Max-Planck-Institut für Informatik
Computer Graphics Group
Saarbrücken, Germany

Master's Thesis in Computer Science

Realistic Real-time Rendering of Refractive Objects

Saarland University
Faculty of Natural Science and Technology I
Department of Computer Science

submitted by

Art Tevs

on July 12, 2007

Supervisor
Prof. Dr. Hans-Peter Seidel, MPI für Informatik, Saarbrücken, Germany

Advisor
Dr. Ivo Ihrke, MPI für Informatik, Saarbrücken, Germany

Reviewers
Prof. Dr. Hans-Peter Seidel, MPI für Informatik, Saarbrücken, Germany
Prof. Dr.-Ing. Philipp Slusallek, Saarland University, Saarbrücken, Germany

Author: Art Tevs
Student number: 2500415
Address: Breslauer Str. 18
66121 Saarbrücken

Statement

Hereby I confirm that this thesis is my own work and that I have documented all sources used.

Saarbrücken, Juli 12. 2007

(Art Tevs)

Declaration of Consent

Herewith I agree that my thesis will be made available through the library of Computer Science Department.

Saarbrücken, Juli 12. 2007

(Art Tevs)

Abstract

Physically correct rendering of inhomogeneous refractive objects in real-time is a difficult task. Many published works which address this problem require either a lot of computational power or can only reproduce a subset of optical effects achievable by a realistic simulation of light behavior inside such structures. In this thesis, we present a way for real-time rendering of complex refractive objects, described by a volumetric representation. Our approach enables us to simulate a variety of physically motivated optical effects. The algorithm is based on the eikonal equation, the main postulate of geometric optics. We derive a system of ordinary differential equations that allows us to simulate the propagation of light rays through an inhomogeneous refractive index field. Afterwards, a powerful image formation model provides for sophisticated rendering effects, such as arbitrary varying refractive index, inhomogeneous attenuation, as well as spatially-varying anisotropic scattering and reflectance properties. We also propose an efficient wavefront propagation technique, achieved with a complexity of a particle tracer, which enables us to compute the distribution of differential irradiance values inside a volume of interest. Efficient GPU implementations enable us to render a combination of visual effects that were previously not reproducible in real-time.

Acknowledgment

I want to thank Prof. Dr. Hans-Peter Seidel for the possibility to work at the Computer Graphics group at Max Planck Institute for Informatik. I also want to thank my supervisor for the duration of my research assistant job, Dr. Christian Theobalt.

I express deep gratitude to Ivo Ihrke and Gernot Ziegler for their endurance with me during this project. I thank Naveed Ahmed for supporting me in reviewing the text.

Thanks to Anders Sundstedt and Michael Schantin for their modeling contributions to the animation demo, created especially for this project. And finally thanks to GusGus for providing us with the permissions, for using their soundtrack in our project demo.

CONTENTS

1 Introduction 9

1.1 Idea 10

1.2 Algorithm 10

2 Related works 13

2.1 Ray tracing and Photon mapping approach 13

2.1.1 Ray Tracing 13

2.1.2 Photon mapping 14

2.2 Utilizing a streaming processor 15

2.2.1 GPU-based interactive refraction 16

2.3 Discussion 17

3 Background 19

3.1 Gradient computation 19

3.2 Optical Effects 20

3.2.1 Surface BRDF 21

3.2.2 Refraction and Caustics 21

3.2.3 Attenuation and Absorption 22

3.2.4 Scattering 23

3.2.5 Reflection and Fresnel equation 24

3.2.6 Emission 25

3.2.7 Dispersion 25

3.3 Conclusion 27

4 Light simulation 29

4.1 Ray optics 29

4.1.1 Light propagation with Snell's law 30

4.1.2 The Eikonal equation and the ray equation of geometric optics . 31

4.2 Light and wavefront definitions 32

4.2.1 Light source 32

4.2.2 Wavefront representation 33

4.2.3 Wavefront propagation 34

4.2.4 Irradiance computation 36

4.3 Conclusion . . . 37

5 Image Formation Model 39
5.1 General image formation . . . 39
5.1.1 General volume rendering equation . . . 39
5.1.2 Attenuation and the background . . . 39
5.1.3 Reflection, Scattering and Emission . . . 41
5.1.4 Discretization of rendering terms . . . 42
5.2 Discrete image formation model . . . 43
5.3 Conclusion . . . 45

6 Implementation 47
6.1 Input Data . . . 47
6.2 Light Simulator . . . 49
6.2.1 Initialization . . . 49
6.2.2 Wavefront patch propagation . . . 50
6.2.3 Voxelization of wavefront patches . . . 50
6.2.4 Patch list analysis . . . 51
6.3 View renderer . . . 52
6.3.1 Ray casting . . . 53
6.4 Conclusion . . . 55

7 Results 57
7.1 Objects . . . 58
7.2 Results . . . 60

8 Conclusions & Future Work 63

A Anisotropic materials 65
A.1 Birefringence . . . 66
A.2 e-Ray propagation in uniaxial crystals . . . 67
A.2.1 Refraction and Internal reflection of e-rays . . . 69
A.3 Absorption/Attenuation . . . 71

B Formulae 73

C Code Listing 77

CHAPTER 1

INTRODUCTION

”In the beginning ... there was light”: these words can be found in a world famous book. Humans knew from the beginning of time that light is indispensable for our perception of the environment. Light particles, photons, start their travel from a light source and traverse at immense speed. They are reflected or absorbed by objects before they reach our eye. These light particles, which are absorbed by our eyes, produce a world full of beautiful colors on our retina. Our brain is then able to reconstruct the environment in our mind, so that we can feel the environment visually.

Computer graphics tries to reproduce the behavior of the light in a machine simulation. To achieve very realistic representations, one can attempt to simulate the behavior of the light based on the actual light particle paths. Having a strong theory of geometric optics, one can deduce how light particles interact with the environment. However, due to limited computational power, one has to lower the expectations. Some parts of the light interaction with the environment can not be simulated and thus the produced images do not look as realistic as one could see them in the real world.

Objects with complex optical properties are needed for a close simulation of reality in a virtual environment. A very simple example for a complex object can be a glass of water. For us, as humans, there is nothing amazing about it and it seems to be a quite simple object. But to correctly simulate the light interactions with such an object in a virtual environment is not as an easy task as one might think.

Light particles from a light source have to pass through more than two media (here: glass, water, air) before they reach our eyes. The light interacts with the medium while it is travelling through it. At material boundaries or even inside a medium with strong inhomogenity, the particle can be reflected or change its path in a complex way. Dust or dirt particles inside a medium can absorb or attenuate light. The absorbed energy is later emitted, usually as heat[1], in many directions, so that a simple representation of real-world light interactions is not really possible.

In this thesis we will introduce a general framework able to simulate the complex light behavior. Most of these effects can be visualized in real-time on commodity graphics hardware. The thesis can be understood as an expansion and a detailed version of the algorithm presented in [IZT+07].

---

[1] Fluorescent and phosphorescent materials emit energy in the visible spectrum [Pri63]

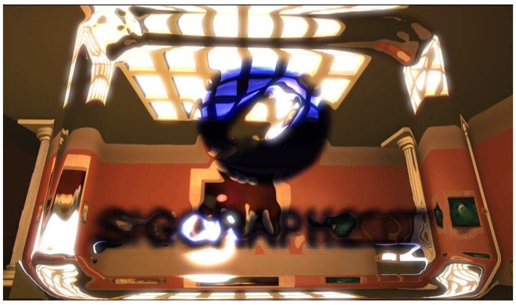


Figure 1.1: Glass block with embedded SIGGRAPH logo. The complex behavior of refraction is combined with a spatially varying attenuation inside the letters. Note the total reflection on the block boundaries.

1.1 Idea

The main idea of the algorithm is based on a simple set of ordinary differential equations derived from the *eikonal equation*, the main postulate of geometric optics [BW99]. The eikonal equation $\mid \vec{\bigtriangledown} S \mid = n$ describes the spatial distribution of light arrival. The iso-surfaces of the eikonal solution S are also called wavefronts.

Using *Fermat's principle*, rays become characteristic of the eikonal equation. A light ray, represented by particles, is always perpendicular to the wavefront. Therefore, we represent the wavefronts by particles and thus provide a simple way to propagate the wavefront from a light source through the scene. Particle system can be efficiently simulated on today's programmable graphics hardware. The radiant energy, transferred by the wavefront, can be used to illuminate the scene. In our case, light energy can also be absorbed by the nature of the object's material, giving us a possibility to render colored objects.

The same idea as for the wavefront propagation is used to cast viewing rays into the scene. For this purpose, we define a rendering equation which is used to compute the radiance of the viewing rays. This, combined into a powerful, physically motivated image formation model, allows us to realistically render refractive objects in real-time.

1.2 Algorithm

We represent our refractive objects by a volumetric data structure storing spatially varying object properties in voxels. Thus, for example, we store the refractive index field, responsible for spatially varying refraction, as a 3D volume V. A more detailed description of the implementation of the scene representation using a volumetric data structure, can be found in Chapter 6.

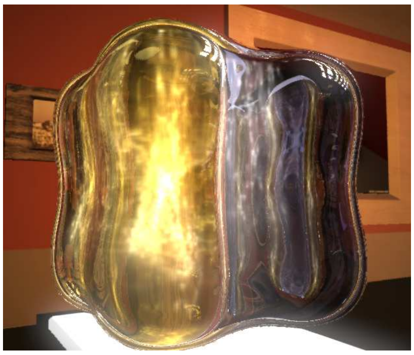


Figure 1.2: Rounded cube consisting of three different glass layers. Light simulator is capable to compute the irradiance distribution inside the object volume providing a nice sparkle like structures during the rendering phase.

The rendering process is performed in two steps. First, a *light simulator*, described in Chapter 4, pre-computes the irradiance distribution, which is needed to simulate the scattering of light particles inside a medium. For this we introduce the concept of a *wavefront*[2]. The eikonal equation is used to derive the ordinary differential equations for the wavefront's propagation. The computation of the light trajectory through a refractive medium is based on the properties of geometric optics.

The pre-computed irradiance distribution is used by the *view renderer* in the second step to realistically render the appearance of translucent objects in scattering participating media, such as smoke. For scenes without scattering, we do not require the pre-computation step, because the represented effects do not depend on the light distribution in the scene.

In general, our framework enables us to reproduce a variety of sophisticated effects on a commodity Shader Model 3.0 graphic hardware. An expressive image formation model (Chapter 5) combined with ray propagation theory (Chapter 4) derived from the *eikonal equation*, supports the rendering of scenes containing objects with arbitrarily varying refractive indices. It is also able to handle surface effects with arbitrary BRDFs and view-dependent single-scattering effects with arbitrary scattering phase functions. Some of the non-geometric effects, like dispersion, can also be realistically simulated within the framework. Advanced effects, such as total reflection, are implicitly obtained at no additional cost. The algorithm achieves real-time viewer performance on objects with complex refractive properties consisting of high resolution voxel volume data. The implementation of our image formation model and the light simulator can be found in Chapter 6.

---

[2] an iso-surface of equal time distance from the light origin

CHAPTER 2

RELATED WORKS

This chapter presents a selection of previous work done in the field of rendering of refractive objects. We show examples, results and discuss the assets and drawbacks of these algorithms with respect to real-time rendering applications.

2.1 Ray tracing and Photon mapping approach

An intuitive way of simulating light propagation through any kind of material can be developed by using a true *ray tracing* approach. Ray tracing gives the possibility to simulate light rays which can be deflected on material boundaries according to the physical laws of reflection and refraction.

Most ray tracing algorithms achieve good results by recursively evaluating Snell's law at material boundaries. But a continuous computation of refractive events along arbitrarily varying paths, like in our method, would be computationally very expensive. To overcome this problem and to be able to render volumetric refractive objects with continuously varying refractive index, one can use iso-surfaces to simulate boundaries of different refractive indices in the data. However, such an approach requires considerable computation time, since explicit material boundaries, iso-surfaces, have to be computed.

2.1.1 Ray Tracing

One of the first ray tracing approaches used to compute caustics and refractions was backward ray tracing shown in [Arv86]. The algorithm uses multiple passes and an illumination map (similar to the photon map) which helps to compute the caustics seen from the view point.

Stam et al. [SL96] have used, as one of the first, the eikonal equation to trace rays through non-constant media with a continuously varying index of refraction. The authors derived the ray equation of geometry optics to handle these media in a standard ray tracer. Their work is motivated by the rendering of natural phenomena, such as mirages.

There is some interesting work on interactive ray tracing simulation including refractions and caustics. One approach is described in [WBS$^+$02]. It can solve various ray tracing problems including refractions. The approach is based on a ray tracing simulation. However, for the computation of caustics, they use photon maps, storing irradiance informations in a 3-dimensional regular grid. The algorithm produces very impressive

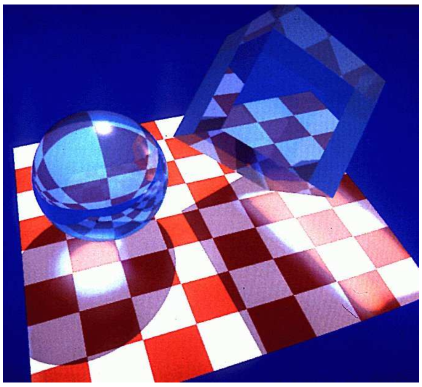


Figure 2.1: One of the first synthetic image showing caustics, refractions and reflections [Arv86]

results, as can be seen in Fig. 2.2. Unfortunately, the algorithm requires more than one fast CPU, and typically runs on a PC cluster to handle the scene in an interactive mode.

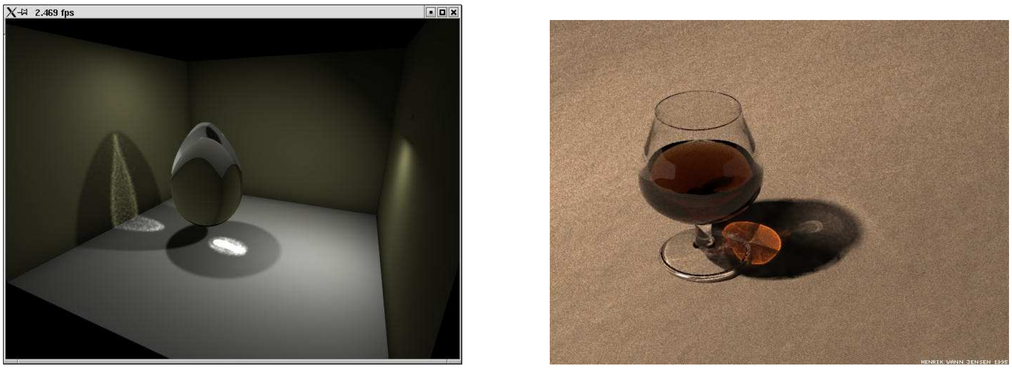
Hakura and Snyder [HS01] propose a slightly different ray tracing approach, which they call hybrid ray-tracing. The authors combine a standard ray tracing, which simulates complicated ray bouncing off local geometry, with environment maps which capture the more distant geometry. Furthermore, their algorithm handles refraction and reflection very well by lowering the costs of computation. The produced results are appealing, but the algorithm does does not run in real-time.

To increase computation speed some approaches utilize the GPU or other special hardware. Very exciting work was done by Schmittler et al. [SWS02]. They developed a special graphics processing unit which is able to render ray-traced images at interactive frame rates. Another algorithm, utilizing special hardware, can be found in [Ohb03].

2.1.2 Photon mapping

A simulation in the opposite direction (from light source to the object) can be achieved using the photon mapping approach. Photon mapping was first introduced by H. W. Jensen [Jen96]. It is probably the most intuitive way to implement light propagation.

In photon mapping the photons are sent out into the scene from the light source. Whenever a photon intersects with a surface it is stored in so called photon map which is usually a hierarchical tree structure, e.g. a kd-tree. If the photon continues bouncing, a new propagation direction of the photon is computed. The photon map is used during the rendering to estimate the density of accumulated photons, as an estimate for the local irradiance.



(a)

(b)

Figure 2.2: (a) Realistic refractions and caustics rendered at 2.5 frame per second on 8 client machines [WBS+02] (b) Caustic computed by the photon mapping approach described in [Jen01]

Rendering of realistic caustics, as with the photon mapping approach, is related to the problem of refraction rendering. An interesting solution of the caustic rendering problem with a photon mapping approach is described in [GWS04]. They parallelize the photon mapping algorithm to achieve interactive frame rates across commodity PCs. Their solution is attractive, especially for the realistic rendering of caustics. But the algorithm requires a PC cluster with up to 36 CPUs. The computed results are realistic, but due to the strong requirements on computational power, this method is not suitable for everyday realistic refraction simulation.

2.2 Utilizing a streaming processor

Recently, researchers have ported native ray tracing or photon mapping algorithms to graphics hardware to achieve real-time performance. Some research on graphics hardware algorithms has explored the idea of simulating global illumination. Ma et al. [MM02] propose a technique to approximate nearest neighbor search in the photon map on a GPU using a block hashing scheme. Their scheme is optimized to reduce bandwidth on the hardware, but requires processing by the CPU to build the data structure. Carr et al. [CHH02] and Purcell et al. [PBMH02] use the GPU to speed up ray tracing.

Purcell et al. [PDC+03] show a stream processor implementation for the photon mapping algorithm. Their implementation uses breadth-first photon tracing to distribute photons using the GPU. The grid-based photon map is constructed directly on the graphics hardware. The results are very impressive, however they can not handle volumetric caustics and inhomogeneous refractive materials.

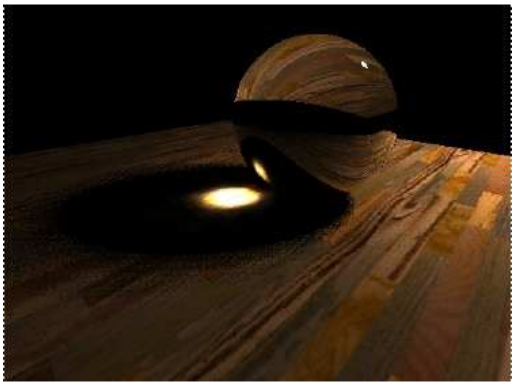


Figure 2.3: Photon mapping computed on the GPU by [PDC$^+$03]

A similar idea was presented by Purcell in his dissertation [Pur04]. He changed the data structure for the photon map to a uniform grid, which can be constructed directly on the hardware. In addition he uses recursive ray tracing for specular reflection and refraction. His ray tracer uses the kNN[1]-grid photon map to compute caustic effects.

2.2.1 GPU-based interactive refraction

Since the common graphics hardware has become programmable, it has been attempted to port CPU-based algorithms to the new hardware . As already mentioned in the previous section, there are multiple algorithms implementing ray tracing or photon mapping on a stream processor like a GPU.

The first refraction effects shown on the GPU were very simple and not physically correct. There, a light ray (in most cases the ray between view point and a point on the mesh surface) is refracted only once. However, in many cases this does not suffice and at least double refraction is required.

One approach was introduced by Wyman [Wym05a]. The method can approximate refraction effects in real-time on the GPU. An extension of this algorithm for nearby geometry was presented in [Wym05b]. Another interesting approach is shown by Wand and Strasser [WS03]. They suggest to compute reflective caustics by approximating surfaces with uniformly sampled light sources.

Wyman and Davis [WD06] propose an interactive image space technique to approximate caustic rendering on the GPU. The authors use a traditional two-pass rendering approach similar to photon mapping. In the first pass, the photons are emitted and their contribution is stored in a buffer by rendering the scene as it was seen by the light source.

---

[1] k-nearest neighbors

In the second pass, the photons are gathered by an image-space nearest neighbor search. The rendering is performed in real-time, but is limited to point light sources and does not consider volumetric scattering effects.

Shah et al. [SK07] propose a full GPU implementation of an image-space technique for real-time caustic rendering. The authors create a caustic-map texture by splatting the vertices of a refractive object onto the receiver's geometry. The rendering is performed in real-time and does not require any pre-computation. However the approach can not reproduce volumetric caustics and is limited to homogeneous refractive media.

Hu and Qin [HQ07] present an interactive, image-based approach for the rendering of reflection, refraction and caustics. The method implements double refraction by computing the refraction vector on the determined back-face and front-face of the refractive object. Furthermore, a new method for nearby geometry rendering is proposed. However the method works only on objects with constant index of refraction and is not capable of simulating volumetric effects, such as volume caustics or scattering.

2.3 Discussion

In contrast to these GPU techniques, we employ a more general model of light ray propagation through continuous refractive media. Similarly, computation of the irradiance distribution in the scene volume enables us to reproduce volumetric effects, e.g. anisotropic scattering or volumetric caustics. A powerful implementation of our image formation model gives us the possibility to render various additional effects, such as dispersion, emission, scattering, BRDFs and spatially varying attenuation within a common framework.

Adaptive wavefront tracing also enables us to simulate error-bounded, non-linear light transport with the complexity of a particle tracer. The computation time during the update, which is required if the light position changes, is comparable to other state-of-the art GPU methods reproducing fewer effects, e.g. only caustics in isotropic media [EAMJ05].

Nevertheless our algorithm is in other aspects not as powerful as some of the related approaches, i.e. photon mapping, which can produce full global illumination solutions. The disadvantages of a volumetric scene representation make our method only suitable for spatially confined refractive objects. However, the rendering performance and the wide range of reproducible, physically plausible[2], realistic looking effects benefits from the simplicity and generality of our method.

---

[2] within the limits of geometrical optics, see [BW99] for details

CHAPTER 3

BACKGROUND

In this chapter we will explain the basics of geometric optics and some of its aspects used in our approach. We will describe particular optical effects which can be simulated with our algorithm. Some equations used in the further computations will also be presented and explained in an intuitive way. We complement with an introduction to gradients computation, which are required to derive valid ray equations presented in the next chapter.

3.1 Gradient computation

The gradient of a scalar field (i.e. spatially varying refractive index field) is a vector field, called *gradient field*, and is defined as (in cartesian coordinates):

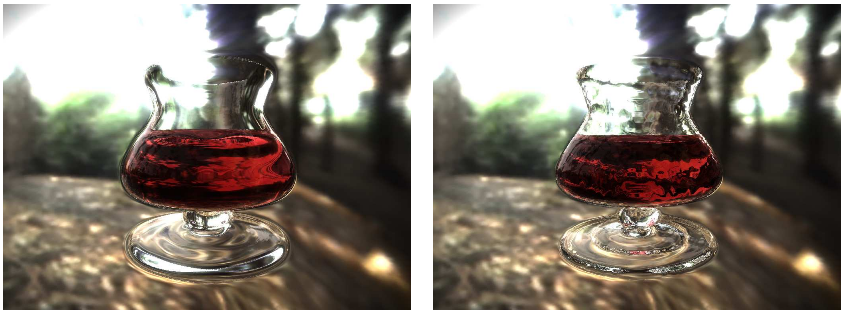
$$\vec{\bigtriangledown} n(x,y,z) = (\frac{dn}{dx}, \frac{dn}{dy}, \frac{dn}{dz}),$$

where $n(x,y,z) = n$ defines the refractive index field. There exist several methods to calculate gradient vectors. In our approach we are using the most common method, *central differences*. This method estimates the derivative by calculating the first terms of a Taylor expansion.

$$\begin{aligned} \vec{\bigtriangledown} f(x,y,z) \approx (&\frac{f(x+\triangle x,y,z)-f(x-\triangle x,y,z)}{2\triangle x}, \\ &\frac{f(x,y+\triangle y,z)-f(x,y-\triangle y,z)}{2\triangle y}, \\ &\frac{f(x,y,z+\triangle z)-f(x,y,z-\triangle z)}{2\triangle z}) \end{aligned} \tag{3.1}$$

In our data representation we are using the six neighbor voxels, i.e. refractive indices, to compute the gradient at position (x,y,z).

Due to its nature, a gradient points in a direction normal to the iso-surface of function f. In our case, the iso-surface of the refractive index function n can be interpreted as the boundary between different media. Thus, the gradient $\vec{\bigtriangledown} n$ becomes indicative of an object's surface normal. However, by using discrete input data (e.g. $n_1 = 1$ for air and $n_2 = 1.5$ for glass sampled at discrete positions in space (x_i, y_i, z_i)), we only receive discretized normals. The gradient directions are thus only pointing in a discrete number



(a) (b)

Figure 3.1: Wine glass object consisting of 128x128x128 voxels. (a) Refractive index field was pre-smoothed before computing the gradients. (b) No smoothing: Glass appears blocky. Note how the smooth filtering unintendedly expands the boundaries.

of directions. This leads to the restriction that smooth surfaces can not be represented properly. To overcome this problem, we either have to use more samples, requiring more processing power and memory, or we have to smooth the gradients (Fig. 3.1).

We use a simple three-dimensional smoothing operator to pre-smooth the refractive index field. The operator is a three dimensional convolution kernel, which is applied a priori to the volumetric object representation. The convolution kernel can be computed by a three dimensional *Gaussian function* as following:

$$g(x,y,z) = e^{-\left((\frac{x-x_O}{\sigma})^2+(\frac{y-y_O}{\sigma})^2+(\frac{z-z_O}{\sigma})^2\right)}. \tag{3.2}$$

For the standard deviations of the filter kernel we typically use values between 0.5 and 1 voxels, resulting in object boundaries which extend over 2-3 voxels. This yields a so called *halo*-effect, produced by interpolated refractive indices on object boundaries.

Other methods are equally valid for gradient computations, like a three dimensional Sobel operator [SHB99]. In general a good smoothing operation on refractive indices is indispensable to provide smooth surfaces.

3.2 Optical Effects

In this section we introduce the optical effects which can be simulated with our approach. We show, in short, how these effects arise in real life and describe how they can be simulated within our framework. Furthermore we define a light ray as a travel path of light particles (i.e. photons).

3.2.1 Surface BRDF

The *bidirectional reflectance distribution function* (also called just *BRDF*) describes the ratio of reflected radiance to the irradiance incident on the surface. The BRDF is a material property of the viewed surface. The common definition of the BRDF, has the form:

$$f_r(\theta_i, \phi_i, \theta_r, \phi_r) \equiv \frac{dL_r(\theta_r, \phi_r)}{L_i(\theta_i, \phi_i) \cos\theta_i d\omega_i}, \tag{3.3}$$

where (θ_r, ϕ_r) is the direction of reflected radiance L_r, (θ_i, ϕ_i) is the incident direction of the irradiance L_i and $d\omega_i$ is a differential solid angle in the incident direction.

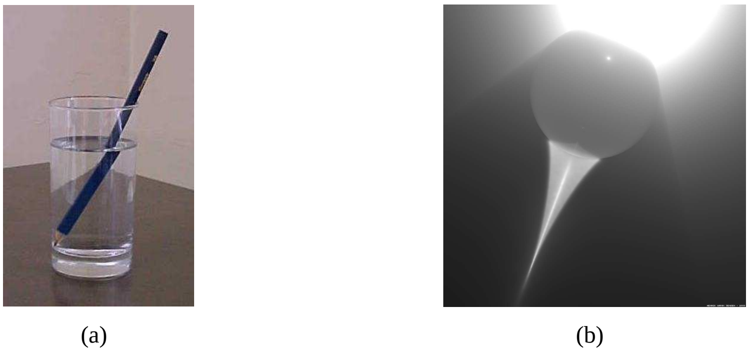
The BRDF can either be described by mathematical models or measured for particular directions and interpolated inbetween. The BRDF obeys the Helmholtz reciprocity principle, i.e. the BRDF remains unchanged if the incoming and outgoing directions are interchanged.

Based on the BRDF, one can define varying surface properties and hence simulate different material surfaces. The ratio between reflected and incident radiance could also be used beyond the Fresnel reflection (Sect. 3.2.5) to simulate more specific materials that can not properly be handled by the Fresnel equations.

3.2.2 Refraction and Caustics

Refraction is the directional change of a wave due to a change in its speed. In optics, refraction occurs on the boundary between two mediums with different refractive indices. For example, a light ray refracts when it enters and leaves a glass. The strength of the refraction depends on the *refractive indices* of the two media bordering the boundary and the angle between the light ray and the line normal to the surface separating the two media (Fig. 4.1). A good optical example for this, is the view inside a bowl of water. Air has a refractive index of just over 1, and water has a refractive index of about 1.3. A straight object, e.g. a ruler, placed partially in the water, will appear to bend at the water's surface (Fig. 3.2 (a)).

Refraction is also the cause for caustics. Caustic effects appear, when a light beam, propagating through a refractive medium, is being focused. The method we present here uses this relation to create realistic caustics. If a participating medium, e.g. smoke, is present in the scene, light can be scattered in all directions at a certain scene point (Sect. 3.2.4). Thus, if the object material through which the light is propagated focuses the light, we obtain visible volume caustics (Fig. 3.2(b)). To create surface caustics we



(a) (b)

Figure 3.2: (a) A real world photo showing the refraction effect. The light rays are bent as they cross from water to air. (b) Volume caustics created by propagating light particles through a glass sphere. Notice how the light rays are being focused [Jen01].

simply intersect a 3D surface mesh with the computed, volumetric irradiance distribution.

3.2.3 Attenuation and Absorption

In physical optics, *absorption* is a process where the energy of a photon is transfered to another entity, for example an atom. The effect is common and can be seen everywhere in our daily life. Almost any object absorbs some portion of incoming light, which makes us see it in different colors. For example, the ink of the text in this paper absorbs almost all light frequencies, giving an impression of black color. In refraction, the translucent absorption of light during light propagation is also called *attenuation*.

Attenuation is the process of decreasing intensity of an electromagnetic radiation due to absorption or scattering of photons. We define attenuation as a scalar field σ_a for each color component, describing how much of a certain wavelength (color) is attenuated at the point $\vec{x} = c(t)$, during the propagation of the ray c.

Since the light ray has to propagate through a medium with some thickness, we can define the attenuation factor $\alpha(t, c)$ according to the *absorption law*:

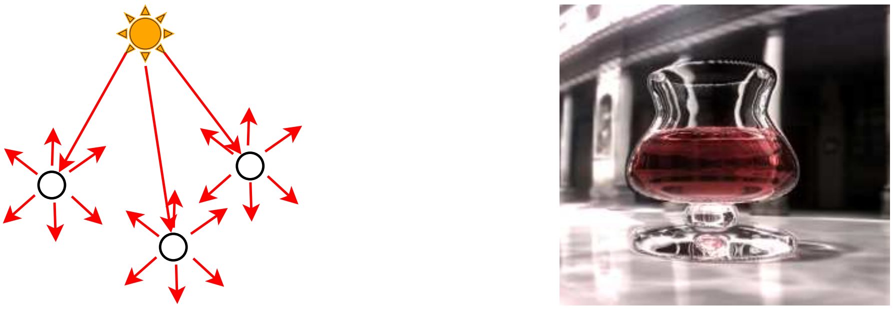
$$\alpha(t, c) = L_0 \cdot e^{-\mu \cdot d}.$$

Here L_0 is the initial radiance on the ray c, μ is the absorption constant, also called *absorbance*, and d is the thickness[1] of the medium.

Now in order to simulate *spatially varying* attenuation, we have to change the equation slightly to

---

[1] here: d is the optical path length of the ray c



(a) (b)

Figure 3.3: (a) Scattering occurs when light originating at the light source *scatters* on material impurities, e.g. dust particles. (b) Wine glass showing refraction and attenuation. Wine attenuates green and blue components of the light resulting in a red colored fluid.

$$\alpha(t,c) = L_0 \cdot e^{-\int_0^t \sigma_a(c(s))ds}. \tag{3.4}$$

The function describes the exponential attenuation of radiance on curve (ray) $c(t)$ due to a spatially varying attenuation function σ_a. We will use this formulation later in Chapter 5 to define the complete image formation model. An example of attenuation can be seen in Fig. 3.3(b).

3.2.4 Scattering

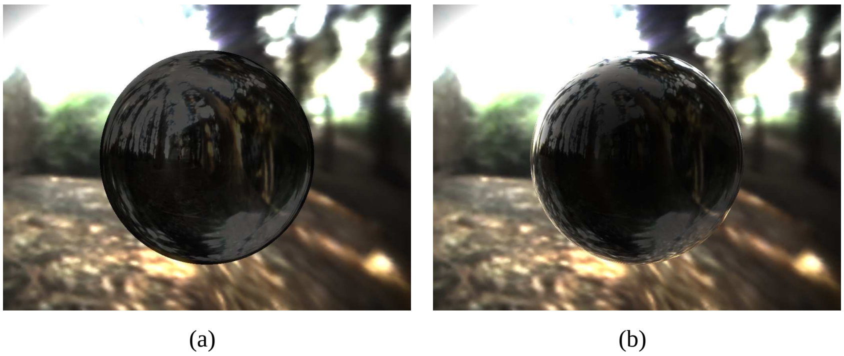
In physics, *scattering* is a process where some forms of radiation are forced to deviate from a straight trajectory by some localized non-uniformities in the medium through which it passes. In optics a light ray is split[2] into an infinite number of rays upon incidence on a scattering particle. To imagine scattering, one can think of small dust particles which reflect the incoming light in all directions (Fig. 3.3(a)). Scattering also helps us to see volumetric caustics directly, since it can be used to simulate fog or dust in the scene volume (Fig. 3.2(b)).

In our approach, we are using an anisotropic scattering phase function. A complete scattering function is currently not suitable for efficient GPU implementation. Therefore we apply an approximation, presented by Henyey and Greenstein [HJ41], which depends on only a few parameters:

$$p = \frac{1-g^2}{2(1-2g\cos\theta+g^2)^{3/2}}. \tag{3.5}$$

---

[2] based on Huygen's Principle a new wavefront is created at a scattering center



(a) (b)

Figure 3.4: Black glass sphere reflecting the environment. The sphere's material absorbs almost the whole light energy such that only the reflection on the surface is visible. (a) Native reflection (b) Fresnel reflection. Note the use of fresnel equations improves the realism of the scene.

g is the anisotropy factor and θ is the angle between the ray $\vec{x}$ and local light direction $\vec{v}$. The anisotropy factor is defined as one of the properties of the scene and can vary spatially.

3.2.5 Reflection and Fresnel equation

Reflection is probably the most intuitive effect of our daily life. We speak about light reflection, when the wavefront changes its moving direction on an interface between two media without leaving the propagating one. When light moves from a medium of a given refractive index n_i into a second medium, with refractive index n_t, both reflection and refraction of the ray may occur.

Since the light ray refracts and reflects on the border between two media, it is divided into two rays[3]. The sum of the radiance, transported by the two light rays, stays equal to the radiance of the incident ray. How the energy is split up can be approximated via the *Fresnel equation*. It describes the behavior of light as it moves between media of different refractive indices. The fraction of the reflected radiance, is given by the *reflection coefficient* R, and the fraction of refracted radiance, by the *transmission coefficient* T.

The amount of reflection is computed with

$$R = \frac{n_i \cos\theta_i + n_t \cos\theta_t}{n_i \cos\theta_i - n_t \cos\theta_t} \tag{3.6}$$

and the transmission fraction is consequently

---

[3] depending on the critical angle total reflection may occur and the ray is not split

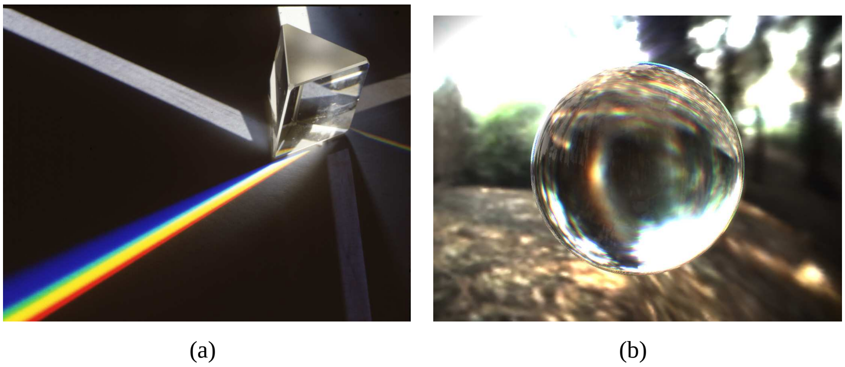

(a) (b)

Figure 3.5: (a) Real world photo of dispersion phenomenon on a prism [MS] (b) Emulated dispersion effect with a three-pass computation of color components red, green and blue.

$$T = 1 - R.$$

n_i and n_t are the refractive indices of the incident and transmitting media. θ_i is the angle between the incident ray and the interface normal and θ_t is the angle between transmitted ray and the normal.

The refraction coefficient depends on the polarization of the light, but in computer graphics it is common to ignore polarization. It is also common to ignore the Fresnel reflection of conductive materials (i.e. metal), since the amount of reflectance varies so little that the human eye has problems to detect them.

Fig. 3.4 shows the difference between native and Fresnel reflection.

3.2.6 Emission

In optics, emission is a physical process where light energy, i.e. photons, are released from another entity. In our case, we define emission as a scalar field $L_e(\vec{x})$ for each color component. The value is also called *emittance* and quantifies how much radiance is emitted. The *emitted radiance* depends on the position $\vec{x}$ and can be evaluated given volumetric descriptions of its distribution.

3.2.7 Dispersion

In optics, *dispersion* is the effect of separating the light ray[4] into its spectral components with different wavelengths. The effect occurs because of the different speed of light for different wavelengths.

---

[4] e.g. a white colored light ray is composed of different colored rays

The common consequence of dispersion is the separation of white light into its color components (i.e. color spectrum). For example, this happens with water drops in the air, which is why we are able to see a rainbow. Another common example is a prism.

The equation to compute the wave speed in a medium is

$$v = \frac{c}{n},$$

where c is a constant (i.e. speed of light in vacuum) and n is the refractive index.

In general the refractive index is a function of the light wavelength λ, so $n = n(\lambda)$. The wavelength dependency is usually quantified by empirical observations. For visible light, most transparent objects have some order for wavelength and refractive indices:

$$1 < n(\lambda_{red}) < n(\lambda_{green}) < n(\lambda_{blue})$$

or in other words

$$\frac{dn}{d\lambda} < 0.$$

This is also called *normal dispersion*, due to decreasing refractive index by increasing wavelength.

We use pseudo-multi-pass-rendering[5] to compute the radiance for each color component. The more passes we use, the more colors we can simulate, yielding increasingly better approximations of the dispersion effect. We use the dependencies above and approximate the refractive index equation $n = n(\lambda)$ for each pass, by using the empirical relationship, also known as *Cauchy's equation* [BW99]. The equation however, works only well for areas with normal dispersion in the visible wavelength region, but this is sufficient for our purpose. We use a two-term equation of the form:

$$n(\lambda) = A + \frac{B}{\lambda^2}, \tag{3.7}$$

where A and B are the coefficients of a material. The coefficients can be determined by measuring refractive indices of known wavelengths.

Having this information available, we are now able to create refractive index fields for each of the relevant wavelengths. If scattering is required, a multi-pass precomputation of radiance distributions for each of the color components is performed. The succeeding rendering step then results in the effect of dispersion. Fig. 3.5 shows a three-pass rendering for the colors red, green and blue.

---

[5] we compute simultaneously the results for each color component in the fragment shader while rendering the scene volume.

3.3 Conclusion

This chapter has demonstrated a variety of optical effects that are all based on the geometric model of light. With the help of multipass rendering we can also provide an emulation of advanced effects like dispersion, for example.

We introduced the concept of gradients and have shown how they can be computed. Prior to gradient computation we have to smooth the refractive index volume to provide smooth objects, excluding the use of high resolution data. However, due to the smoothing we get blurry object boundaries which unfortunately affect the realism of the scene negatively. Here one has to take care not to ”over-smooth” the scene.

By handling reflectance properties with the help of Fresnel equations and BRDFs, we are able to render the refractive and reflective objects in a more realistic way than before. An approximation of the anisotropic scattering function gives us the possibility to simulate various volumetric effects like fog, shadows, and caustics.

Since almost all parameters may vary spatially, we are able to simulate the light propagation more precisely than other algorithms. The formulations of the specific effects shown here help us to derive a proper image formation model which can be efficiently implemented on common graphics hardware. In the following chapter we now derive a ray equation to propagate the light through the scene volume.

CHAPTER 4

LIGHT SIMULATION

This chapter starts off with an introduction to the basics of ray optics[1]. We continue with introducing a wavefront representation for light sources. We will present the *eikonal equation* and describe how this equation can be used to simulate light propagation through the scene volume. The derived first order ordinary differential equations for the light ray propagation assist in understanding how light travels through a medium with spatially varying refraction. Finally, we will show how to gather irradiance values out of the wavefront to compute the illumination in the scene.

4.1 Ray optics

Geometric optics, or ray optics, describes light propagation in terms of ”rays”. Rays are bent at the interface between two dissimilar media, and may be curved in a medium in which the refractive index varies in space. Its geometry thus depends on 3D position (inhomogeneous refractive index field). The ”ray” in geometric optics is the path for a single light particle. A ray is perpendicular to the wavefronts of the actual light waves. Geometric optics provides rules for propagating these particles through an optical system and thus describes how the actual wavefront will propagate. Note that this is only a simple model to describe light propagation. It fails to account for other optical effects such as diffraction and polarization.

Forward or backward ray tracing approaches use this model to trace light through the scene back to the light source or to the view point. Similarly, photon mapping algorithms make use of this model to describe how the photons move along the optical paths.

In our algorithm we will propagate wavefronts based on this ray definition. A wavefront is an iso-surface of constant travel time originating from a light source. Due to Fermat’s Principle, we can use ray optics for wavefront propagation, since light rays always travels normal to these wavefronts (Sect. 4.2).

Due to the computational discretization we are not able to define the wavefront continuously. Therefore, we break the wavefront into a number of *wavefront patches*. Each wavefront patch represents a part of the wavefront as a small area holding a fraction of the light wave energy (Sect. 4.2.2).

---

[1] we handle here only optically *isotropic* materials [BW99]; for *anisotropic* materials see appendix A

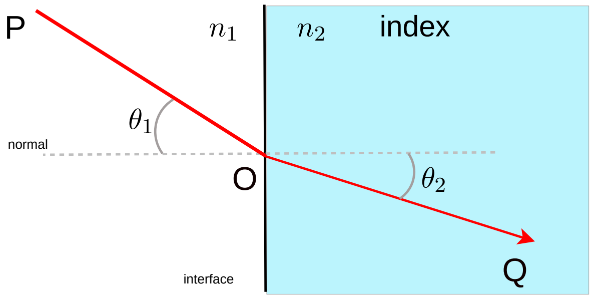


Figure 4.1: ¡¡¡¡¡¡¡ .mine Snell's law; The propagation time of light from P to Q is minimal, if the sines of the ray angles in different media are in ======= Snell's law; The propagation time of light from P to Q is minimal, if the sines of the ray angles in different media are in ¿¿¿¿¿¿¿ .r48 proportion to the refractive indices.

4.1.1 Light propagation with Snell's law

Fermat's Principle states that *the path taken by a ray of light between two points is the path that can be traversed in the least time*. Sometimes this is actually used as the definition for the ray of light. The principle can be used to derive the law of refraction known as Snell's law:

$$n_1 sin(\theta_1) = n_2 sin(\theta_2). \tag{4.1}$$

¡¡¡¡¡¡¡ .mine This equation gives us the possibility to compute a vector of the refraction direction at the boundary of a medium. However, when light moves from a dense to a medium of lower density, such as from water to air, where $\frac{n_2}{n_1} \leq 1$, Snell's law can not be applied. At this point, light is reflected in the incident medium, known as ======= This equation gives us the possibility to compute a vector of the refracted ray direction at the boundary of a medium. However, when light moves from an optically dense medium to a medium of lower density, such as from water to air, where $\frac{n_2}{n_1} \leq 1$, Snell's law can not be applied, i.e. if the incident angle exceeds the so called *critical angle*. At this point, light is reflected back into the incident medium. This behavior is known as ¿¿¿¿¿¿¿ .r48 *internal reflection* or *total reflection*.

If the refractive index of a medium is not constant, but varies with position, then the material is known as a *gradient-index medium*. In our case, we define the *scene volume* V as a gradient-index volume storing gradients of refractive indices in voxels.

Assume now that we want to simulate the propagation trajectory of a light ray c through a scene volume V. A possible implementation is to apply Snell's law at medium boundaries. Furthermore there must be a check for the critical angle, to apply the reflection law if the critical angle is exceeded. This can be avoided if another, more general, formulation for the propagation of light rays is used.

4.1.2 The Eikonal equation and the ray equation of geometric optics

In physics, light is described by a complex wave equation, varying over space and time. The *eikonal equation* [BW99]

$$\mid \vec{\bigtriangledown} S \mid = n, \tag{4.2}$$

is obtained from the wave equation by letting the wavelength go to zero and taking the limit. $S(\vec{x})$ is a real scalar function of position defining the optical path. The function S is also called *eikonal*.

The surface $S(\vec{x}) = const$ is called *(geometrical) wavefront* and describes an iso-surface of constant travel time of light from a light source. Equation 4.2 states that the magnitude of the wavefront gradient[2] is the refractive index.

Let us now derive[3] an equation for ray propagation in a refractive index field[4]. In accordance with Fermat's principle, light rays propagate perpendicular to the wavefronts. A ray in space can be defined as:

$$\begin{aligned} \vec{x} &= \vec{x_0} + s\vec{u} \\ \Leftrightarrow \vec{u} &= \frac{d\vec{x}}{ds}. \end{aligned} \tag{4.3}$$

We assume $\mid \vec{u} \mid = 1$. Since the gradient of the wavefront points in the direction of $\vec{u}$, with magnitude n, we can write

$$\vec{\bigtriangledown} S = n\vec{u} = n\frac{d\vec{x}}{ds} \Leftrightarrow \frac{\vec{\bigtriangledown} S}{\mid \vec{\bigtriangledown} S \mid} = \frac{d\vec{x}}{ds} \tag{4.4}$$

Taking the gradient of the squared eikonal equation we get:

$$\vec{\bigtriangledown}(\vec{\bigtriangledown} S)^2 = 2\vec{\bigtriangledown} S \cdot \vec{\bigtriangledown}(\vec{\bigtriangledown} S) = 2n\vec{\bigtriangledown} n. \tag{4.5}$$

Based on the definition of the nabla operator and applying the chain rule for several variables we have:

$$\frac{d}{ds}\vec{\bigtriangledown} S = \frac{d\vec{x}}{ds} \cdot \vec{\bigtriangledown}(\vec{\bigtriangledown} S). \tag{4.6}$$

---

[2] in contrast to Born and Wolf we are using $\vec{\bigtriangledown}$ (nabla operator) as the notation for a gradient vector, so $grad(f) = \vec{\bigtriangledown} f$

[3] The derivation follows [BW99]

[4] we define the refractive index n as $n = n(\vec{x})$ and stay in cartesian space.

Combining now Eqs. 4.4, 4.5 and Eq. 4.6 yields:

$$2\vec{\bigtriangledown}S \cdot \vec{\bigtriangledown}(\vec{\bigtriangledown}S) = 2n\frac{d\vec{x}}{ds} \cdot \vec{\bigtriangledown}(\vec{\bigtriangledown}S) = 2n\frac{d}{ds}\vec{\bigtriangledown}S = 2n\frac{d}{ds}(n\frac{d\vec{x}}{ds}) = 2n\vec{\bigtriangledown}n$$

Dividing by $2n$ gives:

$$\frac{d}{ds}(n\frac{d\vec{x}}{ds}) = \vec{\bigtriangledown}n. \tag{4.7}$$

The equation describes the trajectory of a light ray in an inhomogeneous refractive index field $n(\vec{x})$ and is known as the ray equation of geometric optics. A simple substitution of $\frac{d\vec{x}}{ds} = \frac{\vec{w}}{n}$ gives us the following set of first order ordinary differential equations:

$$\frac{d\vec{x}}{ds} = \frac{\vec{w}}{n}, \tag{4.8}$$

$$\frac{d\vec{w}}{ds} = \vec{\bigtriangledown}n. \tag{4.9}$$

ds defines an infinitesimal step in the direction of the ray. Equation 4.8 further provides us with a constant spatial step size parameterization, since $\mid \frac{d\vec{x}}{ds} \mid = 1$. This is advantageous for rendering, where the number of particle trajectories should be approximately equal to get optimal performance.

4.2 Light and wavefront definitions

The derived Eq. 4.7, based on geometrical optics from Sect. 4.1.2, gives us a relatively simple way to simulate a light ray propagating through an inhomogeneous refractive index field. But for a complete simulation we need to model the light source and its wavefront as well.

4.2.1 Light source

A light source emits photons which propagate through a medium, according to the laws of physics, until they reach our eye. The same idea is used by ray tracing (Sect. 2.1.1) and photon mapping (Sect. 2.1.2) to simulate the light.

In contrast, we define the light source as an emitter of a wavefront. The light source is associated with a vector field describing the *local light direction* $v(\vec{x}) = \vec{v}$ and a 3D scalar field of *differential irradiance values* $\Delta E_\omega(\vec{x})$. The local light direction can be seen as the travelling direction of a single photon. The differential irradiance value,

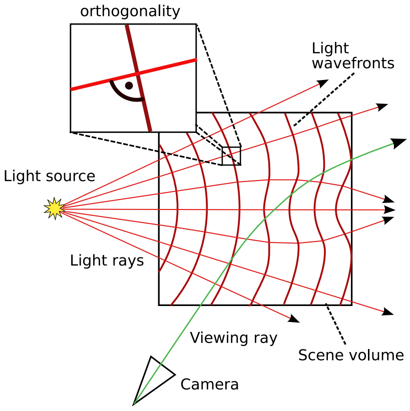


Figure 4.2: 2D illustration of our complex image formation scenario – due to inhomogeneous material distribution, light rays and viewing rays are bent on their way through the *scene volume* V. Light rays always travels orthogonally to the light wavefronts. Light wavefronts are the iso-surfaces of constant travel time from the light source.

sometimes also called *intensity*, describes the energy stored in the wavefront.

4.2.2 Wavefront representation

We see the wavefront as an iso-surface of points, in our case photons, having the same phase or constant travel time, originating from a light source. The simplest form of a wavefront is a *plane wave*. The corresponding rays are parallel and their direction is perpendicular to the wave. We use such a wavefront to model a directional light source. A *spherical wavefront*, described by a sphere of radius R, defines a point light source[5]. Later, we use these representations to initialize a wavefront before it propagates through the scene volume.

We define the wavefront as an adaptive set of inter-connected particles (patches) propagating independently through the scene volume. The connectivity information is needed for differential irradiance computation (Sect. 4.2.4) and error bounding. The wavefront is discretized into so-called wavefront patches. The local light direction $\vec{v}$ is represented by the travelling direction of a particle and the differential irradiance value $\Delta E_\omega(\vec{x})$ can be computed from the area of the wavefront patch. Fig. 4.2 illustrates an example constellation of a point light source and its wavefronts in a scene volume.

---

[5] since the radius of a point is equal 0, we set the initial radius of a spherical wavefront to $R = \varepsilon$, so that it covers the scene volume

4.2.3 Wavefront propagation

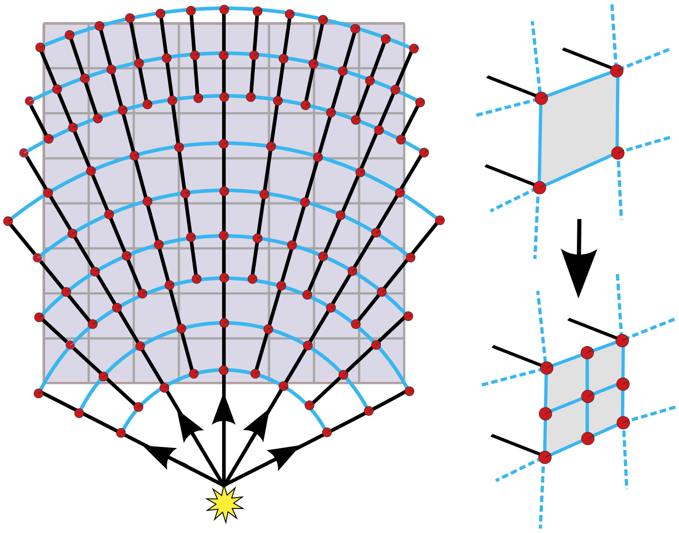
In the previous section we have defined a wavefront as an adaptive set of connected points, or patches. The corners of a patch are given by the four neighboring particles. These are propagated through the inhomogeneous refractive index field. In case the wavefront becomes under-resolved, new particles/patches are generated by subdividing the corresponding patch to preserve a minimum sampling rate. We also define a wavefront patch as a container of some finite differential irradiance $\Delta E_{\omega}(\vec{x})$. Due to *attenuation*, defined as a scalar field σ_a, the irradiance can be consumed by the corresponding scene voxel. In real life the same effect can be seen in colored glass. Some *wavelengths* are attenuated, so that only the colors passing through can reach our eye.

In the end, we obtain an irradiance distribution over a scene volume, produced by a propagating wavefront based on the assumption that the energy of a patch is absorbed by scene points. The computation of such a light distribution is performed with the following pseudo algorithm:

```
- $dt$ = time step
- while in volume
    - propagate wavefront by time $dt$
    - compute irradiance of wavefront patches
    - voxelize data
    - refine wavefront
```

In case a wavefront patch does not touch a voxel with its corner particles, the voxel's refractive index does not influence the propagation of the patch. Hence the wavefront would not be simulated correctly if a patch was larger than one voxel. To alleviate this, we adaptively split a wavefront patch once it grows too large (Fig. 4.3). To be able to voxelize the irradiance distribution we equate the wavefront patches with their midpoints, and store the differential irradiance value and local light direction (patch direction) into the current voxel.

Propagating the patch particles with the derived equations (Sect. 4.1.2) would break the wavefront, since its particles would go out of phase. This means that particles describing a wavefront patch and propagating through a medium while taking constant spatial steps destroy the equitemporal nature of the wavefront patch. This phenomenon comes from different propagation ”speeds” (or in other words; different temporal steps



(a) (b)

Figure 4.3: Adaptive wavefront refinement. (a) 2D illustration: the wavefront is represented by particles (red dots) that are connected to form a wavefront (blue lines). While advancing through the voxel volume (shown in gray) the wavefront is tessellated such that its patches span less than a voxel. (b) 3D illustration of the tessellation for one wavefront patch.

for the same distance) of particles inside a medium. The curvature of a wavefront patch would break and lead to unwanted results (e.g. the patch area is wrong). A solution for this issue would be a scheme for wavefront curvature tracking as shown in [MH92].

Another solution is to derive an equation providing constant temporal steps for the ray equation. This means, that we would like to have an equation giving us the possibility to specify a temporal step dt instead of a spatial step ds to propagate the wavefront particles through the scene volume. We are thus looking for a parameterization where:

$$\frac{dS}{dt} = \vec{\bigtriangledown} S \cdot \frac{d\vec{x}}{dt} = 1, \tag{4.10}$$

so that infinitesimal changes of the eikonal function S with respect to parameter t are constant.

Inserting Eq. 4.4 into Eq. 4.10 and applying the chain rule $\frac{df}{dy} = \frac{df}{dg}\frac{dg}{dy}$ yields

$$\begin{aligned} n\frac{d\vec{x}}{ds} \cdot \frac{d\vec{x}}{dt} &= 1 \\ \Leftrightarrow \frac{d\vec{x}}{ds} \cdot \frac{d\vec{x}}{ds}\frac{ds}{dt} &= \frac{1}{n} \end{aligned}$$

We already know from the eikonal equation that $\frac{d\vec{x}}{ds} = \frac{\vec{\bigtriangledown}S}{|\vec{\bigtriangledown}S|}$. Therefore we can follow from the equation above that:

$$\begin{aligned} \frac{d\vec{x}}{ds} \cdot \frac{d\vec{x}}{ds}\frac{ds}{dt} &= \frac{1}{n} \\ \Leftrightarrow \frac{\vec{\bigtriangledown}S}{\mid \vec{\bigtriangledown}S \mid} \cdot \frac{\vec{\bigtriangledown}S}{\mid \vec{\bigtriangledown}S \mid}\frac{ds}{dt} &= \frac{1}{n} \\ \Leftrightarrow \frac{ds}{dt} &= \frac{1}{n} \end{aligned} \tag{4.11}$$

This is a nice result, because now we can re-parameterize Eqs. 4.8 and 4.9 to use the constant temporal instead of constant spatial step size. Applying the chain rule and substituting the results above yields

$$\frac{d\vec{x}}{dt} = \frac{d\vec{x}}{ds}\frac{ds}{dt} = \frac{\vec{w}}{n} \cdot \frac{1}{n} = \frac{\vec{w}}{n^2} \tag{4.12}$$

and

$$\frac{d\vec{w}}{dt} = \frac{d\vec{w}}{ds}\frac{ds}{dt} = \vec{\bigtriangledown}n \cdot \frac{1}{n} = \frac{\vec{\bigtriangledown}n}{n} \tag{4.13}$$

Combining both equations, by solving for $\vec{w}$, we get the ray trajectory equation for a constant temporal step size,

$$n\frac{d}{dt}(n^2\frac{d\vec{x}}{dt}) = \vec{\bigtriangledown}n \tag{4.14}$$

This formulation gives us the possibility of a fast GPU implementation for the wavefront propagation by using a modified particle system. Once the wavefront can be tracked over time, we can compute differential irradiance values at every point in space from the area of the connected particles.

4.2.4 Irradiance computation

Irradiance is a radiometry term for the incident power of electromagnetic radiation at a surface, per unit area. If a point source radiates light uniformly in all directions and there is no absorption, then its irradiance drops off in proportion to the distance from the object squared, since the total emitted power is constant and it is spread over an area that increases with the square of the distance from the source [DBB06].

In other words, irradiance describes energy per surface area for a given incident radiation. Our light source's irradiance is stored in a wavefront. Therefore, each patch in the wavefront is initialized with some initial irradiance $E_{\omega 0}$, depending on the light

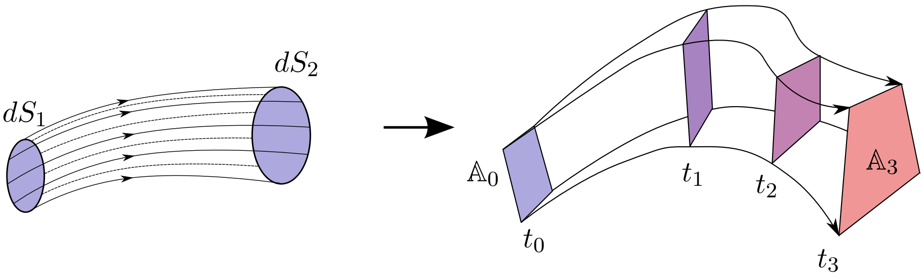


Figure 4.4: The intensity law of geometric optics (left) and its discretized version (right) in the form of a *stream tube*. The product of area and differential irradiance is constant along a tube of rays.

source. By propagating through space, the irradiance value of a patch can grow larger or smaller, depending on the patch area. This is also known as *intensity law of geometric optics* [BW99], see Fig. 4.4. The law states that the energy in a ray tube is always constant:

$$dE_{\omega 1} dS_1 = dE_{\omega 2} dS_2 \tag{4.15}$$

We can thus see, that by increasing the area dS_2, the corresponding energy $dE_{\omega 2}$ has to be decreased. Therefore, the energy is indirectly proportional to the area of a wavefront patch[6]. With this information and Eq. 4.15, we can deduct a discretized version:

$$\Delta E_\omega(t)\mathbb{A}(t) = \Delta E_\omega(0)\mathbb{A}(0) \Leftrightarrow \Delta E_\omega(t) = \frac{\Delta E_\omega(0)\mathbb{A}(0)}{\mathbb{A}(t)} \tag{4.16}$$

where $\Delta E_\omega(0)$ denotes the initial irradiance $E_{\omega 0}$ and $\mathbb{A}(0)$ the initial surface area of a wavefront patch. As already mentioned, these values are set once during initialization and depend on the properties of the light source and dimensions of the wavefront.

The wavefront patch is spanned by four particles representing its corners (Sect. 4.2.2). Due to the connectivity information on the particles, we can compute for every time t the surface area of a wavefront patch $\mathbb{A}(t)$. Hence we can also compute the discretized differential irradiance value $\Delta E_\omega(t)$ associated with a wavefront patch at time t.

4.3 Conclusion

In this chapter we have shown some basics of geometric optics. We have described a way of using Snell's law to simulate light transport through a refractive index field.

---

[6] area of a wavefront patch in Fig. 4.4 is $\mathbb{A}_i$

Afterwards, we introduced the concept of light wavefronts. With the help of the eikonal equation $| \vec{\bigtriangledown} S \ |= n$ and Fermat's principle we were able to derive the ray equation of geometric optics, Eq. 4.7, for a massless particle travelling through an inhomogeneous refractive index field. We use this principle to compute viewing rays, implementing a ray casting approach.

We defined the light source as an emitter of a light wavefront. The wavefront is an adaptive set of patches, with corners represented by particles. The particles are then propagated through the scene volume to compute the irradiance distribution. For updating the wavefront patch positions and sizes we have used Eqs. 4.12 and 4.13. This was done to keep the temporal step size constant. Finally, we have shown a simple way, based on the intensity law of geometric optics, of computing the differential irradiance values of the wavefront patches.

The equations derived in this chapter allow for an efficient GPU implementation, which, combined with a proper image formation model, enables a realistic rendering of refractive objects in real-time.

CHAPTER 5

IMAGE FORMATION MODEL

In the previous chapter we have introduced the basics of geometric optics and presented how a light wavefront can be propagated efficiently through the scene volume. In this chapter we will introduce the image formation model used for the radiance computation of the *viewing rays*. We derive an image formation equation for our approach, and present a simplified form ready to be implemented on the GPU.

Another issue will be to show which effects can be simulated with the stated model. We will also present some results, giving an impression of how different effects, included one by one in the image formation model, contribute to the realistic look of the scene. Finally a pseudo code version of our implementation will be presented.

5.1 General image formation

5.1.1 General volume rendering equation

We have previously defined our scene volume as a three dimensional refractive index field. To compute a viewing ray's radiance, we require a proper formulation. A very common approach is to use a *volume rendering equation*, which describes how the radiance of a light ray changes as it travels through a volume of data.

Virtually all volume rendering algorithms try to find a good approximation for the low albedo volume rendering integral presented by [Bli82] and [KvH84]. Most of the volume rendering algorithms approximate the following integral:

$$L(\tau) = \int_0^{\tau} L_c(t) e^{-\int_0^t \sigma(s) ds} dt, \tag{5.1}$$

where $L_c(t)$ is a radiance function including emission, scattering and reflection. $\sigma(s)$ is the opacity function. In our case $\sigma(s) = \sigma_a(c(s))$ and represents the attenuation function (Sect. 5.1.2). We will expand the above definition to a suitable form, which can be easily discretized and computed on the GPU.

5.1.2 Attenuation and the background

The rendering equation uses an opacity function, as we already have seen in the previous section. Comparing the inner integral of Eq. 5.1 and the attenuation integral, Eq. 3.4,

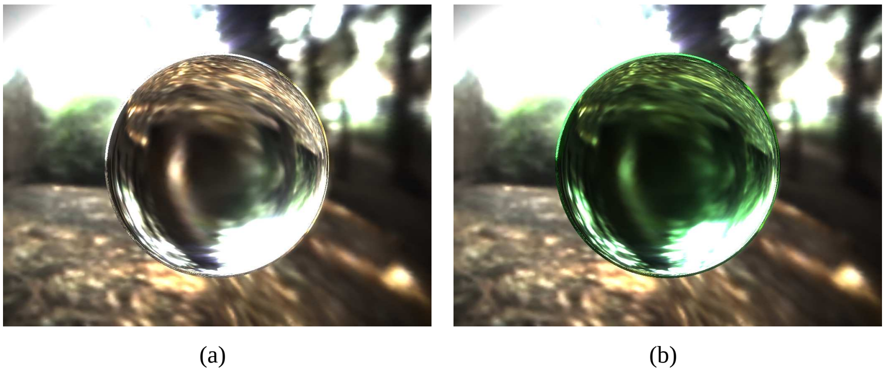

(a) (b)

Figure 5.1: Glass sphere (128^3 voxels) refracting the viewing rays. (a) Refraction only (b) Refraction combined with attenuation; red and blue light components are attenuated, producing green colored glass.

presented in Sect. 3.2.3, we can already see that they are identical. Indeed, our attenuation definition corresponds completely to the opacity function presented in the volume rendering equation.

Hence we expand Eq. 5.1 to the following form

$$L(c) = \int_c L_c(\vec{x}, \vec{v})\alpha(t, c)dt, \tag{5.2}$$

where $L(c)$ denotes the resulting radiance due to the whole optical path length of the ray c. The position in space $\vec{x}$ and the local light direction $\vec{v}$ are defined as $\vec{x} = c(t)$ and $\vec{v} = \frac{dc}{dt}$. The local light directions are precomputed previously in the light simulation step as described in Sect. 4.2.3.

Equation 5.2 does not handle the scene background. Thus the proper rendering of a volumetric object in an environment is not possible. We therefore add a *background term* L_{bg} to the final equation, implicitly assuming the background is at infinite distance. We also have to apply the attenuation factor to the background term for correct approximation[1].

$$L(c) = \int_c L_c(\vec{x}, \vec{v})\alpha(t, c)dt + L_{bg}\alpha(t_\infty, c) \tag{5.3}$$

Defining the background at infinite distance has some disadvantages for rendering nearby geometry seen through the refractive object. This is a current restriction, but our algorithm could easily be extended to support non-volumetric objects at finite distances around the scene volume. A possible implementation of such an approach is described

---

[1] imagine the background as the last, infinitesimally thin layer in the volumetric data.

by Wyman [Wym05b]. A simple implementation of nearby geometry was used to create ground truth comparisons to a ray tracer (Chapter 7).

5.1.3 Reflection, Scattering and Emission

In the previous section we have presented an equation for our image formation. As already shown in the general volume rendering equation, Eq. 5.1, the term L_c is used to compute contributions of emission, scattering and reflection to the light ray. We express L_c in terms of these variables as:

$$L_c(\vec{x},\vec{v}) = \hat{\omega} L_s(\vec{x},\vec{v}) + \delta(\vec{x}) R L_r(\vec{x},\vec{v}) + L_e(\vec{x}) \ . \tag{5.4}$$

L_s denotes the radiance due to inscatter, and $\hat{\omega}$ is the scatter strength, also known as *albedo*. Albedo is the ratio between scattered and incident radiance. Normally, the albedo depends on the local ray direction and directional distribution of incoming radiation. In our case, we set it to $\hat{\omega} = \frac{\sigma_s}{\sigma_s+\sigma_a}$, where σ_s is scattering and σ_a the absorption coefficient.

We formulate L_s in terms of the scattering phase function p (Eq. 3.5) presented in Sect. 3.2.4. The light contribution due to inscatter is integrated over the sphere of all incoming light directions. It is spatially varying and depends on the local ray direction $\vec{v}$ and the differential irradiance dE_ω from the direction ω.

$$L_s(\vec{x},\vec{v}) = \int_\Omega p(\vec{x},\vec{v},\vec{\omega}) dE_\omega \tag{5.5}$$

Similarly to L_s, we define the reflected radiance L_r as an integral over all directions,

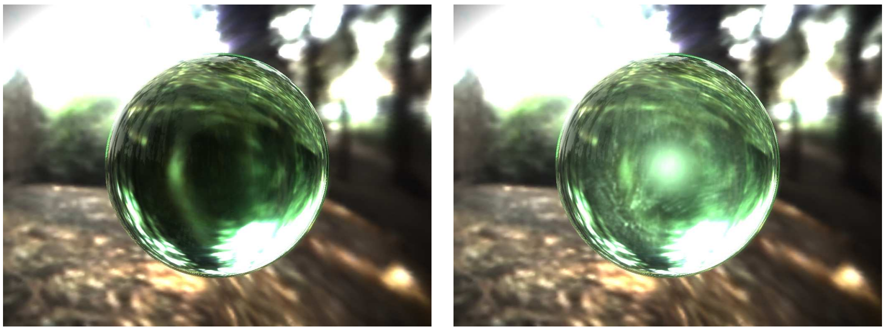
$$L_r(\vec{x},\vec{v}) = \int_{\Omega_+} f_r(\vec{x},\vec{v},\vec{\omega}) \cos\theta dE_\omega, \tag{5.6}$$

where θ is the angle between the surface normal $\hat{n}$ and the incident light direction ω. The surface normal can either be provided as an additional function or computed directly from the gradient of the refractive index field, $\hat{n} = \frac{\vec{\bigtriangledown} n}{|\vec{\bigtriangledown} n|}$ (Sect. 3.1).

The reflected radiance term is triggered by the Dirac delta function $\delta(\vec{x})$ in Eq. 5.4 which serves as a boundary indicator, i.e. it integrates to one over a boundary[2] and is zero elsewhere. This way, we compute the reflection only on boundaries between different materials. As already described in Sect. 3.2.5 the Fresnel factor R determines how much of the reflected radiance contributes to the light ray. The transmission factor T of the

---

[2] The boundary indicator could also be computed from the refractive index gradient, since its length is unequal 0 only on the interface between different refractive indices/media (see Code Listing C).



(a) (b)

Figure 5.2: (a) L_r becomes active on the material boundary between glass and air, and is responsible for reflections of the environment (Fresnel reflections) (b) Additionally, L_s and L_e have become active. Scattering gives a feeling of impurities inside the glass. A small blob in the center of the glass, emitting radiance, gives an impression of phosphorescence.

Fresnel equation enters into Eq. 3.4 and describes the amount of radiance transmitted from the refracted ray to the observer. We compute the global transmission factor $T(t)$ as $T(t) = T_1 \cdot T_2 \cdot ... \cdot T_n$, where T_i is the transmission factor of the Fresnel equation (Sect. 3.2.5) for the refraction event i.

Finally, the function $L_e(\vec{x})$ describes the amount of radiance emitted at the position $\vec{x}$. The function can be used to model multiple scattering effects or self-emission due to fluorescence or phosphorescence. It can either be defined through an implicit function or as one of the properties of the scene volume. It can not affect the global light computation, though.

5.1.4 Discretization of rendering terms

L_s, L_r and L_e depend only on the 3D position $\vec{x}$ in space and the local light direction $\vec{v}$. Therefore their values can be evaluated locally, given volumetric descriptions of their distributions. This locality of L_c is important for the efficient parallelization of the computations on the GPU.

The rendering equation in its general form, as shown in Eq. 5.4, is too complex to be evaluated in real-time. Therefore we have to simplify it with the following assumptions:

- there are only a discrete number of light sources in the scene, and
- for each point in the scene, there are only a discrete number of incoming light rays from each of the light sources

These restrictions allow us to discretize Eqs. 5.5 and 5.6 into discrete sums over all incoming light directions,

$$L_s(\vec{x}, \vec{v}) = \sum_j p(\vec{x}, \vec{v}, \vec{l_j}) \Delta E_{\omega_j} \tag{5.7}$$

and

$$L_r(\vec{x}, \vec{v}) = \sum_j f_r(\vec{x}, \vec{v}, \vec{l_j}) \cos\theta \Delta E_{\omega_j} \tag{5.8}$$

With the help of the algorithm's numerical approximations presented in the previous chapter, we are able to evaluate the above equations and thus Eq. 5.4 with nothing more than the available local values. This leads to an efficient GPU implementation.

Discretization of attenuation factor $\alpha(t, c)$

The attenuation factor $\alpha'(i, c)$ stands for a discrete version of $\alpha(t, c)$, as shown in Sect. 3.2.3, and is defined analogous as

$$\alpha'(t, c) = L_0 \cdot e^{-\sum_{k=0}^{t} \sigma_a(c(k))\Delta s}.$$

The equation can be rewritten as

$$\alpha'(t, c) = L_0 \cdot e^{-\hat{\alpha}_i}, \tag{5.9}$$

where $\hat{\alpha}$ is defined iteratively as

$$\hat{\alpha}_{i+1} = \hat{\alpha}_i + \sigma_a(\vec{x})\Delta s.$$

Here $\vec{x}$ is a spatial point on the ray curve c. This way, the attenuation factor can be updated along with the position and direction of the light ray, Eqs. 5.11 and 5.12

Having discretized the terms, we can now derive a discrete image formation model to compute the radiance of a viewing ray propagated through the refractive object.

5.2 Discrete image formation model

We sample the light trajectory with a constant spatial step size. This is one of the properties of the ray equations derived from the eikonal equation. Due to the nature of the equations of geometric optics, the simulation also supports ray bending and total reflection. We would like to discretize Eq. 5.3, since this yields one general approach for the realistic rendering of refractive objects.

In the previous section, we converted the integrals into sums by making simplifying assumptions about the light sources. We have also derived discrete versions of L_s (Eq. 5.7), L_r (Eq. 5.8) and α (Eq. 5.9). Now, we combine them into L_c and thus obtain a discrete version of the rendering equation, Eq. 5.3, which has to be evaluated during the rendering process for each viewing ray.

We discretize Eq. 5.3 into a discrete sum:

$$L(c) = \sum_c L_c(\vec{x}, \vec{v}) \alpha'(t, c) \Delta s + L_{bg} \alpha'(t_\infty, c), \tag{5.10}$$

using the discrete versions of the attenuation function $\alpha(t, c)$ and $L_c(\vec{x}, \vec{v})$, as show in Sect. 5.1.4.

The trajectory of the light is described by the first order differential equations, Eqs. 4.8 and 4.9, presented in Sect. 4.1.2. Using a simple Euler forward scheme we can discretize the equations:

$$\vec{x}_{i+1} = \vec{x}_i + \frac{\Delta s}{n} \vec{w}_i \tag{5.11}$$

$$\vec{w}_{i+1} = \vec{w}_i + \Delta s \vec{\bigtriangledown} n. \tag{5.12}$$

If necessary, also higher order integration methods could be used (e.g. the Runge-Kutta family [PTVF92]) to discretize the equations.

The algorithm for computing the resulting radiance will therefore look as follows:

```
- while ray is in the volume do
    - acquire all spatial data
    - compute $L_r$, $L_s$, $L_e$
    - $\hat{\alpha}_{i+1} = \hat{\alpha}_i + \sigma_a(\vec{x}_i) \Delta s$
    - $\alpha'(t, c) = L_0 \cdot e^{-\hat{\alpha}_i}$
    - compute $L_c$
    - $L_{i+1}(c) = L_i(c) + L_c(\vec{x}, \vec{v}) \alpha'(t, c) \Delta s$
    - compute $\vec{x}_{i+1}$ and $\vec{w}_{i+1}$
- add the attenuated background term $L_{bg} \alpha'(t_\infty, c)$
```

This pseudo algorithm is the basis of the view renderer in the next chapter. The view renderer evaluates the rendering equation, Eq. 5.3, for each cast viewing ray. A more detailed description of the implementation is presented in the next chapter.

5.3 Conclusion

In this chapter, we have shown a general, physically motivated image formation model, which allows an efficient implementation on the GPU. The computation is based on a volumetric representation of the scene data. We have demonstrated simplifications for the Eqs. 5.5 and 5.6 to enable an efficient implementation of the main rendering equation.

For the realistic representation of refractive objects, we had to incorporate various effects such as Fresnel reflection, surface BRDFs and scattering phase functions. Example images demonstrate why the implementation of these effects is so important for realistic image synthesis. Finally, we have combined the image formation model into one discrete equation (Eq. 5.10).

In the next chapter we present an implementation of our algorithm. We use the equations from the current and the previous chapter to state an efficient and powerful implementation. The light simulation and the view rendering is performed completely on the GPU.

CHAPTER 6

IMPLEMENTATION

We have already shown the underlying mathematics of our rendering approach. In this chapter we present details, describing the implementation of the light simulator and the view renderer. We complement this with pseudo code to simplify the understanding.

The first part of this chapter implements the theory of Chapter 4. The light simulator computes the movement of the wavefront through the scene volume. The output data is comprised of differential irradiance values and local light directions of the light rays for each point in the scene.

The second part explains the inner working of the view renderer. The view renderer implements the theory of the image formation model and the optic effects presented in Chapters 5 and 3. It demonstrates how the rendering equations, derived for the viewing rays, can be implemented with little hassle. Some pseudo code, combined with a shader implementation, shows a practical implementation of the image formation equation.

6.1 Input Data

For proper rendering of any refractive objects, we have to agree on input data formats. Since we evaluate the image formation equations with spatially varying functions, large scale input data have to be provided to the algorithm.

Our volumetric scene V is stored as a set of 3D volume textures. To render a complete scene, including all input data, we require up to eight RGBA volume textures.

T1 Refractive index and gradient field

T2 Attenuation scalar field (R,G,B)

T3 Differential irradiance values (Illumination data (R,G,B))

T4 Local light direction of the wavefront

T5 Emission scalar field (R,G,B)

T6 Opaqueness indicators

T7 Reflection information (only for boundaries)

T8 Auxiliary spatially varying data (i.e. anisotropy factor, scatter strength,...)

The light simulator (as described in Sect. 6.2) requires textures T1 and T2 as input. It outputs the differential irradiance values (T3) and local light directions (T4). We do not have to recompute them as long as the properties of the light source, on which the result depends (i.e. relative light position to the scene volume) do not change. Currently the computation by the light simulator is done within a few seconds[1] depending on sampling rate and complexity of the scene.

In texture T1, we store a spatially varying refractive index field and the corresponding gradients. The refractive indices are pre-smoothed before the gradients are computed, as already described in Sect. 3.1. To render a dispersion effect (see Sect. 3.2.7), we can either use a multi-pass approach or an array of refractive index textures. This poses an advantage for simulation speed but requires considerably more video memory.

The spatially varying attenuation data texture T2 stores the color-specific attenuation factor σ_a for each voxel. Thus, the radiance of the light ray is attenuated continuously along its path through the scene volume. Emission data T5 contributes to the light ray by an additive radiance $L_e(\vec{x})$ to each of the RGB channels.

The generality of our algorithm allows us to render opaque objects, by using the opaque indicators T6 as input, which trigger opaque behavior through the $\delta(\vec{x})$ operator; used in Eq. 5.4. The coefficient has no direct correspondence in the image formation model, as it is only a technical implementation issue, providing us with the possibility to stop the viewing ray propagation through the scene based on specific conditions. The color of the corresponding ray is thus computed by the Fresnel reflection on a surface and the opaque color of the boundary point. One use for the opaque term is to render opaque objects inside a refractive material (e.g. amber with a trapped insect inside).

The reflection texture T7 is used to define special material properties for the reflection on a boundary. Combined with a proper boundary indicator it can be used to simulate a BRDF and thus to compute the $L_r(\vec{x}, \vec{v})$ term. The boundary indicator is a discrete version of Dirac's delta $\delta(\vec{x})$. We compute it by voxelizing the mesh or use the gradient strength of the refractive index gradients.

Finally the auxiliary data T8 contains supplementary, spatially varying information (e.g. scattering strength $\hat{\omega}$ as in Eq. 3.5, anisotropy factor g from Eq. 5.4). For scattering simulation, we are using the Henyey-Greenstein phase function as described in Sect. 3.2.4.

We are using 16 bit floating point 3D volume textures to store scene informations. To render a refractive object of 128^3 voxels, we require up to $8 \cdot 128^3 \cdot 4 \cdot 2 = 134$MB of video memory for an object if all eight input textures are used. This amount of memory can be

---

[1] currently about 8-12 seconds per frame (see Chapter 7)

reduced for particular objects when some data is not required for proper rendering. For example, a glass object does not require any opaque information. Possible optimizations for memory storage (e.g. octree textures [BD02]) can be used, but pose a disadvantage for rendering speed.

The volume data is generated either by an implicit function or created by voxelizing a mesh (see e.g. the Stanford bunny example). The advantage of using a mesh to create volumetric data is that we can use it as *proxy geometry* (see Sect. 6.3.1). This reduces the number of viewing rays that need to be propagated through the scene volume.

6.2 Light Simulator

In Chapter 4 we described the theory behind wavefront propagation. The light simulator, presented in [IZT$^+$07] and utilized for our approach, uses this wavefront theory to simulate adaptive wavefront propagation. The background of the implementation can also be found in Chapter 4, section 4.2.2.

A light simulator run can be subdivided into the following steps[2]:

- Initialization
- Wavefront propagation
    - update wavefront patches
    - wavefront voxelization
    - tessellation/undersampling analysis
- Write output

All computations required during the propagation step are performed on the GPU. To do this, the light simulator implements a *particle system* completely on the graphics processor. The difference to a standard particle system is that the particles are packed into a group of four neighboring particles, each representing one corner of a wavefront patch. The light simulator works on a list of patches. All the properties of the patch particles (position, direction and irradiance) are stored in dynamically growing textures.

6.2.1 Initialization

As already mentioned, the wavefront is subdivided into wavefront patches each containing four neighboring particles. The shape of the initialized wavefront depends on the

---

[2] similar definition can be found in Sect. 4.2.3

light source properties. Directional light produces a planar wavefront, where all corresponding light rays are parallel. A point light source, however, has a spherical wavefront. To overcome singularity problems, a small value ε for the initial radius is used.

The initial differential irradiance (see Sect. 4.2.4) for each wavefront patch is computed through the irradiance of the light source. For a directional light source, the patch's initial irradiances are equal. However, for a point light source, the irradiance per patch is based on the intensity law, Eq. 4.16. By assuming the patches are at a certain distance ε away from the light source we avoid a division by zero area.

6.2.2 Wavefront patch propagation

The wavefront is implicitly propagated through its defining patches. The propagation of the wavefront corners is done using Eqs. 4.12 and 4.13 described in the Sect. 4.2.3. We apply the discrete version

$$\vec{x}_{i+1} = \vec{x}_i + \frac{\Delta t}{n^2}\vec{w} \tag{6.1}$$

and

$$\vec{w}_{i+1} = \vec{w}_i + \frac{\Delta t}{n}\vec{\bigtriangledown} n \tag{6.2}$$

of Eq. 4.12 and Eq. 4.13 to each corner particle of a wavefront patch.

During the update step, the light simulator also re-computes the differential irradiance carried by each patch. It is computed by Eq. 4.16, presented in Sect. 4.2.4, and depends on the surface area spanned by the corners of the wavefront patch. Because the light simulator is working on a patch list, all the required computations can be performed locally. This enables an optimal implementation on the GPU.

6.2.3 Voxelization of wavefront patches

During the propagation of the wavefront patches, the light simulator protocols their contribution into the 3D volume, i.e. it stores the differential irradiance values and the local light directions inside a voxel.

Unfortunately, current graphics hardware does not support rendering into 3D volumes (textures). Therefore another voxelization approach is required, i.e. point primitives and the concept of *Flat 3D textures* introduced by Harris et al. [HBSL03]. Flat 3D textures are also called *atlas textures*. They map texture slices, cut along the z-direction, into a 2D texture. To access any voxel stored in a flat volume texture, a mapping $f : T^3 \mapsto T^2$ is applied to the original, three dimensional texel coordinates or vice versa for writing to a Flat 3D texture.

Point primitives are used to perform the voxelization process. The light simulator sends as many point primitives as there are rays to the graphics pipeline. Each point represents the center of one wavefront patch and thus is equivalent to the light ray represented by the propagation direction of a wavefront patch. The results, irradiance value and light direction, are then rendered into the output textures.

Through statistical analysis of the scenes we found, that it is sufficient to only store the highest energy which passes through a voxel. We thus limit our computations to one contribution per voxel. Possible future implementations could store more than one light's contribution into a voxel with multiple simulation passes, yielding a more accurate computation of the current image formation model.

6.2.4 Patch list analysis

After each update step the patch list is analyzed. This analysis is required to prevent undersampling, as outlined in Sect. 4.2.3, and to remove patches which will not visually contribute to the computed results.

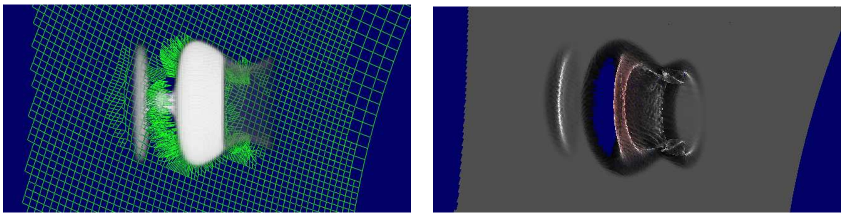
Undersampling is prevented by performing a *divergence tessellation* when a wavefront patch becomes larger than a voxel. To tessellate a patch, it is divided into four smaller ones (see Fig. 4.3). The new patches are then added to the current patch list to advance the simulation.

Some patches do not visually contribute to the results anymore. Thus they are eliminated using an *energy threshold*. Termination usually happens after loss of energy due to attenuation or when a patch has been tessellated too many times. Patches which leave the volume of interest are terminated, since we assume that they can not re-enter the volume.

As mentioned in [BW99], the physical model of ray optics breaks down at wavefront singularities, resulting in an infinite energy result at catastrophic points. Such singularities can produce non-physical caustics. They are detected by examining the patch orientation with respect to its propagation direction. In case, the orientation changes, the corresponding patch is eliminated. Another postulation of the wavefront propagation problem may help, in future work, to prevent these singularities.

The termination repeats until no patches remain. Figure 6.1 shows a wavefront propagating through a wine glass. The irradiance values, computed during the update step, are visualized as colors previewing the beautiful caustics in and around the object.

During patch list termination or tessellation, a data compaction or expansion on the



(a) (b)

Figure 6.1: (a) The refractive index volume of the glass is approached by a spherical wavefront from the right. The adaptive tessellation of the wavefront is also visible. (b) When it passes through the object, beautiful caustic patterns appear in its irradiance distribution.

GPU is performed. This is a non-trivial task, since a GPU is not designed to handle lists[3]. To solve this problem on Shader Model 3.0 hardware, the data compaction algorithm presented by Ziegler [ZTTS06] is utilized. It works on a mipmap-like datastructure to construct a list of retained data entries (in our case: patches) without involving the CPU. An extension of the algorithm is used to handle data expansion (in our case, patch tessellation) [IZT$^+$07].

6.3 View renderer

Our view renderer implementation follows the rules defined in the image formation section in the previous chapter. Having the output of the light simulator available we can render arbitrary views of the scene including complex refractive objects. The view renderer uses the equations derived in Sect. 4.1.2 to propagate viewing rays through the scene volume.

We are using a fast *volume raycasting* approach based on Eqs. 4.8 and 4.9, and their discrete versions, Eqs. 5.11 and 5.12. As already mentioned, the equations support advanced refractive effects, e.g. total reflection, ray bending, without performing explicit ray-surface intersections. The radiance of the viewing ray is computed using the discretized image formation model, Sect. 5.2.

Volume raycasting is an image-based volume rendering technique. The main goal is to render 2D images from 3D data sets. Volume raycasting produces results of very high quality. Combined with the derived rendering equation it is fast enough to render refractive objects in real-time. The basic algorithm for volume raycasting as used in our approach, has the following steps:

---

[3] GPUs with Shader Model 4.0 and above can perform list handling using geometry shaders, but would require a considerable, non-backward-compatible redesign of our implementation.

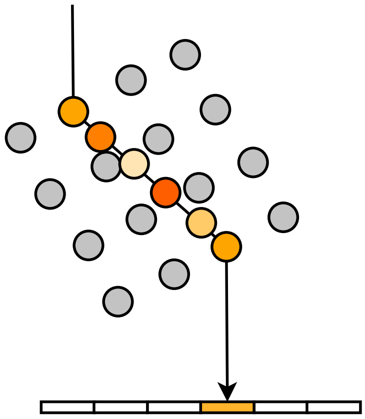


Figure 6.2: Basic volume ray casting of a refractive object. For each pixel of the image plane, a viewing ray is propagated through the volume. The samples are interpolated and composed to produce the resulting pixel value.

- **Ray casting**: For each pixel in the resulting image, a viewing ray is cast through the volume data. It is common to use a bounding geometry for the volume to define if and where viewing rays are issued.
- **Sampling**: The volume data is sampled on equidistant sampling points along the viewing ray trajectory. Since the volume is not aligned with the light trajectory, samples have to be interpolated (i.e. by trilinear interpolation[4] of the surrounding voxels) to acquire correct, approximated values for samples lying inbetween.
- **Local computation**: The samples are used to compute all local data based on the rendering equation.
- **Compositing**: All the samples are composed into a resulting pixel color. The composition is derived directly from the rendering equation.

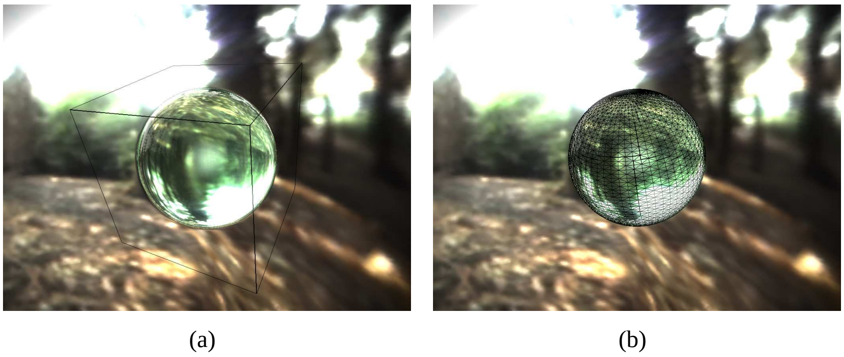
In our case the last three steps are looped while the viewing ray remains in the volume.

6.3.1 Ray casting

For each pixel of the final image, we cast viewing rays through the scene volume. This is done by using a bounding geometry, aka *proxy geometry*. In the simplest case, it is a *cube*.

---

[4] for 3D textures interpolation is performed in hardware, for flat 3D textures (2D atlas texture) we perform the interpolation in software



(a) (b)

Figure 6.3: Refractive object and its proxy geometry. (a) Cube/Box as a bounding primitive. (b) Use of a sphere mesh as bounding primitive.

To optimize rendering, we are using a mesh as a proxy geometry for particular objects (e.g. the sphere). It follows that only the viewing rays, which are propagated through the inner of the surrounded medium, contribute to the rendering results. As can be seen in Fig. 6.3, there is a lot of free space around the sphere which does not contribute to the results at all.

Our view render is able to provide a free view of the refractive object. The rendered object can be seamlessly included into a mesh-based environment. Using a proxy geometry, we apply an object matrix M describing the rotation, translation and uniform scaling of the refractive object. Based on proper matrix manipulation we can perform a coordinate space ”switching” for viewing rays propagated through the scene volume, i.e. we transform the current sampling position into the local object space of the scene volume. This is done by multiplying the ray position $\vec{x}$ with the inverse object matrix M^{-1}. The computed point $\vec{x}' = M^{-1} \cdot \vec{x}$ is then used to sample the volume data.

We are using Shader Model 3.0 hardware to render the refractive objects. A fragment shader is then applied to the proxy geometry, which implements the pseudo code for evaluating the image formation presented in Sect. 5.2. Taking into account correct transformation of the gradient and position vectors, the pseudo shader code to render the refractive objects in real-time looks like this:

```
$\vec{x}_0$ = fragment position in world space
$\vec{w}_0 = \textbf{normalize}(\vec{x}_0 - eyePosition)$
while in volume do
    $\vec{x}' = M^{-1} \cdot \vec{x}_i$
    sample volume data at $\vec{x}'$
    $\vec{\bigtriangledown} n' = (M^{-1})^T \cdot \vec{\bigtriangledown} n$
    compute $L_c$, $\alpha'$
    $L_{i+1}(c) = L_i(c) + L_c(\vec{x}, \vec{v})\alpha'(t, c)$
    $\vec{x}_{i+1} = \vec{x}_i + \frac{\Delta s}{n}\vec{w}_i$
    $\vec{w}_{i+1} = \vec{w}_i + \Delta s \vec{\bigtriangledown} n'$
    $i = i + 1$
$L_{i+1}(c) = L_i(c) + L_{bg}\alpha'(t_\infty, c)T_{fresnel}$
```

After the shader has finished volume traversal we use the viewing ray direction to sample the background term L_{bg}. The sampling is performed with a simple lookup into a dynamic environment map. The term is multiplied with the combined transmission term $T_{fresnel}$ of the fresnel equations and the attenuation factor. The transmission term is needed since only a part of the initial light energy is completely propagated through the volume. This is due to Fresnel reflections on the boundaries as described in Sect. 3.2.5 and 5.1.3.

The algorithm presented in the previous chapters and the powerful image formation model, combined with the view renderer implementation, can handle arbitrary BRDF models. However for simple glass objects, which are close to being perfect reflectors, a good approximation of the first reflection is visually sufficient. Therefore, we use simple environment lookups to simulate reflections on the object boundaries. The reflected rays are not cast back through the volume.

6.4 Conclusion

This chapter presented a sketch of our implementation. Fig. 6.4 illustrates the workflow of our rendering approach. In the first section, we have shown the input data, which consists of three dimensional volume textures storing all the information we need for a proper light simulation and rendering process.

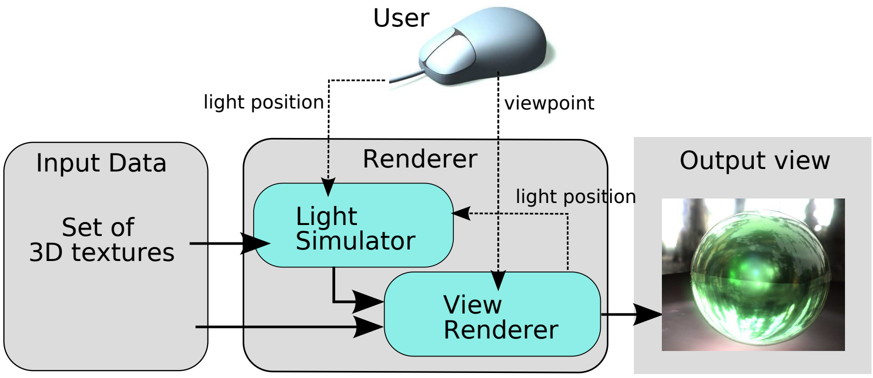


Figure 6.4: Work-flow of our rendering system.

In the second section, we presented the light simulator. The wavefront patch list in the simulator is administrated completely on the GPU without utilizing the geometry shaders, allowing the use of Shader Model 3.0 hardware. The simulation time, required to propagate the wavefront through a scene, depends on the volume resolution and refractive object complexity.

In the last section we have presented a pseudo implementation of our view renderer. The renderer casts viewing rays (rays of sight) through the scene volume and approximately evaluates the image formation model, Eq. 5.3.

All the color computations in the view renderer are performed in high dynamic range, i.e. there is no clamping performed and floating point is used. The results are then mapped to the presentable display range by a tone-mapping approach as in [KMS05]. The results and the statistics about the rendering speed are presented in the next chapter. A full shader code implementation of the pseudo code rendering algorithm is included in Appendix C.

CHAPTER 7

RESULTS

The previous chapters described the complete rendering pipeline for our algorithm. We have shown how to simulate different optical effects and presented a general and efficient image formation model, Sect. 5, for the realistic rendering of refractive objects using a volumetric object representation.

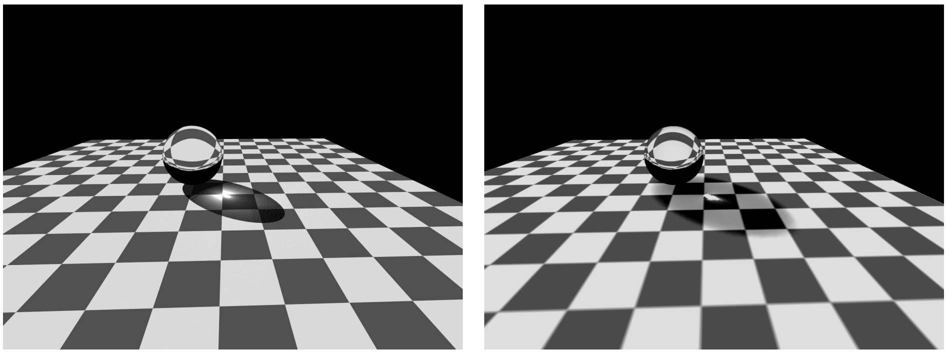
In this chapter, we present results achievable by our approach. Depending on the scene complexity and the count of rendered refractive objects, we obtain suitable rendering performance, i.e. at least at interactive frame rates of about 20-25 FPS.

Unfortunately, we are not always capable of providing a comparison with ground truth results. Currently, we are not handling nearby geometry correctly, producing refraction artifacts for geometry close to the refractive object. In the future, these cases could be handled by combining any suitable approach which enables us to compute a correct environment look-up with respect to near-by geometry. For the purpose of providing comparison results to ground truth, we replaced the environment map lookup by a ray-plane intersection in the fragment shader (see Fig. 7.1). More complex nearby geometry can be rendered accurately using the approach of Hu and Qin [HQ07]. Note that due to the volumetric discretization and refractive index field smoothing prior to gradient computation, we are not able to produce results exactly matching a non-volumetric rendering approach.

Our renderer is capable of computing Fresnel effects and anisotropic scattering phase functions on-the-fly. With the help of spatially varying, as well as color channel-dependent attenuation, beautifully colored objects can be reproduced. Emission and dispersion effects can be simulated by providing one separate refractive index field for each of the color components.

To render our results, we utilized nVidia's GeForce 8800 GTX with 768 MB of video memory. Our scene data is stored in 128^3 volumes, with up to eight volumetric textures, as mentioned in Sect. 6.1. As a CPU we are using a Dual Core AMD Opteron with 2.6 GHz. However, this is mostly irrelevant as most computational work is performed completely on the GPU.

Rendering performance depends on the complexity of the rendered refractive objects and on the view resolution, since this affects the number of rays being cast from the viewpoint through the scene volume. We obtain a suitable real-time performance of 25 frames per second for almost all of our objects rendered at 800x600 resolution. For the



(a) (b)

Figure 7.1: Comparison between a ray-traced image rendered with the Persistence of Vision raytracer (POV-Ray) (a) and our algorithm (b). The differences in the refraction and shadow size as well as the slightly displaced caustic pattern are due to smoothing of the refractive index field.

light simulation step, which is required for each relative movement between light source and refractive object, the algorithm requires 8-12 seconds to re-compute the light distribution inside the volume of interest. For the wine-glass scene, containing an animated light source, we spent around 90 minutes to simulate the light distribution for around 600 frames.

7.1 Objects

For the rendered result sequences we are using up to five different refractive objects in several environments. The objects visualize different combinations of the optical effects that were presented in the previous chapters. A museum scene contains five different refractive objects, all of them rendered simultaneously. Figure 7.2 shows a glass block and a wine glass presented in this environment. The environment maps for the background term (see Chapter 5) are stored in a cube map texture. They are either created on-the-fly for each of the objects (i.e. rendering from the center point of the object), or we utilize an environment map rendered from the view point (i.e. camera position).

The SIGGRAPH logo glass object, shown in Figure 7.2 (a), demonstrates the faithful reproduction of spatially varying refraction and attenuation behavior, in particular close to the logo symbol and the text. On the boundary of the object, total reflection can be observed.

The solid rounded cube, Fig. 7.3(a), is composed of glass layers with different characteristics. The object also shows a combination of varying refractive indices and different attenuation factors. Sparkles are visible inside the object, caused by the anisotropic scattering in its interior. Focusing of light in the glass leads to nice volume caustics.

A simple glass sphere in Figure 7.4 (a) rendered into a captured real-world environment demonstrates similar effects. Additionally, it contains a slight emissive component, which yields a fluorescent structure in the center of the sphere.

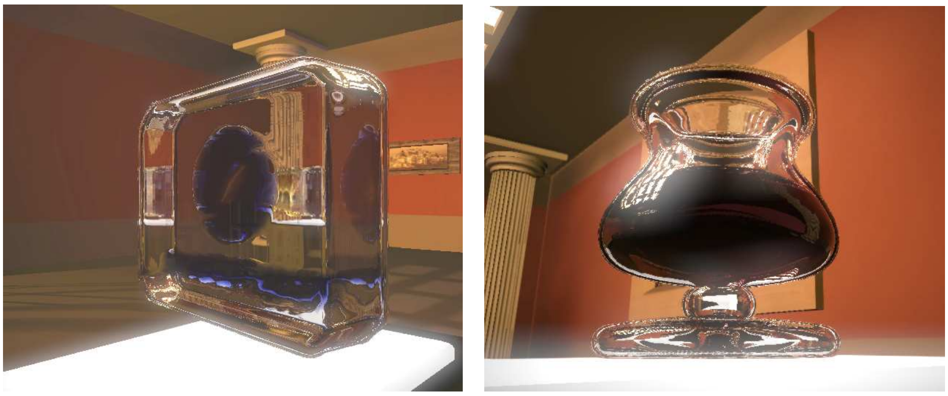
A glass filled with red wine is shown in Figure 7.4(b) and in Figure 7.2(b). In the first case, the glass is illuminated with a directional light source that causes colored caustics on the table. In the second case, the glass is lit from the bottom. However, one can not see any illumination, because of the missing scattering term. Nevertheless, an appealing reflection and complex refraction effects result in a realistic impression of the object.

An example rendering of objects inside scattering participating media can be seen in Figure 7.3(b). The glass bunny in a showing case filled with anisotropically scattering smoke seems to be made of amber with black embeddings. The look of the embeddings excellently shows the spatially varying attenuation possibilities of our algorithm. The glass bunny also anisotropically scatters light in its interior. Please note that this variety of realistic effects is rendered in real-time, in particular the volume caustics inside and outside the object, the glares, the caustics, shadows in the smoke, and the reflections on the surface.

Using transparency sorting, we can render objects one after another. This way, the rendered refractive objects can be seen through other companion pieces (e.g. Fig. 7.2 (a)). This enables a proper integration of volumetric objects into a triangle mesh environment.

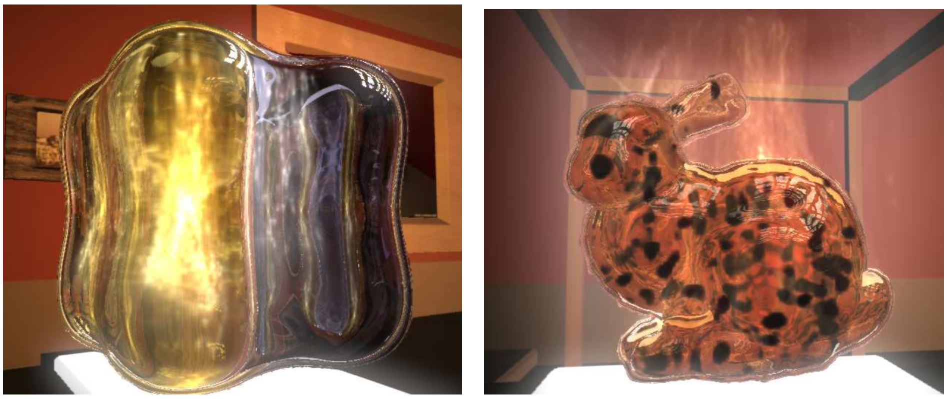
7.2 Results

All screenshots are taken from a real-time FX-demo created during this project. Video of this demo can be downloaded at `http://www.mpi-inf.mpg.de/resources/EikonalRendering/`



(a) (b)

Figure 7.2: (a) Glass block with embedded SIGGRAPH logo made up of varying refraction and attenuation materials, 14.7 FPS, (5 objects in scene). Note the surface reflections and the total reflections within, as well as the rounded cube being visible through the glass block. (b) Complex refraction patterns in the glass, 10.3 FPS, (5 objects in scene).



(a) (b)

Figure 7.3: (a) Rounded cube composed of three differently colored and differently refracting kinds of glass, showing scattering effects and caustics in its interior, 6.5 FPS. (b) Stanford bunny [sta] with spatially varying attenuation, leading to the impression of an amber-like bunny with black embeddings. Since the object is illuminated from underneath, colored volume caustics and shadows are visible in the anisotropically scattering smoke and glass, 13 FPS. – Note that there are 5 objects rendered simultaneously.

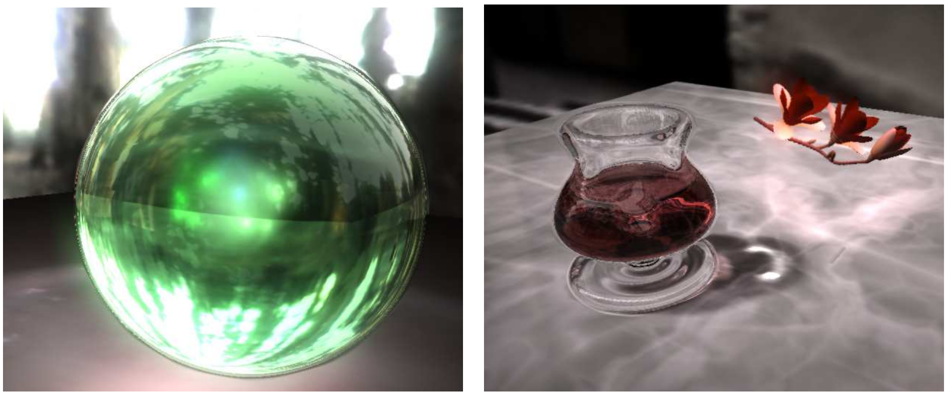
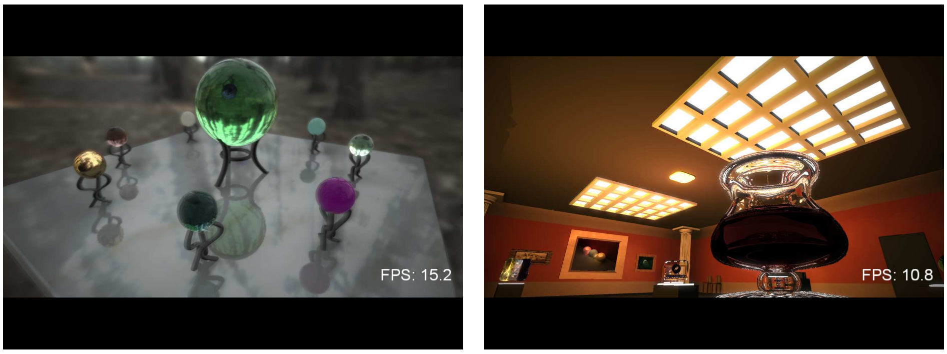

(a) (b)

Figure 7.4: (a) Colored sphere rendered into an HDR environment map, showing slight emission in addition to all other effects, at 26.2 FPS. (c) A screenshot of a wine glass scene. The time to compute the light distribution was around 7 seconds. The time required to render the object is around 0.04 seconds/25 FPS.



(a) (b)

Figure 7.5: (a) Scene showing up to 9 objects simultaneously. The big sphere in the middle has a resolution of 128^3 voxels, while the small spheres surrounding it are composed of only 32^3 voxels. The rendering speed in this scene varies between 9 FPS (when the big sphere fills out the complete viewport) and 60 FPS (whenever only the small spheres are visible). Frame rate of the screenshot is 15.2 FPS. (b) A screenshot from the museum scene, showing all refractive objects simultaneously . The objects exhibit various optical effects. Their resolution is 128^3 voxels, with up to 6 volume data textures for each object. The rendering speed is 10.8 FPS.

CHAPTER 8

CONCLUSIONS & FUTURE WORK

In this thesis we have presented a powerful algorithm for the realistic rendering of refractive objects. The algorithm is based on a continuous, volumetric representation of object data. We have used the eikonal equation, the main postulate of geometric optics, to derive a physically motivated ray equation.

We divided our approach into two stages. In the first stage, we compute the irradiance distribution in the scene volume using a light simulator. The simulator utilizes the derived ray equations to propagate a light wavefront, originating at a light source, through the scene. Using the intensity law of geometric optics and absorption properties of the object, we are able to compute the irradiance distribution in the scene. The computation time depends on the complexity and resolution of the given volumetric data, and is not yet a real-time operation. However an efficient implementation on the GPU provides us with update times of typically as 10 seconds. For static scenes, where the light source does not change relatively to the lit, refractive object, we are only required to computed the irradiance data once, giving the possibility to implement very impressive results in real-time.

We are currently supporting a directional and a point light source. However more complex light sources can be implemented within our approach. This either requires a suitable, single wavefront representation or multiple wavefronts can be used (i.e. multi-directional light source). A propagation of multiple wavefronts is performed via multi-pass simulation.

The data computed by the light simulator is used in the view renderer to efficiently combine the results into a beautiful scene rendering. Our view renderer evaluates a slightly changed volumetric rendering equation to compute the radiance of viewing rays that are cast through the volumetric scene data. A powerful, physically motivated image formation model enables us to render complex volumetric objects with arbitrarily varying refractive index, surface effects with arbitrary BRDFs, and view-dependent single-scattering effects. More advanced effects, described in this thesis, are obtained at very small additional cost. Furthermore, our view renderer is capable of rendering volume caustics and realisticly rendering the appearance of transparent objects in a scattering participating medium, such as smoke.

Due to the physical nature of the used equations, our method enables us to evaluate complex light paths, both from the object towards the viewer and from the light source towards the object, conveniently using the same mathematical framework.

Future work, based on this approach, will focus on increasing the rendering performance through optimization of the consumed data storage in the view renderer. This could be performed by using data-organization structures like octrees or other approaches, helping us to decrease the required video memory. Possible implementations of non-constant step sizes during ray casting, based on distance transforms, could also contribute to increase the rendering performance. Furthermore we would like to study state-of-the-art methods for rendering of nearby geometry. This would improve the attractiveness of volumetric representations for complex refractive objects, enabling their seamless integration into a standard mesh-based environment. A proper combination of opacity and reflection terms together with knowledge of a geometry outside of the refractive object, would allow the combination of volumetric objects with mesh-based scenery. Finally, an extension to optical anisotropic materials (e.g. crystals) would increase the generality of our approach, see Appendix A for a theoretical discussion.

We would like to finish this thesis with the same words that it began with: ”In the beginning ... there was light”. The computer graphics research evolves at a rapid pace. The border between reality and a virtual environment starts to vanish. In this work we have tried to get a step closer to this goal. Future works will only improve the quality and performance of recreating the real world in virtual environments. However, we should never forget how to distinguish the two, since one is the world we live in.

APPENDIX A

ANISOTROPIC MATERIALS

In this chapter we present a method of propagating light rays through an optically *anisotropic material*. In contrast to optically *isotropic* materials the properties of such materials depend on the direction of light propagation. The equations derived in previous chapters, do not hold anymore. Nevertheless, some additional changes in our framework enable a physically motived rendering of even these materials.

The theory of anisotropic materials is very complex (see Born and Wolf [BW99]). We do not handle its backgrounds, but will try to present simple definitions and provide a way on how to include them into our framework.

Possible examples of anisotropic materials can be found in crystals. A crystal is a solid in which the atoms, molecules or irons are packed into a regular structure[1], i.e. a repeating patterns of the structure in all three spatial dimensions. The specific field of optics handling anisotropic materials is also called *optics of crystals* or just *crystal optics*.

The consequences of the optical anisotropic structure of a medium is a couple of effects (e.g. birefringence, trirefringence, polarization effects and conical refraction) which can not properly be handled by an isotropic object representation.

Because of the complexity of light behavior inside anisotropic materials, there is only a small amount of computer graphics research in this area. Wolff and Kurlander [WK90] were the first attempting to include polarization effects in ray tracing. Tannenbaum et al. [TTW94] show some details on the implementation of the birefringence phenomenon. Wilkie et al. [WTP01] present a way of including polarization into rendering, however they do not handle refractive objects at all. Sun et al.[SFD00] present a simple way of rendering diamonds, but do not handle complex effects like birefringence. Guy and Soler [GS04] show an interactive implementation of gemstone rendering with very impressive results. They handle the double refraction phenomenon as well as the dispersion effect including total (internal) reflections inside of gemstones. Due to the complexity of ray propagation computations inside an anisotropic medium they apply a suitable approximation, with a minimal difference to correct simulation.

---

[1] some crystals, e.g. liquid crystals, adopt anisotropy temporarily through external forces, e.g. an electric field

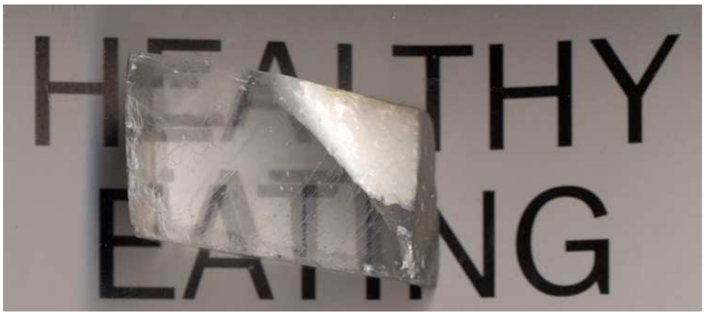


Figure A.1: A birefringence crystal (calcite) showing the double refraction of the light.

A.1 Birefringence

Birefringence, also called *double refraction*, appears in *uniaxial*[2] anisotropic refractive objects, e.g. crystals. A circular or elliptical polarized light ray entering such a material separates into two sub-rays: the *ordinary ray* (o-ray) and the *extraordinary ray* (e-ray). The rays become linearly polarized in orthogonal directions, i.e. their polarization vectors are orthogonal (Fig. A.2) (the effect of *trirefringence* appears in optically *biaxial* materials and is not handled in this work).

The magnitude of birefringence is defined as

$$\Delta n = n_e - n_o,$$

where n_o and n_e are the refractive indices for polarizations perpendicular (ordinary) and parallel (extraordinary) to the axis of anisotropy. Therefore the refractive indices correspond to both rays, created during the entering event, respectively. If $\Delta n < 0$, then the material is *negatively* uniaxial (e.g. feldspar), if $\Delta n > 0$, then it is *positively* uniaxial (e.g. quartz). If $\Delta n = 0$, then the material is optically isotropic (e.g. glass).

Ordinary rays lie in the plane of incidence and obey Snell's law. They behave, as if they were traveling through an isotropic material [BW99]. This enables us to apply the previous ray propagation equations (4.8 and 4.9), to propagate o-rays through the refracted object. Thus we can apply the same strategy as for isotropic materials for the refractive index field of an o-ray, yielding an inhomogeneous anisotropic material with respect to the o-rays. e-rays, however, require a special treatment.

---

[2] one optical axis; the definition of optical axis can be found in [BW99].

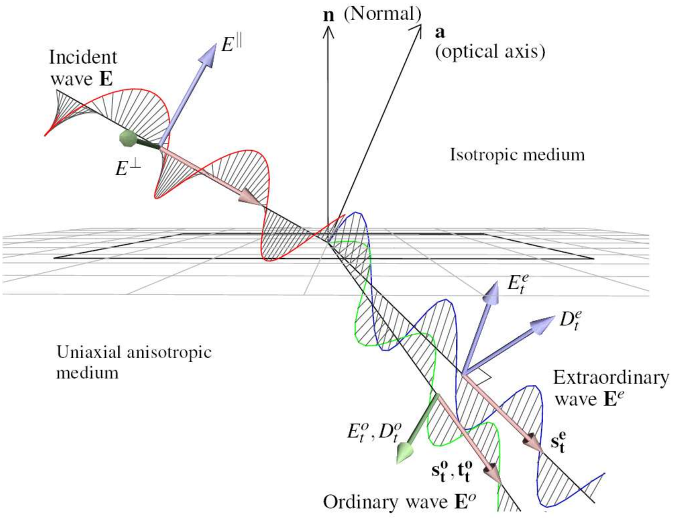


Figure A.2: Formation of an ordinary and extraordinary ray when an incident ray encounters an anisotropic medium [GS04].

A.2 e-Ray propagation in uniaxial crystals

First we define $\vec{a}$ as the *optical axis*. $\vec{a}$ is the direction in which each of the sub-rays experience the same refractive index (Fig. A.2). It can correlate with the symmetry axis of the crystal. It comprises one of the properties of an anisotropic medium. An o-ray is perpendicularly polarized with respect to the optical axis, an e-ray, however, parallel. We further define θ as the angle between the optical axis $\vec{a}$ and the wave propagation direction $\vec{s}$ (i.e. $\cos\theta = \frac{\vec{a}\cdot\vec{s}}{|\vec{a}|\cdot|\vec{s}|}$).

While the energy propagation direction for an ordinary ray is parallel to its wave direction, for an extraordinary ray this is not the case. The propagation direction of an e-ray is not parallel to the wave normal direction. Fig. A.3 illustrates the correspondences between wave propagation direction $\vec{\bigtriangledown} S_e = n_e(\theta)\vec{s_e}$ and the energy or ray propagation direction $\vec{t_e}$, which is relevant for our purpose. However, as shown in [BW99], the wave directions of both rays still obey the laws of refraction (also known as Snell's law). Thus, to propagate an $e-$ray through an inhomogeneous, anisotropic medium, we have to find a mapping $f(\vec{s_e}) = \vec{t_e}$ to be able to compute the ray vector from the wave vector.

Born and Wolf proposed a formula to bring vector $\vec{t_e}$ into a relation with $\vec{s_e}$:

$$\vec{t_k} = \frac{\vec{s_k}}{v_p v_r}\left(v_p^2 + \frac{v_p^2(v_r^2 - v_p^2)}{v_p^2 - v_k^2}\right), (k = x, y, z), \tag{A.1}$$

where v_p is the *phase velocity* (in the direction of $\vec{s}$) and is defined as $v_p = \frac{c}{n}$, v_r is

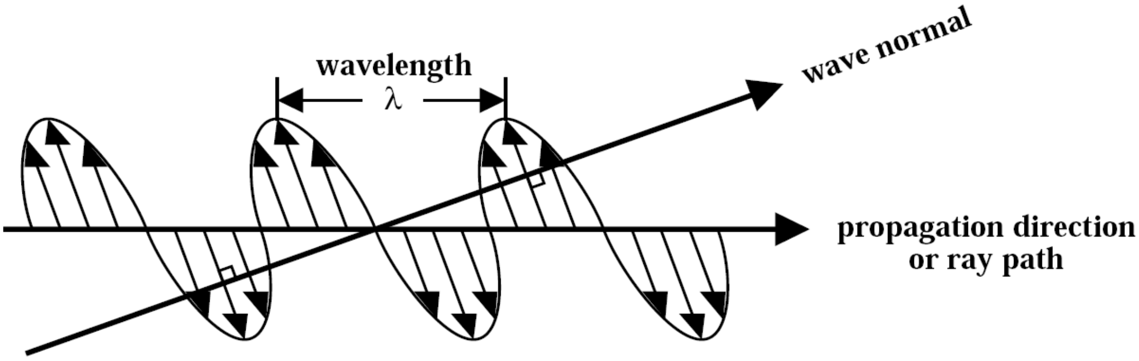


Figure A.3: The electric field of an e-ray is not perpendicular to the wave propagation direction. The consequence is that the energy propagation and wave propagation vectors are not parallel, i.e. $\vec{t_e} \nparallel \vec{s_e}$. (Note that this figure has no directly verifiable physical significance.)

the velocity of energy transport or *ray velocity* with $v_r = v_r \vec{t} \cdot \vec{s}$ and v_k is a *principal velocity of propagation*[3] and is defined as $v_k = \frac{c}{\sqrt{\mu \varepsilon_k}}$ for every $k = (x, y, z)$. The principal velocities are constant and can be pre-computed based on the *dielectric tensor* ε and the *permeability constant* μ[4]. Both values are properties of the refractive medium and can vary spatially to simulate an inhomogeneous, anisotropic substance (e.g. liquid crystal).

We can see in Eq. A.1 that the denominator $v_p^2 - v_k^2$ may vanish. This corresponds to so called *conical refraction*, where a particular wave directions $\vec{s}$ corresponds to an infinite number of ray directions $\vec{t}$. We will not take care of this effect, since the required handling for this special case can not easily be implemented on the GPU.

Typically, there are two possible solutions for the phase velocity v_p. Born and Wolf have shown that both velocities can be computed as following:

$$v_o = \frac{c}{n_o}$$

$$v_e = \left[((\frac{c}{n_o})^2 \cos^2 \theta + (\frac{c}{n_e})^2 \sin^2 \theta \right]^{\frac{1}{2}}.$$

For the special case $\theta = 0$, leading $\vec{s} \parallel \vec{a}$, we do not get the effect of double refraction, since the both ray propagation directions $\vec{t_o}$ and $\vec{t_e}$ are also parallel.

---

[3] see Born and Wolf[BW99] for more details on ε, μ and v_k

[4] μ can be set to 1 to simulate a non-magnetic medium

A.2.1 Refraction and Internal reflection of e-rays

First let us reconsider the dielectric tensor ε from the previous section. The dielectric tensor is a constant for a given material and defined as:

$$\varepsilon = \begin{pmatrix} \varepsilon_{xx} & \varepsilon_{xy} & \varepsilon_{xz} \\ \varepsilon_{yx} & \varepsilon_{yy} & \varepsilon_{yz} \\ \varepsilon_{zx} & \varepsilon_{zy} & \varepsilon_{zz} \end{pmatrix}.$$

Since the dielectric tensor is symmetric ([BW99]), we can rewrite the tensor field in the simpler form based on proper mathematical reformulations:

$$\varepsilon = \begin{pmatrix} \varepsilon_1 & 0 & 0 \\ 0 & \varepsilon_2 & 0 \\ 0 & 0 & \varepsilon_3 \end{pmatrix}.$$

The dielectric displacement vector $\vec{D}$ and the electric field $\vec{E}$[5] are related by the tensor ε:

$$\vec{D} = \varepsilon \vec{E}.$$

In an optically isotropic medium the values of ε are equal, $\varepsilon_1 = \varepsilon_2 = \varepsilon_3$. Hence we see that the dielectric displacement vector and the electric field are collinear, $\vec{D} \cdot \vec{E} = const$. The consequence is that the ray direction and the wave directions are collinear too.

However in an uniaxial anisotropic medium, we have two non-equal constants, $\varepsilon_1 = \varepsilon_2 \neq \varepsilon_3$. This leads to $\vec{D} \nparallel \vec{E}$, which results in a non-parallel wave normal and ray vector (Fig. A.3), since $\vec{s} \perp \vec{D}$ and $\vec{t} \perp \vec{E}$.

As already mentioned in the previous section, Born and Wolf proposed a mapping f to bring both vectors in relation. Beyerle and McDermid [BM98] have shown another relation between them:

$$\vec{t} = \frac{\varepsilon \vec{s}}{\mid \varepsilon \vec{s} \mid}. \tag{A.2}$$

This is also clear, because the corresponding unit vectors $\vec{s}$ and $\vec{t}$ experience the same transformation as the field vectors $\vec{D}$ and $\vec{E}$.

Before we can use this relation, we must first find a suitable way to propagate the wave normal vector $\vec{s}$ of an e-ray in an uniaxial anisotropic medium.

Let us first define the orthonormal *principal system* $\mathcal{E}$ [6] of the crystal through their

---

[5] see Maxwell's equations [BW99]

[6] it is spanned by the wavepropagation vector $\vec{s}$ and the optical axis $\vec{a}$

orthonormal vectors $\{\vec{e_1}, \vec{e_2}, \vec{e_3}\}$:

$$\vec{e_1} = \frac{\vec{a} \times \vec{s}}{\mid \vec{a} \times \vec{s} \mid}, \quad \vec{e_2} = \frac{\vec{s} \times (\vec{a} \times \vec{s})}{\mid \vec{s} \times (\vec{a} \times \vec{s}) \mid}, \quad \vec{e_3} = \vec{a}. \tag{A.3}$$

Now we use the proposal by Beyerle and McDermid [BM98] to build a mapping from $\mathcal{E}$ to a non-orthonormal coordinate system $\mathcal{E}'$ with the help of arbitrarily orthonormal basis vectors $\{\vec{e'_1} = (1,0,0)^T, \vec{e'_2} = (0,1,0)^T, \vec{e'_3} = (0,0,1)^T\}$ (e.g. world space or volume space base vectors):

$$\gamma = O \begin{bmatrix} \frac{1}{n_e} & 0 & 0 \\ 0 & \frac{1}{n_e} & 0 \\ 0 & 0 & \frac{1}{n_o} \end{bmatrix} O^T$$

with

$$O = \begin{pmatrix} \vec{e_1} \cdot \vec{e'_1} & \vec{e_2} \cdot \vec{e'_2} & \vec{e_3} \cdot \vec{e'_3} \\ \vec{e_1} \cdot \vec{e'_1} & \vec{e_2} \cdot \vec{e'_2} & \vec{e_3} \cdot \vec{e'_3} \\ \vec{e_1} \cdot \vec{e'_1} & \vec{e_2} \cdot \vec{e'_2} & \vec{e_3} \cdot \vec{e'_3} \end{pmatrix}.$$

This mapping now allows us to transform the wave specific directions from one coordinate system to another. Beyerle and McDermid have shown that this transformation remaps the *ray normal* surface[7] in $\mathcal{E}$ into a spherical shape in $\mathcal{E}'$. Thus the ray vector $\vec{t'} \in \mathcal{E}'$ corresponds now to the wave normal vector $\vec{s} \in \mathcal{E}$.

In other words, an e-ray propagating through the $\mathcal{E}'$ space behaves as if it was in a "pseudo-isotropic" medium. This is based on the fact that the wave normal vector $\vec{s}$ obeys the refraction/reflection laws with respect to the angle dependent refractive index $n(\theta)$. We can now define the following mappings, as made by Beyerle and McDermid, for the wave normal vector $\vec{s}$ and the ray propagation vector $\vec{t}$:

$$\vec{s'} = \frac{\gamma \vec{s}}{\mid \gamma \vec{s} \mid}$$

$$\vec{t} = \frac{\gamma \vec{s'}}{\mid \gamma \vec{s'} \mid}.$$

As we can see in [BM98] the e-ray traveling in $\mathcal{E}'$ coordinate space obeys Snell's law. However, since the ray is propagated in $\mathcal{E}'$ we have to take care of the normals on

---

[7] a detailed definition of the ray normal surface is given in [BW99], pp.799

the material boundaries. Based on the mapping operator γ we map a surface normal $\hat{n}$ into $\mathcal{E}'$ with:

$$\vec{\hat{n}'} = \frac{\gamma\hat{n}}{\mid \gamma\hat{n} \mid}.$$

Summarizing this we deduce: an e-ray propagated through an anisotropic material in coordinate system $\mathcal{E}'$, behaves in the same way as an o-ray that is propagated in $\mathcal{E}$. Hence we can state similar first order differential equations as in Sect. 4.1.2 to propagate an e-ray:

$$n'\frac{d\vec{r}}{ds} = \vec{s'} \tag{A.4}$$

$$\frac{d\vec{s'}}{ds} = \vec{\bigtriangledown}_\theta n', \tag{A.5}$$

where n' is the corresponding angle dependent refractive index and $\vec{\bigtriangledown}_\theta \in \mathbb{R}^3$ denotes the gradient of refractive index function $n'(x, y, z, \theta) \in \mathbb{R}$ for a fixed θ. For an inhomogeneous, anisotropic material we define the refractive index ”experienced” by an e-ray as $n' = n'_e(x, y, z, \theta) = n'_{e\theta}(x, y, z)$.

Based on these definitions we can now simulate light propagation of an extraordinary ray in inhomogeneous uniaxial anisotropic medium in an elegant way. The gradient function is computed in the same way as shown in Sect. 3.1 and stored as a 4D-lookup texture[8].

A.3 Absorption/Attenuation

As we have already mentioned in Sect. 3.2.3, the energy of a light ray can be attenuated while traveling through a medium. We have defined an attenuation factor α based on the intensity law. For anisotropic materials, Born and Wolf [BW99] proposed two formulas to compute the absorbance of the ordinary and extraordinary rays. We use both of them to define the attenuation function α, as in Sect. 3.2.3,

$$\alpha^o(t, c) = L_0 \cdot e^{-\int_0^t \sigma_a^o(c(s))ds}$$

$$\alpha^e(t, c) = L_0 \cdot e^{-\int_0^t \left(\sigma_a^o(c(s))\cos^2\theta + \sigma_a^e(c(s))(\frac{n_o}{n_e})^2 \sin^2\theta\right)ds},$$

of an o-ray and e-ray, respectively. σ^o and σ^e are characteristical, spatially varying material constants, and θ is the angle between $\vec{s_e}$ and $\vec{a}$, as mentioned previously.

Since the absorbance of an e-ray depends on its direction, the color will be differ-

---

[8] the values can also be pre-smoothed by a four dimensional smoothing operator

ent. This phenomenon is known as *pleochroism*. These approximations of attenuation for each of the refracted rays, enables us to compute the radiance of the viewing rays traveling through an anisotropic medium.

Conclusion

In this chapter we presented a way of propagating light through an optically anisotropic medium. We specified the propagation only for uniaxial (one optical axis) media. Since the propagation of an ordinary ray follows the same rule of refraction as in isotropic media, we can use the equations for isotropic materials as presented in the Chapter 4. However, an extraordinary ray behaves differently. Based on a coordinate space transformation we were able to tackle the problem of e-ray propagation. We have shown that we can simulate the e-ray trajectory inside an inhomogeneous, optically anisotropic material with the same equations as for isotropic objects.

We have presented the computation of attenuation functions for both rays. The absorbance for the e-ray depends on its polarization, which gives rise to the phenomenon of pleochroism.

The computations for ray propagation inside an anisotropic materials are quite complex. In future work, we would like to optimize the propagation of e-rays, and study the practical realization of the theory presented here. We would also like to extend our algorithm to simulate biaxial media and thus cover the complete set of optically anisotropic, inhomogeneous refractive objects.

APPENDIX B

FORMULAE

Chapter 3

- Gradient computation, Eq. 3.1

$$\vec{\bigtriangledown} f(x,y,z) = (\frac{f(x+1,y,z)-f(x-1,y,z)}{2}, \frac{f(x,y+1,z)-f(x,y-1,z)}{2}, \frac{f(x,y,z+1)-f(x,y,z-1)}{2})$$

  and three dimensional gauss smoothing operator, Eq. 3.2

$$g(x,y,z) = e^{-\left((\frac{x-x_O}{\sigma})^2+(\frac{y-y_O}{\sigma})^2+(\frac{z-z_O}{\sigma})^2\right)}.$$

  used in the Sect. 3.1 to present the computation of the gradient field $\vec{\bigtriangledown} n$ from the refractive index field.

- Using the absorption law, we derived the attenuation factor $\alpha(t,c)$ to simulate spatially varying attenuation for a light ray traveling through the scene volume, Eq. 3.4

$$\alpha(t,c) = L_0 \cdot e^{-\int_0^t \sigma_t(c(s))ds}$$

  is presented in the Sect. 3.2.3.

- Henyey and Greenstein approximation of single anisotropic scattering function.

$$p = \frac{1-g^2}{2(1-2g\cos\theta+g^2)^{3/2}},$$

  Eq. 3.5 is shown in Sect. 3.2.4 and allows us to render spatially varying anisotropic scattering effects.

- Eq. 3.6

$$R = \frac{n_i\cos\theta_i + n_t\cos\theta_t}{n_i\cos\theta_i - n_t\cos\theta_t}$$

  and the relation $T = 1 - R$ are used to compute the surface reflection factor based on the Fresnel equation. The factor is used to approximate correct light behavior during reflection at material boundaries.

Chapter **4**

- The eikonal equation, Eq. 4.2

  $$\mid \vec{\bigtriangledown} S \mid = n,$$

  shown in the Sect. 4.1.2, corresponds to the geometric description of the propagation of a massless particle (i.e. photon) through inhomogeneous refractive index fields. We use this equation to solve the volumetric rendering problem by propagating a light ray along the solutions of the eikonal equation.

- The ray equation of geometric optics, Eq. 4.7

  $$\frac{d}{ds}(n\frac{d\vec{r}}{ds}) = \vec{\bigtriangledown} n$$

  derived in the Sect. 4.1.2 from the eikonal equation describes the trajectory of a light ray within a field of inhomogeneous refractive index values. This is the main equation used in the view renderer which casts the viewing rays through the scene volume.

- The re-parameterized ray equation, Eq. 4.14

  $$n\frac{d}{dt}(n^2\frac{d\vec{r}}{dt}) = \vec{\bigtriangledown} n$$

  describes the propagation of wavefront particles, with four of them spanning a wavefront patch. The equation uses a parameterization with constant temporal step sizes, enabling a simple implementation of a light simulator on the GPU.

- A discrete version of the intensity law (Eq. 4.16), presented in the Sect. 4.2.4

  $$\Delta E_\omega(t) = \frac{\Delta E_\omega(0)\mathbb{A}(0)}{\mathbb{A}(t)}$$

  defines the way of computing the differential irradiance values for each of the wavefront patches. The computed patch energy is required to perform correct simulation of anisotropic scattering and reflection effects.

Chapter **5**

- A slightly changed volumetric rendering equation

  $$L(c) = \int_c L_c(\vec{x}, \vec{v})\alpha(t, c)dt + L_{bg}\alpha(t_\infty, c)$$

  presented in the Sect. 5.1.2 is the main postulate of our image formation model. We solve the discrete approximation of this equation while propagating a viewing ray through the scene volume.

- The L_c term of the rendering equation is defined as

  $$L_c(\vec{x}, \vec{v}) = \hat{\omega} L_s(\vec{x}, \vec{v}) + \delta(\vec{x}) R L_r(\vec{x}, \vec{v}) + L_e(\vec{x}, \vec{v})$$

  and combines all the subterms needed for the realistic rendering within our image formation model. The equation is presented in the Sect. 5.1.3.

- Making simplifying assumptions about the light source contribution to the rendering results we could derive a discrete equation for the terms L_s and L_r as

  $$L_s(\vec{x}, \vec{v}) = \sum_j p(\vec{x}, \vec{v}, \vec{l_j}) \Delta E_{\omega_j},$$

  which describes spatially varying scattering, and

  $$L_r(\vec{x}, \vec{v}) = \sum_j f_r(\vec{x}, \vec{v}, \vec{l_j}) \cos\theta \Delta E_{\omega_j},$$

  defining the reflected radiance on a surface boundary, triggered by the Dirac delta function $\delta(\vec{x})$. The term L_e describes the amount of emitted energy. The equations are defined in Sect. 5.1.4.

APPENDIX C

CODE LISTING

Below we are presenting the shader code, which is used in the view renderer to cast the viewing rays through the scene volume. The code is written in the nVidia's C for Graphics (Cg) language. It is supported by the Shader Model 3.0 hardware.

```
//------------------------------------------------------------------------
// Vertex Shader (vp20 profile)
//------------------------------------------------------------------------
struct vertexInputs
{
  float4 position : POSITION;
  float3 normal   : NORMAL;
  float4 color   : COLOR;
};

struct fragmentInputs
{
  float4 pos : POSITION;

  // Position of the ray in world coordinates
  float4 rayInWorld : TEXCOORD1;

  // Position of the ray in volume coordinates
  float4 rayInVolume : TEXCOORD2;
};


void main(
  in vertexInputs IN,
  out fragmentInputs OUT,

  // Object matrix
  uniform float4x4 objMatrix,

  // Modelview-Projection matrix
  uniform float4x4 modelViewProj)
{

  // compute coordinates of the ray position in volume space
  OUT.rayInVolume = IN.position + float4(0.5, 0.5, 0.5, 0);

  // position of the ray in world space
  OUT.rayInWorld = mul(objMatrix, IN.position);

  // compute position in clip space
  OUT.pos = mul(modelViewProj, IN.position);
}



//------------------------------------------------------------------------
// Fragment Shader (fp40 profile)
//------------------------------------------------------------------------
struct fragmentInputs
{
  float4 pos : POSITION;

  // Position of the geometry point in world coordinates
  float4 rayInWorld : TEXCOORD1;

  // Position of the ray in volume coordinates
  float4 rayInVolume : TEXCOORD2;
};

//-----------------------------------------------
```

```
// Constants
//------------------------------------------------------------------
const float3 zero = float3(0,0,0);
const float3 one = float3(1,1,1);

//------------------------------------------------------------------
// Get an interpolated voxel from 2D atlas texture.
//------------------------------------------------------------------
float4 getVoxel(sampler2D atlasTex, float3 texCoord,
                float2 sliceSize, float sliceCount, float slicePerLine)
{
  // our slice number is this one
  float slice = sliceCount * texCoord.z + 0.5;

  // now we have slice number for the both neighbors
  float slice1 = floor(slice);
  float slice2 = ceil(slice);

  // mix factor gives us the lerp factor between two neighboring slices
  float mixfactor = frac(slice);

  // coordinates of the first slice
  float t1 = slice1 / slicePerLine;
  float2 slicecoord1 = float2(frac(t1) * slicePerLine, floor(t1));
  float2 coord1 = sliceSize.xy * (texCoord.xy + slicecoord1);

  // coordinates of the second slice
  float t2 = slice2 / slicePerLine;
  float2 slicecoord2 = float2(frac(t2) * slicePerLine, floor(t2));
  float2 coord2 = sliceSize.xy * (texCoord.xy + slicecoord2);

  // now get both values
  float4 voxel1 = tex2D(atlasTex, coord1);
  float4 voxel2 = tex2D(atlasTex, coord2);

  // now interpolate between both and return the result
  return lerp(voxel1, voxel2, mixfactor);
}


//------------------------------------------------------------------
// Rendering
//------------------------------------------------------------------
float4 main(
  in fragmentInputs IN,

  // input volumes as 2d atlas textures
  uniform sampler2D volumeTex,
  uniform sampler2D volumeAttTex,
  uniform sampler2D volumeAuxTex,
  uniform sampler2D volumeIllumTex,
  uniform sampler2D volumeDirectionTex,
  uniform sampler2D volumeEmissionTex,
  uniform sampler2D volumeOpaqueDataTex,
  uniform sampler2D volumeReflectionDataTex,
  uniform samplerCUBE envMap,

  // Camera position in world coordinates
  uniform float3 eyePos,

  // Delta S, step size
  uniform float stepSize,

  // Object matricies
  uniform float4x4 objMatrix,
  uniform float4x4 objMatrixI,
  uniform float4x4 objMatrixIT,

  // Slices information of the atlas texture
  uniform float2 sliceSize,
  uniform float slicePerLine,
  uniform float sliceCount,
  uniform float2 texelSize,
  uniform float3 voxelSize
) : COLOR{
```

```
float4 vOUT;

// this are ray starting points in volume and in world coordinates
float3 rayVolumePos = IN.rayInVolume;
float3 rayWorldPos = IN.rayInWorld;

// Step through the volume/world space
float3 rayStep = stepSize * 2.0;
float3 rayWorldDir = normalize(rayWorldPos - eyePos);

// use this vectors to iterate through the volume
float3 Ic = 0;  // combined
float3 Is = 0;  // scattering
float3 Ir = 0;  // reflection
float3 Ie = 0;  // emission
float3 I = 0; // final
float3 A = 0; // absorbance
float n = 1.0;  // refraction index
bool bIterate = true;

// Initial fresnel reflection values
float T = 1, R = 0;
bool boundary = false;
bool bOpaque = false;
float4 voxelOpaqueData = float4(0,0,0,0);
sliceCount = sliceCount - 1;

// we iterate through the volume
while (bIterate && !bOpaque)
{
  // Sample voxel data on the current position
  float4 voxel = getVoxel(volumeTex, rayVolumePos,
                          sliceSize, sliceCount, slicePerLine);
  float4 voxelAtt = getVoxel(volumeAttTex, rayVolumePos,
                             sliceSize, sliceCount, slicePerLine);
  float4 voxelAux = getVoxel(volumeAuxTex, rayVolumePos,
                             sliceSize, sliceCount, slicePerLine);
  float3 voxelEmission = getVoxel(volumeEmissionTex, rayVolumePos,
                                  sliceSize, sliceCount, slicePerLine).rgb;
  float4 voxelReflectionData = getVoxel(volumeReflectionDataTex, rayVolumePos,
                                        sliceSize, sliceCount, slicePerLine);
  float3 voxelLightDir = getVoxel(volumeDirectionTex, rayVolumePos, sliceSize,
                                  sliceCount, slicePerLine).xyz;
  float3 voxelIllum = getVoxel(volumeIllumTex, rayVolumePos, sliceSize,
                               sliceCount, slicePerLine).xyz;
         voxelOpaqueData = getVoxel(volumeOpaqueDataTex, rayVolumePos,
                                    sliceSize, sliceCount, slicePerLine);

  // Compute anisotropy factor
  float scatterStrength = voxelAux.r;
  float anisotropyFactor = voxelAux.g;
  float anisotropyFactorSquared = anisotropyFactor * anisotropyFactor;

  // transform gradient into our system
  float3 gradient = mul(float3x3(objMatrixIT), voxel.xyz);

  // Compute Emission factor
  Ie = voxelEmission;

  // Compute Attenuation factor
  A.rgb += rayStep * voxelAtt.xyz;


  // Compute for all incoming lights its contribution
  // Currently we are only using one light source

  // transform light direction into our coordinate system
  voxelLightDir += 0.0001;    // add some epsilon to prevent numeric problems
  float3 lightDir = mul(float3x3(objMatrix), voxelLightDir);
  lightDir = normalize(lightDir);

  // Compute anisotropic scattering term
  float ft = dot(lightDir, normalize(rayWorldDir)) + anisotropyFactorSquared;
  ft = 1 - 2 * anisotropyFactor * ft;
  Is = voxelIllum * 0.5 * (1 - anisotropyFactorSquared) / (pow(ft, 1.5));
```

```
    // Compute new direction, position and refraction index
    float3 oldPos = rayWorldPos;
    rayWorldPos = rayWorldPos + (rayStep / n) * rayWorldDir;
    rayWorldDir = rayWorldDir + rayStep * gradient;
    n +=  dot(gradient, (rayWorldPos - oldPos));

    // compute new volume position
    float4 newVp = mul(objMatrixI, float4(rayWorldPos,1));
    rayVolumePos = newVp.xyz + float3(0.5, 0.5, 0.5);

    // save current transmission factor
    float oldT = T;

    // If we are on a boundary; can be replaced by texture look-up
    if (float(length(gradient) > 0.8) && !boundary)
    {
      boundary = true;

      // check whenever opaque data is specified
      bOpaque = voxelOpaqueData.a > 0;

      // compute fresnel term
      // This is only an approximation of the fresnel reflection term
      // computation, which was presented in the thesis.
      R = 1 / pow(1 + abs(dot(normalize(gradient), normalize(rayWorldDir))), 2);
      R = min(pow(R,3) * voxelAux.a, 1.0);
      T = T * (1 - R);

      // compute reflection
      float3 dir = reflect(normalize(rayWorldDir), normalize(gradient));
      float3 reflectionColor = texCUBE(envMap, dir ).rgb;
      Ir = lerp(reflectionColor, voxelReflectionData.rgb * reflectionColor,
                voxelReflectionData.a);
    }else{
      R = 0;
    }

    // if gradient is too small, so we could not be on a boundary
    if (length(gradient) < 0.01) boundary = false;

    // Compute combined intensity per voxel
    Ic = scatterStrength * Is + R * Ir + Ie ;

    // Compute final integral
    I += Ic * exp(-A) * oldT;

    // check if we are not outside of the volume
    float3 temp1 = rayVolumePos > zero;
    float3 temp2 = rayVolumePos < one;
    float temp3 = dot(temp1, temp2);
    if (temp3 < 3.0) bIterate = false;
  }

  // get color from environment map as if ray is coming out
  float3 envColor = lerp(voxelOpaqueData.rgb, texCUBE(envMap, rayWorldDir).rgb,
                         1.0 - float(bOpaque));

  // Compute resulting color
  vOUT.xyz = envColor * exp(-A) * T  + I;
  vOUT.a = 1;

  // Need this here to prevent possible overflow problems
  vOUT.rgb = min (vOUT.rgb, 65504);

  // return
  return vOUT;
}
```

LIST OF FIGURES

1.1 Glass block with embedded SIGGRAPH logo. The complex behavior of refraction is combined with a spatially varying attenuation inside the letters. Note the total reflection on the block boundaries. 10

1.2 Rounded cube consisting of three different glass layers. Light simulator is capable to compute the irradiance distribution inside the object volume providing a nice sparkle like structures during the rendering phase. . . . 11

2.1 One of the first synthetic image showing caustics, refractions and reflections [Arv86] 14

2.2 (a) Realistic refractions and caustics rendered at 2.5 frame per second on 8 client machines [WBS+02] (b) Caustic computed by the photon mapping approach described in [Jen01] 15

2.3 Photon mapping computed on the GPU by [PDC+03] 16

3.1 Wine glass object consisting of 128x128x128 voxels. (a) Refractive index field was pre-smoothed before computing the gradients. (b) No smoothing: Glass appears blocky. Note how the smooth filtering unintendedly expands the boundaries. 20

3.2 (a) A real world photo showing the refraction effect. The light rays are bent as they cross from water to air. (b) Volume caustics created by propagating light particles through a glass sphere. Notice how the light rays are being focused [Jen01]. 22

3.3 (a) Scattering occurs when light originating at the light source *scatters* on material impurities, e.g. dust particles. (b) Wine glass showing refraction and attenuation. Wine attenuates green and blue components of the light resulting in a red colored fluid. 23

3.4 Black glass sphere reflecting the environment. The sphere's material absorbs almost the whole light energy such that only the reflection on the surface is visible. (a) Native reflection (b) Fresnel reflection. Note the use of fresnel equations improves the realism of the scene. 24

3.5 (a) Real world photo of dispersion phenomenon on a prism [MS] (b) Emulated dispersion effect with a three-pass computation of color components red, green and blue. 25

4.1 ¡¡¡¡¡¡¡ .mine Snell's law; The propagation time of light from P to Q is minimal, if the sines of the ray angles in different media are in ======= Snell's law; The propagation time of light from P to Q is minimal, if the sines of the ray angles in different media are in ¿¿¿¿¿¿¿ .r48 proportion to the refractive indices. 30

4.2 2D illustration of our complex image formation scenario – due to inhomogeneous material distribution, light rays and viewing rays are bent on their way through the *scene volume* V. Light rays always travels orthogonally to the light wavefronts. Light wavefronts are the iso-surfaces of constant travel time from the light source. 33

4.3 Adaptive wavefront refinement. (a) 2D illustration: the wavefront is represented by particles (red dots) that are connected to form a wavefront (blue lines). While advancing through the voxel volume (shown in gray) the wavefront is tessellated such that its patches span less than a voxel. (b) 3D illustration of the tessellation for one wavefront patch. 35

4.4 The intensity law of geometric optics (left) and its discretized version (right) in the form of a *stream tube*. The product of area and differential irradiance is constant along a tube of rays. 37

5.1 Glass sphere (128^3 voxels) refracting the viewing rays. (a) Refraction only (b) Refraction combined with attenuation; red and blue light components are attenuated, producing green colored glass. 40

5.2 (a) L_r becomes active on the material boundary between glass and air, and is responsible for reflections of the environment (Fresnel reflections) (b) Additionally, L_s and L_e have become active. Scattering gives a feeling of impurities inside the glass. A small blob in the center of the glass, emitting radiance, gives an impression of phosphorescence. 42

6.1 (a) The refractive index volume of the glass is approached by a spherical wavefront from the right. The adaptive tessellation of the wavefront is also visible. (b) When it passes through the object, beautiful caustic patterns appear in its irradiance distribution. 52

6.2 Basic volume ray casting of a refractive object. For each pixel of the image plane, a viewing ray is propagated through the volume. The samples are interpolated and composed to produce the resulting pixel value. . . 53

6.3 Refractive object and its proxy geometry. (a) Cube/Box as a bounding primitive. (b) Use of a sphere mesh as bounding primitive. 54

6.4 Work-flow of our rendering system. 56

7.1 Comparison between a ray-traced image rendered with the Persistence of Vision raytracer (POV-Ray) (a) and our algorithm (b). The differences in the refraction and shadow size as well as the slightly displaced caustic pattern are due to smoothing of the refractive index field. 58

7.2 (a) Glass block with embedded SIGGRAPH logo made up of varying refraction and attenuation materials, 14.7 FPS, (5 objects in scene). Note the surface reflections and the total reflections within, as well as the rounded cube being visible through the glass block. (b) Complex refraction patterns in the glass, 10.3 FPS, (5 objects in scene). 60

7.3 (a) Rounded cube composed of three differently colored and differently refracting kinds of glass, showing scattering effects and caustics in its interior, 6.5 FPS. (b) Stanford bunny [sta] with spatially varying attenuation, leading to the impression of an amber-like bunny with black embeddings. Since the object is illuminated from underneath, colored volume caustics and shadows are visible in the anisotropically scattering smoke and glass, 13 FPS. – Note that there are 5 objects rendered simultaneously. 60

7.4 (a) Colored sphere rendered into an HDR environment map, showing slight emission in addition to all other effects, at 26.2 FPS. (c) A screenshot of a wine glass scene. The time to compute the light distribution was around 7 seconds. The time required to render the object is around 0.04 seconds/25 FPS. 61

7.5 (a) Scene showing up to 9 objects simultaneously. The big sphere in the middle has a resolution of 128^3 voxels, while the small spheres surrounding it are composed of only 32^3 voxels. The rendering speed in this scene varies between 9 FPS (when the big sphere fills out the complete viewport) and 60 FPS (whenever only the small spheres are visible). Frame rate of the screenshot is 15.2 FPS. (b) A screenshot from the museum scene, showing all refractive objects simultaneously . The objects exhibit various optical effects. Their resolution is 128^3 voxels, with up to 6 volume data textures for each object. The rendering speed is 10.8 FPS. 61

A.1 A birefringence crystal (calcite) showing the double refraction of the light. 66

A.2 Formation of an ordinary and extraordinary ray when an incident ray encounters an anisotropic medium [GS04]. 67

A.3 The electric field of an e-ray is not perpendicular to the wave propagation direction. The consequence is that the energy propagation and wave propagation vectors are not parallel, i.e. $\vec{t_e} \nparallel \vec{s_e}$. (Note that this figure has no directly verifiable physical significance.) 68

BIBLIOGRAPHY

[Arv86] James R. Arvo. Backward Ray Tracing. In *ACM SIGGRAPH '86 Course Notes - Developments in Ray Tracing*, volume 12, 1986.

[BD02] David Benson and Joel Davis. Octree textures. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 785–790, New York, NY, USA, 2002. ACM Press.

[Bli82] James F. Blinn. 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*, pages 21–29, New York, NY, USA, 1982. ACM Press.

[BM98] G. Beyerle and I. S. McDermid. Ray-tracing formulas for refraction and internal reflection in uniaxial crystals. *Applied Optics*, 37:7947–7953, 1998.

[BW99] Max Born and Emil Wolf. *Principles of Optics, seventh edition*. Cambridge University Press, 1999.

[CHH02] Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 37–46, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[DBB06] Phil Dutre, Kavita Bala, and Philippe Bekaert. *Advanced Global Illumination, 2nd Edition*. A K Peters, Natick, MA, 2006.

[EAMJ05] Manfred Ernst, Tomas Akenine-Möller, and Henrik Wann Jensen. Interactive rendering of caustics using interpolated warped volumes. In *GI '05: Proceedings of Graphics Interface 2005*, pages 87–96, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2005. Canadian Human-Computer Communications Society.

[GS04] Stephane Guy and Cyril Soler. Graphics gems revisited: fast and physically-based rendering of gemstones. *ACM Trans. Graph.*, 23(3):231–238, 2004.

[GWS04] J. Günther, I. Wald, and P. Slusallek. Realtime caustics using distributed photon mapping. In *Rendering Techniques*, pages 111–121, June 2004. (Proceedings of the 15th Eurographics Symposium on Rendering).

[HBSL03] M.J. Harris, W.V. Baxter, T. Scheuermann, and A. Lastra. Simulation of cloud dynamics on graphics hardware. In *Proc. of Graphics Hardware*, pages 92–101, 2003.

[HJ41] L. Henyey and Greenstein J. Diffuse radiation in the galaxy. *Astrophys. Journal*, 93:70–83, 1941.

[HQ07] Wei Hu and Kaihuai Qin. Interactive Approximate Rendering of Reflections, Refractions, and Caustics. *IEEE TVCG*, 13(1):46–57, 2007.

[HS01] Z. S. Hakura and J. M. Snyder. Realistic reflections and refractions on graphics hardware with hybrid rendering and layered environment maps. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 289–300, London, UK, 2001. Springer-Verlag.

[IZT+07] I. Ihrke, G. Ziegler, A. Tevs, C. Theobalt, M. Magnor, and H.-P. Seidel. Eikonal rendering: Efficient light transport in refractive objects. *ACM Trans. on Graphics (Siggraph'07), to appear*, August 2007.

[Jen96] Henrik Wann Jensen. *Global Illumination Using Photon Maps*. Springer-Verlag, London, UK, 1996.

[Jen01] Henrik Wann Jensen. *Realistic image synthesis using photon mapping*. A. K. Peters, Ltd., Natick, MA, USA, 2001.

[KMS05] G. Krawczyk, K. Myszkowski, and H.-P. Seidel. Perceptual effects in real-time tone mapping. In *Proc. of Spring Conference on Computer Graphics*, pages 195–202. ACM, 2005.

[KvH84] James T. Kajiya and Brian P. von Herzen. Ray tracing volume densities. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 165–174, New York, NY, USA, 1984. ACM Press.

[MH92] Don Mitchell and Pat Hanrahan. Illumination from curved reflectors. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 283–291, New York, NY, USA, 1992. ACM Press.

[MM02] Vincent C. H. Ma and Michael D. McCool. Low latency photon mapping using block hashing. Technical report, Aire-la-Ville, Switzerland, Switzerland, 2002.

[MS] R. Merlino and L. Somantri. The wavy face of light: Darkness, shadows, colors and fringes. http://www.physics.uiowa.edu/˜umallik/adventure/physoptics/lightwave.html. [Online; accessed 30-May-2007].

[Ohb03] E. Ohbuchi. A real-time refraction renderer for volume objects using a polygon-rendering scheme. In *Proc. of CGI*, pages 190–195, 2003.

[PBMH02] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 703–712, 2002.

[PDC$^+$03] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. In *SIGGRAPH/EUROGRAPHICS Workshop On Graphics Hardware*, pages 41–50, 2003.

[Pri63] P. Pringsheim. *Fluorescence and phosphorescence.* New York : Interscience Publishers, second edition, 1963.

[PTVF92] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C.* Cambridge University Press, 1992.

[Pur04] Timothy J. Purcell. *Ray tracing on a stream processor*. PhD thesis, 2004. Adviser-Patrick M. Hanrahan.

[SFD00] Y. Sun, F.D. Fracchia, and M.S. Drew. Rendering diamonds. In *Proc. of the 11th WSCG*, pages 9–15, 2000.

[SHB99] Milan Sonka, Vaclav Hlavac, and Roger Boye. *Image Processing, Analysis and Machine Vision*. PWS Publishing, second edition, 1999.

[SK07] Musawir A. Shah and Jaakko Konttinen. Caustics mapping: An image-space technique for real-time caustics. *IEEE Transactions on Visualization and Computer Graphics*, 13(2):272–280, 2007. Member-Sumanta Pattanaik.

[SL96] Jos Stam and Eric Languénou. Ray tracing in non-constant media. In *Proceedings of the eurographics workshop on Rendering techniques '96*, pages 225–ff., London, UK, 1996. Springer-Verlag.

[sta] The Stanford 3D Scanning Repository. http://graphics.stanford.edu/data/3Dscanrep. [Online; accessed 10-January-2007].

[SWS02] Jörg Schmittler, Ingo Wald, and Philipp Slusallek. Saarcor: a hardware architecture for ray tracing. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 27–36, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[TTW94] David C. Tannenbaum, Peter Tannenbaum, and Michael J. Wozny. Polarization and birefringency considerations in rendering. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 221–222, New York, NY, USA, 1994. ACM Press.

[WBS$^+$02] I. Wald, C. Benthin, P. Slusallek, T. Kollig, and A. Keller. Interactive global illumination using fast ray tracing. In *Proc. of Eurographics Rendering Workshop*, pages 15–24, 2002.

[WD06] C. Wyman and S. Davis. Interactive image-space techniques for approximating caustics. In *Proceedings of ACM I3D*, pages 153–160, 2006.

[WK90] Lawrence B. Wolff and David J. Kurlander. Ray tracing with polarization parameters. *IEEE Comput. Graph. Appl.*, 10(6):44–55, 1990.

[WS03] M. Wand and W. Strasser. Real-time caustics. *Computer Graphics Forum (Proc. of Eurographics 2003)*, pages 611–622, 2003.

[WTP01] Alexander Wilkie, Robert F. Tobler, and Werner Purgathofer. Combined rendering of polarization and fluorescence effects. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 197–204, London, UK, 2001. Springer-Verlag.

[Wym05a] C. Wyman. An approximate image-space approach for interactive refraction. *ACM Trans. Graph.*, 24(3):1050–1053, 2005.

[Wym05b] C. Wyman. Interactive image-space refraction of nearby geometry. In *Proceedings of GRAPHITE*, pages 205–211, 2005.

[ZTTS06] G. Ziegler, A. Tevs, C. Theobalt, and H.-P. Seidel. On-the-fly point clouds through histogram pyramids. In *Proc. of VMV*, pages 137–144, 2006.