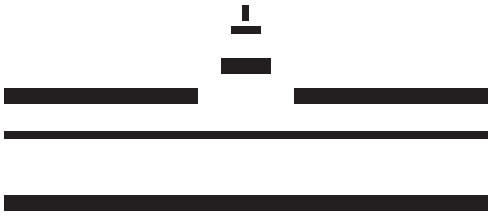


Westfälische
Wilhelms-Universität
Münster

Department of Computer Science
Visualization & Computer Graphics Research Group

Offline BRDF Computation for Microscale Volumes

Bachelor Thesis

submitted by:

Karsten Jeschkies

Matriculation number: 342058

Supervisor: PD Dr. Timo Ropinski

Münster, August 13th 2010

Abstract

While the physics of light are well known, computer scientists still struggle to use this knowledge for real-time rendering of photo realistic images. Especially the phenomena of light in microscale magnitudes are hard to simulate or estimate in real-time.
This thesis will propose a way to pre-calculate the behaviour of light in microscale magnitudes which can be described by Bidirectional Reflectance Distribution Functions (BRDF). The described pre-calculation or offline computation will be specialized on volumes.
A modified Path Tracer named Photon Tracer will be used to compute BRDF data offline. This means the data is calculated and stored and can then be later used to render more realistic images in real-time.

Contents

Chapter 1

Introduction

Ever since the arrival of the computer in the middle of the last century people tried to create a digital copy of the real world on the computer. They started to render the world on the screen. The ultimate goal was and still is to generate a photo-realistic image. When we compare computer generated images from the 1970's and today's images of video games we see that scientists not only have developed ways to create more realistic images, the images can be also rendered in real time to some extent. However, even some of the biggest computer clusters of today's special-effects companies hardly succeed in creating convincingly realistic images. This might come as a surprise because the underlining physics of light are actually well-understood today.

In 1986, Kajiya introduced the rendering equation [Kaj86]. This equation is supposed to describe the behaviour of light when it is emitted and reflected. That would mean one just has to let computer renderers solve the equation to reach the ultimate goal of photo-realistic graphics. Unfortunately, there are several problems. Frist, the equation is not discrete. Thus, computers have to estimate a solution. This leads to a second problem. Estimations with small error cost a lot of performance and are hardly practical for real-time renderings or even non-real-time renderings. For example, even though we can write a program which simulates the reflection of light on a microscale level the program would not generate an image of a landscape in an acceptable time span.

This thesis will propose a way how to render scenes with many objects on a relatively large scale in real-time while still considering the microscale structures of the rendered objects. The solution will be based on the theory of off-line computing *Bidirectional Reflectance Distribution Functions* (BRDF) proposed by Westin et al. [WAT92]. The theory will be extended to renderings of volumes. To compute BRDFs a Path Tracer for volumes will be developed and then modified to trace photons through a volume specimen.

The first part of this paper will describe the problem and proposed solution in detail. A short overview of the theory of BRDFs and the Path Tracing algorithm will be given. At the end of the first part two helper algorithms will be explained which will be used in the implementation of a modified Path Tracer.
The second part of the paper will describe the implementation of the Path Tracer for volumes and its modification to generate BRDF data.
The last part of the paper will discuss several problems of the implementation and draw a conclusion.

Chapter 2

Related Work

2.1 Bidirectional Reflectance Distribution Function (BRDF)

2.1.1 Introduction to BRDF Theory

When a light beam hits a surface of a material several things can happen. The light can be absorbed by the material. It can go right through the material as it happens when light travels through the air. When the beam enters or leaves a material there can be a refraction. That means the light beam bends right at the surface. This effect can be easily observed when we dip a pencil half way into a fish tank: the pencil seems to be bend at the water surface as seen in the figure 2.1. Light which is reflected by the pencil in the water and leaves the water is bend at the surface of the water.
Another well-known effect is the reflection of light at a surface. The angle at which light is reflected can be easily calculated. The angle of incidence equals the angle of reflectance. However, this is only true for perfect reflection on perfectly smooth surfaces. Most materials in the real physical world do not reflect light perfectly, though. They are not perfectly smooth. Brushed metal for instance seems to be very smooth when we look at it with our eyes. Observed with a microscope we discover little bumps at the surface. Light which hits the surface of brushed metal is reflected at these little bumps across the metal and leaves the metal surface at different places as seen in figure 2.2. That means to render realistic light effects the interaction of light with the microgeometry of an object has to be simulated.

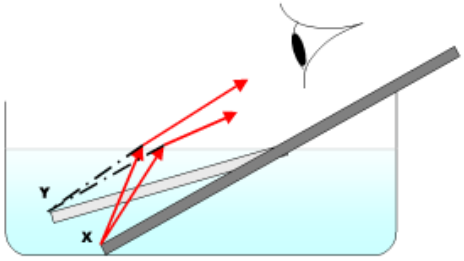


Figure 2.1: The pencil seems to bend at the surface of the water. (The image is taken with permissions from [Wik10])

(a) (b)

Figure 2.2: Light reflection on a rough surface: **(a)** Light rays are reflected across the surface and emitted at different places. **(b)** Light rays are reflected at a rough surface. The amount of light reaching the eye depends on the perspective.

All of the described phenomena can happen at once. The light is scattered on and within the surfaces, it is emitted partially at different places and so on. The result is that the amount of light which reaches our eyes when we look at a point on a surfaces is not equal for all angels we look at the point. In other words, the portion of photons reaching our eyes depends on the direction they hit the point we are looking at and the direction they are reflected into our eyes. The direction we look at the point is our view direction as seen in figure 2.2 b [AMHH08]. Materials which appearance is view dependent are called anisotropic.
In computer graphics the portion is given by the bidirectional distribution function, BRDF. The BRDF is the ratio between differential outgoing radiance and differential irradiance. The basic form is given in equation 2.1.

$$f(l, v) = \frac{dL_o(v)}{dE_i(l)} \tag{2.1}$$

The view vector is l. The direction the photons hit the point we look at is v. dL_o denotes the outgoing light or radiance. It is the radiance which reaches our eyes. dE_i is the irradiance, the amount of photons which reach the point.

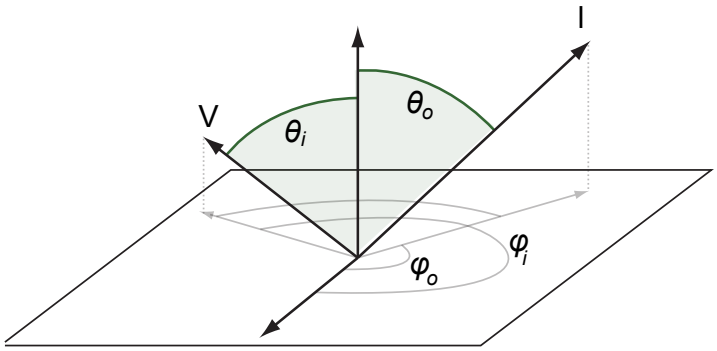


Figure 2.3: The azimuth angles are φ_i and φ_o and zenith angles are θ_i and θ_o.

The function can also be described as a four dimensional function (compare [Com05]) with the azimuth (φ_i and φ_o) and zenith angles as input as seen in equation 2.2. See figure 2.3 for an illustration of the angles.

$$f(\varphi_i, \theta_i, \varphi_o, \theta_o) = \frac{dL_o(\varphi_o, \theta_o)}{dE_i(\varphi_i, \theta_i)} \tag{2.2}$$

If we can determine the BRDF of a surface we can determine how many photons are reflected at a point and reach the view point. This can help us simulate many effects of the real world such as the appearance of satin or brushed metal.
Several models have been proposed to determine the BRDF of a material. However, most models come with problems. The next section will deal with such models and their problems.

2.1.2 BRDF Models and Problems

I have given a small introduction about the idea behind BRDFs. The following is a list of BRDF models which have been developed in the past. I will also discuss the difficulties they front.
A very basic version of a BRDF would be just a constant value for all angles. This approach, however, does not lead to realistic results. To achieve higher realism the BRDF needs more complex models. Basically, there are three approaches to define models for a BRDF. A model can be created using empirical methods, it can be based on physical theories and it can be represented by mathematical fitting functions.
One of the first BRDF models was introduced by Bui Tuong Phong [Pho75]. Phong's model is still widely used today because it is not too complex and delivers acceptable

results most of the time. However, it does not always lead to pleasing and realistic images. Several other models were introduced in the wake of Phong. All of them have the goal to deliver better and more realistic results for most materials. Westin et al. [WLT04] and Ngan et al. [NDM05] give good reviews on these. Most of these models are fast enough to be used for real-time renderings. They have also few coefficients and thus can be fairly easily adjusted by an artist to match certain materials. Comparisons to measured BRDF data from real world materials show, however, that these models cannot generate adequate results for every material (compare [NDM05]). Especially the appearances of anisotropic materials cannot be faithfully rendered with most models (compare [NDM05]).
In contrast, mathematical fitting representations of BRDFs are capable of representing every BRDF exactly. That means that they could be used to render even anisotropic materials. However, these models tend to have many coefficients (into the thousands). An artist would have difficulties to find the right settings for all coefficients to create a special material. Thus, mathematical models are not practical unless there is a way to ease the definition of materials.
Summing up, most BRDF models which can be adjusted easily and are fast enough for real-time rendering are not capable of describing all materials such as anisotropic surfaces. Mathematical fitting representations of BRDFs on the other hand are capable of such things. However, they are too complex to be matched to certain materials. Thus, the lack of an all purpose BRDF model proposes a problem. There is a need for a model which can be easily adapted to a material and is capable of describing every material including anisotropic materials.

2.1.3 A Different Approach and a Solution

When Ngan et al. [NDM05] analysed several BRDF models they compared the modelled data to BRDF data they measured from real world objects. They also used real world data for renderings. Lindemann and Ropinski [LR10] applied BRDF data for illuminations in volume renderings. Both showed that aggregated BRDF data can be used for real-time renderings.
A solution for the stated problem in section 2.1.2 could be to compute BRDF data offline instead of applying a BRDF model while rendering or a combination of both. The only drawback is that the data has to be accumulated somehow. Ngan et al. [NDM05] built a device to measure how light is reflected from certain materials in the real world. This proved to be a time-consuming task. For every angle of incoming light the amount of reflected light has to be measured from every perspective.

Another way of aggregating BRDF data is presented by Westin et al. [WAT92]. They defined different levels of detail. These levels range from macro levels to micro levels. While surface structures on a macro level can be modelled with polygons and effects such as bump mapping, structure details on micro levels need to be described by BRDFs. This could be done by a BRDF model. However, Westin et al. used a different method. They modelled the micro level with polygons. Then, they ray-traced the model surface. They simulated how light was scattered, reflected, refracted, transmitted and absorbed. Rays leaving the surface were measured. That means they measured how the photons hitting a microscale surface were distributed over the surface. In that way they gathered BRDF data on a hemisphere. The BRDF data was then used during real-time rendering on the macro level to predict the distribution of the photons. This approach is reciprocal. Computed BRDF data of one level can be used to computed BRDF data on the next higher level.
The described idea can be also used to gather BRDF data for volume rendering. Instead of simulating the light behaviour on a surface with a hemisphere the light could be measured when it interacts with voxels in a volume specimen with the light coming from a sphere around the specimen.
To simulate the light a modified Path Tracer could be used.

2.2 Path Tracing

In the last section I suggested that a modified Path Tracer could be used to compute the interaction of light with a microscale surface or volume. In this section I will give an introduction to the Path Tracing algorithm and explain how Path Tracing can be used to render volumes.

2.2.1 Introduction

Path Tracing was introduced by Kajiya [Kaj86]. The idea is to estimate the rendering integration which was introduced in the same paper with the help of Monte Carlo methods which will be explained in section 2.3.1. In older ray tracing algorithms rays were sent from each pixel of the resulting image into scene. When a ray hits an object its specular reflection was calculated, it was transmitted or several rays were spawned for diffuse reflection. The algorithms then start over and simulate the interactions of the reflected and newly spawned rays with the objects in the scene. This is done until the

rays hit a light emitter or a specified break condition is reached. The drawback of this technique is that a few diffuse reflections create a massive amount of new rays. The data handling becomes difficult. Kaijiya proposed a different approach. Instead of sending one ray per pixel he suggested to send thousands of rays per pixel. When a ray would be reflected diffusely just one new ray would be spawned. Its direction would be determined by Monte Carlo sampling methods. These methods should ensure that the reflected rays on a diffuse surface are distributed uniformly over the hemisphere of the surface.

2.2.2 Calculating the Specular and the Diffuse Reflection Vector

When a light beam or a ray hits a surface, that means it hits the interface of two different materials, we can observe several phenomena as described above (see 2.1.1). One of these is *reflection*. There can be *specular* and *diffuse* reflection. In this subsection I will describe how the direction of a reflected ray can be determined.

Specular Reflection

Specular reflections are mirror-like. They occur at very smooth surfaces. The direction of light is changed as can be seen in figure 2.4. In an ideal case the angle of incidence equals the angle of reflectance. Equation 2.3 shows how the direction of the reflected ray can be calculated. R_V is a vector pointing in the new direction of the ray. N is the surface Normal and L points *towards* the origin of the ray hitting the surface. The angle of incidence is denoted by θ.

$$R_V = 2 \cdot cos(\theta) \cdot N - L \tag{2.3}$$

Diffuse Reflection

Diffuse reflections occur at very rough surfaces. These surfaces consist of many micro-facets as seen in figure 2.2. An incoming ray is reflected in many directions as can be seen in figure 2.5.
There are several ways to determine new directions for diffuse reflected rays in ray tracing algorithms. Since the ray is reflected in many directions several new rays pointing to many

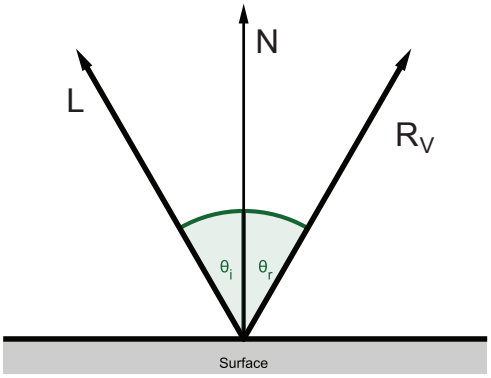


Figure 2.4: The angle of incidence is θ_i. L points towards the light source of the light ray. R_v is the reflection direction.

points on a hemisphere over the hit surface can be spawned. This approach, however, can lead to an overhead of new rays a described in section 2.2.1. In Path Tracing algorithms only one new ray is spawned for diffuse reflections. The new direction is determined by a Monte Carlo sampling method as described in 2.3.1. Instead of spawning a new ray the direction of the old ray can be changed of course. However, this is just an implementation detail.

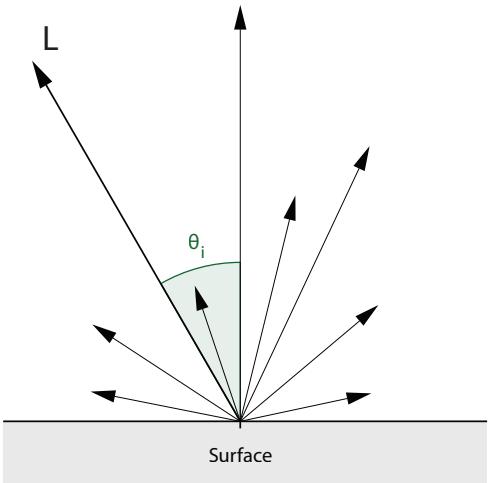


Figure 2.5: Diffuse reflection vectors: The light is reflected across the hemisphere. L is pointing towards the light source.

2.2.3 Color Compositing

When a light beam is reflected on a surface the color or intensity of the reflected light beam needs to be determined. In Path Tracing this can be done with the traditional Phong lighting methods.

For Specular Reflections

If the reflection of a ray is specular its intensity can be calculated with equation 2.4. I_s^{in} is the incoming specular intensity. In the case of Path Tracing this would be the intensity of the ray hitting the surface. p_s and f are parameters controlling the specular reflection of the surface such as the "shininess" of the material. φ is the angle between the vector V pointing towards the view point and the direction of reflection denoted by the vector R_v. $cos(\varphi)$ is calculated by the dot product of V and R_v.

$$I_s = p_s \cdot I_s^{in} \cdot cos(\varphi)^f = p_s \cdot I_s^{in} \cdot (V \cdot R_v)^f \tag{2.4}$$

For Diffuse Reflections

The lighting of diffuse reflections is view independent. The premise of the equation is that the light is equally distributed over the surface. This is only true for a perfect Lambertain surface. However, it is generalized for every diffuse reflection.
The intensity of the reflected ray can be calculated as shown in equation 2.5. p_d is a parameter controlling the diffuse reflection of the surface. I_d^{in} is the incoming diffuse intensity. In Path Tracing the intensity can be the same for specular and diffuse light. θ is the angle on incidence. It is calculated by the dot product of the surface's normal vector N and the vector L which is point to the light's origin.

$$I_d = p_d \cdot I_d^{in} \cdot cos(\theta) = p_d \cdot I_d^{in} \cdot (L \cdot N) \tag{2.5}$$

2.2.4 Ray Tracing of Volumes

So far I have given an introduction to Path Tracing. The focus was on the interaction between the rays and surfaces. In this section I will concentrate on ray tracing of volumes.

Differences to Ray Tracing with Polygons

There are several differences of ray tracing volumes to ray tracing with polygons. A great amount of calculation in ray tracing algorithms such as Path Tracing are denoted to the determination of the intersection points between rays and objects. At each intersection point the ray hits a surface. In a ray tracer for volumes no intersection points need to be calculated. The rays are within the volume. They travel through the volume. So there is no rendering time devoted for intersection calculations. However, it must be determined when a ray hits a surface within a volume. To do so, the ray is sampled at a certain step size. At each sampling point the intensity of the volume at the point is fetched. If intensity is too low, the ray goes on in the same direction. If the intensity is above a certain threshold it marks a surface and the ray hits a surface. Great care needs to be taken in determining the step size.
When a ray hits a surface the angle of incidence needs to be calculated. This is the angle to the surface normal. In volume rendering the normal of a surface can be determined by the gradient as suggested by Levoy [Lev88]. There are several ways to determine the gradient of a volume sample. The implementation in this work uses the central differences method provided by Voreen.

The Volume-Rendering Integral in Volume Path Tracing

When a ray does not hit a surface at a sampling point and keeps its old direction the intensity of the ray is still getting less depending on the intensity of the sampling points. The light the ray transports is absorbed. On the other the hand there is light emission which needs to be simulated. Both effects are described with the absorption / emission integral or volume-rendering integral. The form of the integral can be seen in equation 2.6.

$$I(D) = I_0 \cdot e^{-\int_{s_0}^{D} \kappa(t)\,dt} + \int_{s_0}^{D} q(s) \cdot e^{-\int_{s}^{D} \kappa(t)\,dt} \tag{2.6}$$

Basically, the integral integrates other every emission and its absorption along a line from s_0 to D as seen in figure 2.6.
Since computers usually can only handle discrete numbers the integral needs to be estimated. For the derivation of the estimation see Hadwiger et al. [HKRs*06]. In the end there are two compositing schemes. There is the front-to-back version as seen in equation 2.7 and the back-to-front version as seen in equation 2.8.

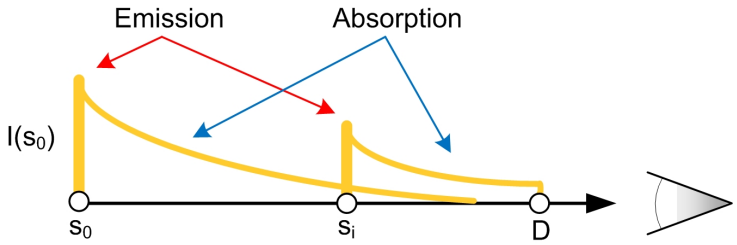


Figure 2.6: Volume-Rendering Integral Illustration (graphic by Maike Dudek)

$$C'_i = C'_{i+1} + T'_{i+1}C_i \qquad T'_i = T'_{i+1}(1-\alpha_i) \tag{2.7}$$

C_i gives the color at position i and C'_i the resulting color of the integral at position i. T'_i is the transparency at point i. α gives the opacity.

$$C'_i = C'_{i-1}(1-\alpha_i) + C_i \qquad T'_i = T'_{i-1}(1-\alpha_i) \tag{2.8}$$

Equation 2.9 describes how the resulting opacity can be determined when using the back-to-front composition.

$$\alpha'_i = \alpha_i + \alpha'_{i-1}(1-\alpha_i) \tag{2.9}$$

The volume-rendering integral is used in volume Path Tracing to determine the intensity of ray travelling through the same material.

2.3 Helper Algorithms

For the development of a Path Tracer several algorithms besides the Path Tracer itself are needed. Points on a unit sphere need to be sampled. For the volume tracing entry points need to be determined. This sections presents two Monte Carlo sampling algorithms for sphere sampling and an algorithm for determining entry points of unit spheres in unit boxes.

2.3.1 Monte Carlo Sampling for Spheres

An important feature of a Path Tracer is Monte Carlo Sampling. In the case of a diffuse reflection the new direction of a ray is determined by sampling a point on a hemisphere over the hit surface. The sampling is distributed uniformly *and* randomly. This sampling can be achieved by Monte Carlo integrations. Monte Carlo integrations are used to estimate integrals by choosing random points and calculating their corresponding integral value. The surface of a hemisphere or sphere can be described by an integral. With the help of a Monte Carlo integration this integral can be estimated.
In this section two Monte Carlo Sampling algorithms are presented. The first is used to sample points on a sphere around the specimen volume for the calculation of the BRDF of the volume. The second variant is used to determine the new direction of a diffusely reflected ray.

First Variant

Dimov et al. [DPS07] describe an algorithm which benefits from the symmetry of spheres. First a sphere is divided into 48 spherical triangles as seen in figure 2.7. Then, points are sampled on one triangle and 47 points for the other triangles are determined by using the symmetry of the sphere.
They present a method to map a unit square to the spherical triangle. The mapping is used to sample a point on one spherical triangle. This approach shifts the problem of distributing samples on a sphere to distributing samples on a unit square. Finding a uniform random distribution for unit square is an fairly easy task. Only two random variables between 0 and 1 need to be determined.
The algorithm including the mapping has the following form. First to random variables u and v are determined. Then the angles φ and θ are calculated as seen in equation 2.10.

$$\varphi = \frac{u\pi}{4} \qquad\qquad \theta = \frac{v}{cos(\frac{u\pi}{4})} \tag{2.10}$$

At last, the sampling points are calculated as described in equation 2.11. The sampling point is then $P = (P_x, P_y, P_z)$. The calculation of the other 47 symmetric sampling points can be seen in appendix B.

$$P_x = cos(\varphi) \cdot sin(\theta) \qquad P_y = sin(\varphi) \cdot sin(\theta) \qquad P_z = cos(\theta) \tag{2.11}$$

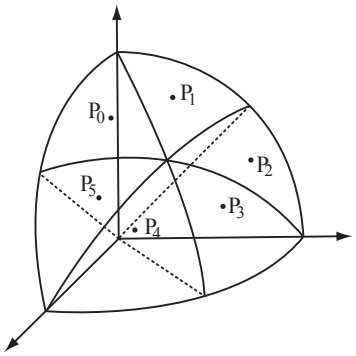


Figure 2.7: Partition of the eighth of a sphere into triangles.

Second Variant

In a Path Tracing algorithm a diffuse reflection is simulated by reflecting one ray in a random direction as described in section 2.2.1. This direction is found by choosing a uniformly distirbuted random point on a sphere or hemisphere. The direction vector points from the center to the point. An algorithm to find such a point is given by Dutre [Dut03, Equation 33]. The point is calculated as seen in equation 2.12. v and u are again random variables uniformly distributed between 0 and 1. The sampling point is $P = (P_x, P_y, P_z)$.

$$\begin{aligned} P_x &= 2 \cdot cos(2\pi \cdot u) \cdot \sqrt{v(1-v)} \\ P_y &= 2 \cdot sin(2\pi \cdot u) \cdot \sqrt{v(1-v)} \\ P_z &= 1 - 2v \end{aligned}$$

2.3.2 Calculating Entry and Exit Points from Unit Sphere Samples

During the off-line BRDF computation from every angle the amount of light has to be measured. The light itself is shot from every angle. Since computers can only calculate

with discrete numbers the view points and light ray origins have to be sampled on a sphere. This is done with a sampling method as described in section 2.3.1. Once the origins of the light rays are sampled on a sphere their entry points into the specimen volume have to be determined. This is done with a simple algorithm described in this section.

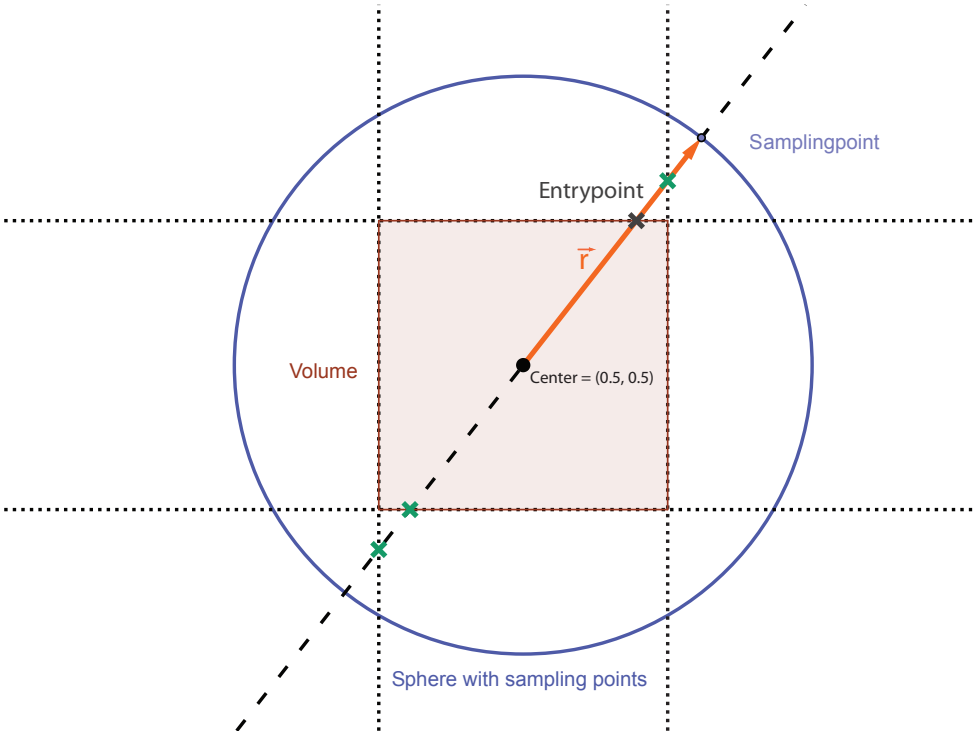


Figure 2.8: Entry point calculation simplified by a projection onto a plane.

The unit sphere has its center at the center of the specimen in volume space. This is $(0.5, 0.5, 0.5)$ as seen in figure 2.8. The entry point of a light ray is the intersection point between a line going through the sampling point and the center (the dashed line in the figure) and the boundaries of the volume (the dotted lines). The idea is to calculate the intersection point with each side of the volume and then decide which point is the entry point. The intersection points are marked with green crosses. The entry point is marked with a black cross.

Equation 2.12 shows how an intersection point between a line through point $(0.5, 0.5, 0.5)$ and a plane is calculated. $\vec{p}$ is a point on the plane. $\vec{u}$ and $\vec{v}$ are vectors on the plane. $\vec{r}$ gives the direction of the line. For the side planes of a volume $\vec{p}$ is a corner of the volume. Note that figure 2.8 shows the case for two dimensions. The planes are represented by the dotted line.

$$\begin{aligned}
\begin{pmatrix}0.5\\0.5\\0.5\end{pmatrix} + k \cdot \vec{r} &= \vec{p} + s \cdot \vec{u} + t \cdot \vec{v} &= \vec{p} + s \cdot \begin{pmatrix}u_x\\u_y\\u_z\end{pmatrix} + t \cdot \begin{pmatrix}v_x\\v_y\\v_z\end{pmatrix} \\
\Rightarrow k &= \frac{p_x + s \cdot u_x + t \cdot v_x - 0.5}{r_x} \\
k &= \frac{p_y + s \cdot u_y + t \cdot v_y - 0.5}{r_y} \\
k &= \frac{p_z + s \cdot u_z + t \cdot v_z - 0.5}{r_z}
\end{aligned} \tag{2.12}$$

To calculate the intersection point between the line and the plane we just have to find k, which is the distance between the center and the entry point. $\vec{u}$ and $\vec{v}$ are always parallel to one axis because the sides of the volume they span are parallel to the axes. Thus, one dimension of $\vec{u}$ and $\vec{v}$ is zero. There are three cases to determine k as shown in equation 2.13. For instance, in the first case $\vec{u}_x$ and $\vec{v}_x$ equal 0. For the cases $r_x = 0$, $r_y = 0$ and $r_z = 0$ there would be a division by zero. These cases have to be tested in an implementation. If $r_x = 0$ the vector is parallel to yz-plane. Thus, it does not cross the sides of the volume which are parallel to the yz-plane and the case can be neglected. The same applies to the cases $r_y = 0$ and $r_z = 0$.

$$k = \frac{p_x - 0.5}{r_x} \qquad k = \frac{p_y - 0.5}{r_y} \qquad k = \frac{p_z - 0.5}{r_z} \tag{2.13}$$

To determine the entry point following steps need to be performed:

1. For each side plane of the volume determine k.
2. Discard every negative k.
3. Choose the smallest k and calculate the intersection point of ray l (the dashed line) and the volume with $P = \begin{pmatrix}0.5\\0.5\\0.5\end{pmatrix} + k \cdot \vec{r}$. P is the intersection point and thus the entry point of the ray into the volume.

Chapter 3

A Path Tracer for BRDF Computation

This chapter is about the implementation of the Path Tracer for off-line BRDF computation. The Path Tracer was implemented in the programming language C++. An implementation for GPUs with e.g. OpenCL might have resulted in better performance. However, debugging proved to be an important factor and was much easier with C++.

3.1 Introduction to Voreen

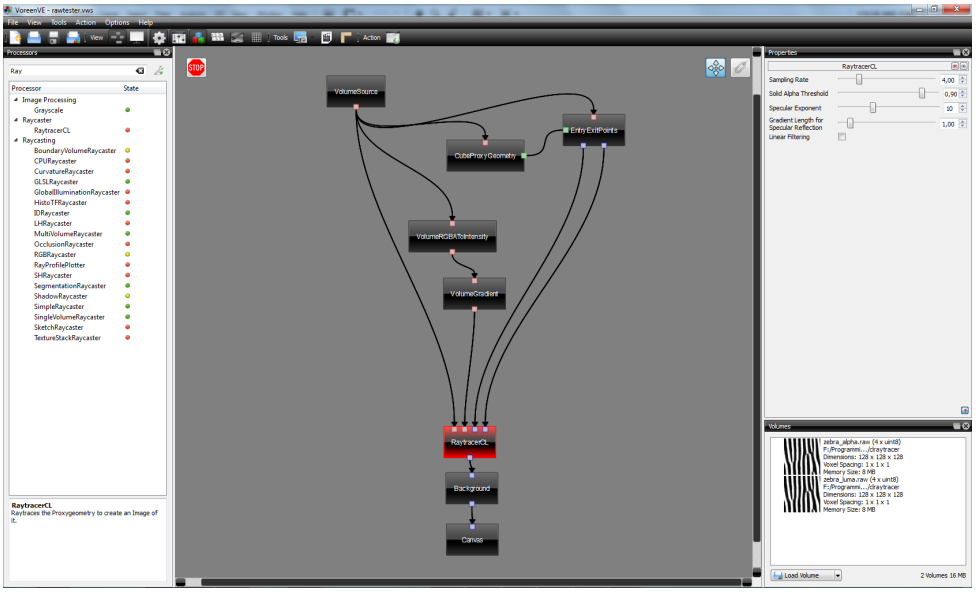


Figure 3.1: Voreen Interface: The nodes of the graph are processors.

Voreen is an acronym for "**vo**lume **re**ndering **en**gine". It is a visualisation framework developed by Ropinski et el. at the WWU in Münster [MSRMH09]. The main focus lies on the visualisation of volumetric data. The framework is used in the application VoreenVe. The structure of Voreen makes it ideal for the implementation of a Path Tracer for volumetric data. Voreen consists of a core which handles basic input and output. The core can be extended with different modules which mainly built up so called processors. Processors can have inputs and outputs. These can be connected. A Processors can e.g. convert an input to a different format. The extensibility is a great advantage of Voreen. The Path Tracer can be easily implemented as a processor module and make use of other processors. For instance, processors provide access to volume data or calculate the gradients for a volume. Thus, there is no need to implement this functionality in the Path Tracer module. The processors can be connected to a graph as shown in figure 3.1.

3.2 The Raytracer Module

This section focuses on the Raytracer module which implements the Path Tracer described in chapter 2. At first a Path Tracer is implemented to render a volume. The rendered image is used to debug the Path Tracer and to find the right settings for the Path Tracer. Once the Path Tracer is finished it is modified to trace the energy of light trough a specimen and compute BRDF data. I call this process "Photon Tracing" which should be not confused with Photon Mapping.

3.2.1 Implementation of the Path Tracer

In this section some parts of the implementation of the Path Tracer are described. The implementation does not consider point light sources yet. An ambient light is assumed. That is a constant light intensity around the volume.
The whole Path Tracer function can be seen in appendixA.

```
1 tgt::vec4 PathTrace(Ray ray, const float stepSize);
```

Listing 3.1: Declaration of the Path Tracer function

Listing 3.1 shows the declaration of the Path Tracing function. *tgt* is the namespace of a graphics utility library called "tiny graphics toolbox" which is develpoed at the University of Münster. The parameter *stepSize* sets the sampling step size. *Ray* is a

struct defining a ray of the Path Tracer. It can be seen in listing 3.2.

```
1 struct Ray {
2   tgt::vec3 next_sample;
3   tgt::vec3 direction;
4   tgt::vec4 old_medium;
5   int recursion_depth;
6 };
```

Listing 3.2: The Ray struct

The implemented Path Tracer is a function which calls itself. It is recursive. To avoid a stack overflow the steps of recursion are counted and saved in the property *recursion_depth*. Once the step count exceeds a certain limit the recursion is stopped. *old_medium* describes the material the ray was in. It is a RGBA color of the last sample. *direction* is the normalized direction of the ray. It is important to note that during the determination of the reflected vector the reversed direction has to be used. *next_sample* is the sample of the volume the ray hits.
At first, the color of the current sample and the gradient for the sample are retrieved. Then three cases for the interaction between the ray and the volume are checked. The ray can be in a transparent part of the volume. This is checked with the alpha value of the current sample. Or the ray hits a solid surface and is reflected. For the reflection there is a diffuse and a specular case.

```
1 // Use direct volume rendering integral if ray is in same medium, gradient
      is not defined or alpha is too low
2 if( tgt::length( color - ray.old_medium ) < 0.001f || tgt::length(gradient)
      < 0.001f || color.a < solid_alpha_threshold_.get() ){
3   ...
4 } else {
5   // Ray hits a different medium. Decide whether we have specular or
        diffuse reflection
6
7   // DIFFUSE case
8   if( tgt::length(gradient) < gradient_specular_threshold_.get() ){
9       ...
10  }
11  // SPECULAR case
12  else {
13      ...
```

```
14    }
15 }
```

Listing 3.3: Case Testing for Ray Material Interactions

The Volume-Rendering Integral

If the ray is in a transparent medium the color is calculated using the volume-rendering integral as seen in listing 3.4.

```
1  // The ray is still in the SAME MEDIUM.
2
3  Ray integral_ray = ray;
4  while( tgt::length( color - integral_ray.old_medium ) < 0.001f || color.a <
       solid_alpha_threshold_.get() )
5  {
6    result.r = result.r + (1.0f - result.a) * color.a * color.r;
7    result.g = result.g + (1.0f - result.a) * color.a * color.g;
8    result.b = result.b + (1.0f - result.a) * color.a * color.b;
9    result.a = result.a + (1.0f - result.a) * color.a;
10
11   // Step forward in volume
12   integral_ray.next_sample += stepSize * tgt::normalize(integral_ray.
         direction);
13
14   // Check if next sample is outside of volume
15   if( integral_ray.next_sample.x > 1.0f || integral_ray.next_sample.y > 1.0
         f || integral_ray.next_sample.z > 1.0f || integral_ray.next_sample.x <
          0.0f || integral_ray.next_sample.y < 0.0f || integral_ray.next_sample
         .z < 0.0f  )
16     return result;
17
18   // Set old medium
19   integral_ray.old_medium = color;
20
21   // Get new sample
22   color = this->getVoxel(integral_ray.next_sample);
23 }
24
25 // Raise recursion depth
26 integral_ray.recursion_depth = ray.recursion_depth + 1;
27
```

```
28 // Get next color
29 tgt::vec4 pre_color = pathTrace(integral_ray, stepSize);
30
31 //Calcualte ray integral (front to back)
32 result.r = result.r + (1.0f - result.a) * pre_color.a * pre_color.r;
33 result.g = result.g + (1.0f - result.a) * pre_color.a * pre_color.g;
34 result.b = result.b + (1.0f - result.a) * pre_color.a * pre_color.b;
35 result.a = result.a + (1.0f - result.a) * pre_color.a;
```

Listing 3.4: Implementation of the Volume-Rendering Integral

The resulting color for the ray is given by the variable *result*. It is calculated using the front-to-back compositing scheme as described in section 2.6. The recursion depth is raised *after* the volume-rendering integral is determined. This is a small performance enhancement, because there is no recursion which could lead to a stack overflow. As long as light is in the same uniform material its direction is not changed. Thus the direction of the ray should not be changed and the ray does not hit a surface. That means no reflection has to be simulated by the *PathTrace* function. Using the volume-rendering integral is an efficient way to simulated this behaviour of the light. Once the ray hits a different material the *PathTrace* function calls itself to calculate the next step as seen in line 29. Within the new function it is checked if the ray hit a new surface or just another transparent material.

Diffuse Reflection Implementation

```
1  tgt::vec3 normal = normalized_grad;
2
3  // Create new ray
4  Ray new_ray;
5  new_ray.recursion_depth = ray.recursion_depth + 1;
6  new_ray.old_medium = color;
7  new_ray.direction = generateRandomDiffuseRay();
8  new_ray.next_sample = ray.next_sample + step_size * new_ray.direction;
9
10 float costheta = std::max(0.0f, tgt::dot(normal, new_ray.direction));
11 tgt::vec4 reflected_color = costheta * PathTrace(new_ray, step_size, log);
12
13 // Add reflected colors
14 result = color * reflected_color;
```

Listing 3.5: Calculation of the Diffuse Reflection Vector

Listing 3.5 is the code sample for diffuse reflection. The new direction of the ray is determined by *generateRandomDiffuseRay()*. The function is an implementation of the second sampling method described in section 2.3.1. The color compositing done in lines 10 till 14 is basically an implementation of the Phong Model explained in section 2.2.3. The variable *reflected_color* is the incoming light I_d^{in} multiplied with $cos(\theta)$. θ is the angle of incidence.

Specular Reflection Implementation

```
1  Ray newRay;
2  tgt::vec3 reversed_ray_direction = - ray.direction;
3  newRay.direction = tgt::normalize( 2 * tgt::dot( normalizedGrad,
       reversed_ray_direction ) * normalizedGrad - reversed_ray_direction );
4  newRay.next_sample = ray.next_sample + stepSize * newRay.direction;
5  newRay.recursion_depth = ray.recursion_depth + 1;
6  newRay.old_medium = ray.old_medium; // since the ray is reflected into the
       old medium, it stays the same
7
8  tgt::vec4 reflected_color = pathTrace(newRay, stepSize, log);
9
10 // Combine colors using Phong
11 // Use Phong here cos(theta) where theta is the angle between V (Viewvector
       , incoming Ray) and R (reflected Ray).
12 int e = specular_exponent_.get();
13 float factor = std::max( 0.0f, tgt::dot( newRay.direction,
       reversed_ray_direction ));
14 factor = pow(factor, e);
15 result = color * factor * reflected_color;
```

Listing 3.6: Specular Reflection Case Handling

The specular reflection is handled as seen in listing 3.6. The direction of the reflected ray is determined in line 3. It is an implementation of the equation 2.3. Line 14 to 20 are the implementation of the Phong Model for specular reflections as described in equation 2.4. In line 8 the *PathTracer* calls itself to retrieve the incoming light at the the hit point of the ray. The variable *reflected_color* is incoming light describe by I_s^{in} in the Phong Model for specular reflections. *newRay.direction* points towards the light source which is in this case the reversed direction of the incoming light.

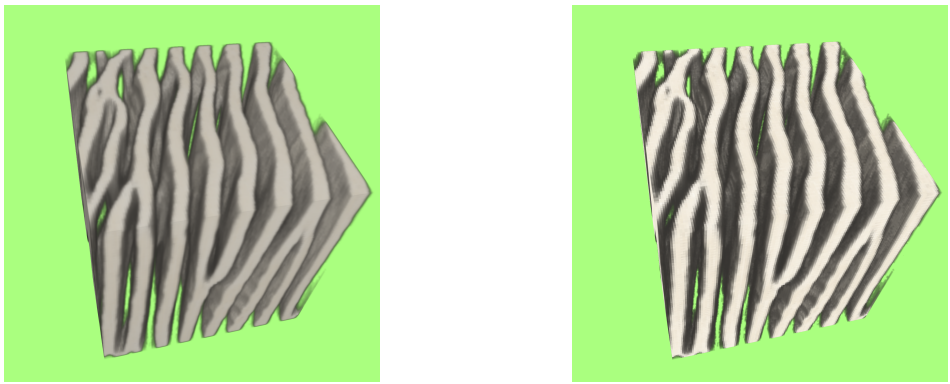
3.2.2 Functions and Properties

The ray tracer module has two functions and several properties. It can render a volume using the Path Tracer algorithm. This function should be used to debug the tracer. That means the rendered image should be used to see if the values of properties of the module are set to result in an acceptable image. If so the module can then be used to compute the BRDF data for a given specimen. This is the second function of the Raytracer module.
A floating point property called "Sampling Rate" is used to control the sampling rate of the Path Tracer. Another floating point property controls what alpha or intensity values are considered solid. That means at which value a voxel reflects and is not transparent. This property is called "Solid Alpha Threshold". There is one integer property which sets the specular exponent for the Phong model. It is called "Specular Exponent". One boolean property controls if voxels should be fetched using linear filtering. Linear filtering proved to be very performance consuming since the filtering is not done on a graphics card. The last property is the "Gradient Length for Specular Reflection" it sets at what gradient length a voxel is specular reflecting and not diffusely.
All properties should be set when rendering with the Path Tracer because their influence can be easily observed on an rendered image. Once the properties are final they can be used for the BRDF computation.

3.2.3 From Path Tracing to Photon Tracing

The path tracer described so far just renders an image of the given volume. The perspective is determined by the entry points of the rays and their direction. To render an image similar to classic ray-casting algorithms used in direct volume rendering the algorithm described by Krüger and Westerman [KW03] can be used to generate the fitting ray entry points and directions. This approach was used to render a debug image. The images in figure 3.2 show a comparison between the GLSL based raycasting renderer of Voreen (a) and a rendering by the implemented Path Tracer (b).

To compute BRDF data for a volume specimen the implemented path trace needs to be changed to calculate the amount of light a ray looses when it exits a specimen after it travelled through it. I call this Photon Tracing. The photons are traced through a specimen. This should not be confused with Photon Mapping which is a different algorithm for different purposes developed by Jensen (for an introduction see [Jen01]).
Several things of the Path Tracer need to be changed. The structure of a ray needs to



(a) (b)

Figure 3.2: Rendering results comparison: **(a)** *GLSL Raycasting*, **(b)** *Path Tracing*

carry a load which describes how much photons were lost. When the ray interacts with a material the lost amount of photons needs to be determined.

```
1 float PhotonTrace(tgt::vec3 lightSamplingPoint, tgt::vec3 viewSamplingPoint
    , const float step_size);
```

Listing 3.7: Declaration of the Photon Tracer function

The *PhotonTrace* function is declared as seen in listing 3.7. The main difference to the *PathTrace* function is that it returns just one float value. That value denotes how much light sent from the *lightSamplingPoint* into to the volume reaches the *viewSamplingPoint*. The whole function can be seen in appendix C.

```
1 struct EnergyRay {
2   tgt::vec3 next_sample;
3   tgt::vec3 direction;
4   tgt::vec4 old_medium;
5   int recursion_depth;
6   float load;
7 };
```

Listing 3.8: Implemented Structure for a Ray of the Photon Tracer

The implemented structure can be seen in listing 3.8. The only new parameter is *load*. It describes what amount of photons the ray carries relatively to a full load. The maximum is 1.0 and the minimum is 0.0. For instance, when 30% of the photons are absorbed at a reflection the load will be 0.7.

Volume Rendering Integral

The Photon Tracer has a modified volume rendering integral implemented. As long as a ray is travelling through the same medium it loses photons. That means the load decreases. This effect is simulated in the code seen in listing 3.9.

```
1 integral_ray.load = integral_ray.load * (1.0f - color.a);
```

Listing 3.9: Calculation of the Load Decrease of a Ray

The equation is similar to the calculation of the transparency in the front-to-back calculation of the volume rendering integral presented in section 2.6. Basically, the new load is the portion of the old load that gets through the sample. The portion is determined by the transparency of the current sample which is given by *(1.0f - color.a)*;

Implementation for Diffuse and Specular Reflection

If a reflection is not ideal, not all photons are reflected in the same direction. That means that some energy is lost. This can be described with the Phong Model. The same calculations described in section 2.2.3 can be used to determine the lost for the photon load of the ray.
The code for specular reflection can be seen in listing 3.10.

```
1   int e = specular_exponent_.get();
2     float factor = std::max( 0.0f, tgt::dot( new_ray.direction,
          reversed_ray_direction ));
3     factor = pow(factor, e);
4
5     new_ray.load *= factor;
```

Listing 3.10: Load Loss at a Specular Reflection

The factor determines how much load is not lost. It is determined by the cosine of the angle between the reflection direction and the light source. The light source is the direction the ray is coming from in this case. The cosine is determined by the dot product.
For diffuse reflection the new load is determined as seen in the code sample of listing 3.11.

```
1  float costheta = std::max(0.0f, tgt::dot(normal, new_ray.direction));
2  new_ray.load = costheta * ray.load;
```

Listing 3.11: Load Loss at a Diffuse Reflection

The variable *costheta* describes how much load is not lost. It is the cosine of the angle between the surface normal and the direction of the reflection.
The rest of the implementation is similar. The recursive structure of the PathTrace function was altered to an iterative form. This avoids the problem of a stack overflow and allows a parallel calculation of the rays.

3.2.4 BRDF Data Representation

Once the BRDF is computed, the data has to be stored somehow to be used later during rendering. This is a difficult task since there is a lot of data. For every view point on a sphere the incoming light from all points on a sphere have to be determined. Depending on the representation of the data and the ability to interpolate it might be sufficient or not.
The focus of this thesis is not to find the perfect representation for BRDF data. However, computing BRDFs is useless if they cannot be stored. Thus a representation is proposed here to show that the storage is at least possible.
As mentioned before, for every view light is shot at the specimen from many samples on a sphere. These samples can be mapped to a spherical environment map. A pixel on such a map would then store how much photons reach the view when shot from a certain point on the sphere. The texture coordinates for a sampling point are determined as seen in equation 3.1.

$$\begin{pmatrix} s \\ t \end{pmatrix} = \begin{pmatrix} \dfrac{P_x}{2 \cdot \sqrt{P_x^2 + P_y^2 + (P_z + 1)^2}} + \dfrac{1}{2} \\ \dfrac{P_y}{2 \cdot \sqrt{P_x^2 + P_y^2 + (P_z + 1)^2}} + \dfrac{1}{2} \end{pmatrix} \tag{3.1}$$

$P = (P_x, P_y, P_z)$ is the light sampling point on a sphere. A texture created for the view point $(1, 0, 0)$ can be seen in figure 3.3. To determine the BRDF for a certain voxel during rendering the right texture for the view needs to be loaded. The texture coordinates are determined for the light hitting the voxel. The pixels corresponding to the texture coordinates are fetched. Every channel of RGB contains the BRDF previously computed.

So just one channel is used to determine the lighting of the voxel.
This BRDF representation has a big flaw. Besides the known distortions of spherical environment maps there has to be one texture per sampled view. This leads to a great amount of textures and thus data which needs to be handled. In addition there is no hardware supported interpolation.

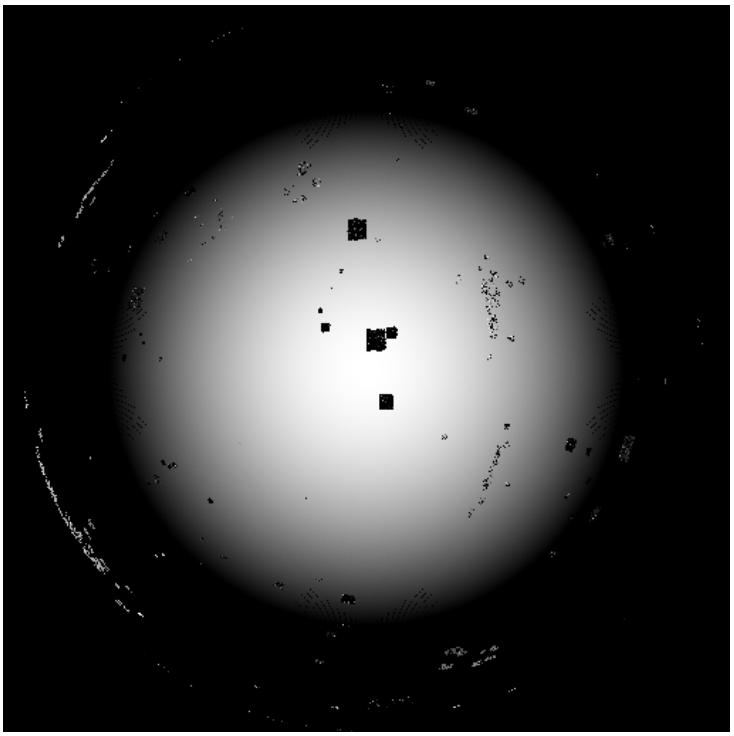


Figure 3.3: BRDF representation for one view. The artefacts show that light form those angles is not reaching the view point.

Chapter 4

Evaluation and Conclusion

4.1 Implementation Problems and Missing Features

This section is about the problems which came up during the implementation. Missing features are also listed.
The implementation of the Path Trace is recursive so far. A recursive version was faster to implement and since the Path Tracer was just one step in between a quick implementation was fine. However, a recursive implementation has several drawbacks. First, the Path Tracer quickly came to a stack overflow. That means the computer ran out of memory. Depending on the settings the Path Tracer could not take more than 500 steps. This problem was avoided in the Photon Tracer by choosing an iterative implementation as seen in appendix C. Another advantage of an iterative implementation is that the tracer can be accelerated with parallel threads. Those threads could also run on a modern GPU.
Another challenge was to create a specimen suitable for rendering and computing of the BRDFs. The specimen has to have a fairly high resolution and has to be very detailed to ensure a high quality BRDF output. A 3D texture generated from a 2D texture seemed to be an appropriate choice to be used as a volume specimen because it eased the creation of the specimen while still ensuring high quality. In the end a 3D texture created by Kopf et al. was modified and used. Kopf et al. [KFCO*07] present a technique to synthesise 3D textures from 2D exemplar.
A missing feature so far is refraction. The equation to determine how a light beam bends at the interface of two translucent materials is well known. However, a special index has to be specified for every material. That means another parameter has to be set in every voxel of the volume. This parameter could be saved in an additional channel.

4.2 Conclusion

Westin et al. [WAT92] proved that BRDFs can be predicted based on off-line computations. They also provided implementations for surfaces. The goal of this work was to adapt the same technique for volumes with the focus on developing a Path Tracer for volumes and modifying it to generate BRDF representations. Even though adequate specimens are missing and the output format is not practical yet the goal was reached. The Path Tracer and its modification render suitable outputs. Suitable specimens can be created manually or with texture synthesis algorithms such as the one developed by Kopf et al. [KFCO*07]. A usable representation aggregated BRDF data was developed and applied by Lindemann and Ropinski [LR10].

Chapter 5

Future Work

Several fields for future work should be considered.
So far, the focus of the implementation of the Photon Tracer was not on performance. This does not to seem to be a drawback because the BRDF data is computed off-line. However, a higher degree of detail can only be achieved with more samples. If the amount of samples is too high, even an off-line computation can take too long. Thus, the performance of the Photon Tracer should be improved. With the recent upcoming of multi-core CPUs a parallel working Photon Tracer should definitely be considered to improve the performance.
In a Path Tracer thousands of rays are send for each pixel to avoid noise. This technique is called supersampling. Noise is not a problem for the Photon Tracer because the samples on the sphere do not represent pixels and a lot samples can be create without supersampling. However, when the rays exit the volume again their photon load has to be saved. How much load is saved depends on the view point and the direction the rays exit the volume. So far the cosine is used to estimate what portion of the load is saved. If this approach is suitable should be subject to further research.
Many light effects were not considered in the implementation. For instance, the effect of refraction is not simulated. Refractions is an important effect for achieving realistic renderings of translucent objects. Volume rendering is used a lot for medical visualization. Organic materials such as fat and muscles have translucent parts. To render these realistically refraction becomes an important feature.
As mentioned before, the quality of the computed BRDF depends on the quality of the specimen. So far, the specimen was an abstract volume created from a 3D texture which in return was generated from a 2D exemplar. This approach promises to ease the creation of high quality specimens. Thus, it is another important aspect of future research.

At last, the computed BRDF data has to be represented in a way that ensures interpolation and fast retrieval during real-time rendering. In section 3.2.4 a representation was proposed. However, it does not yet meet the requirements demanded for real-time rendering. Thus it should be a subject for future research.

Bibliography

[AMHH08] Akenine-Möller T., Haines E., Hoffman N.: Real-Time Rendering 3rd Edition. A. K. Peters, Ltd., Natick, MA, USA, 2008.

[Com05] Comninos P.: Mathematical and Computer Programming Techniques for Computer Graphics. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[DPS07] Dimov I., Penzov A., Stoilova S.: Parallel Monte Carlo sampling scheme for sphere and hemisphere. Numerical Methods and Applications (2007), 148–155.

[Dut03] Dutre P.: Global illumination compendium. the concise guide to global illumination algorithms. `http://www.cs.kuleuven.ac.be/~phil/GI/`, August 2003.

[HKRs*06] Hadwiger M., Kniss J. M., Rezk-salama C., Weiskopf D., Engel K.: Real-time Volume Graphics. A. K. Peters, Ltd., Natick, MA, USA, 2006.

[Jen01] Jensen H. W.: Realistic image synthesis using photon mapping. A. K. Peters, Ltd., Natick, MA, USA, 2001.

[Kaj86] Kajiya J. T.: The rendering equation. In SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques (New York, NY, USA, 1986), ACM, pp. 143–150.

[KFCO*07] Kopf J., Fu C.-W., Cohen-Or D., Deussen O., Lischinski D., Wong T.-T.: Solid texture synthesis from 2d exemplars. ACM Transactions on Graphics (Proceedings of SIGGRAPH 2007) 26, 3 (2007), 2:1–2:9.

[KW03] Krüger J., Westermann R.: Acceleration techniques for gpu-based volume rendering. In VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03) (Washington, DC, USA, 2003), IEEE Computer Society, p. 38.

[Lev88] Levoy M.: Display of surfaces from volume data. IEEE Computer graphics and Applications 8, 3 (1988), 29–37.

[LR10] Lindemann F., Ropinski T.: Advanced light material interaction for direct volume rendering. In IEEE/EG International Symposium on Volume Graphics (2010), Westermann R., Kindlmann G., (Eds.), Eurographics Association, pp. 101–108.

[MSRMH09] Meyer-Spradow J., Ropinski T., Mensmann J., Hinrichs K. H.: Voreen: A rapid-prototyping environment for ray-casting-based volume visualizations. IEEE Computer Graphics and Applications (Applications Department) 29, 6 (Nov./Dec. 2009), 6–13.

[NDM05] Ngan A., Durand F., Matusik W.: Experimental analysis of BRDF models. In Proceedings of the Eurographics Symposium on Rendering (2005), pp. 117–226.

[Pho75] Phong B. T.: Illumination for computer generated pictures. Commun. ACM 18, 6 (1975), 311–317.

[WAT92] Westin S., Arvo J., Torrance K.: Predicting reflectance functions from complex surfaces. ACM Siggraph Computer Graphics 26, 2 (1992), 255–264.

[Wik10] Wikipedia: Refraction — wikipedia, the free encyclopedia, 2010. [Online; accessed 9-August-2010].

[WLT04] Westin S., Li H., Torrance K.: A comparison of four brdf models. Research Note PCG-04-02, Cornell University Program of Computer Graphics (2004).

Appendix A

Path Tracer Source Code

The code in listing A.1 is called to render the volume. The *entry* and *exit* points are generated using the Krüger and Westermann algorithm.
For each view pixel the *PathTrace* function is called. Note that no supersampling is done. The definition for the *PathTrace* function can be seen in listing A.2.

```
1  void RaytracerCL::RenderImage()
2  {
3      LGL_ERROR;
4
5      //output view size
6      int width = outport_.getSize().x;
7      int height = outport_.getSize().y;
8
9      // buffer for output image
10     float *pixels = new float[width*height*4];
11
12     // download eep textures and set buffers
13     float* entry = (float*)entryPort_.getColorTexture()->
           downloadTextureToBuffer(GL_RGBA, GL_FLOAT);
14     float* exit = (float*)exitPort_.getColorTexture()->
           downloadTextureToBuffer(GL_RGBA, GL_FLOAT);
15
16     //assume bpv is 4
17     this->volume_data_ = dynamic_cast<Volume4xUInt8*>(volumePort_.getData()
           ->getVolume());
18     tgt::ivec3 volumeDimensions = volume_data_->getDimensions();
19
```

```
20      // get gradient volume
21      this->volume_gradient_data_ = volumeGradientPort_.getData()->getVolume
            ();
22
23      float samplingstep_size = 1.f / (tgt::min(volumeDimensions) *
            sampling_rate_.get());
24
25      LINFO("Start Path Tracing...");
26
27      for( int w = 0; w< width; w++) {
28          for( int h = 0; h<height; h++) {
29
30              std::cout << ".";
31
32              int index = 4 * (h*width + w);
33
34              tgt::vec3 frontPos = tgt::vec3(&entry[index]);
35              tgt::vec3 backPos = tgt::vec3(&exit[index]);
36              if( frontPos != backPos ){
37                  Ray ray;
38                  ray.recursion_depth = 0;
39                  ray.direction = tgt::normalize(backPos-frontPos);
40                  ray.next_sample = frontPos;
41                  ray.old_medium = tgt::vec4(0.0f);
42                  tgt::vec4 color = PathTrace(ray, samplingstep_size);
43                  pixels[index] = color.r;
44                  pixels[index+1] = color.g;
45                  pixels[index+2] = color.b;
46                  pixels[index+3] = color.a;
47              }
48          }
49      }
50
51      LINFO("Path Tracing done.");
52
53      // copy pixels to reder target
54      outport_.activateTarget();
55
56      glClearColor(0.0, 0.0, 0.0, 0.0);
57      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
58
59      glWindowPos2i(0,0);
```

```
60      glDrawPixels(outport_.getSize().x, outport_.getSize().y, GL_RGBA,
            GL_FLOAT, pixels);
61
62      outport_.deactivateTarget();
63      LGL_ERROR;
64
65      delete[] pixels;
66      entryPort_.getColorTexture()->destroy();
67      exitPort_.getColorTexture()->destroy();
68  }
```

Listing A.1: RenderImage Function Definition

```
1   tgt::vec4 RaytracerCL::PathTrace(Ray ray, const float step_size)
2   {
3       // result color
4       tgt::vec4 result(0.0f);
5
6       if( ray.recursion_depth == this->MAX_DEPTH )
7           return result;
8
9       // check if next_sample is outside of volume or not defined
10      if( ray.next_sample.x > 1.0f || ray.next_sample.y > 1.0f || ray.
            next_sample.z > 1.0f || ray.next_sample.x < 0.0f || ray.next_sample.
            y < 0.0f || ray.next_sample.z < 0.0f  )
11          return result;
12
13      // calculate sampling point
14      tgt::vec3 sample = ray.next_sample;
15
16      // get color for sample
17      tgt::vec4 color = this->getVoxel(sample, use_linear_filtering_.get());
18
19      // calculate gradient of sample
20      tgt::vec3 gradient = getGradient(sample, use_linear_filtering_.get());
21      tgt::vec3 normalized_grad = tgt::normalize(gradient);
22
23      // use oldfashioned absorption / emission integral if ray is in same
            medium, gradient is not defined or if alpha is too low
24      if( tgt::length( color - ray.old_medium ) < 0.001f || tgt::length(
            gradient) < 0.001f || color.a < solid_alpha_threshold_.get() )
25      {
26          // the ray is still in the SAME MEDIUM.
```

```
27
28          Ray integral_ray = ray;
29          while( tgt::length( color - integral_ray.old_medium ) < 0.001f ||
                color.a < solid_alpha_threshold_.get() )
30          {
31              result.r = result.r + (1.0f - result.a) * color.a * color.r;
32              result.g = result.g + (1.0f - result.a) * color.a * color.g;
33              result.b = result.b + (1.0f - result.a) * color.a * color.b;
34              result.a = result.a + (1.0f - result.a) * color.a;
35
36              // set forward in volume
37              integral_ray.next_sample += step_size * tgt::normalize(
                    integral_ray.direction);
38
39              // check if next sample is outside of volume
40              if( integral_ray.next_sample.x > 1.0f || integral_ray.
                    next_sample.y > 1.0f || integral_ray.next_sample.z > 1.0f ||
                     integral_ray.next_sample.x < 0.0f || integral_ray.
                    next_sample.y < 0.0f || integral_ray.next_sample.z < 0.0f  )
41                  return result;
42
43              // set old medium
44              integral_ray.old_medium = color;
45
46              // get new sample
47              color = this->getVoxel(integral_ray.next_sample);
48          }
49
50          // raise recursion depth
51          integral_ray.recursion_depth = ray.recursion_depth + 1;
52
53          // get next color
54          tgt::vec4 pre_color = PathTrace(integral_ray, step_size);
55
56          //calcualte ray integral (front to back)
57          result.r = result.r + (1.0f - result.a) * pre_color.a * pre_color.r
                ;
58          result.g = result.g + (1.0f - result.a) * pre_color.a * pre_color.g
                ;
59          result.b = result.b + (1.0f - result.a) * pre_color.a * pre_color.b
                ;
60          result.a = result.a + (1.0f - result.a) * pre_color.a;
61      } else {
```

```
62
63          // ray hit a different medium. Decide whether we have specular
                reflection, refraction or diffuse reflection
64
65          // REFRACTION is omitted. There needs to be a way to retrieve
                refraction indices
66
67          // REFLECTION case
68
69          // DIFFUSE case
70          // spawn several new rays in Energy Path Tracing. Each ray has the
                Energy cos(theta) where theta is the angle between incoming Ray
                and Normal (gradient)
71          // the new rays should be equally distributed for now
72          if( tgt::length(gradient) < gradient_specular_threshold_.get() )
73          {
74              tgt::vec3 normal = normalized_grad;
75
76              // create new ray
77              Ray new_ray;
78              new_ray.recursion_depth = ray.recursion_depth + 1;
79              new_ray.old_medium = color;
80              new_ray.direction = generateRandomDiffuseRay();
81              new_ray.next_sample = ray.next_sample + step_size * new_ray.
                    direction;
82
83              float costheta = std::max(0.0f, tgt::dot(normal, new_ray.
                    direction));
84              tgt::vec4 reflected_color = costheta * PathTrace(new_ray,
                    step_size);
85
86              // add reflected colors
87              result = color * reflected_color;
88          }
89          // SPECULAR case
90          else
91          {
92              Ray new_ray;
93              tgt::vec3 reversed_ray_direction = - ray.direction;
94              new_ray.direction = tgt::normalize( 2 * tgt::dot(
                    normalized_grad, reversed_ray_direction ) * normalized_grad
                    - reversed_ray_direction );
```

```
95              new_ray.next_sample = ray.next_sample + step_size * new_ray.
                    direction;
96              new_ray.recursion_depth = ray.recursion_depth + 1;
97              new_ray.old_medium = ray.old_medium;
98
99              tgt::vec4 reflected_color = PathTrace(new_ray, step_size);
100
101             // combine colors using phong
102             // use PHONG here cos(theta) where theta is the angle between V
                    (Viewvector, incoming Ray) and R (reflected Ray).
103             int e = specular_exponent_.get();
104             float factor = std::max( 0.0f, tgt::dot( new_ray.direction,
                    reversed_ray_direction ));
105             factor = pow(factor, e);
106             result = color * factor * reflected_color;
107         }
108     }
109
110     return result;
```

Listing A.2: Definition of the PathTrace Function

Appendix B

Calculation of Sampling Points on Sphere

Table B.1 shows how the sampling points for a hemisphere are calculated if the sampling point $P = (x_0, y_0, z_0)$ for the spherical triangle is known. Table B.2 shows how the sampling point for a sphere are calculated from spherical sampling points. Note that these sampling points can be calculated parallel.

$P_0(x_0, y_0, z_0)$	$P'_0(-x_0, y_0, z_0)$	$P''_0(-x_0, -y_0, z_0)$	$P'''_0(x_0, -y_0, z_0)$
$P_1(y_0, x_0, z_0)$	$P'_1(-y_0, x_0, z_0)$	$P''_1(-y_0, -x_0, z_0)$	$P'''_1(y_0, -x_0, z_0)$
$P_2(z_0, y_0, x_0)$	$P'_2(-z_0, y_0, x_0)$	$P''_2(-z_0, -y_0, x_0)$	$P'''_2(z_0, -y_0, x_0)$
$P_3(z_0, x_0, y_0)$	$P'_3(-z_0, x_0, y_0)$	$P''_3(-z_0, -x_0, y_0)$	$P'''_3(z_0, -x_0, y_0)$
$P_4(x_0, z_0, y_0)$	$P'_4(-x_0, z_0, y_0)$	$P''_4(-x_0, -z_0, y_0)$	$P'''_4(x_0, -z_0, y_0)$
$P_5(y_0, z_0, x_0)$	$P'_5(-y_0, z_0, x_0)$	$P''_5(-y_0, -z_0, x_0)$	$P'''_5(y_0, -z_0, x_0)$

Table B.1: Sample Coordinates of Hemisphere

$P_0(x_0, y_0, z_0)$	$P'_0(-x_0, y_0, z_0)$	$P''_0(-x_0, -y_0, z_0)$	$P'''_0(x_0, -y_0, z_0)$
$P_1(y_0, x_0, z_0)$	$P'_1(-y_0, x_0, z_0)$	$P''_1(-y_0, -x_0, z_0)$	$P'''_1(y_0, -x_0, z_0)$
$P_2(z_0, y_0, x_0)$	$P'_2(-z_0, y_0, x_0)$	$P''_2(-z_0, -y_0, x_0)$	$P'''_2(z_0, -y_0, x_0)$
$P_3(z_0, x_0, y_0)$	$P'_3(-z_0, x_0, y_0)$	$P''_3(-z_0, -x_0, y_0)$	$P'''_3(z_0, -x_0, y_0)$
$P_4(x_0, z_0, y_0)$	$P'_4(-x_0, z_0, y_0)$	$P''_4(-x_0, -z_0, y_0)$	$P'''_4(x_0, -z_0, y_0)$
$P_5(y_0, z_0, x_0)$	$P'_5(-y_0, z_0, x_0)$	$P''_5(-y_0, -z_0, x_0)$	$P'''_5(y_0, -z_0, x_0)$
$\bar{P}_0(x_0, y_0, -z_0)$	$\bar{P}'_0(-x_0, y_0, -z_0)$	$\bar{P}''_0(-x_0, -y_0, -z_0)$	$\bar{P}'''_0(x_0, -y_0, -z_0)$
$\bar{P}_1(y_0, x_0, -z_0)$	$\bar{P}'_1(-y_0, x_0, -z_0)$	$\bar{P}''_1(-y_0, -x_0, -z_0)$	$\bar{P}'''_1(y_0, -x_0, -z_0)$
$\bar{P}_2(z_0, y_0, -x_0)$	$\bar{P}'_2(-z_0, y_0, -x_0)$	$\bar{P}''_2(-z_0, -y_0, -x_0)$	$\bar{P}'''_2(z_0, -y_0, -x_0)$
$\bar{P}_3(z_0, x_0, -y_0)$	$\bar{P}'_3(-z_0, x_0, -y_0)$	$\bar{P}''_3(-z_0, -x_0, -y_0)$	$\bar{P}'''_3(z_0, -x_0, -y_0)$
$\bar{P}_4(x_0, z_0, -y_0)$	$\bar{P}'_4(-x_0, z_0, -y_0)$	$\bar{P}''_4(-x_0, -z_0, -y_0)$	$\bar{P}'''_4(x_0, -z_0, -y_0)$
$\bar{P}_5(y_0, z_0, -x_0)$	$\bar{P}'_5(-y_0, z_0, -x_0)$	$\bar{P}''_5(-y_0, -z_0, -x_0)$	$\bar{P}'''_5(y_0, -z_0, -x_0)$

Table B.2: Sample Coordinates of Sphere

Appendix C

Photon Tracer Source Code

The function *ComputeBRDF* is called to compute the BRDF data for a specimen. The code is seen in listing C.1. First, the sphere sampling points are generated. After that the PhotonTrace function seen in listing C.2 is called for only one view. Note that the PhotonTrace function has two steps to separate certain tasks. In *PhotonTrace* the energy ray is set up. *PhotonTraceIterative* seen in listing C.3 is an iterative implementation of the Photon Tracer. Note that *PhotonTraceIterative* can return more than one result. This can happen if more than one new ray is spawned after a diffuse reflection.

```
1  void RaytracerCL::ComputeBRDF()
2  {
3      LGL_ERROR;
4
5      //output view size
6      int width = outport_.getSize().x;
7      int height = outport_.getSize().y;
8      LGL_ERROR;
9
10     // buffer for output image
11     float *pixels = new float[width*height*4];
12
13     //assume bpv is 4
14     this->volume_data_ = dynamic_cast<Volume4xUInt8*>(volumePort_.getData()
           ->getVolume());
15     tgt::ivec3 volumeDimensions = volume_data_->getDimensions();
16     LGL_ERROR;
17
18     // get gradient volume
```

```
19      this->volume_gradient_data_ = volumeGradientPort_.getData()->getVolume
            ();
20      LGL_ERROR;
21
22      float samplingstep_size = 1.f / (tgt::min(volumeDimensions) *
            sampling_rate_.get());
23      LGL_ERROR;
24
25      int MAX_SAMPLES = 256; //* 48 since sphere is divided into 48 parts
26      std::vector<tgt::vec3> sphereSamplingPoints = generateSphereSamples(
            MAX_SAMPLES);
27
28      LINFO("Compute BRDF with " << sphereSamplingPoints.size() << " samples.
            ");
29
30      // clear image
31      for(int i = 0; i<(width*height); i++)
32      {
33          tgt::vec4 color = tgt::vec4(0.0f);
34
35          pixels[i*4] = color.r;
36          pixels[i*4+1] = color.g;
37          pixels[i*4+2] = color.b;
38          pixels[i*4+3] = color.a;
39      }
40
41      // path trace for each sphere smapling point
42      for( size_t i = 0; i < sphereSamplingPoints.size(); i++ )
43      {
44          float e = PhotonTrace( sphereSamplingPoints[i],
                sphereSamplingPoints[0], samplingstep_size);
45
46          // texture coordinates.
47          float s = 0.5f + sphereSamplingPoints[i].x / (2 * std::sqrt( pow(
                sphereSamplingPoints[i].x, 2.0f) + pow(sphereSamplingPoints[i].y
                , 2.0f) + pow(sphereSamplingPoints[i].z+1.0f, 2.0f) ));
48          float t = 0.5f + sphereSamplingPoints[i].y / (2 * std::sqrt( pow(
                sphereSamplingPoints[i].x, 2.0f) + pow(sphereSamplingPoints[i].y
                , 2.0f) + pow(sphereSamplingPoints[i].z+1.0f, 2.0f) ));
49
50          writeImageFloat(pixels, tgt::ivec2(width, height), s, t, tgt::vec4(
                e, e, e, 1.0f));
51      }
```

```
52
53      LINFO("Photon Tracing done.");
54
55      // copy pixels to reder target
56      outport_.activateTarget();
57
58      glClearColor(0.0, 0.0, 0.0, 0.0);
59      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
60
61      glWindowPos2i(0,0);
62      glDrawPixels(outport_.getSize().x, outport_.getSize().y, GL_RGBA,
            GL_FLOAT, pixels);
63
64      outport_.deactivateTarget();
65      LGL_ERROR;
66
67      delete[] pixels;
68  }
```

Listing C.1: ComputeBRDF Function Definition

```
1  float RaytracerCL::PhotonTrace(tgt::vec3 lightSamplingPoint, tgt::vec3
       viewSamplingPoint, const float step_size)
2  {
3      // Create energy ray
4      EnergyRay ray;
5      ray.recursion_depth = 0;
6      ray.direction = -lightSamplingPoint;
7
8      ray.next_sample = sphereSamplingPointToEntryPoint(lightSamplingPoint);
9      ray.old_medium = tgt::vec4(0.0f);
10     ray.load = 1.0f;
11
12     // Photon trace ray
13     std::vector<EnergyRay> results = PhotonTraceIterative(ray, step_size);
14
15     // The ray exited the volume. Calculate its position on unit sphere
16     // Exit sampling point on sphere is normalize( exitPoint - (0.5, 0.5,
           0.5) )
17     tgt::vec3 lightExitSamplingPoint = tgt::normalize( tgt::vec3(0.5f) -
           results[0].next_sample );
18
19     // Check how much energy reaches the view point. Use cosine
```

```
20      float energy = tgt::dot( viewSamplingPoint, lightExitSamplingPoint) *
            ray.load;
21
22      return energy;
23  }
```

Listing C.2: Definition of the PhotonTrace Function

```
1   std::vector<RaytracerCL::EnergyRay> RaytracerCL::PhotonTraceIterative(
        EnergyRay initialRay, const float step_size)
2   {
3       std::vector<EnergyRay> exitingRays;
4
5       std::vector<EnergyRay> raysToTrace;
6       raysToTrace.push_back(initialRay);
7
8       while(!raysToTrace.empty())
9       {
10          EnergyRay ray = raysToTrace[0];
11          raysToTrace.erase(raysToTrace.begin());
12
13          // check if ray meets an exit condition
14
15          // ray reached max depth without ever exiting volume. The volume
                absorped the energy.
16          if( ray.recursion_depth == this->MAX_DEPTH )
17          {
18              EnergyRay e = ray;
19              e.load = 0.0f;
20              exitingRays.push_back(e);
21              continue;
22          }
23
24          // The ray lost all its energy
25          if( ray.load < 0.0001f )
26          {
27              exitingRays.push_back(ray);
28              continue;
29          }
30
31          // check if next_sample is outside of volume or not defined
```

```
32          if( ray.next_sample.x > 1.0f || ray.next_sample.y > 1.0f || ray.
                next_sample.z > 1.0f || ray.next_sample.x < 0.0f || ray.
                next_sample.y < 0.0f || ray.next_sample.z < 0.0f  )
33          {
34              exitingRays.push_back(ray);
35              continue;
36          }
37
38          // PHOTON TRACE
39
40          // calculate sampling point
41          tgt::vec3 sample = ray.next_sample;
42
43          // get color for sample
44          tgt::vec4 color = this->getVoxel(sample, use_linear_filtering_.get
                ());
45
46          // calculate gradient of sample
47          tgt::vec3 gradient = getGradient(sample, use_linear_filtering_.get
                ());
48          tgt::vec3 normalized_grad = tgt::normalize(gradient);
49
50          // use oldfashioned absorption / emission integral if ray is in
                same medium, gradient is not defined or if alpha is too low
51          if( tgt::length( color - ray.old_medium ) < 0.001f || tgt::length(
                gradient) < 0.001f || color.a < solid_alpha_threshold_.get() )
52          {
53              // the ray is still in the SAME MEDIUM.
54
55              EnergyRay integral_ray = ray;
56              bool exit = false;
57              while( tgt::length( color - integral_ray.old_medium ) < 0.001f
                    || color.a < solid_alpha_threshold_.get() )
58              {
59                  integral_ray.load = integral_ray.load * (1.0f - color.a);
60
61                  // set forward in volume
62                  integral_ray.next_sample += step_size * tgt::normalize(
                        integral_ray.direction);
63
64                  // check if next_sample is outside of volume
65                  if( integral_ray.next_sample.x > 1.0f || integral_ray.
                        next_sample.y > 1.0f || integral_ray.next_sample.z > 1.0
```

```
                f || integral_ray.next_sample.x < 0.0f || integral_ray.
                next_sample.y < 0.0f || integral_ray.next_sample.z < 0.0
                f )
66              {
67                  exitingRays.push_back(integral_ray);
68                  exit = true;
69                  break;
70              }
71
72              // set old medium
73              integral_ray.old_medium = color;
74
75              // get new sample
76              color = this->getVoxel(integral_ray.next_sample);
77          }
78
79          if(exit)
80              continue;
81
82          // raise recursion depth
83          integral_ray.recursion_depth = ray.recursion_depth + 1;
84
85          // continue photon trace
86          raysToTrace.push_back(integral_ray);
87          //result.a = result.a + (1.0f - result.a) * pre_color.a;
88      } else {
89
90          // ray hit a different medium. Decide whether we have specular
                reflection, refraction or diffuse reflection
91
92          // REFRACTION is omitted. There needs to be a way to retrieve
                refraction indices
93
94          // REFLECTION case
95
96          // DIFFUSE case
97          // spawn several new rays in Energy Path Tracing. Each ray has
                the Energy cos(theta) where theta is the angle between
                incoming Ray and Normal (gradient)
98          // the new rays should be equally distributed for now
99          if( tgt::length(gradient) < gradient_specular_threshold_.get()
                )
100         {
```

```
101                 tgt::vec3 normal = normalized_grad;
102                 tgt::vec4 average_color(0.0f);
103                 std::vector<tgt::vec3> random_ray_directions =
                        generateRandomDiffuseRays(DIFFUSE_RAYS);
104
105                 for(size_t i = 0; i < random_ray_directions.size(); i++ )
106                 {
107                     // create new ray
108                     EnergyRay new_ray;
109                     new_ray.recursion_depth = ray.recursion_depth + 1;
110                     new_ray.old_medium = color;
111                     new_ray.direction = tgt::normalize(
                            random_ray_directions[i]);
112                     new_ray.next_sample = ray.next_sample + step_size *
                            new_ray.direction;
113
114                     float costheta = std::max(0.0f, tgt::dot(normal,
                            new_ray.direction));
115                     new_ray.load = costheta * ray.load;
116
117                     raysToTrace.push_back(new_ray);
118                 }
119             }
120             // SPECULAR case
121             else
122             {
123                 EnergyRay new_ray;
124                 tgt::vec3 reversed_ray_direction = - ray.direction;
125                 new_ray.direction = tgt::normalize( 2 * tgt::dot(
                        normalized_grad, reversed_ray_direction ) *
                        normalized_grad - reversed_ray_direction );
126                 new_ray.next_sample = ray.next_sample + step_size * new_ray
                        .direction;
127                 new_ray.recursion_depth = ray.recursion_depth + 1;
128                 new_ray.old_medium = ray.old_medium;
129                 new_ray.load = ray.load;
130
131                 // energy is loast according to phong
132                 // use PHONG here cos(theta) where theta is the angle
                        between V (Viewvector, incoming Ray) and R (reflected
                        Ray).
133                 int e = specular_exponent_.get();
```

```
134                 float factor = std::max( 0.0f, tgt::dot( new_ray.direction,
                        reversed_ray_direction ));
135                 factor = pow(factor, e);
136
137                 new_ray.load *= factor;
138                 raysToTrace.push_back(new_ray);
139             }
140         }
141     }
142
143     return exitingRays;
144 }
```

Listing C.3: Iterative Implementation of the Photon Tracer

Erklärung über Eigenständigkeit

Hiermit versichere ich, Karsten Jeschkies, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.
Gedanklich, inhaltlich oder wörtlich Übernommenes habe ich entsprechend kenntlich gemacht, d.h. dieses durch Angabe von Herkunft und Text oder Anmerkung belegt. Dies gilt in gleicher Weise für Bilder, Tabellen und Skizzen, die nicht von mir selbst erstellt wurden.

Ort, Datum und Unterschrift