EFFICIENT RENDERING OF REALISTIC METALS

by

Lazányi István Zsolt

Department of Control Engineering and Information Technology
Budapest University of Technology and Economics

supervisor

Prof. Szirmay-Kalos László, Ph.D.

Department of Control Engineering and Information Technology
Budapest University of Technology and Economics

2008
# Contents

1 Introduction  .......................................................... 4  
   1.1 Lighting calculations ........................................... 6  
      1.1.1 The geometry of lighting .................................. 6  
      1.1.2 Basic radiometric quantities ............................... 7  
      1.1.3 The BRDF and the rendering equation .................... 8  
      1.1.4 Physically plausible BRDF models ......................... 10  
      1.1.5 Global illumination methods ............................... 14  
   1.2 Graphics processing units ...................................... 15  
      1.2.1 Shaders beyond the standard pipeline operation ......... 15  

2 Previous Work .......................................................... 17  
   2.1 Specular Effects on the GPU .................................... 17  
      2.1.1 Ray tracing in the original geometry .................... 17  
      2.1.2 Geometry transformation .................................. 18  
      2.1.3 Image based lighting and rendering ....................... 19  
      2.1.4 Ray tracing in the sampled geometry ..................... 21  
   2.2 Diffuse and glossy effects on the GPU ......................... 21  

3 Fresnel Term Approximations for Metals .......................... 23  
   3.1 Schlick’s approximation ....................................... 23  
   3.2 The new Fresnel term approximation for metals ............... 25  
      3.2.1 Analyzing the Fresnel function ............................ 25  
      3.2.2 Rescaling the Schlick’s function .......................... 26  
      3.2.3 Error compensation ........................................ 27  
   3.3 Implementation .................................................. 29  
   3.4 Results ............................................................ 30  
   3.5 Conclusions ....................................................... 30  

4 Rendering Specular Reflections ...................................... 35  
   4.1 Localization of the environment map ............................ 35  
      4.1.1 Finding initial guesses .................................... 36  
      4.1.2 Refining the initial guesses ............................... 36  
      4.1.3 Iterative refinement ....................................... 37  
      4.1.4 Iteration strategies ....................................... 39  
   4.2 Implementation .................................................. 41  
      4.2.1 Secant method ............................................... 42  
      4.2.2 False position method ..................................... 42  
      4.2.3 Mixed method ............................................... 43  
   4.3 Results ............................................................ 44  
      4.3.1 Analyzing the secant method ............................... 44  
      4.3.2 Comparing the mixed method and the false position method 44  
   4.4 Conclusions ....................................................... 50
Chapter 1

Introduction

The main objective of rendering is to provide the computer user with the illusion of watching real objects on the computer screen [SKAC03, SP92]. This illusion requires the brightness of the pixels be identical to the brightness of the point visible in the direction defined by the pixel. Thus image synthesis should identify the visible points, and then compute the brightness of these points. To query the point that is visible through a pixel, we have to start a line, called ray, from the eye position through the center of the given pixel, and determine the intersection of the ray with the objects of the virtual world. The visible point, also called shaded point, is defined by the intersection that is closest to the eye position.

Assuming that the shaded point itself does not emit light, the brightness of the shaded point can be determined by considering the illumination received from other surface points. That is, we have to consider all possible incoming directions at that point and summarize their contributions to the brightness of the shaded point. The resulting integral is called the rendering equation [Kaj86]. To determine the surface points that are visible from the shaded point, new visibility queries are needed. Furthermore, to calculate their contributions, the brightness of those surface points is also needed. This will recursively generate new surface points and new visibility queries. Finally, the solution of the problem can also be explained as the identification of those light paths that connect the light sources to the eye via single or multiple reflections or refractions. When these multiple reflections or refractions are taken into account, the method belongs to the category of global illumination methods [SK08, Ant04]. On the other hand, if indirect illumination is ignored, i.e. only single reflections and refractions are considered, we talk about local illumination methods.

Figure 1.1: Typical computer game scenes.

In interactive applications such as computer games (Figure 1.1) and real-time systems, there is usually no time to trace multiple reflections and refractions. Thus, to maintain an interactive frame rate, local illumination methods and other fast approximations are applied
In a typical game, for example, the scene is surrounded by diffuse, textured walls that reflect the colorful lights emitted by dynamic objects. These colorful reflections not only provide a pleasing visual result but also may help to hide some artifacts introduced during rendering. Furthermore, reflective, metallic objects such as weapons and armors are also often included. The reflections caused by these objects help to increase the contrast of the scenes and also help to establish the atmosphere of the game, especially for dark scenes (e.g. “knives glinting in the moonlight”). However, to save precious rendering time, metals such as steel, silver and aluminum are rendered in games so that they look equally “gray”, although realistic silver could be easily distinguished from steel due to its yellowish look (Figure 1.2).

![Figure 1.2: Comparing objects made of steel and silver.](image)

Metals have several unique material properties that are worth studying from rendering aspects. For example, the refraction index for metals is a complex number, where the imaginary part describes how the electromagnetic waves die away in the material. Thus, metals do not let electromagnetic waves travel inside the material, as also demonstrated in the Faraday cage experiment [Far]. Furthermore, the refraction index for metals varies significantly on different wavelengths. Thus the Fresnel term, which is derived from the refraction index and determines the amount of reflected light for a given incidence angle, will also depend on wavelength, resulting in colored specular reflections (the specular highlight for most materials, e.g. plastics is white).

Depending on their optical quality, metals can have different reflective properties. While *optically smooth* (polished) metals can be treated as ideal mirrors, *optically rough* (non-polished, oxidized) metals act as glossy mirrors, or as diffuse surfaces in the extreme case. Thus, to render metals in real time, accurate but simple reflection models and rendering methods are needed for both ideal and non-ideal (glossy or diffuse) reflections. Furthermore, instead of considering direct illumination only, at least one indirect bounce of light should be calculated or at least approximated to achieve realistic result.

The organization of the dissertation is as follows. Section 1.1 defines the basic concepts of illumination calculations and introduces the rendering equation, while Section 1.2 discusses the capabilities of the programmable graphics hardware. Chapter 2 reviews the state-of-the-art of the dissertation. Then, the next four chapters present the new scientific results for rendering metals with hardware support. Namely, Chapter 3 proposes a simplification of the Fresnel function for metals, Chapters 4 and 5 present approximation techniques for rendering ideal and non-ideal reflections, respectively, and Chapter 6 presents a CPU-based global illumination method that also exploits the capabilities of the graphics hardware. Finally, Chapter 7 concludes the results and provides a list of publications.
1.1 Lighting calculations

In this section we shall introduce the basic quantities of lighting calculations such as solid angle, power, radiance, and the BRDF function. Using these concepts, we shall introduce the rendering equation and discuss the global illumination problem.

1.1.1 The geometry of lighting

In the two-dimensional space, directions can be represented as unit-length 2D vectors. The set of all vectors forms a unit circle, while a subset of them defines an arc on this circle. Angle (angular extent) of a set of 2D directions is defined as the length of the arc defined by the respective 2D vectors.

Analogously, directions in the 3D space are represented as unit-length 3D vectors, usually denoted by $\omega$ ($\omega \in \mathbb{R}^3$). Again, the set of all directions forms a unit sphere in the 3D space, while a subset of directions can be characterized using the concept of solid angle, which is the three-dimensional equivalent of the angular extent.

Figure 1.3: Defining solid angle subtended by surface $S$.

Solid angle subtended by a set of 3D directions is defined as the area of the surface patch defined by the respective unit-length 3D vectors. Since the surface area of a unit sphere equals $4\pi \cdot 1^2 = 4\pi$, the solid angle subtended by a sphere equals $4\pi$ (steradians), while a hemisphere subtends $2\pi$ (steradians).

Considering $x$ as a reference point, solid angle $\Delta \omega$ subtended by a surface $S$ can be determined by projecting surface $S$ onto the unit sphere around $x$ and measuring the area of the projected surface $S'$ (Figure 1.3). For example, having a small surface patch $S$ facing toward reference point $x$, solid angle subtended by the patch is approximately

$$\Delta \omega \approx \Delta A/r^2,$$

where $\Delta A$ is the size of the patch and $r$ is the distance between the patch and the reference point. This approximation is only acceptable if the surface area $\Delta A$ is small compared to the distance between the patch and point $x$. In the limiting case, if the surface area is infinitesimally small, the subtended solid angle is exactly

$$d\omega = dA/r^2,$$

where $dA$ denotes the area of an infinitesimal surface patch. For a surface patch with arbitrary orientation, the formula of the infinitesimal solid angle becomes

$$d\omega = dA^\perp/r^2,$$

where $dA^\perp$ is the projected (or foreshortened) area seen from point $x$. For example, denoting the angle between the surface normal and the vector pointing from $x$ to the center of the surface patch by $\theta$, the solid angle equals

$$d\omega = dA \cos \theta/r^2.$$
1.1.2 Basic radiometric quantities

A basic quantity in radiometry is power (flux), which describes the rate of energy emission of an emissive surface, that is, it measures energy (or number of photons) emitted per unit time. Power (flux) is measured in Joule/second (Watt) and is usually denoted by $\Phi$:

$$\Phi = \frac{dE}{dt}.$$  

Energy (or light) received by an object can be described by the total incident flux, or by its spatial derivative called irradiance. Irradiance describes the amount of incoming light per unit surface area (Watt/m$^2$) and is usually denoted by $H$:

$$H = \frac{d\Phi}{dA}.$$  

Radiance

The central quantity in radiometry is radiance, which describes the distribution of light in the 3D space. For an arbitrary point and an arbitrary direction, radiance measures the rate of energy transmission (i.e. flux) through that point into that direction (Figure 1.4). Since the energy transmitted through a point is infinitesimally small, we will consider a virtual surface $dA^\perp$ that is perpendicular to the direction of travel, and then, radiance $L$ is defined as flux per unit surface area per unit solid angle (Watt/m$^2$/steradian):

$$L = \frac{d\Phi}{d\omega \cdot dA^\perp}.$$  

Figure 1.4: Defining radiance for energy transmission.

Radiance can also be introduced for energy emission or energy absorption. For energy emission, radiance measures the rate of emission (i.e. flux) at a particular surface location into a particular direction. In this case, radiance is defined as flux per unit solid angle per unit surface area foreshortened (i.e. per unit surface area perpendicular to the direction of travel) (Figure 1.5). For energy absorption, radiance can be defined similarly.

$$L = \frac{d\Phi}{d\omega \cdot dA \cos \theta}.$$  

Figure 1.5: Defining radiance for energy emission.

Radiance has the following important properties. First of all, radiance is constant along unoccluded straight lines, that is, it does not decay or attenuate during the travel of light.
Another important property is that cameras and human eye “see” surface radiance. In other words, image irradiance, i.e. the power detected by a unit-sized sensor in the camera, is proportional to scene radiance. Finally, comparing the definitions of irradiance and radiance, we can establish the following relationship between these quantities:

\[ H = L \cos \theta d\omega. \]

1.1.3 The BRDF and the rendering equation

Let us consider a surface point \( \vec{x} \), incoming direction \( \vec{\omega}_{\text{in}} \), outgoing direction \( \vec{\omega}_{\text{out}} \), and an infinitesimal cone of size \( d\omega_{\text{in}} \) around \( \vec{\omega}_{\text{in}} \) as shown in Figure 1.6. Denoting the radiance arriving from direction \( \vec{\omega}_{\text{in}} \) by \( L_{\text{in}}(\vec{\omega}_{\text{in}}) \), light arriving from the infinitesimal cone generates the following irradiance:

\[ H_{\text{in}}(\vec{\omega}_{\text{in}}) = L_{\text{in}}(\vec{\omega}_{\text{in}}) \cos \theta_{\text{in}} d\omega_{\text{in}}. \]

On the surface, light is scattered in all directions. Let us denote the radiance leaving in direction \( \vec{\omega}_{\text{out}} \) by \( L_{\text{r}}(\vec{\omega}_{\text{out}}) \). According to experimental results, if \( H_{\text{in}}(\vec{\omega}_{\text{in}}) \) increases, there is a proportional increase in \( L_{\text{r}}(\vec{\omega}_{\text{out}}) \):

\[ L_{\text{r}}(\vec{\omega}_{\text{out}}) \propto H_{\text{in}}(\vec{\omega}_{\text{in}}). \]

That is, outgoing radiance \( L_{\text{r}}(\vec{\omega}_{\text{out}}) \) is proportional to the incoming radiance \( L_{\text{in}}(\vec{\omega}_{\text{in}}) \), and also proportional to \( d\omega_{\text{in}} \). Bidirectional reflection distribution function (BRDF) is defined as the rate of proportionality:

\[ f_r(\vec{\omega}_{\text{in}} \rightarrow \vec{\omega}_{\text{out}}) = \frac{L_{\text{r}}(\vec{\omega}_{\text{out}})}{L_{\text{in}}(\vec{\omega}_{\text{in}})} = \frac{L_{\text{r}}(\vec{\omega}_{\text{out}})}{L_{\text{in}}(\vec{\omega}_{\text{in}}) \cos \theta_{\text{in}} d\omega_{\text{in}}}. \]

That is, the BRDF measures outgoing radiance per unit incoming irradiance. Thus, outgoing radiance can be expressed from the incoming radiance as follows:

\[ L_{\text{r}}(\vec{\omega}_{\text{out}}) = L_{\text{in}}(\vec{\omega}_{\text{in}}) f_r(\vec{\omega}_{\text{in}} \rightarrow \vec{\omega}_{\text{out}}) \cos \theta_{\text{in}} d\omega_{\text{in}}. \]

Summarizing this expression for all incoming directions, we get the rendering equation (light transport equation) [Kaj86]:

\[ L_{\text{r}}(\vec{\omega}_{\text{out}}) = \int_{\Omega_{\text{in}}} L_{\text{in}}(\vec{\omega}_{\text{in}}) f_r(\vec{\omega}_{\text{in}} \rightarrow \vec{\omega}_{\text{out}}) \cos \theta_{\text{in}} d\omega_{\text{in}}. \]

Alternatively, \( \omega_{\text{out}} \) can be simply denoted by \( \omega \) and \( \omega_{\text{in}} \) denoted by \( \omega' \). Note that surface point \( \vec{x} \) is also included:

\[ L_{\text{r}}(\vec{x} \rightarrow \vec{\omega}) = \int_{\Omega_{\text{r}}} L_{\text{in}}(\vec{x} \leftrightarrow \vec{\omega}') f_r(\vec{\omega}' \rightarrow \vec{x} \rightarrow \vec{\omega}) \cos \theta' d\omega'. \]
Symmetry of the BRDF

The symmetry of the BRDF means that the incoming and outgoing directions in the BRDF function can be interchanged without changing the function value:

\[ f_r(\vec{\omega}_{\text{in}} \rightarrow \vec{\omega}_{\text{out}}) = f_r(\vec{\omega}_{\text{out}} \rightarrow \vec{\omega}_{\text{in}}). \]

This symmetry is often referred to as the Helmholtz reciprocity principle. However, Helmholtz’s original statement concerned only about specular reflections in an optical system consisting of mirrors, lenses and prisms. He declared that in such an optical system, light paths are reversible so that by reversing a light path the relative power loss along the path remains the same.

For non-specular reflection, the reciprocity principle was first stated by Lord Rayleigh. According to his original statement, “intensity” transferred from a “radiant point” to another point remains the same if the position of the two “points” are exchanged. Examining the attenuation properties of a point light source we can quickly conclude that instead of points infinitesimally small surfaces have to be considered. As paraphrased in [Vea97]: “We are given a small reflective surface, exposed to a small light source and a small irradiance sensor (which measures the power per unit area falling on a square facing toward the reflective surface). His principle states that if the positions of the source and sensor are exchanged, the measured irradiance will be the same. This implies that the corresponding BRDF must be symmetric, as may easily be verified.”

To see this, let us consider a small light source of area \( dA_{\text{light}} \), a small reflective surface of area \( dA \) and a small irradiance sensor, as seen in Figure 1.7. Let us denote the radiance received by the reflective surface by \( L_{\text{in}} \). According to the rendering equation, the radiance leaving the reflective surface equals

\[ L_{\text{out}} = L_{\text{in}} f_r(\vec{\omega}_{\text{in}} \rightarrow \vec{\omega}_{\text{out}}) \cos \theta_{\text{in}} d\omega_{\text{in}}, \]

where \( d\omega_{\text{in}} \) is the solid angle occupied by the light source. Assuming that the irradiance sensor faces toward the reflective surface, the irradiance measured by the sensor can be written as \( H = L_{\text{out}} d\omega_{\text{out}} \), that is,

\[ H = L_{\text{in}} f_r(\vec{\omega}_{\text{in}} \rightarrow \vec{\omega}_{\text{out}}) \cos \theta_{\text{in}} d\omega_{\text{in}} d\omega_{\text{out}}. \]

Assuming that the light source is facing toward the reflective surface, solid angle \( d\omega_{\text{in}} \) can be expressed as \( dA_{\text{light}} / r_{\text{in}}^2 \), where \( r_{\text{in}} \) is the distance between the light source and the reflective surface.
Similarly, solid angle $d\omega_{\text{out}}$ can be expressed as $dA\cos\theta_{\text{out}}/r^2_{\text{out}}$, where $r_{\text{out}}$ is the distance between the reflective surface and the irradiance sensor. Thus, we get:

$$H = L_{\text{in}} f_r(\vec{\omega}_{\text{in}} \to \vec{\omega}_{\text{out}}) \cos \theta_{\text{in}} \cdot dA_{\text{light}}/r^2_{\text{in}} \cdot dA \cos \theta_{\text{out}}/r^2_{\text{out}}$$  (1.1)

Exchanging the position of the light source and the sensor, we get the following irradiance value:

$$H' = L_{\text{in}} f_r(\vec{\omega}_{\text{out}} \to \vec{\omega}_{\text{in}}) \cos \theta_{\text{out}} \cdot dA_{\text{light}}/r^2_{\text{out}} \cdot dA \cos \theta_{\text{in}}/r^2_{\text{in}}$$  (1.2)

Comparing Equations 1.1 and 1.2 we can conclude that the BRDF is symmetric if and only if the measured irradiances are equal ($H = H'$).

### 1.1.4 Physically plausible BRDF models

Material models are usually defined by the BRDF function that describes the chance of reflection for different pairs of incoming and outgoing light directions. Material models can be physically plausible, that is, they do not violate the basic rules of optics, including the Helmholtz reciprocity and energy conservation.

For example, microfacet based material models are usually physically plausible. These models assume that the material consists of tiny, arbitrarily oriented ideal mirrors. The orientation of the mirrors is handled with a probabilistic approach, while reflection from the mirrors is described by Fresnel’s equations. A microfacet based specular BRDF model usually has the following product form \cite{CT81, HTSG91}:

$$P(\vec{N} \cdot \vec{H}) \cdot G(\vec{L}, \vec{N}, \vec{V}) \cdot F(\vec{L}, \vec{H}, \lambda).$$

where $\lambda$ is the wavelength of light, $\vec{N}$ is the surface normal, $\vec{L}$ is the illumination direction, $\vec{V}$ is the viewing direction and $\vec{H}$ is the halfway vector between the illumination and viewing directions ($\vec{H} = (\vec{V} + \vec{L})/(|\vec{V} + \vec{L}|)$).

![Figure 1.8: Notations of the microfacet based BRDF model: (a) $\vec{N}$ is the surface normal, and (b) $\vec{H}$ is the normal of those microfacets that can ideally reflect from the illumination direction $\vec{L}$ to the viewing direction $\vec{V}$.](image)

**Microfacet distribution**

Microfacet distribution $P(\vec{N} \cdot \vec{H})$ defines the roughness of the surface by describing the probability density of those microfacets that can ideally reflect from the illumination to the viewing direction. (Note that the normal of these microfacets is exactly the halfway vector $\vec{H}$, as demonstrated in Figure 1.8). The theory of scattering of electromagnetic waves led to the application of the Beckmann distribution \cite{Gla95}:

$$P(\delta) = \frac{1}{m^2 \cos^4 \delta} \cdot e^{-\tan^2 \frac{s}{m^2}},$$

where parameter $m$ describes the roughness of the surface. If it were too time consuming to evaluate this formula, we can select other probability densities as well which are maximal at $\delta = 0$. A simple cosine lobe is usually sufficient, where exponent $s$ is the roughness parameter:

$$P(\delta) = \frac{s + 2}{2\pi} \cdot \cos^s \delta.$$
Geometric term

Geometric term \( G(\vec{L}, \vec{N}, \vec{V}) \) shows how much of these ideal reflections can actually occur, and is not blocked by another microfacet before (shadowing) or after (masking) reaching the appropriately oriented microfacet. The geometric term is independent of the material properties, and it causes a general reduction of the specular term for certain illumination and viewing directions. Such reduction should be compensated in the diffuse reflection, since we can assume that photons reflected on the microfacets by multiple times contribute to the diffuse (also called matte) part. (This also means that the matte and specular parts are not independent, as assumed by most of the BRDF models, but are coupled by an appropriate weighting, which depends on the viewing direction [APS00, KSK01]).

For example, the Cook-Torrance BRDF model [CT81] proposes the following geometric term:

\[
G(\vec{L}, \vec{N}, \vec{V}) = \frac{1}{2(\vec{N} \cdot \vec{L})} \cdot \min \left\{ \frac{1}{2}, \frac{(\vec{N} \cdot \vec{H})}{(\vec{V} \cdot \vec{H})}, \frac{2(\vec{N} \cdot \vec{H})(\vec{N} \cdot \vec{E})}{(\vec{V} \cdot \vec{H})}, \frac{2(\vec{N} \cdot \vec{H})(\vec{N} \cdot \vec{E})}{(\vec{V} \cdot \vec{H})} \right\}
\]

Note that the geometric term is rather complicated, and usually depends on all input vectors, thus its tabulation is out of question. To cope with this problem, [KSK01] suggested a simplification to the geometric term that factors out the dependence of the light, normal and viewing vectors, and proposed a model that depends only on a single scalar.

\[
G(\vec{L}, \vec{N}, \vec{V}) \approx \frac{1}{2(1 + \vec{L} \cdot \vec{V})} = \frac{1}{4(\vec{L} \cdot \vec{H})^2}
\]

The simplified geometric term is much easier to compute. More importantly, it depends only on the cosine of a single angle that is between the halfway vector and the light vector. Note that the same cosine angle shows up in the Fresnel term as well, thus the two factors can be combined, and we can store their product in a look up table.

Fresnel term

Finally, Fresnel term \( F(\vec{L} \cdot \vec{H}, \lambda) \) equals to the probability that a photon is reflected from the microfacet considered as an ideal mirror. For a single ideal mirror, the Fresnel term depends on the angle of light incidence, usually denoted by \( \theta \). In the microfacet model, the normal of those microfacets that can ideally reflect from illumination direction \( \vec{L} \) to viewing direction \( \vec{V} \) is exactly the halfway vector \( \vec{H} \) (Figure 1.8). Thus, the incidence angle for these microfacets is the angle of the halfway vector \( \vec{H} \) and the light direction \( \vec{L} \). (Thus, let us call the angle between \( \vec{H} \) and \( \vec{L} \) the microfacet incidence angle, while the angle between \( \vec{N} \) and \( \vec{L} \) is the (surface) incidence angle.)

The Fresnel term is the only factor that is wavelength dependent, thus it is the primary source of coloring. This is why the accurate computation of the Fresnel term is so important to present realistic look for materials.

The Fresnel effect itself is the result of the interaction between the light and the material. In addition to the photon concept, another way of understanding light is the wave model. As light travels in a material, the electromagnetic disturbance of the wave forces the electrons of the material to oscillate as well, which in turn radiate electromagnetic waves [FLM69]. This new wave can be observed as a reflected wave on the boundary of different materials, and also combines with the original wave inside the material. This combination makes the original wave apparently change its speed, i.e. the speed will be different from the speed in vacuum. Furthermore, maintaining forced oscillation requires energy, thus the wave will die away in the material, and the wave amplitude diminishes exponentially. Let us consider a one dimensional wave of the electric field in point \( x \) and time \( t \), for example:

\[
\Psi(x, t) = A \cdot e^{-\delta x} \cdot e^{j\omega(t-x)}
\]
where $A$ is the initial amplitude, $\delta$ is the attenuation factor, $j$ is the imaginary unit on the complex plane, $\omega$ is the phase frequency of the wave, and $v$ is the speed of the wave inside the material. Expressing the speed of the electromagnetic wave relative to the speed in vacuum, i.e. $v = v_0/n$, where $n$ is the index of refraction we can write:

$$\Psi(x, t) = A \cdot e^{-\delta x} \cdot e^{j\omega\left(t - \frac{xn}{v_0}\right)}.$$  

After substituting identity $-\delta x = jj\delta x = j\omega \frac{\delta x}{\omega}$, we can factor out $j\omega$ in the exponent:

$$\Psi(x, t) = A \cdot e^{j\omega\left(t - \frac{xn}{v_0} + \frac{j\delta x}{\omega}\right)}.$$  

Expressing the phase frequency as $\omega = 2\pi f = 2\pi v_0/\lambda$, where $\lambda$ is the wavelength measured in vacuum, we get:

$$\Psi(x, t) = A \cdot e^{j\omega\left(t - \frac{xn}{v_0} + \frac{j\delta x}{2\pi}\right)}.$$  

That is:

$$\Psi(x, t) = A \cdot e^{j\omega\left(t - \frac{xn}{v_0} (n - j\frac{\delta \lambda}{2\pi})\right)}.$$  

Introducing $k = \delta \lambda / 2\pi$ we get:

$$\Psi(x, t) = A \cdot e^{j\omega\left(t - \frac{xn}{v_0} (n - jk)\right)}.$$  

The $n - kj$ term is called the complex refraction index of the material. According to the definition above, its real part $n$ describes how fast the light traverses inside the material, while the imaginary part $k$, called the extinction coefficient, defines how quickly the wave dies away (Figure 1.9).

![Wave extinction at a material boundary at wavelength $\lambda = 400\text{nm}$ for different extinction coefficient values: $n = 1.2$, $k = 0$ (dashed lines), (a) $n = 1.2$, $k = 0.1$ (solid line), and (b) $n = 1.2$, $k = 0.5$ (solid line). Note that the wavelength inside the material is larger than 400nm. The dotted lines show attenuation factor $e^{-\delta x}$.](image)

Based on the explained model of the forced oscillation of electrons, it is possible to derive a similar wave equation, thus the complex refraction index can also be expressed by material constants. A simple and practical formula is [FLM69]:

$$\Psi(x, t) = A \cdot e^{j\omega\left(t - \frac{xn}{v_0} (n - jk)\right)}.$$  

![Figure 1.9: Wave extinction at a material boundary at wavelength $\lambda = 400\text{nm}$ for different extinction coefficient values: $n = 1.2$, $k = 0$ (dashed lines), (a) $n = 1.2$, $k = 0.1$ (solid line), and (b) $n = 1.2$, $k = 0.5$ (solid line). Note that the wavelength inside the material is larger than 400nm. The dotted lines show attenuation factor $e^{-\delta x}$.](image)
\[ n(\omega) - jk(\omega) = 1 + \frac{q^2}{2\epsilon_0 m} \sum_k \frac{Nf_k}{\omega_k^2 - \omega^2 + j\delta_k \omega}, \]  

where \( q \) and \( m \) are the charge and mass of the electron, \( \epsilon_0 \) is the permittivity constant, and \( N \) is the number of atoms in a unit volume. Parameter \( k \) runs over the natural angular frequencies \( \omega_k \) of the electrons in the atoms of the material, \( \delta_k \) is the dumping coefficient of the oscillation due to energy loss, and \( f_k \) is the weight belonging to this resonance frequency.

We can make the following observations by examining this formula. If the natural frequencies are much greater than the frequency of the light, then the imaginary part is negligible and the real part is greater than 1. This is the case for most of the dielectric materials. However, if the natural frequency is close to the frequency of the visible light, then the imaginary part can be large, and the real part can be either smaller or greater than 1 (this also means that the phase speed inside the material can be larger than the speed of vacuum). Metals do not let the electromagnetic wave travel inside the material (as demonstrated in the Faraday cage experiment). The wave dies away quickly due to the larger extinction coefficient.

Applying the Maxwell equations [Gla95], it is possible to compute the ratio of the amplitude of the electric field reflected off a plane boundary and the amplitude of the incident wave. In rendering we work with power (or radiance, which is the density of power). Since the power of the electric field reflected off a plane boundary and the amplitude of the incident wave. In the final result for these cases, the complex refraction index also plays a crucial role:

\[ F_\parallel = \left| \frac{\cos \theta - (n + kj) \cos \theta'}{\cos \theta + (n + kj) \cos \theta'} \right|^2, \quad F_\perp = \left| \frac{\cos \theta' - (n + kj) \cos \theta}{\cos \theta' + (n + kj) \cos \theta} \right|^2. \]

In these formulae, \( \theta \) is the angle of incidence and \( \theta' \) is the angle of refraction. Assuming that the light is not polarized, the ratio of the reflected and incident radiance can be expressed by the Pythagoras theorem:

\[ F = \frac{F_\parallel + F_\perp}{2}. \]

Since on a given wavelength the power and radiance are proportional to the number of photons, the resulting Fresnel term can also be interpreted as the probability that a particular photon is reflected from the boundary of the material.

Using Snell’s law [Gla95], the angle of refraction can be eliminated from the formula, thus the Fresnel term depends on three arguments: \( n \) and \( k \) are the real and imaginary parts of the refraction index, respectively, and \( \theta \) is the angle of incidence for the given microfacet. Expressing the absolute value of the complex numbers, the following form of the Fresnel formula can be obtained [Gla95].

\[ F_\parallel = \frac{a^2 + b^2 - 2a \cos \theta + \cos^2 \theta}{a^2 + b^2 + 2a \cos \theta + \cos^2 \theta}, \]

\[ F_\perp = \frac{a^2 + b^2 - 2a \sin \theta \tan \theta + \sin^2 \theta \tan^2 \theta}{a^2 + b^2 + 2a \sin \theta \tan \theta + \sin^2 \theta \tan^2 \theta}, \]

where \( a \) and \( b \) are defined by the following equations:

\[ 2a^2 = \sqrt{(n^2 - k^2 - \sin^2 \theta)^2 + 4n^2k^2} + (n^2 - k^2 - \sin^2 \theta), \]

\[ 2b^2 = \sqrt{(n^2 - k^2 - \sin^2 \theta)^2 + 4n^2k^2} - (n^2 - k^2 - \sin^2 \theta). \]
The computation of the exact Fresnel function is quite expensive even on the graphics hardware. In real time applications we may need its approximation, which is much cheaper to evaluate, but is accurate enough not to destroy image quality [NV02]. Such Fresnel approximations are proposed in Chapter 3.

### 1.1.5 Global illumination methods

Rendering requires the identification of those light paths that connect the light sources to the eye via reflections and refractions, and then the computation of the sum of their contribution [SKSS08]. Let us consider the rendering equation, which expresses the radiance of an arbitrary point \( \mathbf{x} \) as a function of the radiance values of illuminating points \( \mathbf{y} \).

\[
L(\mathbf{x}, \omega) = L^e(\mathbf{x}, \omega) + \int_{\Omega} L(\mathbf{y}, \omega') f_1 \cos \theta_x' \, d\omega',
\]

where \( f_1 = f_r(\omega', \mathbf{x}, \omega) \) and \( L^e \) is the emitted radiance. Radiance \( L(\mathbf{y}) \) at illuminating point \( \mathbf{y} \) is not known, but we can express it using the rendering equation inserting \( \mathbf{y} \) into \( \mathbf{x} \),

\[
L(\mathbf{y}, \omega') = L^e(\mathbf{y}, \omega') + \int_{\Omega} L(\mathbf{y'}, \omega_2) f_2 \cos \theta_{y1} \, d\omega_2,
\]

where \( f_2 = f_r(\omega_2', \mathbf{y}, \omega') \). Using this the radiance at point \( \mathbf{x} \) can be expressed by the radiance values of points that can be reached by a single reflection:

\[
L(\mathbf{x}, \omega) = L^e(\mathbf{x}, \omega) + \int_{\Omega} f_1 \cos \theta_x' \left( L^e(\mathbf{y}, \omega') + \int_{\Omega} L(\mathbf{y}, \omega_2) f_2 \cos \theta_{y1} \, d\omega_2 \right) \, d\omega'.
\]

Repeating the same step, the radiance caused by single, double, triple, etc. reflections can be obtained. It means that to consider not only single bounce but also multiple bounce light paths, the integral of the rendering equation should be recursively evaluated at illuminating points \( \mathbf{y}, \mathbf{y'}, \ldots \), which leads to a sequence of high-dimensional integrals:

\[
L = L^e_1 + \int_{\Omega} f_1 \cos \theta_x' \left( L^e_2 + \int_{\Omega} f_2 \cos \theta_{y1} \left( L^e_3 + \ldots \right) \, d\omega_2 \right) \, d\omega_1.
\]

From mathematical point of view, global illumination rendering means the solution of the rendering equation, or alternatively, the evaluation of the sequence of high-dimensional integrals for the representative wavelengths.

In order to avoid the dimensional explosion of classical quadrature rules, Monte-Carlo or quasi-Monte Carlo integration techniques can be applied. The fundamental idea of Monte-Carlo quadrature is to convert integrals to expected values estimated by the average of random samples:

\[
I = \int_V f(z) \, dz = \int_V \frac{f(z)}{p(z)} \cdot p(z) \, dz = E \left[ \frac{f(z)}{p(z)} \right] \approx \frac{1}{M} \sum_{i=1}^{M} \frac{f(z_i)}{p(z_i)}, \tag{1.8}
\]

where \( z = [z_1, \ldots, z_d] \) is the integration variable, \( p(z) \) is a probability density in \( d \)-dimensional integration domain \( V \), and points \( z_i \) are selected according to this probability density.

In order to reduce the variance (i.e. the error) of the estimate, the probability density should mimic the integrand, that is, it is worth applying probability distributions that are large where the integrand is large and small where the integrand is negligible. This variance reduction technique is called importance sampling [Sobh91]. For example, in case of random walk algorithms, the probability density should be at least approximately proportional to the
contribution of the light path. However, since a path carries light on several wavelengths, the contribution is a vector, thus the selection of a “proportional” probability density requires further considerations. That is, we have to define a scalar contribution function that represents, for example, the luminance of the carried light. Then, importance sampling should obtain light paths with a probability density that mimics the scalar contribution function.

Since the contribution of the path is the product of the light source emission at the beginning and the cosine weighted BRDFs at the visited points, local importance sampling constructs the probability density step-by-step taking into account these factors independently. That is, light source sampling [SWZ96] finds a point with a probability that is proportional to the emission, BRDF sampling [Kaj86] obtains a random direction that mimics the cosine weighted BRDF, and Russian roulette generates the next step or obtains zero contribution with the probability of the reflection or absorption, respectively.

Even effective importance sampling techniques require a large number of light paths to obtain an accurate pixel color. However, high resolution images have many pixels, which results in unacceptable rendering times. To speed up the computation, we can try to reuse paths or subpaths generated for a pixel for the computation of other pixels as well [BSH02, SK99]. A particularly effective approach based on this idea is the virtual light sources method, which will be discussed in Chapter 6.

1.2 Graphics processing units

To implement an illumination algorithm efficiently, we must not forget the recent advance of the graphics cards and their processing units (GPU). The average GPU computation time of a pixel, including transformations, local illumination, projection, clipping, texturing, blending and visibility testing, is about a few nanoseconds, i.e. close to the time of a single memory cycle. This incredible speed is the result of a massive pipelining, parallelization and special Arithmetic–Logical Units (ALUs) along the pipeline. For example, a GeForce 3 GPU may operate with 800 pipeline levels, while the Intel P4 processor has at most 20. The pipeline is broken to parallel channels at computation-intensive parts, and all ALUs along the pipes can handle four floating point values in parallel and execute complex operations such as the multiplication of a $4 \times 4$ matrix and a 4-dimensional vector.

Two phases of the pipeline have become programmable, the vertex and pixel processing units. Vertex processing, which was originally responsible for vertex transformation and local illumination computations at the vertices, can be controlled by a vertex shader program. Similarly, pixel processing, which was originally responsible for (multi)texturing, can be governed by a pixel shader program. The programmability together with the high speed have made the GPU a primary candidate for computation-intensive tasks, which might be non-local illumination and even non-graphics problems. Examples include ray-shooting [PDC+03], progressive radiosity [CHL03], Voronoi diagrams, FFT and the solution of linear equations, etc. (see http://www.gpgpu.org).

1.2.1 Shaders beyond the standard pipeline operation

Programmable vertex and fragment shaders offer a higher level of flexibility on how the data from the vertex buffer is processed, and how shading is performed. However, the basic pipeline model remains the same: a vertex is processed, the results are linearly interpolated, and they are used to find the color of a fragment. The flexibility of the programmable stages will allow us to change the illumination model: implement per-fragment (per-pixel) lighting, consider more and different light sources, combine several textures or render unfolded triangle charts instead of the models themselves, among the infinite number of other possibilities.

What programmable vertex and fragment shaders alone cannot do is non-local illumination. All the data passed to shaders are still only describing local geometry and materials, or global constants, but nothing about other pieces of geometry. When a point is shaded with a global
The illumination algorithm, its radiance will be the function of all other points in the scene. From a programming point of view this means that the basic pipeline model remains the same. While this is granted in CPU based ray-tracing systems [WKB+02, WBS03], the stream processing architecture of current GPUs fundamentally contradicts this requirement. When a point is shaded on the GPU we have just its limited amount of local properties stored in registers, and may access texture data. Thus the required global properties of the scene must be stored in textures.

In order to make use of the processing power of the graphics hardware, textures themselves have to be computed on the GPU. The render-to-texture feature allows this: anything that can be rendered to the screen, may also be stored in a texture. Such texture render targets may also require depth and stencil buffers. Along with programmability, various kinds of data may be computed to textures [SKSS08]. These data may also be stored in floating point format in the texture memory, unlike the color buffer which usually stores data of 8 bit precision.

To use textures generated by the GPU, the rendering process must be decomposed to passes, where one pass may render into a texture and may use the textures generated by the previous passes. Since the reflected radiance also depends on geometric properties, these textures usually contain not only conventional color data, but they also encode geometry and prepared, reusable illumination information as well.

Programmability and render-to-texture together make it possible to create some kind of processed representation of geometry and illumination as textures. These data can be accessed when rendering and shading other parts of the scene. This is the key to addressing the self-dependency of the global illumination rendering problem. In all GPUGI algorithms [SKSS06], i.e. in algorithms that address global illumination on the GPU, we use multiple passes to different render targets to capture some aspects of the scene like the surrounding environment, the shadowing or the refracting geometry, illumination due to light samples, etc. These passes belong to the illumination information generation part of rendering. In a final pass, also called final gathering, scene objects are rendered to the frame buffer making use of previously computed information to achieve non-local illumination effects like shadows, reflections, caustics, or indirect illumination.

Pases of the illumination information generation part are responsible for preparing the reusable illumination information and storing it in the texture memory, from which the final gathering part produces the image for the particular camera. To produce continuous animation, the final gathering part should run at high frame rates. Since the scene may change, the illumination information generation should also be repeated. However, if the illumination information is appropriately defined, then its elements can be reused for many points and many frames. Thus the illumination information data structure is compact and might be regenerated at significantly lower frequency than the final gathering frame rate.
Chapter 2

Previous Work

In this chapter we briefly summarize those closely related previous results that were used and extended by our work.

2.1 Specular Effects on the GPU

Computing specular reflections and refractions involves tracing a primary ray up to the reflective/refractive surface, computing the reflected or refracted ray, and recursively tracing the successive rays reflecting or refracting on specular surfaces until a non-specular surface is reached. Most of the GPU based techniques take advantage of the GPU rasterizing capabilities to compute the first intersection point with the primary rays. In order to trace secondary rays on the GPU, we should ensure that GPU processors have access to the scene geometry. Vertex and fragment shaders may access global (i.e. uniform) parameters and textures, thus this requirement can be met by storing the scene geometry in uniform parameters or in textures.

The computation of real-time specular effects on the GPU can divided into four categories based on the different ways of representing the scene and handling ray intersections with the aid of the GPU:

- **Ray tracing in the original geometry** where traditional, object space accurate ray tracing algorithms are adapted to work on the GPU, which is possible due to the high parallelism inherent to these computations.
- **Geometry transformation** where the geometry of the scene is transformed to build the virtual scene that is seen on the reflector surface.
- **Image based lighting and rendering** where a part of the scene is represented by images, which are looked up with the parameters of the ray.
- **Ray tracing in the sampled geometry** where the scene is sampled in such a way that it is feasible to store samples in textures and compute the intersection of a ray with reasonably few texture fetches.

2.1.1 Ray tracing in the original geometry

Ray tracing is a classical image synthesis approach, which offers accurate non-local lighting phenomena, including specular effects. Efficient implementations of ray tracing algorithms that work with the original geometry need fast ray–object intersection calculation and complex space partitioning data structures to quickly exclude objects that are surely not intersected by a ray. Many GPU data structures and algorithms have been developed having a CPU implementation in mind [AW87, Arv91, AK87, FTK86, Gla89, OM87, Hav01, WKB+02, Sze03,
RSH05, HHS06. GPU ray tracing have benefited from the advancements in accelerating CPU
ray tracing, but not all techniques run equally well on both platforms. Most ideas had to be
adapted to account for the special nature of GPUs.

Intersection calculation
The fundamental operation of ray tracing is evaluating the ray–object (ray–triangle) inter-
section test. Depending on the acceleration scheme, a high number of intersection tests may
have to be performed, making it a time-critical operation. The test can be speeded up by fast
intersection calculation [MT97] or by applying pre-computation and passing other information
instead of the triangle vertices [Wal04, SKSS06, Ken07].

Spatial acceleration schemes
As GPUs were advancing, the GPU implementation of more and more complex acceleration
schemes became possible. Additionally, parallel algorithms for the construction of these data
structures have also been proposed, making them applicable for real-time rendering of dynamic
scenes. A detailed state-of-the-art analysis focusing on the construction and traversal of such
object hierarchies for dynamic scenes (considering both CPU and GPU methods) has been
given in [WMG07].

The ray engine
The ray engine [CHH02] method uses the rasterization hardware and the fragment shader
to initiate and evaluate the ray–triangle intersection tests. The method associates a ray to
every pixel of the render target. The origin and direction of rays to be traced are stored in
textures that have the same dimensions as the render target. One after the other, a single ray
casting primitive is taken, and it is rendered as a full-screen quad (FSQ), i.e. a quadrilateral
that covers all pixels of the viewport, with the primitive data attached to the quad vertices.
Since the pixel shaders for every pixel will receive the primitive data, and can also access the
ray data via texture reads, the ray–primitive intersection calculation can be performed in the
shader. Then, using the distance of the intersection as a depth value, a depth test is performed
to verify that no closer intersection has been found yet. The pitfall of the ray engine method is
that it implements the naive ray casting algorithm of testing every ray against every primitive.
Since the method follows a per primitive processing scheme instead of the per ray philosophy,
it is also worth building an acceleration hierarchy over the rays, not over the objects [Sze06].

2.1.2 Geometry transformation
Searching the hit point in the environment after finding the reflection point on the shaded
object is not the only way to render reflections. Searches can also be organized differently,
for example, we can start at the vertices of the reflected environment, and search for the
reflection points, i.e. identify those points of the shaded object that reflect the input points.
This means searching on or projecting onto the shaded reflector surface rather than searching
on the reflected environment surface.

The first attempts focused on planar reflectors. Multi-pass rendering [DB97, MBGN00]
used a modified projection matrix to project a reflected object through the plane of each
reflector, and mapped its image onto the reflector plane. Reflections are implemented by first
rendering the scene without the mirrored surfaces. A second rendering pass is performed for
each reflected viewpoint, and the resulting image is applied to the reflector. This process can
be repeated recursively in scenes with multiple reflectors. Stencil operations are arranged so
the reflected images are masked by the stencil buffer. Pre-computed radiance maps [BHWL99]
can be used to avoid the projection step.
For curved reflectors, the situation is more complicated. An interesting solution was proposed by Ofek and Rappoport [OR98]. Their method warps the surrounding scene geometry such that it appears as a correct virtual image when drawn on the reflector.

Since this method transforms all scene geometry and usually requires fine tessellation, it can be either computationally expensive or exhibit accuracy problems. An analytic approach has also been developed [CA00], using a preprocessing step based on path perturbation theory. When objects move, the preprocessing step needs to be repeated.

Schmidt [Sch03] used geometry transformation for both planar and curved refractions. He noted that multi-pass rendering developed for reflections is only approximately correct for refractions, and also proposed a method to handle refractions on curved surfaces.

Guy and Soler [GS04] used geometry transformation to realistically render gem stones, which have planar faces. A light path in a gem stone contains two refractions and arbitrary number of internal reflections in between. Planar reflection transforms triangles to triangles, thus triangles can be exactly handled by geometry transformation, but refraction distorts geometry. This method “linearized” the refraction of the eye ray, i.e. approximated the refraction by a linear transformation similar to reflections. Furthermore, the refraction of the ray connecting the light source was computed by the fragment shader with per pixel accuracy.

2.1.3 Image based lighting and rendering

Environment mapping

A GPU friendly approximation technique is environment mapping, which assumes that the illuminating environment is very (infinitely) far from the rendered objects. Thus, when tracing rays from the virtual objects toward the environment, the location of the ray origin becomes irrelevant, i.e. the hit point becomes independent of the ray origin. In this case rays can be translated to the same reference point, so we get that case back for which the GPU is an optimal tool.

To render a reflective or refractive object, environment mapping takes images about the environment from a point, called reference point, that is in the vicinity of the object, then the environment of the object is replaced by a cube, or by a sphere, which is textured by these images (Figure 2.1). When the incoming illumination from a direction is needed, instead of sending a ray we can look up the result from the images constituting the environment map.

![Figure 2.1: Steps of environment mapping](image)

Environment mapping has been originally proposed to render ideal mirrors in local illumination frameworks [BN76], then extended to approximate general secondary rays without expensive ray-tracing [Gre84, RTJ94, Wil01]. Environment mapping has also become a standard technique of image based lighting [MH84, Deb98].

A fundamental problem of environment mapping is that the environment map is the correct representation of the direction dependent illumination only at a single point, the reference point of the object. For other points, accurate results can only be expected if the distance...
of the point of interest from the reference point is negligible compared to the distance from the surrounding geometry. However, when the object size and the scale of its movements are comparable with the distance from the surrounding surface errors occur, which create the impression that the object is not properly embedded in its environment. However, if the object moves, the environment map can be frequently re-generated from the translated reference point to provide *motion parallax*.

**Multiple environment maps**

In order to increase the accuracy of environment mapping, Cabral et al. [CON99] and Hakura et al. [HS01] used *multiple environment maps*, and for each shaded point they looked up that map where the shaded point is the closest to the reference point of the environment map. However, the transitions between different environment maps are difficult to control and may lead to discontinuities. One elegant way to solve the discontinuity problem is to approximate the hit point by tracing the ray in a proxy geometry, to look up the hit point from the environment maps having centers closest to the hit point, and blending the colors based on the distances [ML03]. In this paper the proxy geometry was the height map of an urban model.

**Proxy geometry**

In order to provide more accurate environment reflections and the missing parallax effect, the geometry of the scene should be taken into account, and the ray tracing process should be executed or at least approximated. A simple approximation is to use some *proxy geometry* [Bre02, Bjo04] (e.g. a sphere or a cube) of the environment, which is intersected by the reflection ray to obtain a point, whose direction is used in the environment map lookup (Figure 2.2). For a fixed and simple proxy geometry, the ray intersection calculation can be executed by the pixel shader of the GPU. However, the assumption of a simple and constant environment geometry creates visible artifacts that make the proxy geometry apparent during animation.

Hargreaves [Har04] applied further simplifications and assumed that the environment is an infinite hemisphere, the object is always at a fixed distance from the base plane of the hemisphere, and reflection rays are always perpendicular to the base plane. These assumptions make even ray tracing unnecessary, but significantly limit the applicability of the algorithm (the target application was a bike racing game).

Popescu et. al [PMDS06] proposed a method that used displacement mapped rectangles as the proxy geometry.

---

**Figure 2.2:** Application of a proxy sphere when tracing reflection ray of origin $\vec{x}$ and direction $\vec{R}$. The environment map is looked up in the direction of the proxy hit.

Hargreaves [Har04] applied further simplifications and assumed that the environment is an infinite hemisphere, the object is always at a fixed distance from the base plane of the hemisphere, and reflection rays are always perpendicular to the base plane. These assumptions make even ray tracing unnecessary, but significantly limit the applicability of the algorithm (the target application was a bike racing game).

Popescu et. al [PMDS06] proposed a method that used displacement mapped rectangles as the proxy geometry.
2.1.4 Ray tracing in the sampled geometry

Instead of storing the original geometry, the scene geometry can also be represented in a sampled form in textures. A conventional texture map encodes a color (i.e. radiance) function over a 2D surface in the 3D space, thus a single texel represents a point of a 2D surface, and the texture coordinate pair navigates in this 2D subset. In order to allow the representation of points of 3D point sets, we can either use 3D textures or store additional coordinates in the texture data that complements the texture coordinate pair. One alternative is to store all three coordinates in the texture data, which is the basic idea of geometry images [GGH02]. On the other hand, if the texture coordinate pair is also directly utilized to identify the represented point, a single additional coordinate is enough. This additional coordinate can be a distance from a reference surface, when it is called the “height”, or from a reference point, when it is called the “depth”.

The oldest application of the idea of using distances from a given point as a sampled representation of the scene is the depth map of shadow algorithms [Wil78]. A shadow map samples just that part of the geometry which is seen from the light position through a window. The depth map generated from the camera — i.e. assuming that the reference point is the eye — is also sufficient in many applications if rays leaving the visible frustum can be ignored [Wym05, WD06, SP07].

Sampled representations based on distances from a surface of simple geometry were applied in displacement mapping [Bli78, SKU08].

2.2 Diffuse and glossy effects on the GPU

According to the rendering equation, the reflected radiance of point \( \vec{x} \) in viewing direction \( \vec{\omega} \) can be expressed by the following integral

\[
L^r(\vec{x} \rightarrow \vec{\omega}) = \int_{\Omega'} L^in(\vec{x} \leftarrow \vec{\omega}') \cdot f_r(\vec{\omega}' \rightarrow \vec{x} \rightarrow \vec{\omega}) \cdot \cos \theta \vec{x} \cdot d\omega',
\]

where \( \Omega' \) is the set of possible illumination directions, \( L^in(\vec{x} \leftarrow \vec{\omega}') \) is the incoming radiance arriving at point \( \vec{x} \) from illumination direction \( \vec{\omega}' \), \( f_r \) is the BRDF, and \( \theta \vec{x} \) is the angle between the surface normal at point \( \vec{x} \) and the illumination direction.

This integral can be estimated by Monte Carlo methods, also called stochastic sampling [Coo86], which involves tracing rays in the reflection directions and the summation of their contribution. Partial analytic techniques can greatly improve the performance [SSSK04]. The integral can also be interpreted as a convolution, which can be speeded up by finite-element approximations. Spherical harmonics are particularly popular in this aspect [KSS02].
Shadow maps
If only one-bounce indirect illumination of a relatively small number of light sources is considered, we can use shadow maps to calculate indirect illumination. As pointed out in [DS05], points where the light bounces for the first time are stored by default in the shadow map. By adding illumination information to the depth values, the indirect illumination, that is, the secondary light transport, can be computed as the reflection of the shadow map.

Diffuse and specular environment maps
The idea of environment mapping can also be applied for glossy and diffuse reflections as well. The usual trick is the convolution of the angular variation of the cosine-weighted BRDF with the environment map during preprocessing [KVHS00, KM00, RH01, HSC+05]. When the object is moving, similarly to ordinary environment maps, the diffuse (or specular) environment maps have to be frequently re-generated. This may cause a performance bottleneck unless the resolution of the environment map is kept low and fast algorithms are applied.

Multiple environment maps
A possible alternative is to use multiple environment maps [GSHG98, ZHL+05], which can be compressed using spherical harmonics [RH01, KAMJ05] or wavelets [ZHL+05]. For example, Greger et al. [GSHG98] calculate and store the direction dependent illumination in the vertices of a bi-level grid subdividing the object scene. During run-time, irradiance values of a point are calculated by trilinearly interpolating the values obtained from the neighboring grid vertices. They reported good results even with surprisingly coarse subdivisions. That is, the smooth irradiance function seems to be especially suitable for interpolation. While Greger et al. used a precomputed radiosity solution to initialize the data structures, Mantiuk et al. [MPM02] calculated these values during run time using an iterative algorithm that simulates the multiple bounces of light.

Unfortunately, the generation and compression of many environment maps require considerable time which is not available during real-time rendering. Thus most of this computation should be done during preprocessing, which imposes restrictions on dynamic scenes.

Global precomputations
Another possible approach is to perform global computations to reduce the computation requirement of many lights. This issue has been addressed by Precomputed Radiance Transfer [SKS02, KAMJ05], by ignoring possibly unimportant lights [War94, SWZ96, WBS03], and by replacing a cluster of lights by a single point light source [WFA+05].
Chapter 3

Fresnel Term Approximations for Metals

Colored reflections are governed by the Fresnel term in the BRDF models, which can be expressed from the refraction index of the material. For metals, where the refraction index is a complex number, the evaluation of the Fresnel function is rather computation intensive. We present an accurate simplification, which can also be applied for metals as well and is simple enough to be implemented on the vertex or pixel shader, and used in games.

3.1 Schlick’s approximation

For many non-metallic materials, the extinction coefficient $k$ is quite small, which allows us to ignore the imaginary part altogether. The assumption of the extinction coefficient being zero has also been made by Schlick, who has found the following simple rational approximation for the Fresnel term [Sch93]:

$$F_{\text{Schlick}} = F_\perp + (1 - F_\perp)(1 - \cos \theta)^5,$$

(3.1)

where $F_\perp$ is the function value at perpendicular illumination (i.e. at $\cos \theta = 1$):

$$F_\perp = \frac{(n - 1)^2}{(n + 1)^2}, \quad 1 - F_\perp = \frac{4n}{(n + 1)^2}$$

This formula provides a fairly good approximation if the extinction coefficient is really zero ($k = 0$), as shown in Figure 3.1.

![Figure 3.1](image-url)

Figure 3.1: Comparing the original Fresnel function (solid) and Schlick’s formula (dashed) for different refraction indices. (a) $n = 1.1$, (b) $n = 1.5$, and (c) $n = 3.0$. 23
Now let us concentrate on metals and other materials having complex refraction indices. For different metals, an overview of the possible \((n, k)\) pairs is given in Figure 3.2, while Figure 3.3 also displays the wavelength dependence explicitly. Note the the real part of the reflection index can also be smaller than 1.

If we apply Schlick’s formula for metals, the result will be unacceptable (Figure 3.4). In the following we shall examine the Fresnel function and propose an improvement to significantly reduce the error of Schlick’s approximation.

Figure 3.2: Complex refraction indices for different metals at wavelengths \(\lambda = 400\text{..}800\text{ nm}\).

Figure 3.3: Refraction indices for different metals, at different wavelengths: (a) real part (b) imaginary part.

Now let us concentrate on metals and other materials having complex refraction indices. For different metals, an overview of the possible \((n, k)\) pairs is given in Figure 3.2, while Figure 3.3 also displays the wavelength dependence explicitly. Note the the real part of the reflection index can also be smaller than 1.

If we apply Schlick’s formula for metals, the result will be unacceptable (Figure 3.4). In the following we shall examine the Fresnel function and propose an improvement to significantly reduce the error of Schlick’s approximation.

Figure 3.4: A copper ring rendered (a) with the original Fresnel term and (b) with Schlick’s approximation.
3.2 The new Fresnel term approximation for metals

3.2.1 Analyzing the Fresnel function

Having a look at the Fresnel function for a fixed $n$ value ($n = 1.5$ in case of Figure 3.5), we can notice that for small extinction coefficients the Fresnel term is monotonous (approximately for $k \leq 2$ in this case), while for larger $k$ values a local minimum appears. However, the maximal $k$ value where the function is still monotonous depends on the actual $n$ value as shown in Figure 3.6. Thus, it seems to be rather difficult to create a rational approximation that catches the dual nature of the original Fresnel function.

![Figure 3.5](image1)

Figure 3.5: The Fresnel function for $n = 1.5$, $k = 0..10$, (a) as a 3D plot, (b) as a series of 2D plots.

![Figure 3.6](image2)

Figure 3.6: For $(n,k)$ pairs inside the plotted curve, the Fresnel function is monotonous. For other $(n,k)$ pairs a local minimum exists in range $\cos \theta = 0..1$.

Instead of establishing a new rational approximation we chose to extend Schlick’s formula. For metals, although Schlick’s formula can reproduce the original function value at $\cos \theta = 0$, it produces an erroneous result at $\cos \theta = 1$, as shown in Figure 3.7a. The simplest way to “fix” this problem is to rescale the approximation.
3.2.2 Rescaling the Schlick’s function

To reduce the error of the approximation we shall rescale Schlick’s formula so that it will obey the value of the original function not only at $\cos \theta = 0$, but at $\cos \theta = 1$ as well (Figure 3.7a). To achieve this, we first “flip” the functions vertically so that both the original Fresnel function and Schlick’s approximation become zero at $\cos \theta = 0$ (Figure 3.7b). For example, by “flipping” Schlick’s approximation we get:

$$1 - F_{\text{Schlick}} = 1 - (F_{\perp} + (1 - F_{\perp})(1 - \cos \theta)^5) =$$

$$= 1 - F_{\perp} - (1 - F_{\perp})(1 - \cos \theta)^5 = (1 - F_{\perp})(1 - (1 - \cos \theta)^5).$$

Now we can freely rescale (multiply) Schlick’s approximation without affecting its value at $\cos \theta = 0$. To determine the appropriate scaling factor, let us calculate the exact and approximated function values at $\cos \theta = 1$ using the notations of Figure 3.7b:

$$F_1 := 1 - F(n, k, \cos \theta = 1) = \frac{4n}{(n + 1)^2 + k^2},$$

$$S_1 := 1 - F_{\text{Schlick}}(n, \cos \theta = 1) = 1 - \frac{(n - 1)^2}{(n + 1)^2} = \frac{4n}{(n + 1)^2}.$$

If we multiply Schlick’s approximation by $F_1/S_1$, we shall get correct values at $\cos \theta = 1$. This scaling factor equals:

$$\frac{F_1}{S_1} = \frac{(n + 1)^2}{(n + 1)^2 + k^2}.$$

Now the modified (rescaled) Schlick’s formula $F^*$ can be expressed as follows:

$$1 - F^*(n, k, \cos \theta) := \frac{F_1}{S_1}(1 - F_{\text{Schlick}}(n, \cos \theta)).$$

Let us “flip” the function back (by subtracting 1 and multiplying by $-1$):

$$F^*(n, k, \cos \theta) := 1 - \frac{F_1}{S_1}(1 - F_{\text{Schlick}}(n, \cos \theta)) = 1 - \frac{F_1}{S_1}(1 - F_{\perp})(1 - (1 - \cos \theta)^5) =$$

$$= 1 - \frac{(n + 1)^2}{(n + 1)^2 + k^2} \cdot \frac{4n}{(n + 1)^2} \cdot (1 - (1 - \cos \theta)^5) =$$
\[ = 1 - \frac{4n}{(n+1)^2 + k^2} \cdot (1 - (1 - \cos \theta)^5) = 1 - \frac{4n}{(n+1)^2 + k^2} + \frac{4n}{(n+1)^2 + k^2}(1 - \cos \theta)^5 = \]
\[= \frac{(n-1)^2 + k^2}{(n+1)^2 + k^2} + \frac{4n}{(n+1)^2 + k^2}(1 - \cos \theta)^5. \]

Introducing
\[F^*_\perp = \frac{(n-1)^2 + k^2}{(n+1)^2 + k^2},\]
we get a formula similar to Schlick’s original proposal:
\[F^*(n, k, \cos \theta) := F^*_\perp + (1 - F^*_\perp)(1 - \cos \theta)^5. \quad (3.2)\]

The resulting formula \(F^*\) obeys the values of the original Fresnel function at \(\cos \theta = 0\) and \(\cos \theta = 1\), thus it is able to deal with complex refraction indices and is simple enough for practical applications.

After examining the relative error of the modified approximation (Figure 3.8) we can conclude that this simple modification enables us to extend the original Schlick’s formula to complex refraction indices without significant increase of the relative error (compare error for \(k = 0\) and \(k > 0\)). Note that for \(k = 0\) we get back Schlick’s original formula.

![Figure 3.8: (a) Absolute (solid) and relative error (dashed) of Schlick’s formula for different \(n\) values. (b) Relative error of the rescaled Schlick’s formula for different \(n\) and \(k\) values.](image)

### 3.2.3 Error compensation

To further improve our approximation, we shall compensate the difference between the original Fresnel function \(F\) and our new approximation \(F^*\) with a rational approximation that is simple enough to enable fast calculation and its shape is close to the shape of the error function. After examining the error term \(F - F^*\) (Figure 3.9a) we chose the following approximation:
\[-ax(1 - x)^\alpha \quad (3.3)\]

As an illustration, Figure 3.9b shows this rational approximation for different \(a\) and \(\alpha\) values. Note that the characteristics of the rational term is quite similar to the error term.

Since the derivative of the rational term equals to \(-a\) at \(\cos \theta = 0\), it seems to be reasonable to determine the value of the parameter \(a\) by calculating (or approximating) the derivative of the original error term at \(\cos \theta = 0\).
Figure 3.9: (a) Difference between the exact Fresnel term $F$ and our rescaled proposal $F^*$ for different metals at multiple wavelengths. (b) Rational approximation $-ax(1 - x)^\alpha$ with different parameter values.

To find an $\alpha$ value so that the approximation exactly reproduces the local minimum of the original error term seems to be a hard task since it would require the derivation of the quite complicated Fresnel term. However, for most $(n, k)$ values, the local minimum of the error term is located somewhere in the range of 0.1..0.15. (The only exceptions are those $(n, k)$ pairs where both $n$ and $k$ are small, but in these cases — as we have seen earlier — the error term is negligible.)

Our proposal is to use an approximation that obeys the value of the original error function at $x = 0.15$ (which is close to its local minimum), that is:

$$-0.15a(1 - 0.15)^\alpha = F(n, k, 0.15) - F^*(n, k, 0.15)$$

$$(1 - 0.15)^\alpha = \frac{F(n, k, 0.15) - F^*(n, k, 0.15)}{0.15a}$$

$$\alpha = \frac{\ln |(F(n, k, 0.15) - F^*(n, k, 0.15))/0.15a|}{\ln(1 - 0.15)}$$

Using these formulae, we can dramatically improve the quality of the approximation (Figure ??).

Values $a$ and $\alpha$ should be regarded as derived material properties. When the CPU program instructs the GPU to use a different material model, it should compute these parameters and download them to the GPU as uniform variables.
3.3 Implementation

Based on the Fresnel equations (Equation 1.5 and Equation 1.6), the original Fresnel function can be calculated as follows:

```cpp
float3 CalcFresnel(float3 n, float3 k, float ctheta)
{
    float ctheta2 = ctheta*ctheta;
    float stheta2 = 1 - ctheta2;
    float stheta = sqrt(stheta2);
    float3 n2 = n*n;
    float3 k2 = k*k;
    float3 n_k_stheta = n2-k2-stheta2;
    float3 n_k_stheta2 = n_k_stheta * n_k_stheta;
    float3 sq = sqrt( n_k_stheta2 + 4*n2*k2);
    float3 a2 = ( sq + n_k_stheta )/2;
    float3 a = sqrt(a2);
    float3 b2 = ( sq - n_k_stheta )/2;
    // F ||
    float3 Fs = (a2 + b2 - 2*a*ctheta + ctheta2 ) /
                (a2 + b2 + 2*a*ctheta + ctheta2);
    float3 c = a2 + b2 + stheta2*stheta2/ctheta2;
    float3 d = 2*a*stheta2/ctheta;
    // F _|_
    // float3 Fp = Fs * ( (c-d) / ( c+d ));
    // float3 F = (Fs + Fp) / 2;
    // an equivalent form, where Fp is left implicit
    float3 F = Fs * ( c / ( c+d ));
    return F;
}
```

Schlick’s approximation, our rescaled proposal, and our error compensated proposal can be implemented as follows. Note that in case of error compensation, additional parameters \( a \) and \( \alpha \) are also passed.

```cpp
float3 CalcSchlick(float3 n, float ctheta) {
    return ( (n-1)*(n-1) + pow(1-ctheta,5) * 4*n ) / ( (n+1)*(n+1) );
}

float3 CalcModSchlick(float3 n, float3 k, float ctheta) {
    return ( (n-1)*(n-1) + k*k + pow(1-ctheta,5) * 4*n ) / ( (n+1)*(n+1) + k*k );
}

float3 CalcModCompSchlick(float3 n, float3 k, float3 a, float3 alpha, float ctheta) {
    return CalcModSchlick(n,k,ctheta) - a*ctheta*pow(1-ctheta,alpha);
}
```

Finally, let us illustrate how these functions can be embedded into the shader code. While parameter \( \cos \theta \) (ctheta) is derived from illumination direction \( L \) and viewing direction \( V \), parameters \( n \), \( k \), \( a \), and \( \alpha \) should be passed as uniform variables.

```cpp
float3 H = normalize( L+V ); // halfway vector
float ctheta = dot(H,L); // microfacet incidence angle
float3 fresnelcolor = CalcFresnel(n,k,ctheta);
//float3 fresnelcolor = CalcSchlick(n,ctheta);
//float3 fresnelcolor = CalcModSchlick(n,k,ctheta);
//float3 fresnelcolor = CalcModCompSchlick(n,k,a,alpha,ctheta);
```
3.4 Results

Figures 3.10 and 3.11 compare the original Fresnel term, our rescaled proposal, and our error compensated proposal both visually and analytically. The functions are evaluated on three representative wavelengths (435 nm, 550 nm, 700 nm for blue, green and red, respectively). After examining the figures the following results can be concluded. In case of the rescaled proposal (i.e. without error compensation) the maximum relative error is approximately 2%, 5% and 6% for silver, copper and gold, respectively. However, in case of aluminum, the difference between the original (non-monotonous) function and our monotonous approximation is substantial, resulting in a relative error of nearly 18%. The difference is clearly visible on the colorful stripes that compare the two functions for different microfacet incidence angles, especially in range $\cos \theta = 0 \ldots 0.25$, where the relative error exceeds 10%, and for $\cos \theta = 0.1$, where the relative error is maximal. Thus, in case of aluminum, error compensation is a reasonable choice.

Although there is a clearly visible difference between the exact value and the rescaled approximation on the colorful stripes of Figures 3.10 and 3.11, real rendering results show only subtle differences between the two methods. For example, when we display the upper and lower half of the same sphere with the original Fresnel function and our approximation, respectively, no clear distinction between the two halves can be made (Figure 3.12).

A possible explanation is that the “problematic” region where the relative error is large (i.e. $\cos \theta = 0 \ldots 0.25$) corresponds to large microfacet incidence angles ($\theta > 75^\circ$). Especially, the maximal relative error ($\cos \theta = 0.1$) corresponds to $\theta = 84^\circ$ as shown in Table 3.1.

<table>
<thead>
<tr>
<th>$\cos \theta$</th>
<th>0</th>
<th>0.05</th>
<th>0.1</th>
<th>0.15</th>
<th>0.2</th>
<th>0.25</th>
<th>0.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\theta$</td>
<td>90°</td>
<td>87°</td>
<td>84°</td>
<td>81°</td>
<td>78°</td>
<td>75°</td>
<td>73°</td>
</tr>
</tbody>
</table>

Table 3.1: Angles corresponding to small cosine values.

Large microfacet incidence angles imply that the angle between illumination direction $\vec{L}$ and viewing direction $\vec{V}$ is also large. In other words, the object to be illuminated is located somewhere between the camera and the light source. That is, only a minor part of the object is visible from both the camera and the light source (note that the maximal relative error corresponds to the second image of Figure 3.12). This small region is exposed to grazing light, that is, light intensity may be large, thus it is difficult to discover the error of the approximation. Even with proper zooming and scaling, the differences can hardly be seen (Figure 3.13).

3.5 Conclusions

We proposed a new approximation of the Fresnel function. Unlike previous approaches, we did not assume that the imaginary part of the refraction index is negligible, thus our model can be applied for a wider range of materials, e.g. metals. Our approximation produces a visible error compared to the original Fresnel function only if the object is placed between the camera and the light source.

According to our tests, the proposed simplification does not reduce image quality, but reduces rendering times to their third. Table 3.2 summarizes the relative computation times of the original Fresnel function and its approximations:

<table>
<thead>
<tr>
<th></th>
<th>Relative computation time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fresnel formula</td>
<td>100%</td>
</tr>
<tr>
<td>Schlick’s formula</td>
<td>28%</td>
</tr>
<tr>
<td>Rescaled Schlick’s</td>
<td>33%</td>
</tr>
<tr>
<td>Error compensation</td>
<td>54%</td>
</tr>
</tbody>
</table>

Table 3.2: Relative computation times for different methods.
Figure 3.10: Comparing the original Fresnel function and our proposals for different microfacet incidence angles ($\cos \theta = 0..1$).
(a) The original Fresnel term (solid), our rescaled proposal (dashed) and our error-compensated proposal (dotted) at three different wavelengths.
(b) Relative error of our rescaled proposal (dashed) and our error-compensated proposal (dotted) at three different wavelengths.
(c) The original Fresnel term (solid), our rescaled proposal (dashed).
(d) Relative error of our rescaled proposal.
Figure 3.11: Comparing the original Fresnel function and our proposals for different microfacet incidence angles ($\cos \theta = 0..1$).
(a) The original Fresnel term (solid), our rescaled proposal (dashed) and our error-compensated proposal (dotted) at three different wavelengths.
(b) Relative error of our rescaled proposal (dashed) and our error-compensated proposal (dotted) at three different wavelengths.
(c) The original Fresnel term (solid), our rescaled proposal (dashed).
(d) Relative error of our rescaled proposal.
Figure 3.12: Aluminum sphere illuminated by a single directional light source for different θ values. The upper and lower half of the sphere is displayed using the original Fresnel function and our proposal, respectively.

Figure 3.13: A closer look to the aluminum sphere for microfacet incident angles θ = 80, 75, and 70 degrees. (Differences between the two methods are magnified to 200%).
Figure 3.14: Rings of different materials rendered with (a) Schlick’s approximation, (b) our proposal, and (c) the original Fresnel term.
Chapter 4

Rendering Specular Reflections

Efficient rendering of the reflections caused by mirror-like objects such as polished metals is usually based on environment mapping. However, the classical environment mapping technique assumes that the environment is infinitely far, and if this assumption fails, visible artifacts occur. To solve this problem, we present a fast approximation method that produces localized environment mapped reflections. Unlike previous localization methods, we do not ray-trace the neighborhood or the proxy geometry, but rely solely on environment map lookups. The geometric information required by the localization process is stored in the environment map. This information is the distance of the source of the illumination from the reference point where the environment map was taken. The method is fast and accurate if the scene consists of larger planar surfaces. In this case, the results are similar to that of ray-tracing. The algorithm is simple and can be executed by current vertex and pixel shaders at very high frame rates. Furthermore, since the algorithm can tolerate small-scale object movements, there is no need to refresh the environment map in every frame. Thus, by reusing a single environment map in multiple frames, the amortized cost of its generation becomes negligible.

4.1 Localization of the environment map

The idea to localize environment maps is discussed using the notations of Figure 4.1. Let us assume that the origin $\vec{o}$ of our coordinate system is the reference point of the environment map and we are interested in the illumination of point $\vec{x}$ from direction $\vec{R}$. We suppose that direction vector $\vec{R}$ has unit length.

Classical environment mapping would look up the illumination selected by direction $\vec{R}$, that is, it would use the radiance of point $\vec{r}$. However, $\vec{r}$ is usually not equal to point $\vec{q}$, which

Figure 4.1: Localization of the environment map having reference point $\vec{o}$. The illumination of point $\vec{x}$ from direction $\vec{R}$ is needed.
is in direction $\vec{R}$ from $\vec{x}$, and thus satisfies the following ray equation for some distance $d$:

$$\vec{q} = \vec{x} + \vec{R} \cdot d.$$ 

Our localization method finds an approximation of $d$ using an iterative process working with distances between the environment and reference point $\vec{o}$. The required distance information can be computed during the generation of the environment map. While a normal environment map stores the illumination for each direction in R,G,B channels, now we also obtain the distance of the visible point for these directions and store it, for example, in the alpha channel. We call these extended environment maps as **distance impostors**.

### 4.1.1 Finding initial guesses

To start the iteration process, we need two initial guesses for the ray hit. Straightforwardly, we can use shaded point $\vec{x}$ and default point $\vec{r}$, corresponding to zero and infinite ray parameters. Alternatively, we can assume that the environment surface at $\vec{r}$ is perpendicular to ray direction $\vec{R}$ (Figure 4.2), and pair the resulting hit point $\vec{p}$ with default point $\vec{r}$ to obtain the initial guesses.

To express the ray parameter $d_p$ corresponding to point $\vec{p}$, let us multiply ray equation

$$\vec{x} + \vec{R} \cdot d_p = \vec{p} \quad (4.1)$$ 

by direction vector $\vec{R}$. Substituting $\vec{R} \cdot \vec{p} = |\vec{r}|$, which is the consequence of the , we get:

$$d_p = |\vec{r}| - \vec{R} \cdot \vec{x}. \quad (4.2)$$

Note that if we used the direction of point $\vec{p}$ to lookup the environment map, we would obtain the color of point $\vec{p}'$, which is in the direction of $\vec{p}$ but is on the surface.

![Figure 4.2: Identifying first approximation point $\vec{p}$ assuming that the surface is perpendicular to $\vec{R}$.

### 4.1.2 Refining the initial guesses

During the iteration, we have to consider those cases separately when one of the guesses is $\vec{r}$ since in this case the corresponding ray parameter becomes infinite. Let us denote the other guess by $\vec{p}$ and the surface point in that direction by $\vec{p}'$ (Figure 4.3).

In this case, if the surface were a plane between $\vec{p}'$ and $\vec{r}$, then the intersection of the surface and the plane defined by points $\vec{r}$, $\vec{x}$ and $\vec{o}$ would be a line, and the point visible from $\vec{x}$ at direction $\vec{R}$ would be $\vec{l}$. Let us assume that the surface can be well approximated by a plane between points $\vec{r}$ and $\vec{p}'$, and find intersection $\vec{l}$ of the plane with the ray. The intersection is on the ray, thus it satisfies the following ray equation:

$$\vec{l} = \vec{x} + \vec{R} \cdot d_l. \quad (4.3)$$

36
Figure 4.3: Identifying second approximation point \( \vec{l} \) supposing that the surface is planar between points \( \vec{r} \) and \( \vec{p}' \).

Furthermore, point \( \vec{l} \) is also on the line of \( \vec{r} \) and \( \vec{p}' \), thus it can be expressed as a combination of these two points with unknown weight \( \alpha \):

\[
\vec{l} = \vec{r} \cdot \alpha + \vec{p}' \cdot (1 - \alpha).
\]

Note that \( \vec{l} \) is not necessarily between \( \vec{r} \) and \( \vec{p}' \), thus \( \alpha \) is not restricted to \([0, 1]\) but can have an arbitrary value. Substituting identities \( \vec{p}' = \vec{p} \cdot |\vec{p}'| / |\vec{p}| \), \( \vec{r} = \vec{R} \cdot |\vec{r}| \), as well as the ray equation for \( d_p \) (equation 4.1), we obtain

\[
\vec{l} = \vec{r} \cdot \alpha + \vec{p} \cdot (1 - \alpha).
\]

Comparing this expression with equation 4.3, we obtain the following requirements for unknowns \( \alpha \) and \( d_l \):

\[
(1 - \alpha) \cdot |\vec{p}'| / |\vec{p}| = 1, \\
d_p \cdot (1 - \alpha) \cdot |\vec{p}'| / |\vec{p}| + \alpha \cdot |\vec{r}| = d_l.
\]

From the first requirement, unknown \( \alpha \) can be expressed as \( \alpha = 1 - |\vec{p}'| / |\vec{p}| \). Substituting it into the second requirement we get:

\[
d_l = d_p + |\vec{r}| \cdot \left(1 - \frac{|\vec{p}'|}{|\vec{p}|} \right).
\]

Substituting this distance into the ray equation, we obtain hit point approximation \( \vec{l} \), which can be used to look up the environment map.

4.1.3 Iterative refinement

Suppose that we have two guesses of the ray parameter \( d_p \) and \( d_l \), and consequently two points \( \vec{p}' \) and \( \vec{l} \) that are on the ray, but are not necessarily on the surface, and two other points \( \vec{p}' \) and \( \vec{l}' \) that are on the surface, but are not necessarily on the ray (Figure 4.4).

These guesses can be refined by an iteration process, which computes new hit point approximation \( \vec{l}_{\text{new}} \) assuming that the surface is planar between sample points \( \vec{p}' \) and \( \vec{l}' \). Since new approximation \( \vec{l}_{\text{new}} \) is on the ray, it satisfies the following ray equation:

\[
\vec{l}_{\text{new}} = \vec{x} + \vec{R} \cdot d_{\text{new}}.
\]
Point $\vec{l}_{\text{new}}$ is also on the line of $\vec{l}$ and $\vec{p}'$, thus it can be expressed as their combination with unknown weight $\alpha$:

$$\vec{l}_{\text{new}} = \vec{l} \cdot \alpha + \vec{p}' \cdot (1 - \alpha).$$

Substituting identities $\vec{p}' = \vec{p} \cdot |\vec{p}'|/|\vec{p}|$ and $\vec{l} = \vec{l} / |\vec{l}|$, as well as the ray equation for $d_p$ and $d_t$, we get:

$$\vec{l}_{\text{new}} = (\vec{x} + \vec{R} \cdot d_t) \cdot \frac{|\vec{l}|}{|\vec{l}|} \cdot \alpha + (\vec{x} + \vec{R} \cdot d_p) \cdot \frac{|\vec{p}'|}{|\vec{p}|} \cdot (1 - \alpha) =

\begin{align*}
= \vec{x} \cdot \left( \frac{|\vec{p}|}{|\vec{l}|} \cdot \alpha + \frac{|\vec{p}'|}{|\vec{p}|} \cdot (1 - \alpha) \right) + \vec{R} \cdot \left( d_t \cdot \frac{|\vec{l}|}{|\vec{l}|} \cdot \alpha + d_p \cdot \frac{|\vec{p}'|}{|\vec{p}|} \cdot (1 - \alpha) \right).
\end{align*}

Comparing this expression with equation 4.5, we obtain the following requirements for unknowns $\alpha$ and $d_{\text{new}}$:

$$\frac{|\vec{l}|}{|\vec{l}|} \cdot \alpha + \frac{|\vec{p}'|}{|\vec{p}|} \cdot (1 - \alpha) = 1,$n

$$d_t \cdot \frac{|\vec{l}|}{|\vec{l}|} \cdot \alpha + d_p \cdot \frac{|\vec{p}'|}{|\vec{p}|} \cdot (1 - \alpha) = d_{\text{new}}.$$

Using the first requirement, unknown $\alpha$ can be expressed as $\alpha = (|\vec{p}'|/|\vec{p}|) - 1) / (|\vec{l}|/|\vec{l}| - |\vec{p}'|/|\vec{p}|)$. Putting it into the second requirement, we get:

$$d_{\text{new}} = d_t \cdot \frac{|\vec{l}|}{|\vec{l}|} \cdot \frac{|\vec{l}|/|\vec{l}| - |\vec{p}'|/|\vec{p}|} + d_p \cdot \frac{|\vec{p}'|}{|\vec{p}|} \cdot \frac{1 - |\vec{l}|/|\vec{l}|}{|\vec{l}|/|\vec{l}| - |\vec{p}'|/|\vec{p}|}.$$

Let us multiply both numerators and denominators by $(|\vec{l}|/|\vec{l}|) \cdot (|\vec{p}'|/|\vec{p}|)$ to reduce the number of factors. Note that fractions $|\vec{l}|/|\vec{l}|$ and $|\vec{p}'|/|\vec{p}|$ are eliminated and fractions $|\vec{l}|/|\vec{l}|$ and $|\vec{p}'|/|\vec{p}|$ appear instead:

$$d_{\text{new}} = d_t \cdot \frac{1 - |\vec{l}|/|\vec{l}|}{|\vec{l}|/|\vec{l}| - |\vec{p}'|/|\vec{p}|} + d_p \cdot \frac{|\vec{l}|/|\vec{l}| - |\vec{p}'|/|\vec{p}|}{|\vec{l}|/|\vec{l}| - |\vec{p}'|/|\vec{p}|}.$$

Finally, the expression is brought into a convenient form:

$$d_{\text{new}} = d_t + (d_t - d_p) \cdot \frac{1 - |\vec{l}|/|\vec{l}|}{|\vec{l}|/|\vec{l}| - |\vec{p}'|/|\vec{p}|}. \quad (4.6)$$

A step of the iteration evaluates this formula and replaces either $d_p$ or $d_t$ by new ray parameter approximation $d_{\text{new}}$, depending on the applied iteration strategy.
4.1.4 Iteration strategies

From a mathematical point of view, to obtain the hit point, we have to solve the ray equation:

\[ f(\vec{x} + \vec{R} \cdot d) = 0, \]

where \( f(\vec{r}) = 0 \) is the implicit equation of the environment surface, which is represented by discrete distance samples of the environment map. To solve this equation, numerical root finding methods can be applied. One possibility is to use the general *secant root finding method* [Wei03a], which simply pairs the last approximation (guess) with the last but one approximation to establish a new approximation, as shown in Table 4.1. The secant method usually converges very quickly, but it may not converge for high variation functions [Wei03a]. For example, the right side of Figure 4.5 shows a test case where the step size of the algorithm increases dramatically.

![Figure 4.5: The secant method always pairs the last estimate with the last but one estimate.](image)

For example, starting with initial guesses \((l_1, l_2)\), the secant method will pair guesses \((l_2, l_3)\), \((l_3, l_4)\), \((l_4, l_5)\), etc.

![Figure 4.6: The false position method pairs the last estimate with the last estimate of the opposite sign.](image)

For example, starting with initial guesses \((l_1^+, l_2^+)\), the false position method will pair guesses \((l_2^+, l_3^-)\), \((l_3^+, l_4^-)\), \((l_4^+, l_5^-)\), etc.

Note that in equation 4.6 the fraction

\[
\frac{1 - |\vec{l}|/|\vec{p}|}{|\vec{l}|/|\vec{p}| - |\vec{p}|/|\vec{p}'|}
\]

determines how the step size changes during the iteration. Since the absolute value of denominator \(|\vec{l}|/|\vec{p}| - |\vec{p}|/|\vec{p}'|\) can be smaller than the absolute value of numerator \(1 - |\vec{l}|/|\vec{p}|\), step size \(|d_l - d_p|\) may also increase. While increasing the step size where necessary improves convergence, it is also the cause of divergence.
\(d_p, d_l \leftarrow \text{initial guesses of the ray parameter} \)

Iteration:
\[d_{\text{new}} \leftarrow \text{guess of the ray parameter using parameters } d_p \text{ and } d_l \text{ (eq. 4.6.)} \]
\[d_p \leftarrow d_l \]
\[d_l \leftarrow d_{\text{new}} \]

Table 4.1: Pseudocode of the secant method for approximate raytracing.

\(d_{\text{under}}, d_{\overline{\text{over}}} \leftarrow \text{initial guesses of the ray parameter} \)

Iteration:
\[d_{\text{new}} \leftarrow \text{guess of the ray parameter using parameters } d_{\text{under}} \text{ and } d_{\overline{\text{over}}} \text{ (eq. 4.6.)} \]
if \(d_{\text{new}} \) is an undershooting
\[d_{\text{under}} \leftarrow d_{\text{new}} \]
else
\[d_{\overline{\text{over}}} \leftarrow d_{\text{new}} \]

Table 4.2: Pseudocode of the false position method for approximate raytracing.

To address the convergence issue, the basic iteration should be slightly modified. Let us first recognize that ratios \(|\vec{l}|/|\vec{l}'|\) and \(|\vec{p}|/|\vec{p}'|\) showing up in equation 4.6 express the accuracy of approximation points \(\vec{l}\) and \(\vec{p}\). If the point were the real ray hit, then the ratio would be 1. Values smaller than 1 indicate that the approximation point is an undershooting, i.e. it is in front of the surface (Figure 4.7a). On the other hand, when the ratio is greater than one, the approximation is an overshooting since the approximation point is behind the surface (Figure 4.7b).

Note that when \(\vec{l}\) and \(\vec{p}\) are of different types, the absolute value of denominator \(|\vec{l}|/|\vec{l}'|\) – \(|\vec{p}|/|\vec{p}'|\) cannot be smaller than the absolute value of enumerator \(1 - |\vec{l}|/|\vec{l}'|\), which results in a step size reduction. This condition can be enforced if last approximation \(\vec{l}\) is paired not necessarily with the last but one approximation, but the last approximation of opposite type. This method is equivalent to the false position root finding method [Wei03b], as shown in Table 4.2 and Figure 4.6.

We can switch from the secant method to the false position method if we have at least one overshooting point and one undershooting approximation. Since default point \(\vec{r}\) corresponds to an infinite ray parameter, which is a sure overshooting, we can use this ideal point at infinity if
there are no other overshooting results. On the other hand, if there is no undershooting point, we can either follow the secant rule, or default point $\vec{r}$ again to substitute the undershooting point. We have found that the last option is safe and works well.

Note that even with guaranteed convergence, the proposed method is not necessarily equivalent to exact ray tracing in the limiting case. Small errors may be due to the discrete surface approximation, or to view dependent occlusions. For example, should the ray hit a point that is not visible from the reference point of the environment map, the presented approximation scheme would obviously be unable to find that. However, when the object is curved and moving, these errors can hardly be recognized visually.

4.2 Implementation

The computation of distance impostors is very similar to that of classical environment maps. The only difference is that the distance from the reference point is also calculated, which can be stored in a separate texture or in the alpha channel of the environment map. Since the distance is a non linear function of the homogeneous coordinates of the points, correct results can be obtained only by letting the pixel shader compute the distance values.

Having the distance impostor, we can place an arbitrary object in the scene and illuminate it with its environment map using custom vertex and pixel shader programs. The vertex shader transforms object vertices ($\text{pos}$) to normalized screen space by the model-view-projection transformation ($\text{TMVP}$), and also to the coordinate system of the environment map first applying the modeling transform ($\text{TM}$), then translating to the reference point ($\text{refpos}$). View vector $V$ and normal $N$ are also obtained in world coordinates. Note that the normal vector is transformed with the inverse transpose of the modeling transform ($\text{TMIT}$).

```
OUT.hpos = mul(TMVP, IN.pos); // to normalized screen space
float3 xw = mul(TM, IN.pos).xyz; // to model space
OUT.x = xw - refpos; // to space of environment map
OUT.N = mul(TMIT, IN.norm); // normal vector
OUT.V = xw - eyepos; // view vector
```

Having the graphics hardware computed the homogeneous division and filled the triangle with linearly interpolating all vertex data, the pixel shader is called to find ray hit $l$ and to look up the cube map in this direction. The pixel shader calls function $\text{Hit}$ and looks up the cube map again to find illumination $I$ of the visible point, and possibly multiplies it with the BRDF.

```
N = normalize(N); V = normalize(V);
R = reflect(V, N); // reflection dir.
float3 I = Hit(x, R, envmap); // ray hit
float3 I = texCUBE(envmap, l).rgb; // radiance of the hit point
return I * BRDF(...); // reflected radiance
```

Function $\text{Hit}$ gets ray origin $x$ and direction $R$, as well as cube map $\text{mp}$, and returns the hit point approximation. We suppose that the distance values are stored in the alpha channel of the environment map. The implementation of this function based on different root finding methods will be discussed below.
4.2.1 Secant method

The following version of function \( \text{Hit} \) computes hit point approximation \( l \) with the general secant method, using points \( x \) and \( r \) as initial guesses.

```c
float3 Hit(float3 x, float3 R, sampler mp) {
    float rl = texCUBE(mp, R).a; // |r|
    float dp = 0;
    float ppp = length(x)/texCUBE(mp,x).a; // |x|/|x’|
    float dl = max(dp + rl * (1 - ppp), 0); // eq. 4.4
    float3 l = x + R * dl; // ray equation
    for(int i = 0; i < NITER; i++) { // iteration
        float llp = length(l)/texCUBE(mp,l).a; // |l|/|l’|
        float den = abs(llp - ppp);
        if (den > EPS || den > abs(1 - llp)) {
            float ddl = (dl - dp) * (1 - llp)/(llp - ppp); // eq. 4.6
            dp = dl; ppp = llp;
            dl = max(dl + ddl, 0); // avoid flip
            l = x + R * dl; // ray equation
        }
        //else i = NITER;
    }
    return l; // computed hit point
}
```

The following version is also based on the secant method, but uses the perpendicular surface assumption (Equation 4.2) to obtain the initial guesses.

```c
float3 Hit(float3 x, float3 R, sampler mp) {
    float rl = texCUBE(mp, R).a; // |r|
    float dp = rl - dot(x, R); // eq. 4.2
    float3 p = x + R * dp;
    float ppp = length(p)/texCUBE(mp,p).a; // |p|/|p’|
    float dl = max(dp + rl * (1 - ppp), 0); // eq. 4.4
    float3 l = x + R * dl; // ray equation
    for(int i = 0; i < NITER; i++) { // iteration
        float llp = length(l)/texCUBE(mp,l).a; // |l|/|l’|
        float den = abs(llp - ppp);
        if (den > EPS || den > abs(1 - llp)) {
            float ddl = (dl - dp) * (1 - llp)/(llp - ppp); // eq. 4.6
            dp = dl; ppp = llp;
            dl = max(dl + ddl, 0); // avoid flip
            l = x + R * dl; // ray equation
        }
        //else i = NITER;
    }
    return l; // computed hit point
}
```

4.2.2 False position method

The following version of function \( \text{Hit} \) computes hit point approximation \( l \) with the false position method: Note that variables \( \text{dun} \) and \( \text{dov} \) store the last undershooting and overshooting ray parameters. If there has been no overshooting approximation, point \( r \) takes the role of the overshooting point.

```c
float3 Hit(float3 x, float3 R, sampler mp) {
    float rl = texCUBE(mp, R).a; // |r|
    float dp = x + R * dp;
    float ppp = length(p)/texCUBE(mp,p).a; // |p|/|p|
    float dl = max(dp + rl * (1 - ppp), 0); // eq. 4.4
    float3 l = x + R * dl; // ray equation
    for(int i = 0; i < NITER; i++) { // iteration
        float llp = length(l)/texCUBE(mp,l).a; // |l|/|l’|
        float den = abs(llp - ppp);
        if (den > EPS || den > abs(1 - llp)) {
            float ddl = (dl - dp) * (1 - llp)/(llp - ppp); // eq. 4.6
            dp = dl; ppp = llp;
            dl = max(dl + ddl, 0); // avoid flip
            l = x + R * dl; // ray equation
        }
        //else i = NITER;
    }
    return l; // computed hit point
}
```
float3 Hit(float3 x, float3 R, sampler mp) {
    float rl = texCUBE(mp, R).a;  // |r|
    float ppp = length(x)/texCUBE(mp, x).a;  // |x|/|x’|
    float dun = 0, pov = 0, dov = 0, pov;
    float d1 = rl * (1 - ppp);  // eq. 4.4
    float3 l = x + R * d1;  // ray equation

    for(int i = 0; i < NITER; i++) {  // iteration
        float llp = length(l)/texCUBE(mp,l).a;  // |l|/|l’|
        if (llp < 1) {  // undershooting
            dun = dl; pun = llp;
            dl += (dov == 0) ? rl * (1 - llp) : // eq. 4.4
                (dl-dov) * (1-llp)/(llp-pov);  // eq. 4.6
        } else {  // overshooting
            dov = dl; pov = llp;
            dl += (dl-dun) * (1-llp)/(llp-pun);  // eq. 4.6
        }

        l = x + R * dl;  // ray equation
    }

    return l;  // computed hit point
}

4.2.3 Mixed method

The following version of function Hit computes hit point approximation l with a mixed method. Ratios |l’|/|l| and |p’|/|p| are represented by variables llp and ppp, respectively. Note that variables dun and dov store the last undershooting and overshooting ray parameters. If there has been no such approximation, the ray parameters are zero. In this case default point r takes their roles. In order to avoid ray flipping, the algorithm limits ray parameters for the non-negative domain.

float3 Hit(float3 x, float3 R, sampler mp) {
    float rl = texCUBE(mp, R).a;  // |r|
    float dp = rl - dot(x, R);  // eq. 4.2
    float3 p = x + R * dp;

    float ppp = length(p)/texCUBE(mp,p).a;  // |p|/|p’|
    float dun = 0, dov = 0, pun = ppp, pov = ppp;
    if (ppp < 1) dun = dp; else dov = dp;

    float d1 = max(dp + rl * (1 - ppp), 0);  // eq. 4.4
    float3 l = x + R * d1;

    for(int i = 0; i < NITER; i++) {  // iteration
        float ddl;
        float llp = length(l)/texCUBE(mp,l).a;  // |l|/|l’|
        if (llp < 1) {  // undershooting
            dun = dl; pun = llp;
            dl1 = (dov == 0) ? rl * (1 - llp) : // eq. 4.4
                (dl-dov) * (1-llp)/(llp-pov);  // eq. 4.6
        } else {  // overshooting
            dov = dl; pov = llp;
            dl1 = (dun == 0) ? rl * (1 - llp) : // eq. 4.4
                (dl-dun) * (1-llp)/(llp-pun);  // eq. 4.6
        }

        dl = max(dl1 + ddl, 0);  // avoid flip
        l = x + R * dl;

    }

    return l;
}
4.3 Results

4.3.1 Analyzing the secant method

Figure 4.8 provides an overview of the different methods. Comparing the secant method with initial guess $\vec{x}$ (Figure 4.8a) and the secant method with the perpendicular surface assumption (Figure 4.8b), we can notice that initial guess $\vec{x}$ produced unacceptable results. A possible explanation is that in this case the angle between reflection direction $\vec{R}$ and initial guess $\vec{x}$ may be large, which increases the extent and the variation of the search domain. Thus, the secant method is more likely to diverge in this case.

In Figure 4.9, we can see that the occluded face of the cube is correctly represented on the reflective sphere only in case of the mixed method and the false position method. Thus, let us concentrate on these methods.

4.3.2 Comparing the mixed method and the false position method

Figure 4.10 compares the methods for 3 and 10 iterations and also visualizes the iteration process. Note that for 3 iterations, the mixed method performs better.

Figures 4.11 and 4.12 show the typical weakness of the mixed method. Namely, choosing point $\vec{p}$ instead of $\vec{x}$ usually narrows down the search domain, thus the mixed method is more likely to miss the real hit point. (Interestingly, in case of the secant method, narrowing down the search domain improves convergence, while in our case it is the source of false results.)

Examining the test results, we can conclude that although it is possible to create test cases where both methods fail, the overall performance of the false position method was better.

The presented false position algorithm has been implemented in a test application that computes the environment map only once to show that the proposed localization gives good results even if the objects moved significantly from their original positions.

Figure 4.13 compares the images rendered by the proposed method with standard environment mapping and ray tracing. Note that for such scenes where the environment is convex from the reference point of the environment map, and there are larger planar surfaces, the new algorithm converges very quickly. The FPS values are measured with $700 \times 700$ resolution on an NV6800GT.

Figure 4.14 shows a difficult case where a box of checker board texture makes the environment surface concave and of high variation. Note that the convergence is still pretty fast, but the converged image is not exactly what we expect. We can observe that the green edge of the box is visible in a larger portion of the reflection image, because that part of the wall is not visible from the reference point of the environment map, but are expected to show up in the reflection. In such cases the algorithm can go only to the edge of the box and substitutes the reflection of the occluded points by the blurred image of the edge. The proposed method can be used not only for reflection but also for refraction calculations if the reflect operation is replaced by the refract function in the pixel shader.

We also included the proposed method in a game executing shadow computation, collision detection, etc. (Figures 4.15, 4.16, and 4.17) that can run with about a hundred FPS [P2]. In this game we used $6 \times 256 \times 256$ resolution cube maps that are recomputed in every 150 msec. We have realized that the speed improves by an additional 20 percent if the distance values are separated from the color data and stored in another texture map. The reason of this behavior is that the pixel shader reads the distance values several times from different texels before the color value is fetched, and separating the distance values increases texture cache utilization.
Figure 4.8: Reflections of a tiger rendered with (a) the secant method, (b) the secant method with perpendicular surface assumption, (c) the false position method and (d) the mixed method. When using the secant method with initial guesses $\vec{x}$ and $\vec{r}$ (a), the results are far worse than in the other three cases.

Figure 4.9: Comparing (a) the secant method with perpendicular surface assumption, (b) the false position method, and (c) the mixed method for 10 iterations. In the bottom row, the ray is represented by a thick horizontal red line, while the guesses are represented by green lines.
Figure 4.10: Comparing (a) the secant method, (b) the false position method, and (c) the mixed method for 3 and 10 iterations (first and second row) and visualizing the algorithm (third row). Note that the secant method is unable to handle occluded surfaces.
Figure 4.11: Comparing (a) the false position method and (b) the mixed method. Owing to the initial guess $\bar{x}$ the false position method could hit the corner of the crate while the mixed method failed. The ray is represented by a thick horizontal red line.

Figure 4.12: Comparing (a) the false position method and (b) the mixed method. A typical scenario where the mixed method fails. The ray is represented by a thick horizontal red line.
Figure 4.13: Comparison of classical and localized environment map reflections with ray traced reflections placing the reference point at the center of the room and moving a reflective sphere to different locations.

Figure 4.14: A more difficult case when the room contains a box that makes the scene strongly concave and is responsible for view dependent occlusions.
Figure 4.15: A reflective box with shadows in a stone environment illuminated by dynamic lights (160 FPS).

Figure 4.16: A knight in reflective armor in textured environment illuminated by dynamic lights (105..130 FPS).

Figure 4.17: Reflective and refractive spheres in a game environment running with 105 and 85 FPS, respectively.
4.4 Conclusions

We proposed a localization method for environment maps, which uses the distance values stored in environment map texels. The localization method is equivalent to approximate ray-tracing, which solves the ray equation by numerical root finding. The proposed solution can introduce effects in games that are usually simulated by ray tracing, such as reflections and refractions on curved surfaces.

Since the proposed algorithm [P2, P3] represents just those surfaces that are directly visible from the reference point, the ray hits that are occluded from the reference point cannot be computed accurately. This issue was later addressed by multiple articles. If occluded surfaces are also needed, a texel should store multiple distances or, alternatively, we should use multiple layers as suggested by layered depth impostors [DSSD99]. The set of layers representing the scene could be obtained by depth peeling [Eve01, LWX06] in the general case. However, we may often apply an approximation that limits the number of layers to two, representing front faces and back faces, respectively.

A robust root finding method was proposed by Umenhoffer et al. [UPSK07] where the intersection was identified by ray marching, i.e. by generating an increasing sequence of ray parameter $d$ and detecting the first pair of subsequent ray points where one point is an overshooting while the other is an undershooting.

The complete search can be implemented by drawing a line and letting the fragment program check just one texel [KBW06]. However, in this case it is problematic to find the first intersection from the multiple intersections since fragments are processed independently. In their implementation, Krieger et al. solved this problem by rendering each ray into a separate row of a texture, and found the first hits by additional texture processing passes, which complicates the algorithm, reduces the flexibility, and prohibits early ray termination.

On GPUs supporting dynamic branching, the complete ray marching process can also be executed by a single fragment program [UPSK07]. This program can not only identify the first pair of overshooting and undershooting samples, but can also refine the hit point by a secant or a binary search. The linear search was accelerated by computing minimum and maximum distance values for each distance map. Furthermore, by marching on a projected line segment, the algorithm could avoid redundant cube map lookups.

Hu et al. [HQ07] and Oliveira et al. [OB07] used binary search steps to refine the approximations obtained by a linear search similarly to popular displacement mapping algorithms. Binary search simply halves the interval of the ray parameter. Since it does not use as much information as the secant search its convergence is slower.

Instead of using multiple or layered cube maps, another solution of the occlusion problem is the application of multiperspective cameras [HWSG06, MPS05], where a “flattened” high-dimensional texture is used, which is stored as a tiled 2D texture.
Chapter 5

Rendering Glossy and Diffuse Reflections

Depending on their surface properties (roughness, oxidation, etc.), metals can not only act as ideal mirrors, but as non-ideal reflectors as well. In this case, precalculated diffuse and specular environment maps can be applied. However, due to their computational cost, these maps cannot be recalculated for every frame like ordinary environment maps. Thus, disturbing popping effects will occur when the environment maps are regenerated, as seen in many computer games.

In this chapter we propose a method that reduces these popping artifacts by providing localized environment map lookups. Our proposal does not require preprocessing, and thanks to the geometric localization procedure, it can use one cubemap for an object or even for several close objects. The cubemap is regenerated when the respective object moves significantly, but less frequently than the rendering frame rate of the application, amortizing the cubemap generation cost in multiple frames. Thus the new method can cope with fully dynamic scenes and changing illumination environments.

5.1 The new method for glossy and diffuse reflections

5.1.1 Proposed simplifications

Final gathering, i.e. the computation of the reflection of the indirect illumination toward the eye, is one of the most time consuming steps of realistic rendering. According to the rendering equation, the reflected radiance of point \( \vec{x} \) in viewing direction \( \vec{\omega} \) can be expressed by the following integral

\[
L^r(\vec{x} \rightarrow \vec{\omega}) = \int_{\Omega'} L^{in}(\vec{x} \leftarrow \vec{\omega}') \cdot f_r(\vec{\omega}' \rightarrow \vec{\omega} \rightarrow \vec{x}) \cdot \cos \theta_{\vec{x}} \, d\omega',
\]

where \( \Omega' \) is the set of possible illumination directions, \( L^{in}(\vec{x} \leftarrow \vec{\omega}') \) is the incoming radiance arriving at point \( \vec{x} \) from illumination direction \( \vec{\omega}' \), \( f_r \) is the BRDF, and \( \theta_{\vec{x}} \) is the angle between the surface normal at point \( \vec{x} \) and the illumination direction.

The evaluation of this integral usually requires many sampling rays from each shaded point. Ray casting finds illuminating points \( \vec{y}_i \) for shaded point \( \vec{x} \) at different directions (Figure 5.1), and the radiance of these illumination points is inserted into a numerical quadrature approximating the rendering equation. In practice, number \( P \) of shaded points is over hundred thousands or millions, while number \( D \) of sample directions is about a hundred or a thousand to eliminate annoying sampling artifacts. However, in games and in real-time systems, render-
ing cannot take more than a few tens of milliseconds. This time does not allow tracing \( P \cdot D \), i.e. a large number of rays.

![Figure 5.1: Indirect illumination with sampling rays.](image)

To solve this complexity problem, we can exploit the fact that in games the dynamic objects are usually significantly smaller than their environment. Thus the global indirect illumination of the environment can be computed separately, since it is not really affected by the smaller dynamic objects. Furthermore, when the indirect illumination of dynamic objects is evaluated, their small size makes it possible to reuse illumination information obtained when shading its other points.

The first idea of our method is to trace rays with the graphics hardware just from a single reference point being in the vicinity of the dynamic object. Then the illuminating points selected by these rays and their radiance are used not only for the reference point, but for all visible points of the dynamic object. From a different point of view, tracing rays locates virtual light sources which illuminate the origin of these rays. We propose to find a collection of virtual light sources from a reference point, and then use these virtual light sources to illuminate all shaded points (Figure 5.2).

![Figure 5.2: The basic idea of the proposed method. (a) First virtual lights sampled from reference point \( \vec{d} \) are identified, then these point lights are grouped into large area lights. (b) The illumination of a relatively small number of area lights at shaded points \( \vec{x} \) is computed without visibility tests.](image)

This approach has two advantages. On the one hand, instead of tracing \( P \cdot D \) rays, we solve the rendering problem by tracing only \( D \) rays. On the other hand, these rays form a bundle meeting in the reference point and are regularly spaced. Such ray bundles can be very efficiently traced by the graphics hardware. However, this simplification also has disadvantages. Assuming that the same set of illuminating points are visible from each shaded point, self-shadowing effects are ignored. However, while shadows are crucial for direct lighting, shadows from indirect lighting are not so visually important. Thus the user or the gamer finds
this simplification acceptable. Additionally, the used virtual light sources must be visible from
the reference point. In concave environments, however, it may happen that an environment
point is not visible from the reference point but may illuminate the shaded point. However,
indirect illumination is usually quite smooth, thus ignoring smaller parts of the potentially
illuminating surfaces does not lead to noticeable errors.

Unfortunately, this simplification alone cannot allow real time frame rates. The evaluation
of the reflected radiance at a shaded point still requires the evaluation of the BRDF and the
orientation angle, and the multiplication with the radiance of the illuminating point by $D$
times. Although the number of rays traced to obtain indirect illumination is reduced from
$P \cdot D$ to $D$, but the illumination formula must be evaluated $P \cdot D$ times. These computations
would still need too much time.

In order to further increase the rendering speed, we propose to carry out as much compu-
tation globally for all shaded points, as possible. Clearly, this is again an approximation, since
the weighting of the radiance of each illumination point at each shaded point is different. From
mathematical point of view, we need to evaluate an integral of the product of the illumination
and the local reflection for every shaded point. To allow global computation, the integral of
these products is approximated by the product of the integrals of the illumination and the
local reflection, thus the illumination can be handled globally for all shaded points.

Intuitively, global computation means that the sets of virtual light sources are replaced by
larger homogeneous area light sources. Since the total area of these lights is assumed to be
visible, the reflected radiance can be analytically evaluated once for a whole set of virtual light
sources.

5.2 Previous work

As discussed earlier, environment mapping is a hardware-accelerated technique to approximate
general secondary rays without expensive ray-tracing. This technique can also be applied for
glossy and diffuse reflections as well. The usual trick is the convolution of the angular variation
of the BRDF with the environment map during preprocessing [RH01].

These convolution results are stored in different texels of the resulting map. In diffuse case,
the surface normal is used as query direction to read one of these precalculated values, while
in specular case, supposing the Phong BRDF model, the ideal reflection direction can be used
for the same purpose. For example, Figure 5.3 shows a simple scene and the corresponding
diffuse and specular environment maps, generated from the middle of the room. Then, the
tiny ball at the middle of the room was rendered using the respective environment map. Note
that the lamp in the corner is included for easier interpretation only.

When the object is moving, similarly to ordinary environment maps, the diffuse (or spec-
ular) environment maps have to be frequently re-generated. However, if the object is large
compared to the distance from the environment (Figure 5.4), a single environment map – even
if frequently updated – will not provide sufficient information to shade the entire object prop-
perly. For example, although the glossy Happy Buddha, rendered with the “classic” specular
environment mapping method (Figure 5.4a) looks impressive, its illumination does not reflect
the proximity of the environment properly. For example, the fireballs, located close to the
middle of the statue, should not affect Buddha’s head and the basement of the statue, where,
on the other hand, the contribution of the luminous ceiling and floor should be much more
significant (see Figure 5.4b).

For dynamic scenes, a possible alternative is to use multiple environment maps [GSHG98,
ZHL+05].

5.2.1 Reusing illumination information

Let us assume that we use a single environment map that records illumination information for
reference point $\vec{o}$. Our goal is to reuse this illumination information for other nearby points

53
Figure 5.3: (a) The original scene, the generated (b) diffuse and (c) specular environment maps. The ball at the middle of the room was rendered using the respective environment map.

Figure 5.4: Happy Buddha between fireballs, displayed with (a) the “classical” specular environment mapping method and (b) our proposal (shininess = 10).
as well. To do so, we use approximations that allow us to factor out those components from the rendering equation, which strongly depend on shaded point \( \vec{x} \).

In order to estimate the integral of the rendering equation, the set of possible illumination directions \( \Omega' \) is partitioned to solid angles \( \Delta \omega'_i, i = 1, \ldots, N \), where the radiance is roughly uniform in each domain. After partitioning, the reflected radiance is expressed by the following sum:

\[
L'(\vec{x} \to \vec{\omega}) = \sum_{i=1}^{N} \int L^m(\vec{x} \to \vec{\omega}') \cdot f_r(\vec{\omega}' \to \vec{x} \to \vec{\omega}) \cdot \cos \theta_{\vec{x}} \, d\omega'.
\]

Let us consider a single term of this sum representing the radiance reflected from solid angle \( \Delta \omega'_i \). Supposing that \( \Delta \omega'_i \) is small, then incoming radiance \( L^m \) has small variation and is approximately constant. Furthermore, while reflectance \( f_r(\vec{\omega}' \to \vec{x} \to \vec{\omega}) \cdot \cos \theta_{\vec{x}} \) depends strongly on the shaded point \( \vec{x} \), incoming radiance \( L^m \) is independent of \( \vec{x} \) if the environment is diffuse, and can be supposed to be approximately independent of \( \vec{x} \) if the environment is moderately glossy. Thus, as an approximation, the incoming radiance is assumed to be constant and is factored out:

This factoring is acceptable because if \( \Delta \omega'_i \) is small, then \( L^m \) has small variation and depends on the illuminating point, while \( f_r(\vec{\omega}' \to \vec{x} \to \vec{\omega}) \cdot \cos \theta_{\vec{x}} \) depends on the shaded point.

\[
\int L^m(\vec{x} \to \vec{\omega}') \cdot f_r(\vec{\omega}' \to \vec{x} \to \vec{\omega}) \cdot \cos \theta_{\vec{x}} \, d\omega' \\
\approx \bar{L}^m(\Delta A_i) \cdot \int f_r(\vec{\omega}' \to \vec{x} \to \vec{\omega}) \cdot \cos \theta_{\vec{x}} \, d\omega' \tag{5.1}
\]

where \( \bar{L}^m(\Delta A_i) \) is the average incoming radiance coming from surface \( \Delta A_i \) seen at solid angle \( \Delta \omega'_i \). Expressing average incoming radiance \( \bar{L}^m(\Delta A_i) \) on surface area \( \Delta A_i \), we obtain:

\[
\bar{L}^m(\Delta A_i) = \frac{1}{\Delta A_i} \int_{\Delta A_i} L(\vec{y} \to \vec{\omega} \to \vec{x}) \, dy.
\]

Note that this average is independent of shaded point \( \vec{x} \) if the environment is diffuse, and can be supposed to be approximately independent of the shaded point if the environment is moderately glossy.

The second factor of the product in equation 5.1 is the reflectivity integral, which is also expressed as the product of the average integrand and the size of the integration domain:

\[
\int_{\Delta \omega'} f_r(\vec{\omega}' \to \vec{x} \to \vec{\omega}) \cdot \cos \theta_{\vec{x}} \, d\omega' = a(\Delta \omega'_i \to \vec{x} \to \vec{\omega}) \cdot \Delta \omega'_i
\]

where \( a(\Delta \omega'_i \to \vec{x} \to \vec{\omega}) \) is the average reflectivity from solid angle \( \Delta \omega'_i \). Using these notations, the reflected radiance can be approximately expressed as

\[
L'(\vec{x} \to \vec{\omega}) \approx \sum_{i=1}^{N} \bar{L}^m(\Delta A_i) \cdot a(\Delta \omega'_i \to \vec{x} \to \vec{\omega}) \cdot \Delta \omega'_i. \tag{5.2}
\]

### 5.2.2 Calculating average incoming illumination

As concluded in the previous section, average incoming radiance values \( \bar{L}^m(\Delta A_i) \) are independent of the shaded point in case of diffuse or moderately glossy environment, thus these values can potentially be reused for all shaded points. To exploit this idea, visible surface areas \( \Delta A_i \) need to be identified and their average radiances need to be computed first. These areas are
found and the averaging is computed with the help of a cube map placed at reference point \( \vec{o} \) in the vicinity of the shaded object. We render the scene from reference point \( \vec{o} \) onto the six sides of a cube. In each pixel of these images we store the radiance of the visible point and also the distance from the reference point. The pixels of the cube map thus store the radiance and also encode the position of small indirect lights.

The small virtual lights are clustered into larger area light sources while averaging their radiance, which corresponds to downsampling the cube map. A pixel of the lower resolution cube map is computed as the average of the included higher resolution pixels. Note that both radiance and distance values are averaged, thus finally we have larger lights having the average radiance of the small lights and placed at their average position. The total area corresponding to a pixel of a lower resolution cube map will be elementary surface \( \Delta A_i \), and its average radiance is stored in the texel.

![Image](image-url)

**Figure 5.5:** Solid angle in which a surface seen through a cubemap pixel.

### 5.2.3 Determining solid angles

According to equation 5.2 the reflected radiance at the reference point is:

\[
L'(\vec{o} \rightarrow \vec{\omega}) \approx \sum_{i=1}^{N} \tilde{L}_{\text{in}}^{i}(\Delta A_i) \cdot a(\Delta \omega_i^0 \rightarrow \vec{o} \rightarrow \vec{\omega}) \cdot \Delta \omega_i^0.
\]

Now let us now consider another point \( \vec{x} \) close to the reference point \( \vec{o} \) and evaluate a similar integral for point \( \vec{x} \) while making exactly the same assumption on the surface radiance, i.e. it is constant in areas \( \Delta A_i \):

\[
L'(\vec{x} \rightarrow \vec{\omega}) \approx \sum_{i=1}^{N} \tilde{L}_{\text{in}}^{i}(\Delta A_i) \cdot a(\Delta \omega_i^0 \rightarrow \vec{x} \rightarrow \vec{\omega}) \cdot \Delta \omega_i^0,
\]

(5.3)

where \( \Delta \omega_i^0 \) is the solid angle subtended by \( \Delta A_i \) as seen from shaded point \( \vec{x} \).

For reference point \( \vec{o} \), solid angle \( \Delta \omega_i^0 \) can be easily determined by considering the solid angle subtended by the corresponding texel in the cube map:

\[
\Delta \omega_i^0 \approx \frac{\Delta A_{\text{texel}}}{|\tilde{L}_i|^2} \cos \theta_{\text{texel}}^i,
\]

where \( \Delta A_{\text{texel}} \) is the area of the texel, \( \Delta A_{\text{texel}} \cos \theta_{\text{texel}}^i \) is the projected area of the texel as seen from reference point \( \vec{o} \), and \( \tilde{L}_i \) is the (non-normalized) vector pointing from the middle
of the cube map to the respective texel center. Supposing that the edge size of the cube map is 2 (Figure 5.5), the formula becomes

$$\Delta \omega'_i \approx \left(\frac{2}{M}\right)^2 \frac{|\vec{L_i}|^2}{|\vec{L_i}|^3},$$

where $M$ is the resolution of a single cube map face.

Unfortunately, the solid angle values can be obtained directly from the geometry of the cubemap only if the shaded point is the center of the cube map. In case of other shaded points, special considerations are needed that are based on the distances from the environment surface.

![Figure 5.6: The notations of the subtended solid angles for (a) reference point $\vec{o}$ and (b) shaded point $\vec{x}$.](image)

Solid angles $\Delta \omega'_i$ and $\Delta \omega^*_i$ can be expressed without the disc to point form factor approximation:

$$\Delta \omega'_i = \frac{\Delta A_i \cdot \cos \theta'_i}{|\vec{o} - \vec{y_i}|^2}, \quad \Delta \omega^*_i = \frac{\Delta A_i \cdot \cos \theta^*_i}{|\vec{x} - \vec{y_i}|^2},$$

where $\theta'_i$ and $\theta^*_i$ are the angle between the surface normal at $\vec{y}_i$ and the vector pointing from $\vec{y}_i$ to $\vec{o}$, and the angle between the surface normal and the vector pointing from $\vec{y}_i$ to $\vec{x}$, respectively (Figure 5.6).

Assume that the environment surface is not very close compared to the distances of the reference and shaded points, thus the angles between the normal vector at $\vec{y}_i$ and reflection vectors from $\vec{o}$ and from $\vec{x}$ are similar:

$$\cos \theta'_i \approx \cos \theta^*_i.$$

In this case, we can establish the following simple relationship between $\Delta \omega^*_i$ and $\Delta \omega'_i$:

$$\Delta \omega^*_i \approx \Delta \omega'_i \cdot \frac{|\vec{o} - \vec{y}_i|^2}{|\vec{x} - \vec{y}_i|^2}. \quad (5.4)$$

### 5.2.4 Determining solid angles with point-to-disc approximations

Supposing that the distance between shaded point $\vec{x}$ and surface center $\vec{y}_i$ is small ($r^*_i \to 0$), formula 5.4 tends to infinity. To get more accurate results, point-to-disc approximations will be used. That is, assuming that the light source is a disc of area $\Delta A_i$, the subtended solid angle can be calculated analytically:
\[
\Delta \omega'_i = 2\pi \left( 1 - \frac{1}{\sqrt{1 + \frac{\Delta A_i \cos \theta'_i}{\pi |\vec{y}_i - \vec{y}'_i|^2}}} \right),
\]
\[
\Delta \omega^*_i = 2\pi \left( 1 - \frac{1}{\sqrt{1 + \frac{\Delta A_i \cos \theta^*_i}{\pi |\vec{y}_i - \vec{y}'_i|^2}}} \right),
\]

Using the same assumptions as above, i.e. \( \cos \theta'_i \) and \( \cos \theta^*_i \) are similar, the following relationship can be derived between \( \Delta \omega^*_i \) and \( \Delta \omega'_i \):

\[
\Delta \omega^*_i \approx 2\pi \left( 1 - \frac{1}{\sqrt{1 + \left( \frac{2\pi}{\Delta \omega'_i} \right)^2} \cdot \frac{|\vec{x} - \vec{y}_i|^2}{|\vec{x} - \vec{y}'_i|^2}} \right)
\]

### 5.2.5 Calculating reflectivity

Average reflectivity \( a(\Delta \omega'_i \rightarrow \vec{x} \rightarrow \vec{\omega}) \) (Equation 5.2) is evaluated using only one directional sample \( \omega_{\vec{y}_i \rightarrow \vec{x}} \) pointing from \( \vec{y}_i \) being the center of \( \Delta A_i \) toward \( \vec{x} \), that is, the integrand is assumed to be constant in domain \( \Delta \omega'_i \):

\[
a(\Delta \omega'_i \rightarrow \vec{x} \rightarrow \vec{\omega}) \approx f_r(\omega_{\vec{y}_i \rightarrow \vec{x}} \rightarrow \vec{x} \rightarrow \vec{\omega}) \cdot \cos \theta_{\vec{x}}.
\]

Clearly, such approximation is acceptable for diffuse and moderately glossy materials, but fails for highly specular surfaces.

Supposing that the object does not approach the environment excessively, the reflectivity can be assumed to be constant in a domain and can be approximately calculated using only one directional sample (e.g. considering the center of the domain)

\[
a(\Delta \omega'_i \rightarrow \vec{x} \rightarrow \vec{\omega}) \approx f_r(\omega_{\vec{y}_i \rightarrow \vec{x}} \rightarrow \vec{x} \rightarrow \vec{\omega}) \cdot \cos \theta_{\vec{x}}.
\]

However, if the object may approach the illuminating environment, using only one sample causes the centers of the area light sources become visible (Figure 5.7b). To avoid this, the reflectivity integral can be precisely precalculated for all possible values and stored in a 2D texture, where one of the texture coordinates corresponds to the size of the integration domain while the other corresponds to the cosine of the angle between the surface normal \( \vec{N} \) and the illumination direction \( \vec{L} \) pointing to the center of the area light source. Using this texture, artifacts can be greatly reduced (Figure 5.7c).

Since the incoming illumination values get weighted with the reflectivity, the number of area light sources (i.e. texels) that may have a significant contribution depends on how quickly the average reflectivity changes. On the one hand, for diffuse materials the reflectivity is a simple cosine term that changes slowly, thus we can obtain good results using a low-resolution map (4 × 4, or even 2 × 2 texels per cube map face).

On the other hand, for specular materials the reflectivity changes quickly and is non-zero only for a few texels. Therefore, a high-resolution environment map is needed to provide these quickly changing regions with detailed illumination information. However, since the reflectivity is mostly zero there is no need to visit all texels. We set the resolution of the map according to the specularity of the material and evaluate just a few (say 5) terms.

### 5.3 Implementation

The proposed algorithm first computes an environment cube map from the reference point and stores the radiance and distance values of the points visible in its pixels. We usually generate
Figure 5.7: (a) The diffuse ball at the wall demonstrates the importance of an accurate reflectivity calculation. Let us compare the side view of the same scene with (b) simplified and (c) texture-based reflectivity calculation.

6 × 256 × 256 pixel resolution cube maps. Then the cube map is downsampled to have \( M \times M \) pixel resolution faces (\( M \) is 4 or even 2). Texels of the low-resolution cubemap represent elementary surfaces \( \Delta A \), whose average radiance and distance are stored in the texel. The illumination of these elementary surfaces is reused for an arbitrary point \( x \), as shown by the following pixel shader program calculating the reflected radiance at this point:

```c
float4 RefRad ( half3 N : TEXCOORD0,
                half3 V : TEXCOORD1,
                half3 x : TEXCOORD2 ) : COLOR0
{
    half4 Lr = 0;
    V = normalize( V ); N = normalize( N );
    for (int X = 0; X < M; X++) // for each texel
        for (int Y = 0; Y < M; Y++)
        {
            half2 t = half2((X+0.5f)/M, (Y+0.5f)/M);
            half2 l = 2 * t - 1; // [0,1]->[-1,1]
            Lr += Contrib(x, half3(l.x,l.y, 1), N, V);
            Lr += Contrib(x, half3(l.x,l.y,-1), N, V);
            Lr += Contrib(x, half3(l.x, 1,l.y), N, V);
            Lr += Contrib(x, half3(l.x,-1,l.y), N, V);
            Lr += Contrib(x, half3( 1,l.x,l.y), N, V);
            Lr += Contrib(x, half3(-1,l.x,l.y), N, V);
        }
    return Lr;
}
```

The `Contrib` function calculates the contribution of a single texel of downsampled, low resolution cubemap `LREnvMap` to the illumination of the shaded point. Arguments \( x, L, N, \) and \( V \) are the relative position of the shaded point with respect to the reference point, the unnormalized illumination direction of the reference point, the unit surface normal at the shaded point, and the unit view direction, respectively.

```c
half4 Contrib(half3 x, half3 L, half3 N, half3 V) {
    half l = length(L);
    half dw = 4 / (M*M*l*l*l + 4/PI);
    half doy = texCUBE(LREnvMap, L).a;
    half doy2 = doy * doy;
    half3 y = L / l * doy;
    half3 dxy2 = dot(y-x, y-x);  
```
First the solid angle subtended by the texel from the reference point is computed and stored in variable \( d_{w} \), then illuminating point \( y \) is obtained looking up the distance value of the cube map. The square distances between the reference point and the illuminating point, and between the shaded point and the illuminating point are put into \( d_{y2} \) and \( d_{xy2} \), respectively. These square distances are used to calculate solid angle \( d_{ws} \) subtended by the illuminating surface from the shaded point. Assuming Phong-Blinn BRDF of diffuse reflectance \( k_{d} \), specular reflectance \( k_{s} \), and shininess \( n \), illumination direction \( I \) and halfway vector \( H \) are calculated, and the reflection of the radiance stored in the cube map texel is obtained according to equation 5.3.

### 5.4 Results

In order to demonstrate the results, we took a simple environment consisting of a cubic room with a divider face in it. The object to be indirectly illuminated is the bunny, happy buddha, and the the dragon, respectively. Each of these models consists of approximately 50-60 thousand triangles. Frame rates were measured in 700 × 700 windowed mode on an NV6800GT graphics card and P4/3GHz CPU.

The first set of pictures (Figure 5.8) shows three diffuse Stanford Bunnies next to a fireball. In case of “classic” environment mapping (first image), since the precalculated convolution cannot deal with the position of the objects, all bunnies look identical. Using our method, however, pleasing localization effects occur even with an extremely low resolution (2 × 2). Increasing the resolution to 4 × 4 and 8 × 8, the results get more realistic at the cost of a reduced frame rate. (Frame rates apply to the rendering a single bunny in the room.)

The second set of pictures (Figure 5.9) shows glossy skulls between fireballs. Again, the first image presents the “classic” environment mapping technique (340 FPS for a single skull). The second image shows our proposal for glossy materials, i.e. considering only the five “most important” directional domains during the calculation, producing approximate localization effects at a reasonable speed (264 FPS for a single skull). The other two images perform a full convolution on the fly, resulting in a better approximation and reduced frame rates.

The following three picture sets (Figures 5.10, 5.11 and 5.12) systematically compare the different techniques. For each picture set, the first column presents the “classic” environment mapping technique while the other three columns present the results of our localized method. Examining the picture sets we can conclude that using our method, we get more pleasing results than the classical technique even with an extremely low resolution (2 × 2).

We have also implemented the proposed method in a game running at 30 FPS. Images of this game are shown in Figure 5.13.
5.5 Conclusions

We presented a localization method for computing diffuse and glossy reflections of the incoming radiance stored in environment maps. The localization uses the distance values stored in the environment map texels. The presented method runs in real-time and provides visually pleasing results.

The cubemap is downsampled, which corresponds to clustering lights and replacing a cluster by an area light source. Comparing to Lightcuts [WFA+05], our clustering is simpler, is executed in real-time by the graphics hardware, and replaces the cluster by an area light. The radiance values of the single cube map are then localized for the shaded points taking into account the distances of the environment, the shaded points, and of the reference point, similarly to the approximate ray-tracing method.

However, the new technique uses a different localization approach developed particularly for diffuse and glossy reflections.

The method computes indirect illumination although approximately, but providing very pleasing visual quality. The method suits very well the GPU architecture, and can render these effects interactively.
Figure 5.8: Diffuse Stanford Bunny at different positions of the scene. Comparing classical environment mapping (a) and our localized approach at cube map resolutions (b) $2 \times 2$, (c) $4 \times 4$, and (d) $8 \times 8$. 
Figure 5.9: Glossy skull at different positions of the scene. Comparing classical environment mapping (a) and our localized approach at resolutions (b) 4 × 4 (considering the five most important contributions), (c) 4 × 4 (all contributions), and (d) 8 × 8 (all contributions).
Figure 5.10: Diffuse bunny at different positions of the scene. Comparing the classical environment mapping (a) and our algorithm using different environment map resolutions.

Figure 5.11: Specular buddha (shininess = 5) rendered with the classical environment mapping (a) and with our algorithm using different environment map resolutions.
Figure 5.12: Specular dragon (shininess = 5) at different positions of the scene. Comparing the classical environment mapping (a) and our algorithm using different environment map resolutions.

Figure 5.13: Glossy objects in a game running at 30 FPS.
Chapter 6

Rendering Multiple Interreflections

6.1 Introduction

In Chapters 4 and 5 single reflections were considered. In order to calculate multiple reflections we also propose a GPU friendly global illumination method. A particularly efficient global illumination method is the virtual light sources algorithm that is based on ray-tracing and is running on the CPU in its original form. The method enables us to reuse complete shooting paths at the cost of a single shadow ray per pixel, thus it can solve the rendering problem effectively. However, since the tracing of the shadow rays is rather computation intensive, it seems to be reasonable to trace them only where it is necessary. We propose two techniques to speed up the algorithm. Firstly, the proposed method evaluates the contribution of the virtual sources without taking care of the visibility factor. Then, for each of these partially evaluated samples, we only complete the evaluation by tracing the shadow ray if the sample can have a significant effect on the result. Since the contributions can be computed independently of the visibility tests, we can utilize the computing power of the graphics card to increase performance. Namely, we shall discuss how the partially evaluated samples can be obtained quickly by the graphics card without increasing the load of the CPU.

6.2 The virtual light source method

The virtual light sources method is a simple and effective global illumination method for reusing light paths. Virtual light sources appeared first in instant radiosity [Kel97], then were generalized for non-diffuse environments in [WKB+02]. Virtual light sources can be generated by a shooting type random walk algorithm, where all the visited points are stored together with the incoming power $\Phi^{in}$ and incoming direction $\omega^{in}$ (Figure 6.1a). The hits act as abstract, point-like light sources. For each pixel, the points on the surfaces visible through the pixels are connected to all of these virtual light sources by shadow rays in a deterministic manner, as depicted in Figure 6.1b. Note that while the reflection of light sources causes direct illumination, the reflection of virtual light sources approximates indirect illumination. The estimator for the radiance caused by the illumination of virtual light sources is:

$$L^r(\vec{x} \rightarrow \vec{\omega}) = \sum_{\vec{y}} \Phi^{in}_{\vec{y}} \cdot f_r(\vec{y} \rightarrow \vec{x}) \cdot \cos \theta_{\vec{y}} \cdot \cos \theta_{\vec{x}} \cdot |\vec{x} - \vec{y}|^2 \cdot v(\vec{x}, \vec{y}),$$

(6.1)

where $\vec{y}$ is the location of a given virtual light source, $\vec{x}$ is the point visible from the camera, $f_r(\vec{y}) = f_r(\omega^{in}_{\vec{y}}, \vec{y}, \omega_{\vec{y} \rightarrow \vec{x}})$ is the BRDF at the virtual light source, $f_r(\vec{x}) = f_r(\omega_{\vec{x} \rightarrow \vec{y}}, \vec{x}, \omega_{\text{eye}})$ is
the BRDF at the visible point, $\omega_{\gamma}^{\gamma'}$ is the incoming direction at point $\gamma$, $\omega_{\gamma'\rightarrow\gamma}$ is the direction from $\gamma$ to $\gamma'$, and $\omega$ is the direction from $\gamma'$ to the eye. Furthermore, $\theta_{\gamma}$ and $\theta_{\gamma'}$ are the angles between the direction connecting the virtual light source with the illuminated point and the normals (Figure 6.1b), and $v$ is the visibility indicator, which is 1 if the two points are visible from each other, and zero otherwise.

If we implement the classical virtual light sources algorithm, we can notice that the generated images are quite good even when the number of the shooting paths is moderate (i.e. hundred), but annoying spikes appear at the corners of the scene (Figure 6.2). The generally good behavior is due to the fact that we reuse all shooting paths for each pixels, thus a pixel color is estimated by reasonably many paths. However, these paths cannot be optimal from the point of view of importance sampling. This is obvious if we inspect equation 6.1. Note that this is a Monte-Carlo estimate, which is good if it is close to a constant value. However,
this estimator contains the BRDFs and the cosine angles at the visible point and at the virtual light source, which can change quickly. Furthermore, the formula contains a division by the square distance, making the estimate unbounded and very rapidly growing at certain regions where \( \vec{x} \) and \( \vec{y} \) are close to each other, e.g. in the corners of the virtual scene (Figure 6.2).

These artifacts can be greatly reduced by bounding (clamping) the estimate. We have found the following clamping heuristic particularly efficient. If the distance from the virtual light source is smaller than a given threshold, then the threshold is used instead of the actual distance in equation 6.1. Figure 6.3 shows the effect of setting the threshold to different values. Although this scheme successfully eliminates the annoying artifacts, it makes the method biased \([WKB^+02]\) because it reduces the perceived intensity, making the corners of the scene look darker than expected.

Figure 6.3: Using increasing clamping thresholds in a unit-sized box to eliminate corner spikes

### 6.3 The new method for calculating the contributions

We shall improve the classical virtual light sources algorithm by establishing a priority scheme for the light sources, thus we can use only the relevant samples and do not waste the computation time on those, whose contribution would be negligible anyway. The samples are only partially evaluated, but very quickly and with the support of the graphics card. Then only those samples are selected from the partially evaluated set, which can have significant contribution to the result.

When the impact of the virtual light sources onto a point visible from the camera is computed, first we have to determine whether or not each virtual light source is visible from that point, and in case of visibility its contribution is added. According to the notations of equation 6.1, let us introduce the term called potential contribution:

\[
C(\vec{y}, \vec{x}) = \Phi^\text{in} \frac{f_r(\vec{y}) \cdot \cos \theta_y \cdot f_r(\vec{x}) \cdot \cos \theta_x}{|\vec{x} - \vec{y}|^2}.
\]

If points \( \vec{x} \) and \( \vec{y} \) are visible from each other, the real contribution is equal to \( C(\vec{y}, \vec{x}) \). If the two points are occluded, then the real contribution is zero. The commonly used way to calculate the real contribution is as follows. First we perform the visibility test, which traces a shadow ray from \( \vec{x} \) to \( \vec{y} \), and in case of visibility we calculate the real contribution:
for each virtual source x {
    if (IsLightSourceVisible(x,y))
        intens += GetContribution(x,y);
}

However, the visibility test seems to be the most time consuming part of the application
even if sophisticated space-partitioning schemes (e.g. kd-tree) are applied. Thus we propose
to change the order of the computation: the potential contributions are computed first, and
the shadow ray is only traced if a non-negligible contribution is possible.

CalculatePotentialContributions(y);
for each virtual source x {
    if (x.potential_contribution > threshold)
        if (IsLightSourceVisible(x,y)
            intens += x.potential_contribution;
}

In the following cases the potential contribution is zero or negligible, so we can omit the
visibility test:

• the distance between $\vec{x}$ and $\vec{y}$ is large,
• the surfaces are not facing toward each other,
• the BRDFs in $\vec{x}$ and $\vec{y}$ do not spectrally overlap.

Thus we can obtain speedup if we evaluate the potential contributions first, and check the
visibility for only those virtual lights that can make any difference in the result. However,
reducing the time spent on visibility check may increase the relative time spent on the com-
putation of potential contributions. This computation takes about 10-20% of the total time
even if simple BRDF models are used, and when the visibility of all virtual lights are eval-
uated. This part can easily grow over 50% if the BRDFs are more complex or when less than
50% of the virtual lights are checked for visibility. Thus we also propose the calculation of
the potential contributions with the graphics hardware, which is an order of magnitude faster
than on the CPU, and can be done parallel to the visibility checks.

6.3.1 Reducing the number of evaluations by truncation

If the point of interest is close enough to one or more virtual sources, their contributions may
be large enough to make the contribution of other virtual lights negligible. In this case, the
evaluation of the remaining sources becomes unnecessary. So, in order to save computational
time, large contributions should be evaluated first.

Let us sort the virtual light sources according to the $C(\vec{y}, \vec{x})$ potential contributions, i.e.
we do not take care of the visibility factors while sorting the contributions (Figure 6.4).

\[
\begin{array}{cccc}
C(y_1) & C(y_2) & C(y_3) & C(y_4) \\
\end{array}
\]

Figure 6.4: Sorted potential contributions

Then we check the visibility of the virtual light sources in this sorted order, starting with
the source that is having the highest potential contribution. If a virtual light source is visible,
its contribution is added to the pixel color.

This processing should continue until the not yet processed virtual lights might have just
a negligible effect on the pixel color. This is the case if:
• the pixel color (in all channels) is larger than the most intensive color that can be displayed, thus any further addition will “burn in”,

• the pixel color is significantly larger than the expected contribution of the not yet processed lights.

If these termination criteria are met, the processing of the virtual lights can stop. Thus we do not have to trace unnecessary shadow rays, saving a lot of processing time.

When evaluating the second criterion, we can assume that the probability of the visibility is similar in case of the not yet processed lights as in the case of the already processed ones. (In fact, a small contribution is usually the result of the large distance between the visible point and the virtual source, which increases the probability of occlusion. So our estimation is usually pessimistic in this way.) Thus, during the evaluation of the stop criterion the potential contributions can be used instead of the real ones. Let us introduce the term evaluation rate (\(x\%\)) as follows:

\[
x\% = \frac{\text{evaluated contribution}}{\text{evaluated contribution} + \text{remaining contribution}}
\]

\[
x\% \approx \frac{\text{evaluated potential contribution}}{\text{evaluated potential contribution} + \text{remaining potential contribution}}.
\]

Note that due to the large variation of the potential contributions, it can happen that just a few virtual lights are responsible for almost all contributions. In this case the visibility of only these virtual lights should be determined, and the remaining virtual lights and their potential contributions can be discarded without further computation, which saves a lot of visibility checks. Figure 6.5 shows the empirical cumulative distribution of the contribution, and demonstrates that about 50% of the virtual light sources are responsible for about 90% of the illumination of a given point.

![Figure 6.5: Empirical cumulative distribution of the contributions of the virtual light sources onto a given point](image)

In each step, no matter whether or not the light source is visible, the evaluated potential contribution is increased by the potential contribution of the given light so it always stores the potential contribution of the processed virtual lights. When the termination condition becomes true, we return the calculated intensity. In order to compensate the bias caused by ignoring some virtual lights, the contribution of the really computed virtual lights is scaled up by the ratio of the total potential contribution and the potential contribution of the evaluated lights.
6.3.2 Exploiting coherence between neighboring pixels

In most cases, when moving from one pixel to an adjacent one, only slight differences occur in the potential contributions of the respective virtual light sources. This statement is especially true for high-resolution images. Thus, if we have sorted the virtual lights according to their potential contributions at one pixel, stepping to the next pixel and recalculting the contributions will generally cause small changes, breaking the sortedness a couple of times only. In our naive implementation we applied quicksort (qsort) for each pixel, but the aforementioned partial sortedness enables us to use special sorting techniques.

Probably the simplest way to exploit partial sortedness is the reduction of the number of qsort calls. That means that we choose an appropriate integer value \( N \) and evaluate qsort only once for every \( N \) pixels. Obviously, the overhead caused by the quicksort algorithm can be greatly reduced in this way. But there is another factor that should also be taken into account. If we perform quicksort too rarely, array may become dramatically unsorted after a given number of pixels. Thus, to achieve the same cumulated intensity we may need to evaluate more virtual sources by performing expensive visibility tests, increasing computation time. For the presented 320 × 200 resolution test scene, we have found that the optimal \( N \) value is between 5 to 10 (Figure 6.6a).

A bit more sophisticated way to exploit partial sortedness is as follows. We apply one run of the bubblesort algorithm for each pixel, i.e. we examine the neighboring potential contributions and swap the corresponding virtual sources if necessary. Since we apply only one run of the algorithm, only linear time is needed. Then we examine the sortedness of the array as follows. Let us call the most recently evaluated contribution as the “critical contribution”. In sorted case, all evaluated contributions are greater and all discarded ones are smaller than the critical contribution. So, the sortedness of the array can be described by counting how many values are not in the aforementioned relation with the critical contribution. If the number of these wrongly placed contributions exceeds a predefined limit, the calculation for the current pixel will be “rolled back” and repeated from scratch after performing a qsort step to sort the array completely. As Figure 6.6b shows, the acceptable rate of the wrongly placed elements is approximately 10-30% in case of the test scene (Figure 6.7).

Since the two methods use different approaches and different parameters, Figure 6.6a and Figure 6.6b cannot be compared directly. To compare them, the average relative error was also
Figure 6.7: Evaluating quicksort only if the number of wrongly placed contributions exceeds 5% of the number of virtual light sources. White points represent evaluation points.

calculated for different parameter settings, and paired with the computation time (Figure 6.8). All test cases evaluate 90% of the possible contributions.

The result of Figure 6.8 may be surprising at the first glance. Namely, the adaptive algorithm, shown with dashed lines did not perform better at all. A possible explanation is that while the adaptive algorithm can detect abrupt illumination changes such as sharp edges and corners and corrects the resulting errors immediately, it may fail for smooth transitions such as soft shadows, accumulating the error along several pixels. In the latter case, the regular algorithm can even outperform the adaptive one.

Figure 6.8: Displaying point pairs of (computation time, average relative error) for the test scene. Evaluating quicksort regularly, for every Nth pixel (solid), and adaptively, only when the array becomes too unsorted (dashed).

6.4 Implementation

The potential contribution must be computed for every pixel and every virtual light source. In practical cases the number of pixels and of the virtual lights are a million and a few hundreds, respectively, which means that the simple formula of the potential contribution is evaluated nearly a billion times. To speed up this computation, we evaluate the potential contributions by the GPU. The following Cg program gets visible points $x$ and virtual light points $y$, unit normals $xnormal$ and $ynormal$, and diffuse BRDFs $xbrdf$ and $ybrdf$, and computes the required
potential contribution in variable $C$:

```c
float3 xtoy = y - x;
float xydist2 = dot( xtoy, xtoy );
xtoy = normalize( xtoy );
float cthetax = dot( xtoy, xnormal );
float cthetay = -dot( xtoy, ynormal );
cthetax = (cthetax > 0) ? cthetax : 0;
cthetay = (cthetay > 0) ? cthetay : 0;
float scale = cthetay * cthetax / xydist2;
C = ybrdf * xbrdf * scale;
C.a = (C.r + C.g + C.b) / 3; // luminance
```

Note that the luminance of the potential contribution is also placed into the alpha-channel of the result. This luminance value will be the base of sorting. In case of non-diffuse materials, the BRDF computation can also be included in this program. In order to speed up the parameter downloading to the GPU, the virtual lights, together with their normals, BRDFs and serial number are stored in a display list.

The computed potential contributions are written into the `pbuffer` of the graphics card, which can store floating point values. The pbuffer is organized as an image, i.e. as a two dimensional array. This array is filled up by all the potential contribution values of subsequent pixels, and when it is full, the content is uploaded to the main memory by a `glReadPixels` call. The number of potential contribution values is the product of the number of virtual light sources and the number of pixels of the image. The number of values that can be stored in the pbuffer depends on its resolution. This means that the number of the pbuffer uploads equals to ratio of the pixels of the of the image and of the pbuffer, multiplied by the number of virtual light sources.

According to our measurements, the uploading is the most time consuming part of the process, which is responsible for about 50% of the computation time. On the other hand, note that when the potential contribution of subsequent pixels are uploaded, the CPU can immediately start sorting them and checking the visibility. Thus while potential contributions of the next group of pixels are evaluated by the GPU, the CPU can work on the previous group of pixels. The two processes run in parallel.

Thus we also proposed the evaluation of the potential contributions on the graphics hardware. This is not only about ten times faster than on the CPU, but the two process can run in parallel, so the potential contributions can be obtained at no added time. The presented Cg program can be can be subdivided, executed partly on the vertex shader and partly on the pixel shader, thus the processing power of both processors can be taken advantage of. Since the reading of the results is the bottleneck of the GPU operation, the subdivision of the work does not significantly affect the computation time.

Figure 6.9 shows the computation times for a $512 \times 512$ resolution image and for different virtual lights. For example, when the number of virtual lights is 1000, we computed $1000 \times 512 \times 512 \approx 2.6 \cdot 10^8$ potential contributions in about 10 seconds, which means less than 39 nsec per each potential contribution, including both the calculation and the average pbuffer upload. This is an order of magnitude faster than on the CPU.

### 6.5 Results

The proposed algorithm has been implemented in C++/OpenGL/Cg environment. The following running times were measured on an Intel P4/1.3GHz CPU and ATI/Radeon 9700 Pro graphics card. The images were rendered on $320 \times 200$ resolution.

On the scene of Figure 6.10 all of the materials are diffuse, and therefore about 85% and 15% of the computation time were devoted to the visibility calculation and to the evaluation of potential contributions, respectively, when the classical method has been applied. When
turning to the new method, we concluded that the sorting, i.e. the overhead of the new method, might correspond to 5% of the total time. By applying sorting in every 5th pixel only, this overhead could be decreased below 1%.

To render the images of Figure 6.10, we initiated 20 shooting paths, which resulted in 117 virtual lights (the average albedo of the scene is 0.8). Figure 6.10 also contains the threshold ($x\%$), the average shadow rays per pixel and the total rendering time for each image. The average number of shadow rays per pixel was 80 when the original algorithm was executed (the remaining $117-80=37$ virtual lights could be excluded because the surfaces at the virtual light and at the visible point were not facing toward each other). When turning to the new algorithm, setting the threshold to 90% and 80%, the average number of shadow rays could be reduced to 57 and 45, respectively. We can conclude that if setting $x\%$ to 80–90% the quality is close to that of the original image, but the new method halved the computation time. Note that this speed up is available even for the simplest BRDF models. For more complicated BRDFs, we can expect even higher improvements.

The same experiments have been repeated in a scene containing both diffuse and specular components. In this case, 432 virtual lights were generated in 50 shooting walks (Figure 6.11). The general trend was similar than in the previous experiment, the threshold is worth setting at 0.8–0.9, which results in a speedup of factor two.

### 6.6 Conclusions

We proposed a speedup of the virtual light sources algorithm based on the recognition that the contribution of the virtual light sources can be very different depending on their distance from the point of interest. Thus ignoring those virtual light sources whose potential contribution is much smaller than that of other sources, we make an acceptable error but save tracing a lot of shadow rays. As a rule of thumb, we concluded that in normal scenes only 50% of the virtual light sources are responsible for 90% of the illumination of a given point. When this idea is exploited or the material models are more complex, the calculation of the potential contribution can be a bottleneck of the algorithm. Thus we also proposed the evaluation of the potential contributions on the graphics hardware. This is not only about ten times faster than on the CPU, but the two process can run in parallel, so the potential contributions can be obtained at no added time.
Figure 6.10: Comparison of classical virtual light sources method and the new method using 117 virtual light sources

(a) original method
16.4 secs

(b) $x\% = 90$, # virtual lights = 57
9.8 secs

(c) $x\% = 80$, # virtual lights = 45
7.8 secs

Figure 6.11: Comparison of classical virtual light sources method and the new method using 432 virtual light sources

(a) original method
75.5 secs

(b) $x\% = 90$
48.7 secs

(c) $x\% = 80$
37.7 secs
Figure 6.12: Comparing (a) the original method and the new method for $x\% = 90$. The relative error (right) has been inverted for printing purposes, i.e. darker color means larger error.
Figure 6.13: Comparing the original method (top) and the new method for $x\% = 80$. The relative error (right) has been inverted for printing purposes.
Chapter 7

Conclusions

7.1 New scientific results

Thesis I: Fresnel term approximations for metals [P1]

By modifying the formula proposed by Schlick, I have proposed an accurate simplification of the Fresnel function:

\[
F^*(n, k, \cos \theta) := F^*_\parallel + (1 - F^*_\parallel)(1 - \cos \theta)^{\frac{5}{2}},
\]

where

\[
F^*_\parallel = \frac{(n - 1)^2 + k^2}{(n + 1)^2 + k^2}.
\]

The error of the approximation can be further reduced by applying a rational approximation to the residual error. The proposed formulas can also cope with complex refraction indices, thus it can present realistic metals and is simple enough to be implemented on the graphics hardware, and used in games.

Thesis II: Rendering reflective surfaces using approximate raytracing on the GPU [P2, P3]

I have proposed a fast approximation method to obtain the point hit by a reflection ray. The method solves ray equation \( \vec{q} = \vec{x} + \vec{R} \cdot d \) using the distance values stored in environment map texels. An initial guess \( \vec{l} \) can be refined as follows:

\[
d_l = d_p + |\vec{r}| \cdot \left(1 - \frac{|\vec{l}|}{|\vec{p}|}\right),
\]

while guesses \( \vec{l} \) and \( \vec{p} \) can be further refined by interpolation using the following formula:

\[
d_{new} = d_l + (d_l - d_p) \cdot \frac{1 - |\vec{l}|/|\vec{p}|}{|\vec{l}|/|\vec{p}| - |\vec{l}|/|\vec{p}|},
\]

where \( d_l \) and \( d_p \) are the corresponding ray parameters. The approximation method can be used to localize environment mapped reflections, that is, to make them depend on where they occur. The method is fast and accurate if the scene consists of larger planar faces, when the results are similar to that of ray-tracing.
Thesis III: Approximate rendering of glossy and matte reflections on the GPU [P4, P5, P6]

I have proposed a fast approximation method to obtain the indirect diffuse or glossy reflection on a dynamic object, caused by a diffuse or a moderately glossy environment. Instead of tracing rays to find the incoming illumination, we look up the indirect illumination from a cube map rendered from the reference point that is in the vicinity of the object. To cope with the difference between the incoming illumination of the reference point and of the shaded points, we apply a correction based on the geometric information also stored in cube map texels. The resulting formula allows us to localize the solid angle $\Delta \omega$ with regard to the shaded point:

$$\Delta \omega^* \approx 2\pi - \frac{2\pi}{\sqrt{1 + \left((\frac{2\pi}{2\pi - \Delta \omega})^2 - 1\right) \cdot \frac{r^2}{r^*^2}}}$$

where $r$ and $r^*$ are the distances from the environment, measured from the reference point and the shaded point, respectively. The method computes indirect illumination although approximately, but providing very pleasing visual quality.

Thesis IV: Speeding up the virtual light sources algorithm with the GPU [P7]

I proposed two techniques to speed up the classical virtual light sources algorithm. Firstly, the contribution of the virtual sources is calculated with the support of the graphics hardware, without taking care of the visibility factor. Secondly, by establishing a priority scheme, the evaluation of a sample is only completed if the sample can have a significant effect on the result. Since the contributions can be computed independently of the visibility tests, we can utilize the computing power of the graphics card to increase performance.

7.2 Application of the results

The primary application area of the proposed method is the introduction of these effects in games. The proposed methods were integrated into a rendering engine as a part of the GameTools FP6 (IST-2-004363) project (www.gametools.org).
Publications


Bibliography


83


Index

ALU, 15
angle (angular extent), 6
attenuation factor, 12
average incoming radiance, 55

bounding, 68
BRDF, 8
BRDF sampling, 15
brightness, 4
bubblesort, 71

clamping threshold, 68
complex refraction index, 12
computer games, 4
convolution, 22
corner spikes, 68
diffuse environment map, 22, 51
directions, 6
displacement mapping, 21
distance impostors, 36

diffuse environment map, 22, 51
directions, 6
displacement mapping, 21
distance impostors, 36
energy conservation, 10
environment mapping, 35, 53
error term, 27
evaluation rate, 70
extinction coefficient, 12

false position root finding method, 40
Faraday cage, 5
final gathering, 16
flux, 7
foreshortened area, 6
Fresnel term, 5
full-screen quad, 18
global illumination, 4, 16, 66
GPUGI, 16
Helmholtz reciprocity, 9, 10

Helmholtz reciprocity, 9, 10
illuminating points, 51
illumination information generation, 16
image based lighting, 19
image based rendering, 21
importance sampling, 15

index of refraction, 12
interactive applications, 4
irradiance, 7, 22
irradiance sensor, 9

light source sampling, 15
local illumination, 4
Lord Rayleigh, 9
luminance, 15
Maxwell equations, 13
microfacet based material model, 10
microfacet incidence angle, 11
motion parallax, 20
multiple environment maps, 20, 22
optically rough, 5
optically smooth, 5
overshooting, 40

partial sortedness, 71
perpendicular surface assumption, 36
physically plausible, 10
planar surface assumption, 36
point-to-disc, 57
potential contribution, 68
power, 7
Precomputed Radiance Transfer, 22
projected area, 6
proxy geometry, 20
quadrature, 14
quicksort, 71

radiance, 7
random walk, 15, 66
rational approximation, 27
ray, 4
ray engine, 18
ray flipping, 43
ray marching, 50
ray parameter, 36
reciprocity, 9
reference point, 19
reflectivity, 55
render-to-texture, 16
rendering, 14
rendering equation, 4, 8
Russian roulette, 15

secant root finding method, 39
shaded point, 4, 51
shadow maps, 22
shadow ray, 66
Snell’s law, 13
solid angle, 6
specular environment map, 22, 51
spherical harmonics, 22
symmetry, 9

undershooting, 40

virtual light source, 66
virtual light sources, 15, 52

wave model, 11
wavelets, 22