline needs less than half the number of photons to produces an image of equal quality as the reference. In the sphere example, for the reference pipeline 122'000 photons were traced and for the adaptive pipeline only 50'000 photons. Even though the adaptive pipeline uses only two fifth of the number of photons, there are visually no difference at all (Figure 4.7). Only with a difference image can we show where the differences are (Figure 4.8) . In the cube example, for the reference pipeline 125'000 photons were used and 46'000 for the adaptive pipeline. Here, differences are visible, namely, the noise in the shadow area on the table. Both images show high frequency noise which is slightly different but of the same amplitude (Figure 4.9). Also the difference image shows that mostly the differences are in the shadow area (Figure 4.10) For the armadillo example, 106'000 photons were used for the reference and 49'000 photons



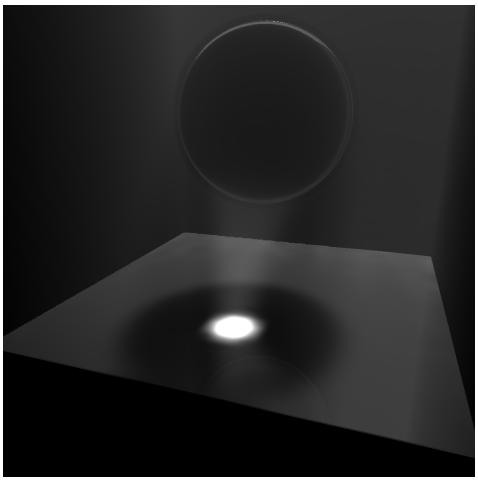
(a) Reference



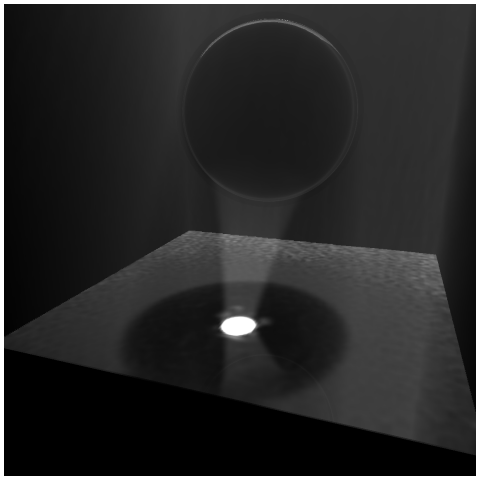
(b) Adaptive Filter



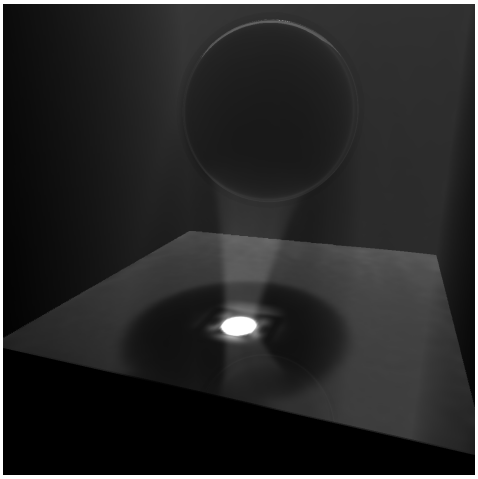
(c) Small constant radius



(d) Large constant radius



(e) $k = 1$



(f) $k = 20$

Figure 4.6: (b) Our new filter is capable of reconstructing the reference image. (c) - (d) The constant radius estimator is not applicable for our adaptive photon distribution. Either the caustic is sharp or the homogeneous region is smooth but not both. (e) - (f) The k-nearest-neighbor estimator is not capable of reconstructing the shadow properly

for the adaptive pipeline. Here we have differences in the projection of the right fist of the armadillo. The caustic and the shadow border are slightly less sharp in the adaptive case. (Figure 4.11). Interestingly, the difference image shows that apart from the tiny differences on the table there is a major difference in the caustic between the armadillo's legs (Figure 4.13). However, this difference is hardly visible.

In all scenes, the differences occur mostly on the table which can be seen in the difference images. If there is a difference in the participating media, as in the armadillo scene, then it is hardly visible.

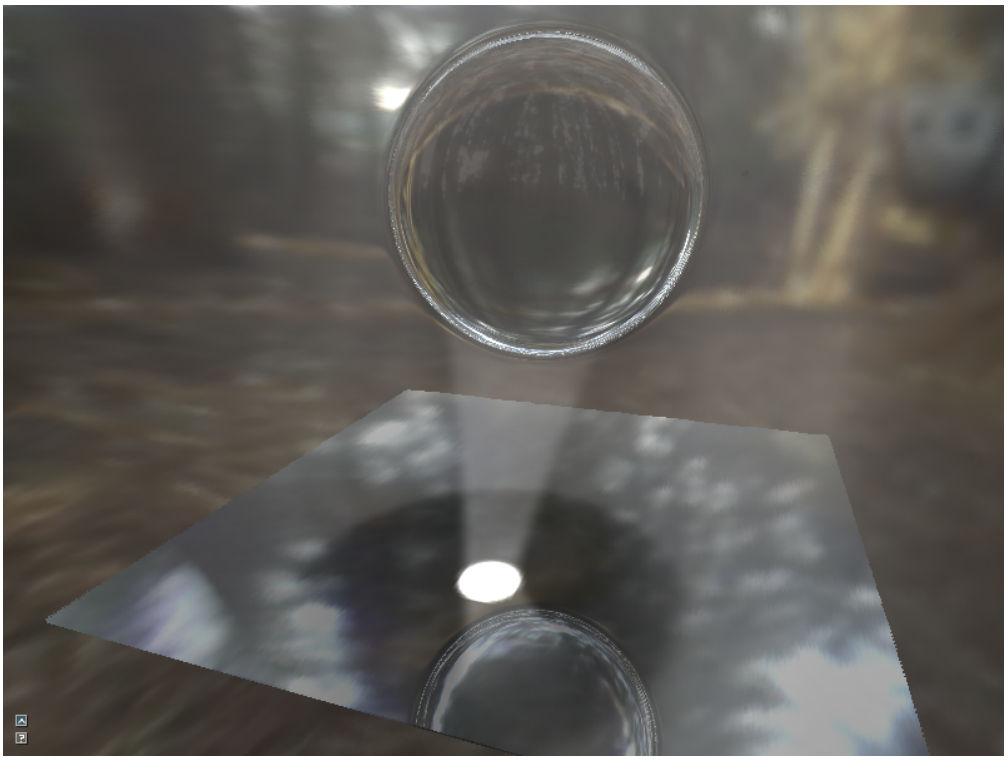
In conclusion, it can be said that the adaptive pipeline is capable of rendering images with hardly any quality loss with less than half the number of photons.

Performance

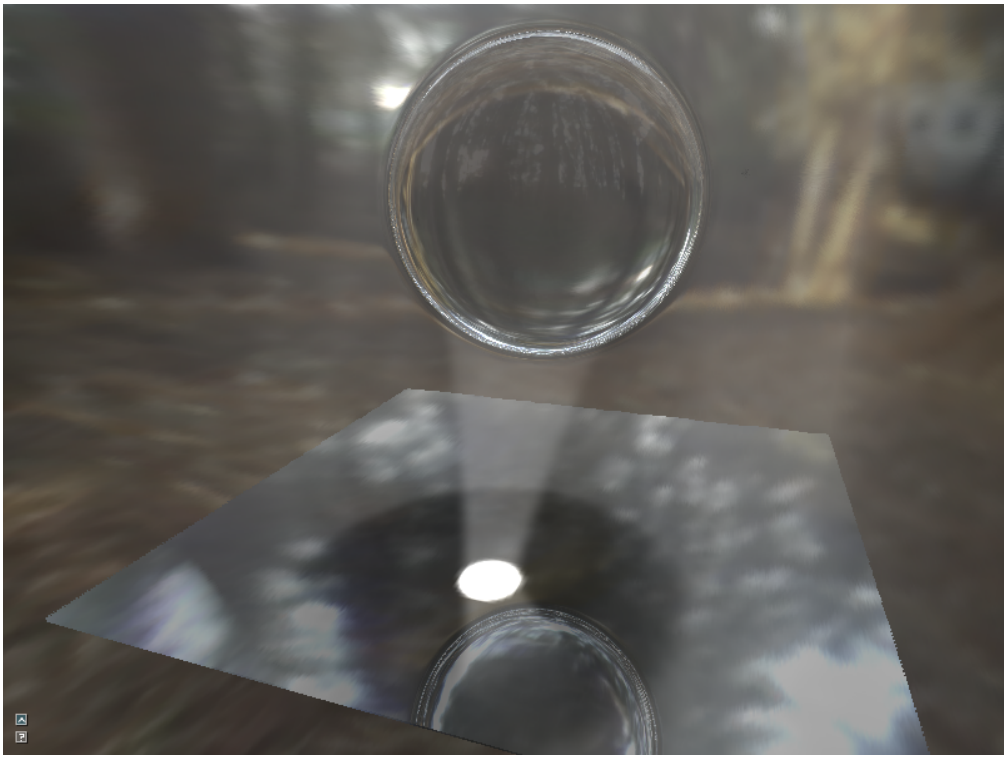
We saw that the adaptive pipeline needs less than half the number of photons to produce images equal to the reference images. However, in this section we see that the execution time of the photon marching of the adaptive pipeline is not scaled by the same factor as the photon counts. Furthermore, it is shown that the constant overhead from the additional passes in the adaptive pipeline consume more time than we could gain by marching less photons. In the following tables we measured the execution time of each pass of the pipeline. Additionally, a copy entry can be found which is the time needed for copying voxel grid data to `cudaArrays`. These arrays are needed to access the data as textures in the viewing pass.

Table 4.2, 4.3 and Table 4.4 show two issues. The first is that the additional passes (pilot photon generation and marching, photon distribution generation, mipmap generation and filtering) in the adaptive pipeline are a constant overhead of approximately 48 ms. The overhead is constant because the additional passes operate mostly on the voxel grid. The other passes are very fast such that hardly any time differences can be observed. The second issue to notice is that the photon marching pass in the adaptive pipeline is slower than the photon marching in the reference pipeline. In all examples less than half the photons are used but the duration of the photon marching does not halve compared to the reference. The reason is that in the adaptive photon marching we collect additional data per voxel, the photon count and average photon weight. This leads to an additional atomic operation. In order to verify the hypothesis that the additional atomic operation is responsible, we added an additional artificial atomic operation in the reference pipeline. As expected, the performance lowered to the level of the adaptive pipeline (Figure 4.14).

An additional problem is that in the adaptive pipeline we march exactly the photons that cause many atomic conflicts. These are the photons that are focused in a caustic. In an experiment, we could show that the photons generated in the importance photon distribution pass (Section 3.2.2) are expensive to march. Instead of using the 2D photon distribution that shoots many photons through the inhomogeneous regions we use the complement of this distribution. This means that we use the distribution that we obtain by computing "1− 2D photon distribution". This way we split a uniform distribution into two non uniform distributions: the regular adaptive sampling and its complement. Figure 4.15 illustrates this. We measured the execution time and conflicts of the photon marching pass of the adaptive pipeline for the uniform, adaptive and complement sampling (Table 4.5). We expect that the timing, conflict count and photon count of the uniform sampling are split up between the adaptive and complement sampling.

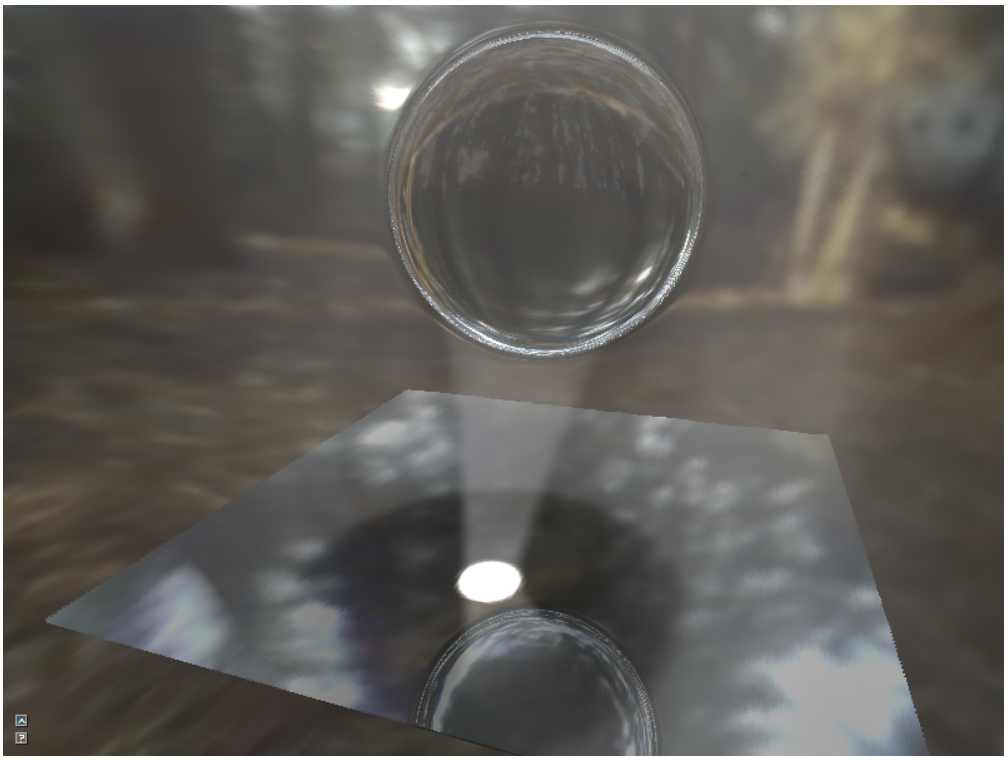


(a) Reference pipeline



(b) Adaptive pipeline

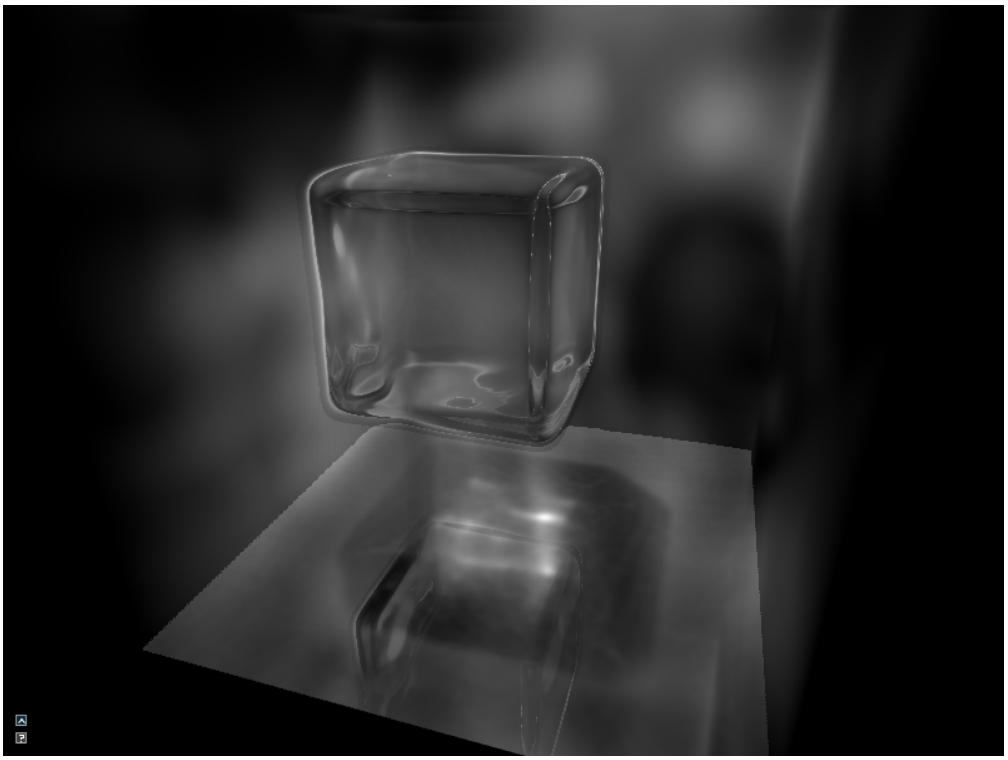
Figure 4.7: In this scene the adaptive pipeline shows no visual difference at all compared to the reference. The reference pipeline uses 122'000 photons with 2.6 FPS and the adaptive pipeline 50'000 photons with 2.5 FPS.



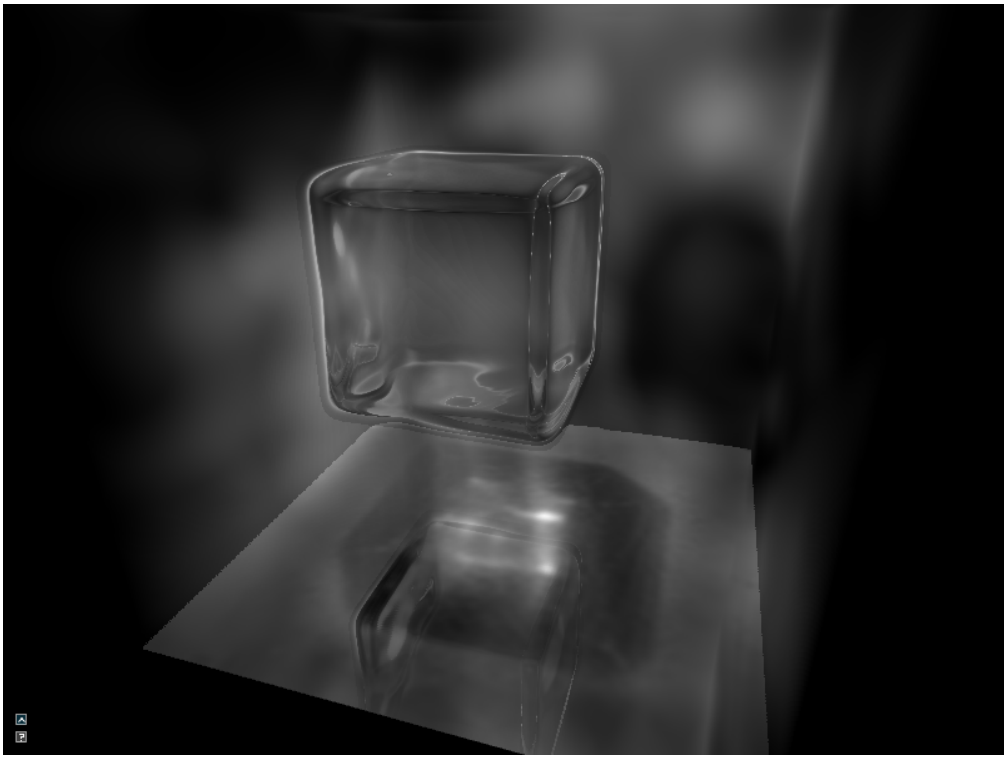
(a) Reference pipeline

(b) Difference image

Figure 4.8: Only with the difference image between the adaptive and the reference can we see where differences are.

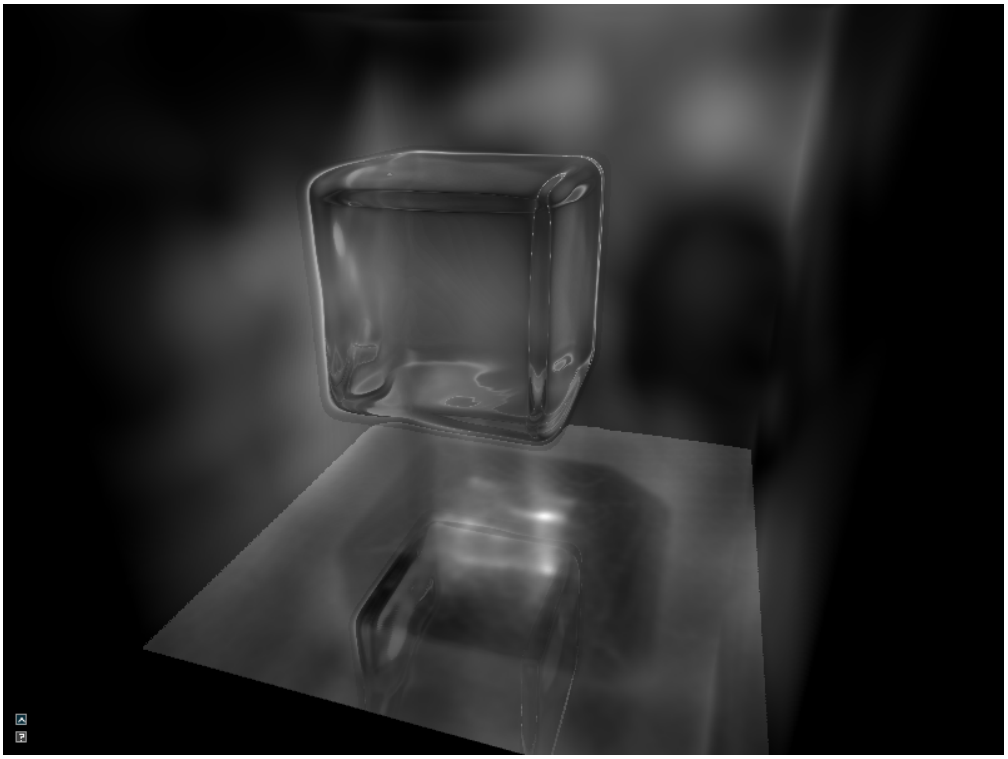


(a) Reference pipeline

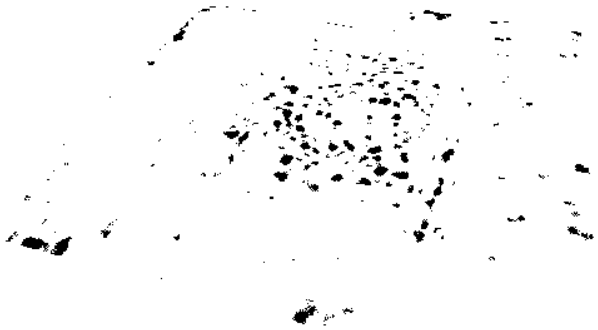


(b) Adaptive pipeline

Figure 4.9: The surfaces of the cube are not even because perturbation to the refraction indices was added with a perlin noise. The same holds for the scattering coefficients. The images show hardly any difference. Only in the shadow the noise is different but not stronger. The reference pipeline uses 125'000 photons with 2.2 FPS and the adaptive pipeline 46'000 photons with 2.2 FPS.



(a) Reference pipeline



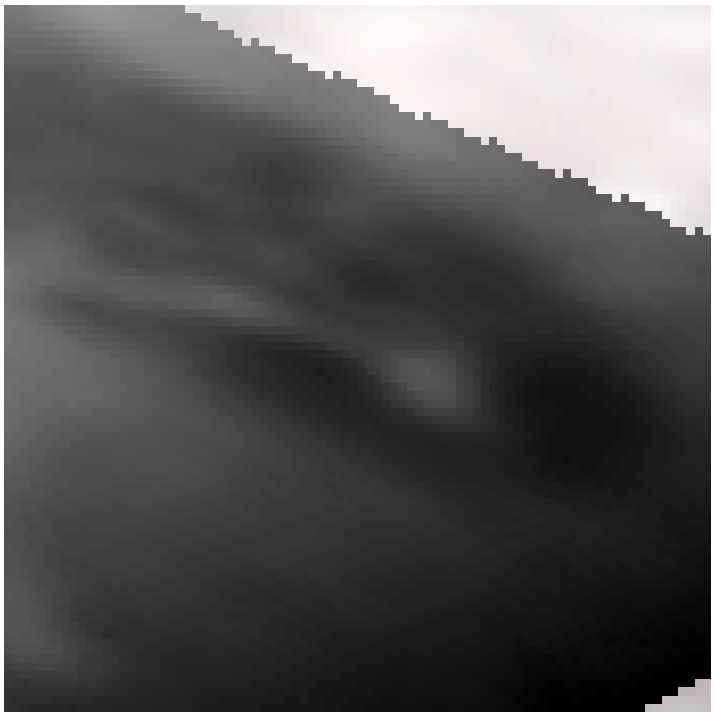
(b) Difference image

Figure 4.10: The difference image shows that the differences are mostly in the shadow area.

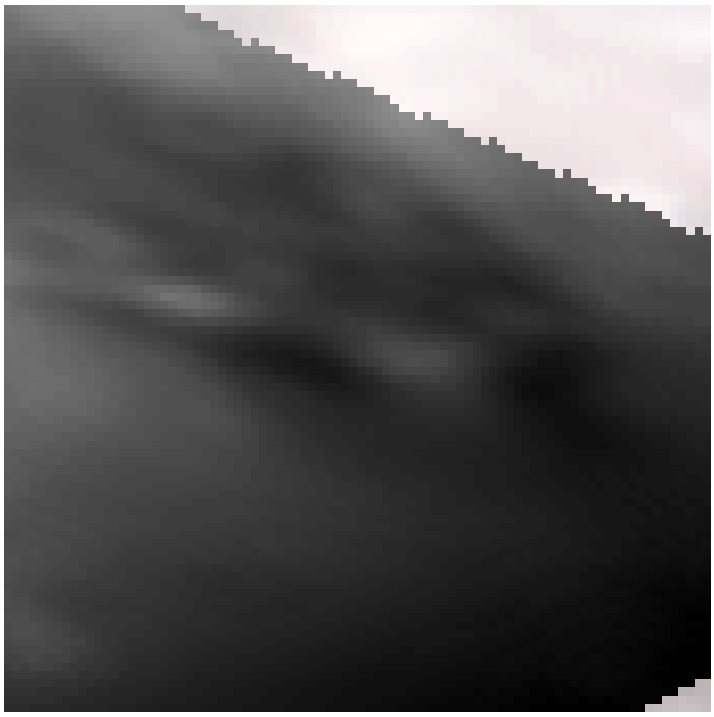
(a) Reference pipeline

(b) Adaptive pipeline

Figure 4.11: The caustic and the shadow border of the right fist of the armadillo in the adaptive pipeline produces a slightly less sharp edge (red square). Figure 4.12 shows the red squares enlarged. The reference pipeline uses 106'000 photons with 2.1 FPS and the adaptive pipeline 49'000 photons with 2.0 FPS.



(a) Close up reference Pipeline



(b) Close up adaptive pipeline

Figure 4.12: Close up images from Figure 4.11 show that the shadow from the adaptive pipeline is less sharp.

(a) Reference pipeline

(b) Difference image

Figure 4.13: The difference image shows that apart from the tiny differences on the table there is a major difference in the caustic between the legs of the armadillo. Yet, this difference is hardly visible in a side by side comparison.

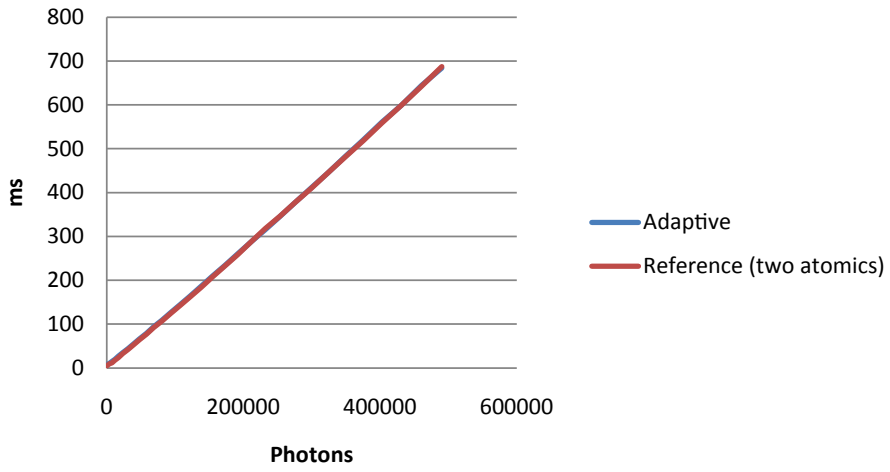


Figure 4.14: The reference photon marcher has the same performance as the adaptive photon marcher when a second artificial atomic write is added. The scene rendered is the same as in Figure 4.1

Pass	Reference	Adaptive
Voxelization	25 ms	25 ms
Octree	5 ms	5 ms
Pilot Photon Generation	-	< 1 ms
Pilot Photon Marching	-	10 ms
Photon Distribution	-	5 ms
Photon Generation	< 1 ms	2 ms
Photon Marching	110 ms	85 ms
Mipmap Generation	-	16 ms
Filtering	-	14 ms
Copy	44 ms	44 ms
Viewing Pass	189 ms	189 ms
Total	**373 ms**	**395 ms**
Photon Count	122'000	50'000

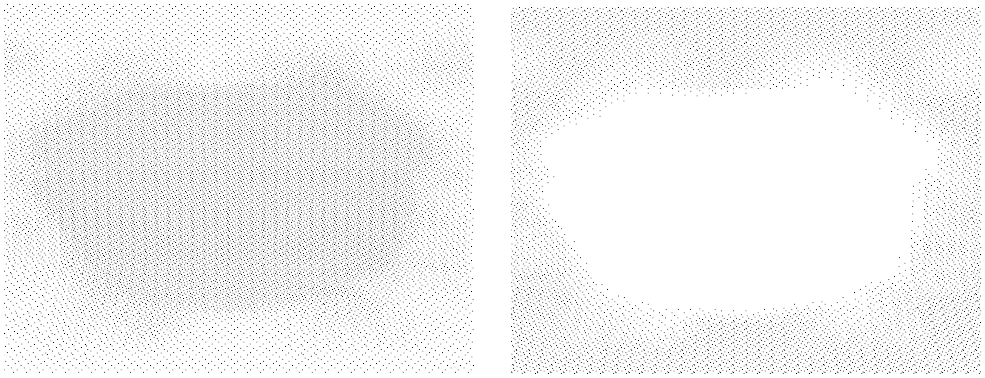
Table 4.2: Duration of the separate passes and the photon count for the sphere example from Figure 4.7.

Pass	Reference	Adaptive
Voxelization	43 ms	43 ms
Octree	5 ms	5 ms
Pilot Photon Generation	-	< 1 ms
Pilot Photon Marching	-	11 ms
Photon Distribution	-	5 ms
Photon Generation	< 1 ms	2 ms
Photon Marching	144 ms	100 ms
Mipmap Generation	-	17 ms
Filtering	-	13 ms
Copy	48 ms	48 ms
Viewing Pass	212 ms	212 ms
Total	**452 ms**	**456 ms**
Photon Count	125'000	46'000

Table 4.3: Duration of the separate passes and the photon count for the cube example from Figure 4.9.

Pass	Reference	Adaptive
Voxelization	68 ms	68 ms
Octree	5 ms	5 ms
Pilot Photon Generation	-	< 1 ms
Pilot Photon Marching	-	11 ms
Photon Distribution	-	6 ms
Photon Generation	< 1 ms	2 ms
Photon Marching	163 ms	134 ms
Mipmap Generation	-	16 ms
Filtering	-	13 ms
Copy	44 ms	44 ms
Viewing Pass	179 ms	179 ms
Total	**459 ms**	**478 ms**
Photon Count	106'000	49'000

Table 4.4: Duration of the separate passes and the photon count for the armadillo example from Figure 4.11.



(a) Adaptive sampling (b) Complement sampling

Figure 4.15: In both images the black dots are photons. The adaptive sampling is the sampling the we obtain from the 2D probability density function from the adaptive rendering pipeline. The complement is the the probability density function "1− adaptive probability density function". Therefore, with the complement we create many photos where we shoot few in the adaptive sampling and vice versa.

The measurements show that our expectation are approximately satisfied. The photons are split up such that the complement sampling produces only half of the photons of the adaptive sampling. Yet, the photon marching pass with the adaptive sampling takes roughly four times as long as with the complement sampling. When looking at the conflict count, we realize that the adaptive sampling causes, as expected, much more conflicts than the complement sampling. This explains why the adaptive sampling is more expensive. The adaptive sampling causes even more conflicts than the uniform sampling. We assume this is because it is more likely that photons which cause conflicts are marched in parallel. From this we can derive that with a uniform sampling, the photon marching pass spends most of the time marching the photons that are also used in the adaptive sampling. This behavior fits perfectly with our hypothesis. The adaptive sampling produces the photons which are likely to cause conflicts and are thusly the most expensive photons to be marched. As a result, we cannot expect that the adaptive photon marching pass is twice as fast with only half the photons.

In conclusion, we saw that even though less photons need to be used in the adaptive pipeline, the pipeline does not have a better performance. The reason is that the additional passes in the adaptive pipeline consume more time than we could save by using less photons. However, since the adaptive pipeline needs only half the photons and the overhead of the additional passes is constant the adaptive pipeline should become more efficient when the reference uses many photons. This behavior is shown in Figure 4.16. It shows the time consumption of the reference photon marcher for a given number of photons. The line of the adaptive pipeline shows the execution time of its photon marcher to obtain the same image quality as the reference. Therefore, the photon count axis is scaled by the factor two because the adaptive pipeline needs only half the number of photons. It can be seen that the adaptive pipeline becomes more efficient when the reference uses more than 200'000 photons. This means that the adaptive pipeline uses only 100'000 photons. In this situation the time we save by using less photons is more than we have to spend in the additional passes.

Photon Count			Execution Time			Conflicts		
U	A	C	U	A	C	U	A	C
38799	26414	11948	54	43	11	24889	44066	5649
47904	32262	15660	66	52	13	39701	77790	11675
57958	38315	19198	79	59	15	60110	88351	17045
68979	45372	23608	94	71	17	93078	132654	27423
80959	52668	27898	109	82	20	128068	147412	40329
93892	60965	32994	127	94	24	179978	252559	54267
107789	69362	37942	145	107	27	228849	306161	76077
122637	78860	43830	165	122	30	281110	360183	100619

Table 4.5: Photon count, timing and conflicts in the photon marching pass for a uniform (U), adaptive (A) and complement (C) photon sampling.

Unfortunately, all our scenes needed at most 150'000 photons with the reference pipeline.

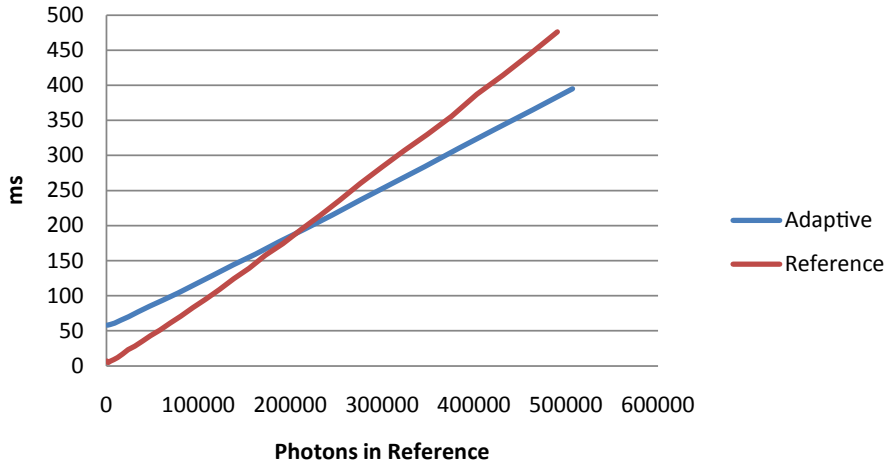


Figure 4.16: Since the adaptive pipeline needs only half the photons and the overhead of the additional passes is constant the adaptive pipeline becomes more efficient when the reference uses more than 200'000 photons. The scene we rendered is the same as in Figure 4.1

Chapter 5

Conclusion and Future Work

In this master's thesis we built two pipelines to render images with refractive objects in participating media in real time. We first reimplemented an adaption of the rendering pipeline by Sun et al. [SZS+08]. In the a second phase we tried to accelerate the existing pipeline. Our approach was to reduce the number of photons by applying a smarter photon distribution. This resulted in the new adaptive rendering pipeline, which was assumed to be faster. To evaluate the adaptive pipeline we compared the performance as well as the image quality of the two pipelines. The results showed that we were able to reduce the number of photons while maintaining the image quality. However, the adaptive pipeline has additional passes, which are necessary to, firstly, determine the photon distribution and, secondly, to reconstruct a noise-free image. Unfortunately, the additional passes consume too much time. The time saved from marching less photons is not enough to compensate for the additional passes. Only when the reference uses more than 200'000 photons has the adaptive pipeline a performance advantage. Unfortunately, we never needed more than 150'000 photon to obtain a smooth image for all the scenes we rendered.

We showed that the ordering of the photons in the photon list influences the performance of the photon marching strongly. In this sense, it would be interesting to attempt to find a better ordering in the photons list for the Hammersley sequence.

We saw that the adaptive pipeline spends much time on the additional passes. Thus, to increase the efficiency of the adaptive pipeline one could try to reduce the overhead produced by the additional passes in order.

An unhandy situation is that our adaptive rendering pipeline relies on several user supplied thresholds. This is problematic because some thresholds depend on the lighting condition. An example is the luminance threshold in the importance photon distribution pass (Section 3.2.2). As a result, it might be necessary to change the thresholds when the lighting changes. Therefore, it would be worth implementing an automatic thresholds adjustment.

In all photon marching passes a voxel grid of radiances is constructed. The radiance distribution in a voxel is coarsely approximate with only a weighted average direction and the average color of all photons passing through this voxel. Although this leads to acceptable results, a more accurate representation would result in a closer approximation to the physically correct solution. A better approximation could be achieved, for example, by using spherical harmonics [M66̈].

A drawback of both rendering pipelines is that they require a lot of GPU memory. At

the time only small scenes can be rendered within a grid of maximally 128^3. Thus, a smarter representation of the voxel grids is required to use the pipeline for large scenes.

Further possible improvements would be to include multiple-scattering, color shift or the inclusion of physical animation models such as fluid simulation.

Appendix A

Reference Pipeline

A.1 Voxelizer Pass

Listing A.1: Voxelizer GLSL Shader

```
#version 130
#extension GL_EXT_gpu_shader4 : enable
#extension GL_EXT_texture_integer  : enable
#extension GL_ARB_draw_buffers : enable

// required number of framebuffers/textures for voxelizing the model
uniform int textureCount;

// = 2^32 -1 = all 32 bits are set to one
const uint maxUint = uint(4294967295);

out uvec4 gl_FragData[gl_MaxDrawBuffers];

void main()
{
   //128 because we have RGBA hence 4*32 bit
   float sliceSize = 1.0/float(textureCount);
   float voxelSize = sliceSize/128.0;

   // tex is the texture slice in the 3d array
   int tex = int((gl_FragCoord.z) / sliceSize);

   // texCoord is the voxel slice in the 3d array slice
   int texCoord = int((gl_FragCoord.z - float(tex) * sliceSize)/voxelSize);

   // set all slices closer to cam to maxUint (= all bits to one)
   for(int i = 0; i < tex; i++)
      gl_FragData[i] = uvec4(maxUint);

   // set all slices farther from cam to 0
   for(int i = tex+1; i < textureCount; i++)
      gl_FragData[i] = uvec4(0);

   // compute slice color correctly
   if(texCoord < 32)
      gl_FragData[tex] = uvec4((1<<(texCoord))-1, 0, 0, 0);
   else if(texCoord < 64)
```

```
        gl_FragData[tex] = uvec4( maxUint, (1<<(texCoord-32))-1, 0, 0);
    else if(texCoord < 96)
        gl_FragData[tex] = uvec4( maxUint, maxUint, (1<<(texCoord-64))-1, 0);
    else
        gl_FragData[tex] = uvec4( maxUint, maxUint, maxUint, (1<<(texCoord-96))
            -1);
}
```

A.2 Octree

Listing A.2: CUDA kernels for the octree construction

```
__global__ void buildOctreeLevel(unsigned char *octreeDst, unsigned char *
    octreeSrc, float *refractionSrc, int level, float epsilon)
{
   dim3 dstCoord(threadIdx.x, blockIdx.x, blockIdx.y);
   dim3 octreeSrcCoord(dstCoord.x>>1,dstCoord.y>>1,dstCoord.z>>1);
   unsigned int dimSrcOctree = blockDim.x>>1;
   unsigned int dimDst = blockDim.x;

   if(octreeSrc[octreeSrcCoord.x + octreeSrcCoord.y*dimSrcOctree +
       octreeSrcCoord.z*dimSrcOctree*dimSrcOctree] > 0)
   {
      octreeDst[dstCoord.x + dstCoord.y*dimDst + dstCoord.z*dimDst*dimDst] =
          octreeSrc[octreeSrcCoord.x + octreeSrcCoord.y*dimSrcOctree +
          octreeSrcCoord.z*dimSrcOctree*dimSrcOctree];
   }
   else
   {
      float min = refractionSrc[2*(dstCoord.x + dstCoord.y*dimDst + dstCoord.z*
          dimDst*dimDst)];
      float max = refractionSrc[2*(dstCoord.x + dstCoord.y*dimDst + dstCoord.z*
          dimDst*dimDst) + 1];
      if(max-min < epsilon)
         octreeDst[dstCoord.x + dstCoord.y*dimDst + dstCoord.z*dimDst*dimDst] =
             level;
   }
}

__global__ void buildIntermediateRefractionLevel(float *minmaxRefrDst, float *
    refractionSrc, int level)
{
   dim3 dstCoord(threadIdx.x, blockIdx.x, blockIdx.y);
   dim3 srcCoord(dstCoord.x<<1,dstCoord.y<<1,dstCoord.z<<1);
   unsigned int dimSrc = blockDim.x<<1;
   unsigned int dimDst = blockDim.x;

   // certainly big/small enough as a refraction index
   float min=1000.0f, max=-1000.0f;

   for(int k= 0; k<2;k++)
   for(int j= 0; j<2;j++)
   for(int i= 0; i<2;i++)
   {
      // finest refractionSrc has only one value per element and is aligned
      // differently
      if(level == 1)
      {
         min=fminf(refractionSrc[(srcCoord.x+i)+ (srcCoord.y+j)*dimSrc + (
             srcCoord.z+k)*dimSrc*dimSrc],min);
         max=fmaxf(refractionSrc[(srcCoord.x+i)+ (srcCoord.y+j)*dimSrc + (
             srcCoord.z+k)*dimSrc*dimSrc],max);
      }
      else
      {
```

```
            min=fminf(refractionSrc[2*(srcCoord.x+i+ (srcCoord.y+j)*dimSrc + (
                srcCoord.z+k)*dimSrc*dimSrc)],min);
            max=fmaxf(refractionSrc[2*(srcCoord.x+i+ (srcCoord.y+j)*dimSrc + (
                srcCoord.z+k)*dimSrc*dimSrc) + 1],max);
        }
    }

    minmaxRefrDst[2*(dstCoord.x + dstCoord.y*dimDst + dstCoord.z*dimDst*dimDst)]
         = min;
    minmaxRefrDst[2*(dstCoord.x + dstCoord.y*dimDst + dstCoord.z*dimDst*dimDst)
        +1] = max;
}

__global__ void buildLowestOctreeLevel(unsigned char *octreeDst, float *
    refractionSrc, int maxLevel, float epsilon)
{
    if(refractionSrc[1] - refractionSrc[0] < epsilon)
        octreeDst[0] = maxLevel;
    else
        octreeDst[0] = 0;
}

/**
  * octrees stores the level assingments for the top down part
  * refrSrc stores the minimum and maximum refraction indices. refrSrc[0] holds
  * the refraction indices from the voxelization pass
  */
void runOctreeKernel(unsigned char **octrees, float **refrSrc, int maxLevel,
    float epsilon)
{
    unsigned int dimOfLevel;

    // bottom up
    for(int i= maxLevel; i > 0 ; i--)
    {
        dimOfLevel = 1<<(i-1);
        buildIntermediateRefractionLevel<<<dim3(dimOfLevel, dimOfLevel),
            dimOfLevel>>>(refrSrc[maxLevel-i+1], refrSrc[maxLevel-i], maxLeve-i+1)
            ;
    }

    // top down
    buildLowestOctreeLevel<<<1,1>>>(octrees[maxLevel],refrSrc[maxLevel],
        maxLevel, epsilon);
    for(int i = maxLevel-1; i >= 0 ; i--)
    {
        dimOfLevel = 1 <<(maxLevel - i);
        buildOctreeLevel<<<dim3(dimOfLevel, dimOfLevel), dimOfLevel>>>(octrees[i
            ], octrees[ i + 1], refrSrc[i], i, epsilon);
    }
}
```

A.3 Photon Generation

Listing A.3: Photon Generation Kernel

```
/**
  * c_data is a structure which holds commonly used data in constant memory
  */
__global__ void calcPhotonsReference(float3 *positions, float3 *directions,
    float3 *radiances, float *hammersley)
{
   int photonId = threadIdx.x+blockDim.x*blockIdx.x;
   if(photonId > c_data.photonCount*c_data.photonCount)
      return;

   // for a point light source the direction is enough the compute the
   // intersection
   directions[photonId] = getPhotonDirection(hammersley[photonId*2],
       hammersley[photonId*2+1]*c_data.photonCount);

   // when photon misses grid then NAN to mask this photon
   positions[photonId] = intersectWithGrid(directions[photonId]);

   // weight photon color
   radiances[photonId] = make_float3(c_data.lightCol.x/c_data.photonCount,
       c_data.lightCol.y/c_data.photonCount, c_data.lightCol.z/c_data.
       photonCount);
}

/**  KERNEL INVOCATION
  *
  * BLOCK_SIZE is the CUDA block size = number of threads per block
  * d_positions, d_directions, d_radiances are float3 arrays of the same size
  *      as photonCount
  * d_hamm is an array with the Hammersley sequence. Organized as
  *      [x1, y1, x2, y2, ....]
  */
calcPhotonsReference<<<photonCount/BLOCK_SIZE+1, BLOCK_SIZE>>>(d_positions,
    d_directions, d_radiances, d_hamm);
```

A.4 Viewing Pass

Listing A.4: Viewing Pass Kernel

```
typedef struct _Ray{
    float3 pos;
    float3 dir;
    float3 col;
} Ray;

/*
 * extTex, scatTex, gradTex, colTex are CUDA texture reference
 * c_data is a structure which holds commonly used data
 */
__device__ void marchRay(Ray *outputRay)
{
    const float stepSize = c_data.voxelSize;
    float3 oldPos = outputRay->pos;
    float3 oldDir = outputRay->dir;
    // transmittance
    float3 T = make_float3(1.f,1.f,1.f);

    // variables for isosurface reflection
    float currentRefrIdx;
    bool reflected = false;

    // scattering variables
    float attenuation;
    float phaseFunction;
    float cosPhi;
    const float scattering = c_data.rayStepSize*(1.0f - c_data.phaseFuncK*c_data
        .phaseFuncK);

    // voxel coordinates
    float3 voxelCoord;

    // general helper variables
    float colScale                          = 1.f;
    float delta_s;

    while(!isPosOutside(outputRay->pos, c_data.bbMin, c_data.bbMax)) {
        // coordinate transformation from global to voxel coordinates
        voxelCoord = getVoxelCoordinates(outputRay->pos)

        // reflective table
        if(outputRay->dir.y != 0.0f && (int) voxelCoord.y == 0)
        {
            // determine exact intersection
            float scale = __fdividef(c_data.bbMin.y + c_data.voxelSize - oldPos.y,
                oldDir.y);
            outputRay->pos.x = oldPos.x + scale*oldDir.x;
            outputRay->pos.y = oldPos.y + scale*oldDir.y;
            outputRay->pos.z = oldPos.z + scale*oldDir.z;

            // coordinate transformation from global to voxel coordinates
            voxelCoord = getVoxelCoordinates(outputRay->pos)
```

```
        // get color of bottom voxels and do scaling (needed when isosurface
            reflection occurred earlier)
        outputRay->col.x += tex3D(colTex, voxelCoord.x, voxelCoord.y,
            voxelCoord.z).x*colScale*TABLE_COL_R;
        outputRay->col.y += tex3D(colTex, voxelCoord.x, voxelCoord.y,
            voxelCoord.z).y*colScale*TABLE_COL_G;
        outputRay->col.z += tex3D(colTex, voxelCoord.x, voxelCoord.y,
            voxelCoord.z).z*colScale*TABLE_COL_B;

        outputRay->dir = getReflectionDir(oldDir, make_float3(0,1,0));
    }
    // reflective isosurface
    else if(!reflected && c_data.reflectionPercentage > 0 && (currentRefrIdx
        = tex3D(refrTex, voxelCoord.x, voxelCoord.y, voxelCoord.z)) >= c_data.
        isosurfIdx)
    {
        // find exact intersection with a binary search
        voxelCoord = binSearchPosition(outputRay, currentRefrIdx, c_data.
            isosurfIdx);
        float3 interpolatedDir = binSearchPosition(outputRay, currentRefrIdx,
            c_data.isosurfIdx);

        // the refraction index gradient is the normal of the isosurface
        float3 surfaceNormal = tex3D(gradTex, voxelCoord.x, voxelCoord.y,
            voxelCoord.z);
        negate(surfaceNormal);
        normalize(surfaceNormal);
        interpolatedDir = getReflectionDir(interpolatedDir, surfaceNormal);

        // just do environment map lookup
        envMapLookUp(outputRay, interpolatedDir, colScale*c_data.
            reflectionPercentage, T );

        reflected = true;
        colScale *= 1 - c_data.reflectionPercentage;
}

// coordinate transformation from global to voxel coordinates
voxelCoord = getVoxelCoordinates(outputRay->pos)

// scattering phase function here we compute the cosine between the ray
// direction and the radiance direction (normalization of the ray direction
// is done in the next line)
cosPhi = -outputRay->dir.x * tex3D(dirTex, voxelCoord.x, voxelCoord.y,
    voxelCoord.z).x +
   -outputRay->dir.y * tex3D(dirTex, voxelCoord.x, voxelCoord.y, voxelCoord.
       z).y +
   -outputRay->dir.z * tex3D(dirTex, voxelCoord.x, voxelCoord.y, voxelCoord.
       z).z;
// here we also normalize direction
cosPhi =  1 - c_data.phaseFuncK*cosPhi/length(outputRay->dir);

phaseFunction = scattering * tex3D(scatTex, voxelCoord.x, voxelCoord.y,
    voxelCoord.z).y/ (4 * MATH_PI * cosPhi*cosPhi);

// attenuation and transmittance factor
```

```
    attenuation = expf(-tex3D(extTex, voxelCoord.x, voxelCoord.y, voxelCoord.z)*
        stepSize);
    T.x *= attenuation;
    T.y *= attenuation;
    T.z *= attenuation;

    // final contribution and scaling  (needed when isosurface reflection
    // occurred earlier)
    outputRay->col.x += colScale*phaseFunction * tex3D(colTex, voxelCoord.x,
        voxelCoord.y, voxelCoord.z).x * T.x;
    outputRay->col.y += colScale*phaseFunction * tex3D(colTex, voxelCoord.x,
        voxelCoord.y, voxelCoord.z).y * T.y;
    outputRay->col.z += colScale*phaseFunction * tex3D(colTex, voxelCoord.x,
        voxelCoord.y, voxelCoord.z).z * T.z;

    // make sure we move position by stepSize
    delta_s = stepSize/length(outputRay->dir);

    // set new position and direction
    oldPos = outputRay->pos;
    outputRay->pos.x += delta_s*outputRay->dir.x;
    outputRay->pos.y += delta_s*outputRay->dir.y;
    outputRay->pos.z += delta_s*outputRay->dir.z;

    oldDir = outputRay->dir;

    outputRay->dir.x += stepSize *
       tex3D(refrTex, voxelCoord.x, voxelCoord.y, voxelCoord.z)*
       tex3D(gradTex, voxelCoord.x, voxelCoord.y, voxelCoord.z).x;
    outputRay->dir.y += stepSize *
       tex3D(refrTex, voxelCoord.x, voxelCoord.y, voxelCoord.z)*
       tex3D(gradTex, voxelCoord.x, voxelCoord.y, voxelCoord.z).y;
    outputRay->dir.z += stepSize *
       tex3D(refrTex, voxelCoord.x, voxelCoord.y, voxelCoord.z)*
       tex3D(gradTex, voxelCoord.x, voxelCoord.y, voxelCoord.z).z;
    }

    envMapLookUp(outputRay, colScale, T);
}

__global__ void setPixelColors(GLuint *d_frameBuffer)
{
    // retrieve pixel coordinate of ray
    int2 pixel = make_int2(threadIdx.x+blockIdx.x*BLOCK_WIDTH, threadIdx.y+
        blockIdx.y*BLOCK_HEIGHT);
    if(pixel.x >= c_data.winSize.x ||
       pixel.y >= c_data.winSize.y)
       return;
    int pixelId = pixel.x+pixel.y*c_data.winSize.x;

    Ray primRay;

    // fill position and direction of ray
    prepareRay(&primRay, pos);

    // place ray on the grid
    intersectWithGrid(&primRay);
```

```
    marchRay(&primRay);

    // clamp RGB values to 1 and pack into an unsigned int
    d_frameBuffer[pixelId] =
        (((GLuint) (fminf(1,primRay.col.x)*255))<<24) |
        (((GLuint) (fminf(1,primRay.col.y)*255))<<16) |
        (((GLuint) (fminf(1,primRay.col.z)*255))<<8 ) ;
}

/**  KERNEL INVOCATION
  *
  * BLOCK_WIDTH, BLOCK_WIDTH defines the size of a CUDA block = width*heitgh
  *       threads with this we split the image into smaller subblocks. This
  *       leads to faster result than just splitting the image row or column
  *       wise.
  * d_frameBuffer, is a OpenGL Pixel Buffer Object which is then used to draw
  *       the image with glDrawPixels
  * g_data is a structure which holds commonly used data
  */
dim3 gridDim(g_data.winSize.x/BLOCK_WIDTH + 1, g_data.winSize.y/BLOCK_HEIGHT+1)
    ;
dim3 blockDim(BLOCK_WIDTH, BLOCK_HEIGHT);
setPixelColors<<<gridDim, blockDim>>>(d_frameBuffer);
```

Appendix B

Adaptive Pipeline

B.1 Importance Photon Distribution

Listing B.1: The GLSL Shader for rendering the importance photon distribution

```
attribute float gradient;
attribute float n;
attribute vec4 radiance;
attribute vec2 sumPosSqrd;
uniform float gradThreshold;
uniform float radThreshold;


#define MATH_PI 3.14159265

void main()
{
    gl_Position = gl_Vertex;
    // mean as center
    gl_Position.x = 0.5 + gl_Position.x / n;
    gl_Position.y = 0.5 + gl_Position.y / n;

    // variance as point size (diameter of circle). At least draw the center
    // point thus max(1, ..)
    gl_PointSize = max(1.0, sqrt(sumPosSqrd.x / n - gl_Position.x*gl_Position.x)
        + sqrt(sumPosSqrd.y / n - gl_Position.y*gl_Position.y));

    float radius = gl_PointSize;
    gl_Position = gl_ModelViewProjectionMatrix*gl_Position;


    if(n > 0 && (gradient >= gradThreshold || ((radiance.r+radiance.g+radiance.b
        ) <= radThreshold && radiance.r+radiance.g+radiance.b != 0.0)))
       gl_FrontColor = vec4(1);
    else
       gl_FrontColor = vec4(0);
}
```

B.2 Mipmap Generation

Listing B.2: Box Filter for the Photon Weights

```
__global__ void buildMipmapPhotonWeight(float *out, float *src)
{
   dim3 srcCoord(threadIdx.x<<1,blockIdx.x<<1,blockIdx.y<<1);
   int srcDim = blockDim.x<<1;

   float weight = 0;
   int c = 0;
   for(int k= 0; k<2;k++)
   for(int j= 0; j<2;j++)
   for(int i= 0; i<2;i++)
   {
      if(src[(srcCoord.x+i) + (srcCoord.y+j)*srcDim + (srcCoord.z+k)*srcDim*
          srcDim] > 0)
      {
         weight += src[(srcCoord.x+i) + (srcCoord.y+j)*srcDim + (srcCoord.z+k)*
             srcDim*srcDim];
         c++;
      }
   }

   if(c != 0)
      out[threadIdx.x + blockIdx.x*blockDim.x + blockIdx.y*blockDim.x*gridDim.x
          ] = weight/c;
   else
      out[threadIdx.x + blockIdx.x*blockDim.x + blockIdx.y*blockDim.x*gridDim.x
          ] = 0;
}
```

B.3 Filtering

Listing B.3: Filtering Kernel

```
/**
  * d_finalRadianceCol, d_finalRadianceDir are the color and direction array
  *      which is passed to the viewing pass
  * d_mipmapCol, d_mipmapDir, d_mipmap_n, d_mipmap_photonWeight hold the color,
  *      direction, photon count and photon weight mipmap pyramid
  * maxLevel is the largest possible mipmap level
  * deltaWeight is the user selected value
  */
__global__ void buildFinalRadiances(cudaPitchedPtr d_finalRadianceCol,
    cudaPitchedPtr d_finalRadianceDir, float3 **d_mipmapCol, float3 **
    d_mipmapDir, int **d_mipmap_n, float **d_mipmap_photonWeight, int maxLevel,
    float deltaWeight)
{
   // PHOTON WEIGHT FILTER

   // photon count to determine if the photon weight on the lowest level is
       used
   float pc = getInterpolatedValue(d_mipmap_n, wLevel);

   // when 0 photons passed through this voxel then the photon weight cannot be
   // used. Then we used the photon weight from a hight mipmap level
   while(pc < 1)
   {
      wLevel++;
      weight = getInterpolatedValue(d_mipmap_n, wLevel);
   }

   // the photon weight we use to determine the mipmap level
   float weight = getInterpolatedValue(d_mipmap_photonWeight, wLevel);

   // stores the level suggested by the photon weight filter
   int wLevel = max(0,min(maxLevel, (int)(weight / deltaWeight)));


   // CENTER SURROUND FILTER

   // stores the level suggested by the center surround filter
   int cenLevel = 0;
   // stores the color value of the inner circle
   float3 center = getInterpolatedValue(d_mipmapCol, curLevel);
   // stores the color value of the outer circle
   float3 surround = getInterpolatedValue(d_mipmapCol, curLevel+1);
   // stores the difference between the inner and outer circle
   float diff = fabsf(center.x+center.y+center.z -surround.x-surround.y-
       surround.z);

   // to detect when the difference increases we need to know the current and
   // the next difference
   center = surround;
   surround = getInterpolatedValue(d_mipmapCol, curLevel+2);
   float nextDiff = fabsf(center.x+center.y+center.z -surround.x-surround.y-
       surround.z);
```

```
    // we increase the level/filter radius as long as the difference does not
    // decrease
    while( nextDiff <= curDiff && curLevel+2 < maxLevel)
    {
        curLevel++;
        center = surround;
        surround = getInterpolatedValue(d_mipmapCol, curLevel+2);
        curDiff = nextDiff;
        nextDiff =  fabsf(center.x+center.y+center.z -surround.x-surround.y-
            surround.z);
    }

    // FILTER COMBINATION
    curLevel = min(wLevel, curLevel);

    setInterpolatedData(curLevel, d_finalRadianceCol, d_finalRadianceDir,
        d_mipmapCol, d_mipmapDir);
}
```

List of Tables

List of Figures

Bibliography

[Bli82] Blinn, James F.: Light reflection functions for simulation of clouds and dusty surfaces. In: *SIGGRAPH '82: Proceedings of the 9th annual conference on Computer graphics and interactive techniques*. New York, NY, USA : ACM, 1982. – ISBN 0–89791–076–1, S. 21–29

[Bor99] Born, Max: *Principles of Optics*. Cambridge : Cambridge University Press, 1999. – ISBN 0521642221

[CLT07] Crane, K. ; Llamas, I. ; Tariq, S.: Real-Time Simulation and Rendering of 3D Fluids. In: *GPU Gems 3*, Addison-Wesley, 2007, S. Ch. 30, 633 – 675

[CT09] Cederman, Daniel ; Tsigas, Philippas: GPU-Quicksort: A practical Quicksort algorithm for graphics processors. In: *J. Exp. Algorithmics* 14 (2009), S. 1.4–1.24. `http://dx.doi.org/http://doi.acm.org/10.1145/1498698.1564500`. – DOI http://doi.acm.org/10.1145/1498698.1564500. – ISSN 1084–6654

[DBMS02] Dmitriev, Kirill ; Brabec, Stefan ; Myszkowski, Karol ; Seidel, Hans-Peter: Interactive global illumination using selective photon tracing. In: *EGRW '02: Proceedings of the 13th Eurographics workshop on Rendering*. Aire-la-Ville, Switzerland, Switzerland : Eurographics Association, 2002. – ISBN 1–58113–534–3, S. 25–36

[ED06] Eisemann, Elmar ; Décoret, Xavier: Fast Scene Voxelization and Applications. In: *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM SIGGRAPH, 2006, 71-78

[ED08] Eisemann, Elmar ; Décoret, Xavier: Single-pass GPU Solid Voxelization and Applications. In: *GI '08: Proceedings of Graphics Interface 2008* Bd. 322, Canadian Information Processing Society, 2008 (ACM International Conference Proceeding Series), 73–80

[FFCC00] Fang, Shiaofen ; Fang, Shiaofen ; Chen, Hongsheng ; Chen, Hongsheng: Hardware Accelerated Voxelization. In: *Computers and Graphics* 24 (2000), S. 200–0

[GS92] Grimmett, G. R. ; Stirzaker, D. R.: *Probability and random processes*. Second. New York : The Clarendon Press Oxford University Press, 1992. – xii+541 S. – ISBN 0–19–853666–6; 0–19–853665–8

[GTGB84] Goral, Cindy M. ; Torrance, Kenneth E. ; Greenberg, Donald P. ; Battaile, Bennett: Modeling the interaction of light between diffuse

surfaces. In: *SIGGRAPH Comput. Graph.* 18 (1984), Nr. 3, S. 213–222. http://dx.doi.org/http://doi.acm.org/10.1145/964965.808601. – DOI http://doi.acm.org/10.1145/964965.808601. – ISSN 0097–8930

[Hul81] Hulst, H. C. d.: *Light Scattering by Small Particles*. Dover Publications, 1981. – ISBN 0486642283

[IZT+07] Ihrke, I. ; Ziegler, G. ; Tevs, A. ; Theobalt, C. ; Magnor, M. ; Seidel, H.-P.: Eikonal Rendering: Efficient Light Transport in Refractive Objects. In: *ACM Trans. on Graphics (Siggraph'07)* (2007), August, S. to appear

[JC95] Jensen, Henrik W. ; Christensen, Niels J.: Photon maps in bidirectional Monte Carlo ray tracing of complex objects. In: *Computers & Graphics* 19 (1995), Nr. 2, S. 215–224

[JC98] Jensen, Henrik W. ; Christensen, Per H.: Efficient simulation of light transport in scences with participating media using photon maps. In: *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*. New York, NY, USA : ACM, 1998. – ISBN 0–89791–999–8, S. 311–320

[Jen09] Jensen, Henrik W.: *Realistic Image Synthesis Using Photon Mapping*. Natick, MA, USA : A. K. Peters, Ltd., 2009. – ISBN 1568814623

[Kaj86] Kajiya, James T.: The rendering equation. In: *SIGGRAPH Comput. Graph.* 20 (1986), Nr. 4, S. 143–150. http://dx.doi.org/http://doi.acm.org/10.1145/15886.15902. – DOI http://doi.acm.org/10.1145/15886.15902. – ISSN 0097–8930

[Mö66] Müller, Claus: *Spherical harmonics*. Berlin : Springer-Verlag, 1966

[Nie92] Niederreiter, Harald: *Random Number Generation and Quasi-Monte Carlo Methods*. Philadelphia : Society for Industrial Mathematics, 1992. – ISBN 0–89–871295–5

[NVI09a] NVIDIA (Hrsg.): *NVIDIA CUDA C Programming Best Practices Guide*. 2.3. NVIDIA, 2009

[NVI09b] NVIDIA (Hrsg.): *NVIDIA CUDA Programming Guide*. 2.3. NVIDIA, 2009

[PH04a] Pharr, Matt ; Humphreys, Greg: *Infinite area light source with importance sampling*. 2004

[PH04b] Pharr, Matt ; Humphreys, Greg: *Physically Based Rendering: From Theory to Implementation (The Interactive 3d Technology Series)*. Morgan Kaufmann, 2004 http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/012553180X. – ISBN 012553180X

[SH01] Siegel, Robert ; Howell, John: *Thermal Radiation Heat Transfer*. Taylor and Francis, 2001. – ISBN 1560328398

[SZS+08] Sun, Xin ; Zhou, Kun ; Stollnitz, Eric ; Shi, Jiaoying ; Guo, Baining: Interactive relighting of dynamic refractive objects. In: *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*. New York, NY, USA : ACM, 2008, S. 1–9

[War91] Ward, Greg: Real Pixels. In: *In Graphics Gems II*, 1991, S. 80–83

[Whi79] Whitted, T.: An improved illumination model for shaded display. In: *SIGGRAPH Comput. Graph.* 13 (1979), Nr. 2, S. 14. http://dx.doi.org/http://doi.acm.org/10.1145/965103.807419. – DOI http://doi.acm.org/10.1145/965103.807419. – ISSN 0097–8930

[WLH97] Wong, Tien-Tsin ; Luk, Wai-Shing ; Heng, Pheng-Ann: Sampling with Hammersley and Halton Points. In: *journal of graphics, gpu, and game tools* 2 (1997), Nr. 2, S. 9–24

[WN09] Wyman, Chris ; Nichols, Greg: Adaptive Caustic Maps Using Deferred Shading. In: *Comput. Graph. Forum* 28 (2009), Nr. 2, S. 309–318