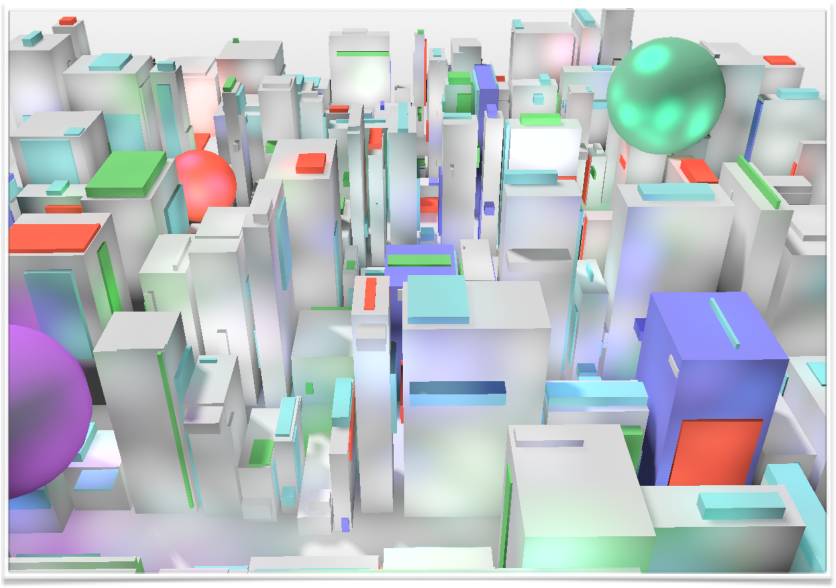
Real-Time Global Illumination for In- and Outdoor Scenes



By: John McLaughlin

Supervisor: Jun.-Prof. Grosch

2nd Supervisor: Prof. Theisel

Institute for Simulation and Graphics,

Otto-von-Guericke University, Magdeburg

Thursday, 15 December 2011

Table of Contents

1.Introduction and Goals

1.1. Introduction

This thesis, *Real-Time Global Illumination for In- and Outdoor Scenes*, covers the computation and use of indirect lighting in real-time environments, such as modern video games. These environments place great limitations on the available resources and through the necessary computation, as there are several necessary systems running at the same time, almost all of them needing to be updated each frame, at 30-60 frames per second on normal consumer hardware. Even when aiming to use modern graphics hardware features, rendering realistic global illumination in real-time is still a very complex problem.

Indirect illumination can give important cues to improve the visibility and understanding of complex 3-dimensional scenes to the viewer. Especially in scenes with large shadow areas where the light does not directly reach any surfaces, indirect illumination is important to perceive additional details and geometric structure of the scene. Improved light calculation and rendering can be used to improve various fields, such as *visual effects in films*, either for completely digital productions or to add virtual elements to a live movie production seamlessly; in *architectural design* for creating realistic visual simulations with in- and outdoor illumination and to design illumination inside a building for creating improved work or living areas; or in computer games for creating an immersive interactive environment meeting ever increasing demands for realistic rendering at high speeds [DBB06]. The transition between different lighting conditions, usually meaning different light sources of inside and outside illumination, can also be an important way to deliver a realistic representation of an interactive environment with the added possibility of effectively combining both.

The main difficulty of rendering indirect illumination at interactive speeds is the great complexity of various lighting problems like ambient occlusion, soft shadows, reflection, refraction and caustics, which need to be calculated in a very short amount of time, usually employing various approximations in a way which still creates seemingly realistic images.

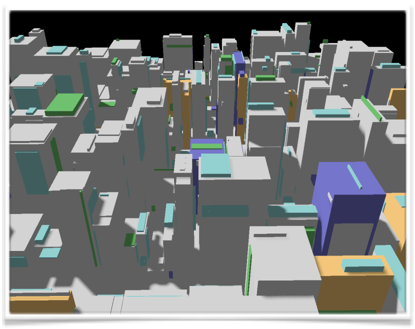
On the other hand, offline or unbiased rendering methods without approximations can take minutes or even hours to complete for a single image and achieve far better and physically more accurate results. Some of the concepts of these approaches can be adapted to be used for precomputing the lighting information for later use during real-time rendering, but can take hours or even days to complete ([BEAST], [ME10]). Delivering a complete, realistic global illumination rendering at real-time speeds including multiple dynamic lights and objects is a complex undertaking, requiring further research and improvement.

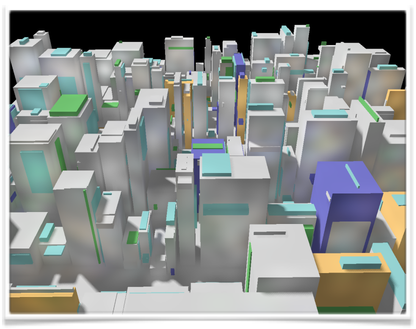
In this thesis I aim to find an approach for rendering indirect illumination for in- and outdoor scenes including refraction and reflection effects at real-time speeds, efficient enough to be used in modern interactive environments. It should have no pre-processing requirements, should handle arbitrary complex geometry and should process all lighting computations on the GPU, using parallel computations to improve performance.

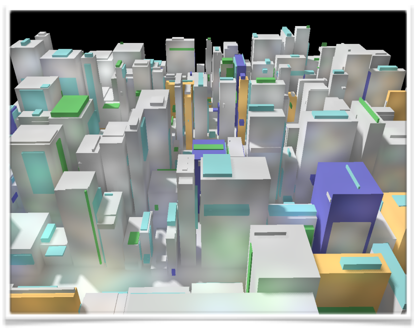
1.2. Motivation

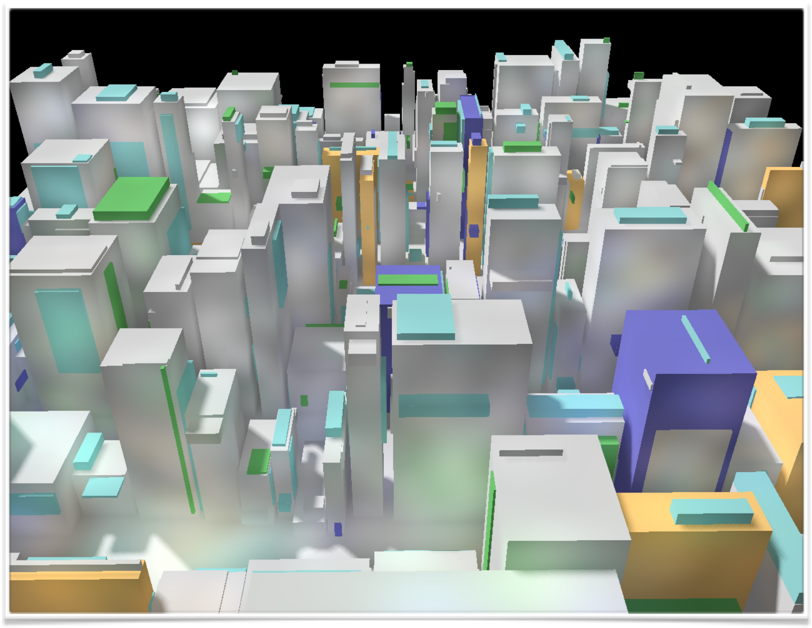
In the following I would like to further detail why global illumination, esp. indirect illumination, is an important (and difficult) problem in modern computer graphics. Correct indirect shadows and light can display a much more accurate version of a scene, including details and effects which would be otherwise lost if only direct lighting was used. Even direct lighting with added ambient occlusion only provides a rough approximation and can fail to accurately deliver the full lighting information. Important factors such as color bleeding, caustics and soft shadows give the viewer a better indication of the relationships between different objects and environments.

It may seem to difficult to produce realistic results when calculating global illumination at real-time speeds and even unrewarding, because the necessary approximations introduce new artifacts and problems. As an example, when rendering light bouncing off different surfaces, it may seem unlikely to get a realistic result without an unlimited number of light bounces, at least until light is completely absorbed. In real-time implementations, usually only one or two of these bounces or indirections are computed per light ray, light path or photon. As can be seen in the following pictures, the perceptual difference when calculating multiple light bounces becomes increasingly negligible. These approximations, however, do demand careful consideration and a precise configuration for different scene or geometry complexities.


direct only


first bounce


second bounce


third bounce

A faster calculation and rendering of completely lit scenes can also speed up production time of 3D content even more than "*Lightspeed*" [RKS07], which delivers a high quality preview of global lighting for film production using specialized framebuffers, shaded and rendered on the GPU to deliver faster results than traditional CPU rendering. This rendering preview and similar approaches used in modern 3D content creation software require several seconds or minutes to render completely. An interactive approach would allow an active 3D-view using global illumination, instead of direct illumination and simple shading, as is used in most current modeling programs. Of course, content creation has different time constraints than an interactive environment like a computer game, usually requiring higher quality but lower frame rates. This should also be considered while creating the implementation, making it adaptable enough to be used in both contexts.

In this approach I use the concept of photon mapping, which is further described in the following chapter, [2. Prerequisites and Theory]. It solves multiple problems of both the raytracing and radiosity concepts, e.g. handling of diffuse, specular and transmissive materials or effects like color bleeding and caustics. It also allows certain approximations to significantly speed up the process using a highly parallelized GPU implementation, one of the main optimization advantages being the parallel nature of photons.

There are of course already a number of approaches for rendering global illumination in real-time, the most relevant of these to my thesis topic are further detailed in [3.1. Previous Approaches], including advantages and drawbacks.

1.3. Goals

The main goal is to compute realistic indirect lighting for in- and outdoor scenes. The secondary goal is to speed up this calculation so it can be used in modern high-end applications. The implemented prototype should show the basic function of the approach and its ability to handle arbitrary scenes and complexity while still rendering at real-time speeds. This prototype should include the basic photon mapping approach and should show that a realistic and visually pleasing result is possible. At least diffuse scattering should be implemented, since specular and transmissive materials are not essential to the prototype, as the basic workings of this solution can be shown with only diffuse scattering. The theory does include specular and transmissive materials and the system is built to also handle these materials.

The concept and implementation should provide an example of possible integration into existing pipelines, which should allow commonly used post-processes and effects, e.g. bloom, motion blur, depth blur and tone mapping. It should also not require any pre-computations or additional work such as generating simplified geometry or pre-computing light-maps as this would limit its usefulness and application possibilities.

2. Prerequisites and Theory

In the following sections, short descriptions of various important theories, key words and variables used in the implementation and the remaining parts of the thesis are presented.

It is assumed that the reader will have basic knowledge of computer graphics, esp. rendering equations, light computation and also Real-Time GPU programming. Resources to further study these topics relating to this thesis include [AHH08], [J01] and [DBB06].

The reader should also be aware of the capabilities of current generation graphics hardware, i.e. the feature set of DirectX11 [DIRECTX], including Shader Model 5.0, as well as GPGPU environments such as [CUDA] or Compute Shaders.

Most of the following concepts are further explained in [J01] and [DBB06]. Only the parts which are necessary for understanding this thesis are detailed here.

2.1. Light Models

This is an overview of the four main light models commonly used, as described in [DBB06] and [J01] As is most common in computer graphics, photon mapping also uses the geometric model as its basis, thus this thesis uses the same model. This also means that certain effects like diffraction and interference are not included [J01].

2.1.1. Geometric Model

The geometric model, also known as ray optics or geometric optics, describes light as rays which travel through the scene independent of each other. In this model only the interactions between these rays and objects bigger than the wavelength of the light are considered. These interactions are modeled through geometric rules. This makes it possible to render reflection and refraction effects.

2.1.2. Wave Model

The wave model represents light as electromagnetic waves, in which objects that are the same size as the different possible wavelengths of light can be considered. This enables the computation of the same effects as the geometric model, reflection and refraction, with the addition of diffraction and interference.

2.1.3. Electromagnetic Model

The electromagnetic model uses wave optics like the wave model, but additionally explains polarization and dispersion.

2.1.4. Photon Model

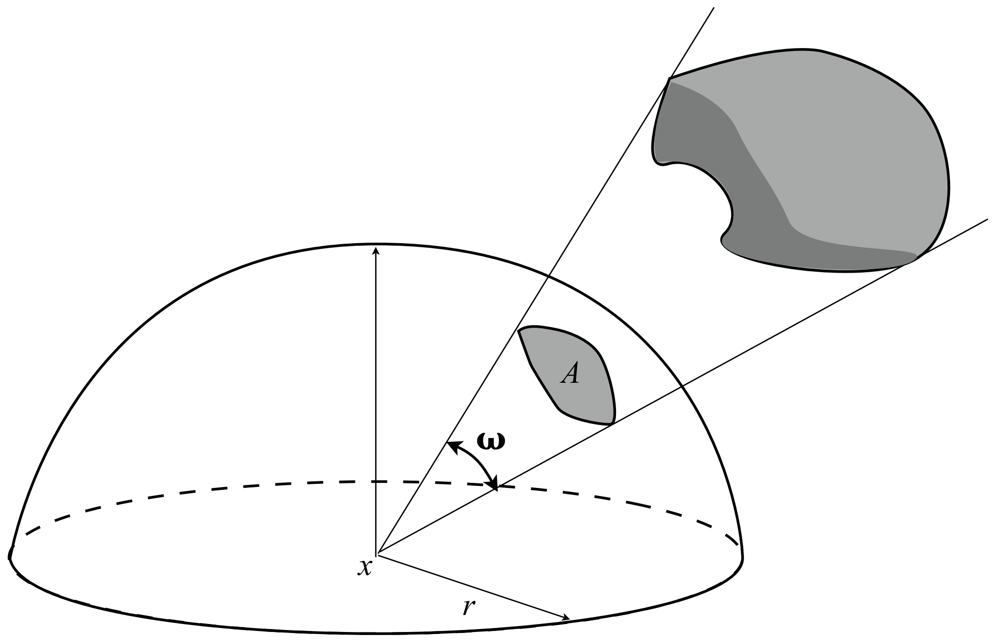
The photon model, also known as quantum model, describes light as light particles or photons. With this the interaction of light and matter on an atomic and subatomic level is explained. Even though the name *photon mapping* might be misleading, this model is not used in this thesis. Please note that a single *photon* in *photon mapping* would be the rough equivalent of a large number of photons in the quantum model.

2.2. Lighting Terminology

The following is the basic terminology to describe light used in this thesis and is mostly based on the terminology descriptions in [J01], [DBB06], [K86] and [N11]. Radiometry is mostly used for measuring and describing electromagnetic radiance. In contrast, photometry also includes the human perception of light in its representation.

2.2.1. Solid Angle

The solid angle describes the area which is taken by an object projected onto the unit sphere, determined by the opening angle needed to encompass this object. This can be thought of as the perceptual size of an object as seen from the center of the unit sphere.



solid angle of an arbitrary object

The solid angle **ω** can be represented by the ratio of the area *A* on the unit sphere and the squared radius *r* of the sphere:

$$\omega = \frac{A}{r^2}$$

solid angle

On the unit sphere the solid angle is equal to the area projected by it. The solid angle also describes the direction of the projected object and as such is usually seen as five-dimensional in a three-dimensional environment.

The differential of the solid angle is used in Monte Carlo ray tracing as it can be used to relate the radiant flux to the intensity of light. The direction and size can be described using spherical coordinates (θ, ϕ) , which requires a base coordinate system, usually built with the surface normal *n* and the surface tangent vectors b_x and b_y. The size of a differential solid angle in spherical coordinates is defined as:

$$dw = \sin\theta d\theta d\phi$$

differential solid angle

Here θ represents the angle between the direction and the normal *n*, ϕ represents the angle between the direction on the surface tangent plane and b_x. With these two angles, the direction of the solid angle can be calculated as follows:

$$sin\theta cos\phi b_x + sin\theta sin\phi b_y + cos\theta n$$

solid angle direction

Even though the solid angle is seen as dimensionless, its SI units can be described using steradian *sr*, e.g. the solid angle of a sphere from its interior would be 4π *sr*.

2.2.2. Radiometry

2.2.2.a. Photon Energy

The photon describes the quantity with which radiance can be measured, i.e. radiance can be seen as consisting of photons. Each photon has energy with the wavelength λ, defined as:

$$e_\lambda = \frac{hc}{\lambda}$$

photon energy

where *h* is Plancks constant and *c* is the speed of light (*c* = 299 792 458 *m/s*). [J01] Please note that a photon in photon mapping represents a collection of physical photons, since the computations are based on geometric concepts in contrast to radiometry where radiance is calculated and measured in terms of the photon model.

2.2.2.b. Radiance

The spectral radiant energy Q_λ of n_λ photons with wavelength λ is

$$Q_\lambda = n_\lambda e_\lambda$$

spectral radiant energy

Using this, the radiant energy of all photons can be calculated by integrating their spectral energy over all (light) wavelengths:

$$Q = \int_0^\infty Q_\lambda d\lambda$$

radiant energy

Radiant flux Φ is the rate of flow of radiant energy over time and is calculated as:

$$\Phi = \frac{dQ}{dt}$$

radiant flux

This can also be calculated wavelength dependent as the spectral radiant flux Φ_λ, i.e. the rate of spectral radiant energy over time.

Radiant flux area density describes the change of radiant flux Φ per differential area A. This is commonly separated into two parts: Radiant exitance *M*, i.e. radiant flux *leaving* the surface A, also known as radiosity *B*, and irradiance *E*, which is the radiance *arriving* at a surface A:

$$\underbrace{E(x) = \frac{d\Phi}{dA}}_{irradiance}$$

The radiant flux reaches surface point *x* from every direction and without considering the material properties of the surface at point *x*. Radiant exitance *M* then describes the emitted and reflected light in every direction.

The radiant intensity *I* represents the radiant flux per solid angle, *dw:*

$$\underbrace{I(w) = \frac{d\Phi}{dw}}_{radiant\ intensity}$$

Finally the radiance *L* is calculated as radiant flux per differential area dA per solid angle:

$$\underbrace{L(x,w) = \frac{d^2\Phi}{\cos\theta dAdw}}_{radiance}$$

L(x,w) is the radiant flux that reaches surface point *x* from direction *w*.

Radiance is very important for global illumination, since it most closely describes the color of an object. It can be thought of as the number of photons reaching a certain area in a certain amount of time and can also be used to describe the light intensity at a given point in a given direction. Most ray-tracing approaches, as well as photon mapping, make use of the fact that the light intensity is constant along a straight line inside a vacuum. This of course changes when introducing participating media [CPPSS05].

2.2.3. Photometry

In addition to radiometry, photometry also plays an important role, since it includes the perception of light by the human eye. In the calculation of lighting usually only radiometry is used, but photometric properties can be calculated as post-effects. This process is commonly known as tone mapping.

2.2.3.a. Luminance

Luminous flux Φ_v describes the perceptual equivalent to radiant flux:

$$\Phi_v = \int\limits_{\Lambda} \Phi_\lambda V(\lambda) d\lambda$$

luminous flux

where *V(λ)* is the visual response of the observer and Λ represents the wavelengths of light in the visual spectrum.

The luminous flux area density, $\frac{d\Phi_v}{dA}$ similar to radiant flux area density, is called illuminance (similar to irradiance) E_v if it is incoming. If outgoing it is called luminous exitance, similar to radiant exitance, M_v.

Luminous intensity, I_v is the flux per solid angle, similar to radiant intensity:

$$I_v = \frac{d\Phi_v}{dw}$$

luminous intensity

Finally the luminance, i.e. the perceptual radiance for a human observer is:

$$L_v(x, w) = \frac{d^2\Phi_v}{cos\theta dA dw}$$

luminance

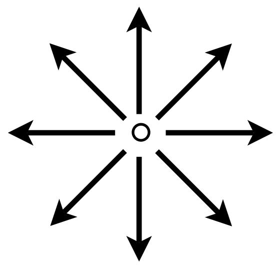
2.3. Light emission

Light emission describes the amount of light or photons emitted by each light source and the way it is emitted, i.e. how much is emitted in each direction. This can take the form of various light sources which can be modeled as point lights, spotlights, area lights, directional lights or complex lights.

Directional light is mainly used to approximate sunlight, as this can be seen as virtually parallel, since the difference of angles between light rays reaching the earth is negligible.

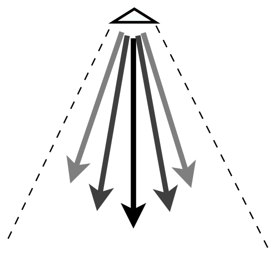
In the following I name and describe the light source types relevant for this thesis.

2.3.1. Point Light


point light

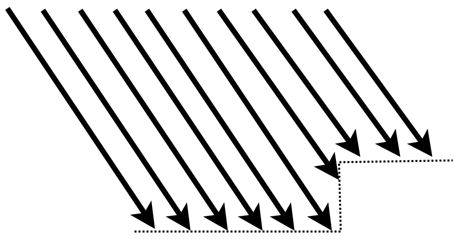
A point light is easiest to describe since it emits an equal amount of light or photons in every direction. They are usually defined by their position and intensity. The intensity of a point can vary by direction, but is usually considered constant. Point lights with constant intensity are also known as omni lights. Even though this type of light does not occur in nature, it is an approximation commonly used in computer graphics. In this thesis only spotlights and directional light (representing the sun) are implemented, albeit a Point light could be easily approximated by using six spotlights and rendering into a cube map. [AHH08]

2.3.2. Spotlight


spotlight

The spotlight is defined by its position, direction and falloff, it sends out light with a certain falloff described with an opening angle and distance, this can also be simulated by a point light with a special falloff function, i.e. the intensity decreases when approaching the maximum opening angle of the spotlight. The falloff of these lights can be divided into three zones: An inner cone with constant intensity, in between inner and outer cone with the intensity decreasing, and beyond the outer cone where there is no direct light. [AHH08]

2.3.3. Directional Light


directional light

The simplest light to simulate is that which comes from very distant light sources, like the sun, and can be approximated with parallel light rays. This means that every surface is reached by the light from the same direction, with the same intensity. A directional light thus only needs a direction and the amount of light emitted to be fully defined.

2.4. Light Scattering

Light scattering describes how the light is reflected or refracted by different materials, which are usually divided into diffuse, specular or reflective and transmissive materials.

2.4.1. Bidirectional Reflectance Distribution Function

The Bidirectional Reflectance Distribution Function (BRDF) is used to describe the reflectance properties of different materials. To include subsurface scattering in the function, the bidirectional surface scattering reflectance distribution function (BSSRDF) would have to be used, but since I approximate the incident light exiting at the same point it hit, I only need to calculate the reflectance using the BRDF.

The BRDF is used to describe the reflectance properties of diffuse, specular and glossy materials with varying roughness.

The BRDF f_r describes the relation between the radiance reflected in direction *w* and the incident irradiance from direction *w'* at point *x*:

$$f_r(x, w', w) = \frac{dL_r(x, w)}{dE_i(x, w')} = \frac{dL_r(x, w)}{L_i(x, w')(w' \cdot n)dw'}$$

bidirectional reflectance distribution function

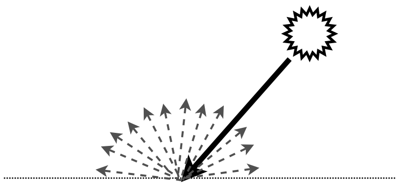
Assuming the incident irradiance for every direction at point *x* is known, the reflected radiance in every direction can be computed by evaluating the integral of L_i over the hemisphere Ω of all incoming directions:

$$L_r(x, w) = \int_{\Omega} f_r(x, w', w) dE_i(x, w') = \int_{\Omega} f_r(x, w', w) L_i(x, w')(w' * n')dw'$$

radiance BRDF

The BRDF has the following important properties: Its value for each incoming direction is independent of the incident irradiance from other directions; interchanging incoming and outgoing direction of the reflectance results in the same radiant flux, which allows the tracing of light rays in both directions; and last but not least the BRDF must adhere to the law of conservation of energy, meaning the total amount of light or power reflected over all directions must be less or equal to the amount of power incident on the surface.

2.4.2. Diffuse Materials


diffuse reflection

Diffuse materials generally scatter light in every direction. Lambert surfaces are a special representation of diffuse surfaces, approximating the scattering as uniform over all directions. The reflected radiance is then constant for every direction and can be calculated as:

$$L_r(x, w) = f_{r,d}(x) \int_{\Omega} dE_i(x, w') = f_{r,d}(x) E_i(x)$$

diffuse BRDF radiance

With the diffuse reflectance ρ_d calculated as:

$$\rho_d(x) = \frac{d\Phi_r(x)}{d\Phi_i(x)} = \frac{L_r(x)dA\int_\Omega dw}{E_i(x)dA} = \pi f_{r,d}(x)$$

diffuse reflectance

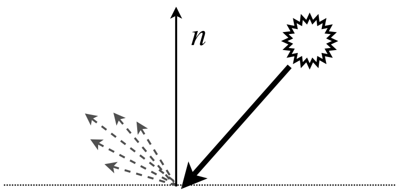
This also means that diffuse Lambert surfaces do not change dependent on view position or direction. The distribution of the reflected direction w_d with this surface is calculated as

$$w_d = (\theta, \phi) = (\cos^{-1}(\sqrt{\xi_1}), 2\pi\xi_2)$$

reflected direction

where $\xi_1 \in [0, 1]$ and $\xi_2 \in [0, 1]$ are uniformly distributed random numbers.

2.4.3. Specular Materials



specular reflection

Specular materials have a smooth enough surface to reflect the light in a more focussed manner. If the surface is without imperfection it can reflect the light perfectly without any scattering. Usually some imperfections are present though, which can be described by a parameter, often referred to as roughness or gloss.

The reflected radiance from specular reflection can be calculated as

$$L_r(x, w_s) = \rho_s(x) L_i(x, w')$$

specular reflected radiance

The mirrored direction of perfect specular reflection is:

$$w_s = 2(w' \cdot n)n - w'$$

perfect specular reflection

2.4.3.a. Fresnel Equations

The amount of light reflected by smooth materials can be calculated using the Fresnel equations. For a given ray which is in a medium with the refraction index η_1 and hits a medium with a different refraction index η_2 the amount of reflected light can be calculated as:

$$\rho_{||} = \frac{\eta_2 \cos\theta_1 - \eta_1 \cos\theta_2}{\eta_2 \cos\theta_1 + \eta_1 \cos\theta_2}$$
$$\rho_{\perp} = \frac{\eta_1 \cos\theta_1 - \eta_2 \cos\theta_2}{\eta_2 \cos\theta_1 + \eta_1 \cos\theta_2}$$

amount of reflected light

This calculation takes polarization into account, $\rho_{\|}$ and $\rho_{\perp}$ representing the reflection coefficient for either light with an electric field parallel or orthogonal to the plane of incidence. The refraction index for most materials can be found in most written resources about optics, e.g. air (η = 1.0), water (η = 1.33) or glass (η = 1.5-1.7).

For unpolarized light the specular reflectance or Fresnel reflection coefficient F_r, can be calculated as:

$$F_r(\theta) = \frac{1}{2}(\rho_{\|}^2 + \rho_{\perp}^2) = \frac{d\Phi_r}{d\Phi_i}$$

Fresnel reflection coefficient

Which can be further approximated as derived by Schlick [S93]:

$$F_r(\theta) \approx F_0 + (1 - F_0)(1 - cos\theta)^5 \approx \frac{d\Phi_r}{d\Phi_i}$$

approximation for Fresnel reflection of unpolarized light

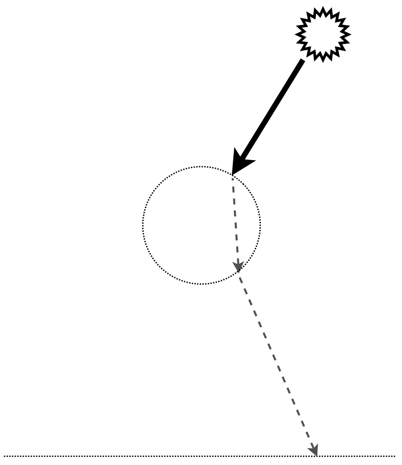
where F_0 is the real Fresnel reflection coefficient at normal incidence:

$$F_0 = \left(\frac{\eta_1 - \eta_2}{\eta_1 + \eta_2}\right)^2$$

Fresnel reflection F_0 [G10]

2.4.4. Transmissive Materials

Transmissive materials not only reflect light but also refract light and transmit it through the material where it is again refracted into another medium, e.g. a light ray could travel from air into glass and into air again. It is approximated that the material only refracts the light on entering and leaving the medium.


transmissive refraction

The reflection and refraction with transmissive materials can also be calculated using the Fresnel equations. The reflection can be calculated in the manner described above. To calculate the refraction we use Snell's law, as shown in [J01]:

$$\frac{\sin\theta_1}{\sin\theta_2} = \frac{\eta_1}{\eta_2}$$

Snell's law

With this the direction w_r of the refracted light can be calculated as:

$$w_r = -\frac{\eta_1}{\eta_2}(w - (w * n)n) - \left(\sqrt{1 - \left(\frac{\eta_1}{\eta_2}\right)^2 (1 - (w * n)^2}\right)n$$

refraction direction

The amount of light refracted is calculated as the remaining amount from subtracting the reflected light from the total amount (1 - F_r).

2.5. Rendering Equation

The rendering equation describes the light transport in a scene and forms a basis for all global illumination calculations. As shown in [K86] the following rendering equation is calculated by every rendering approach, including the Monte Carlo ray-tracing and photon mapping approaches, which are described in the subsequent section:

$$L_o(x, w) = L_e(x, w) + L_r(x, w)$$

rendering equation

where L_o is the outgoing radiance at point x in direction w, as the sum of emitted radiance L_e and reflected radiance L_r. For L_r the BRDF can be used to get the following:

$$L_o(x, w) = L_e(x, w) + \int_\Omega f_r(x, w', w) L_i(x, w')(w' * n) dw'$$

rendering equation with BRDF for reflected radiance

The rendering equation also describes the final state of the lighting. Since this can require a great number of reflections and refractions along each path, it is usually only calculated for a finite number of reflections and refractions.

The equation used here describes the light transport without participating media. For the inclusion of media, which would introduce additional volume scattering, one would have an additional function such as the volume scattering function:

$$\frac{dL(x)}{dx} = \underbrace{\beta^a(x)L_e(x)}_{emission} + \underbrace{\frac{\beta^s(x)}{4\pi}\int_s L(x, \omega_i)p(\omega_0, \omega_i)d\omega}_{in-scattering}$$
$$- \underbrace{\beta^a(x)L(x)}_{absorption} - \underbrace{\beta^s(x)L(x)}_{out-scattering}$$

volume scattering function

This describes the change of light transport inside participating media, i.e. scattering and absorption. β_a describes the absorption coefficient, β_s the scattering coefficient and $L(x, \omega)$ the outgoing radiance. Furthermore, $p(\omega_0, \omega_i)$ is the phase function of scattering method used (e.g. *Rayleigh* or *Mie* for atmospheric scattering), which defines the intensity of the scattered light. More information on participating media and light scattering can be found in [CPPSS05][M10].

2.6. Monte Carlo Ray Tracing

Most ray-tracing approaches for calculating global illumination are based on the classic Monte-Carlo ray tracing. For these approaches the rays are usually traced through the scene, calculating the radiance in the direction of the ray. Each intersection can also be described as 'bounce' since the ray is reflected or refracted in a new direction. One of the advantages of Monte Carlo ray-tracing is that the quality of the rendering is dependent on the number of samples or rays used and not on the complexity of the objects or their materials. The actual computation of the rendering equation is solved by stochastically integrating it over several samples. [DBB06]

Before I describe a number of different ray-tracing approaches using Monte Carlo techniques, a short introduction to Monte Carlo integration is given. For more detail on this topic please see [DBB06].

2.6.1. Continuous Random Number Variables

To better understand Monte Carlo techniques, important concepts of probability theory are briefly reviewed, since the Monte Carlo process is a sequence of a number of random events.

Continuous random variables may assume any value in *[a, b]*, in contrast to discrete random variables which only assume a finite set of values. The mean or expected value of these continuous random variables can be defined as:

$$E(x) = \int_{-\infty}^{\infty} f(x)p(x)dx$$

expected value

where *p(x)* is the probability distribution function (PDF) for the random number variable *f(x)*. The integral over *p(x)* is the cumulative distribution function, which determines the probability of *f(x)* assuming a certain random value in *[a, b]*.

The variance σ of a set of a random numbers describes its spread, i.e. how far the numbers lie from the mean or expected value. This can be defined as:

$$\sigma^2 = E[(x - E(x))^2] = \int_{-\infty}^{\infty} (x - E(x))^2 p(x) dx$$

variance σ

The standard derivation, a way of describing the distribution of random numbers in relation to the mean of the set, can be computed as the integral over the square root of the variance.

2.6.2. Monte Carlo Integration

Monte Carlo integration can be used to evaluate multi-dimensional integration problems of functions with many discontinuities, as are often encountered in rendering, esp. in global illumination approaches. Standard approaches to solve these integrals often do not prove to be efficient enough. An example for a multi-dimensional integral is the aforementioned rendering equation.

The quality of the calculation of the Monte Carlo integration is mainly dependent on the number of samples. It usually needs a very large number of samples to solve an integral with minimal error. It is also has a slow convergence rate of $\frac{1}{\sqrt{N}}$, where N is the number of samples, as described in [J01], [N11] and [DBB06], and thus is most often not the best option to solve an integral. It is however, a viable option for high-dimensional integrals, where there are often no other viable alternatives.

2.6.2.a. Monte Carlo Estimator

The Monte Carlo estimator is the weighted sum of random variables $f(x_i)$, where every x_i has the same probability distribution function $p(x)$.

For an integral

$$I = \int_{a}^{b} f(x) dx$$

integral for function f(x)

the estimator is

$$\langle I \rangle = \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{p(x_i)}$$

estimator

With the calculation of the mean or expected value as described above, it can be shown that the expected value of $\langle I \rangle$ is equal to the Integral I, for an infinite number of samples N:

$$
\begin{aligned}
E(\langle I\rangle) &= E\Big(\frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{p(x_i)}\Big)\\
&= \frac{1}{N}\sum_{i=1}^{N}E\Big(\frac{f(x_i)}{p(x_i)}\Big)\\
&= \frac{1}{N}N\int_a^b\frac{f(x)}{p(x)}p(x)dx\\
&= \int_a^b f(x)dx\\
&= I
\end{aligned}
$$

equation: estimator = integral I

The calculation for the variance can be used to show the analogue:

$$
\begin{aligned}
\sigma^2(\langle I\rangle) &= \sigma^2\Big(\sum_{i=1}^{N}\frac{1}{N}\frac{f(x_i)}{p(x_i)}\Big)\\
&= \sum_{i=1}^{N}\sigma^2\Big(\frac{1}{N}\frac{f(x_i)}{p(x_i)}\Big)\\
&= \frac{1}{N^2}\sum_{i=1}^{N}\sigma^2\Big(\frac{f(x_i)}{p(x_i)}\Big)\\
&= \frac{1}{N}\int_a^b\Big(\frac{f(x)}{p(x)}-I\Big)^2p(x)dx
\end{aligned}
$$

variance equation [DBB06]

This means that the variance is linearly anti-proportional to the number of samples used. The error of the Monte Carlo integration is proportional to the standard deviation of ⟨*I*⟩.

2.6.2.b. Bias

In addition to the variance, the error of the Monte Carlo integration also depends on the estimator being biased. It is unbiased if its expected value is equal to the value of the integral of *I*, as shown above. If it is not equal, the estimator has the bias value equal to the difference between the expected value of the estimator and the actual value of the integral:

$$\underbrace{B[\langle I\rangle] = E[\langle I\rangle] - I}_{bias}$$

The total error of the Monte Carlo integration can be described as the sum of the standard deviation and the bias. A consistent biased estimator is present if the bias disappears with an increasing number of samples. In some cases biased estimators can be useful, if they result in a low variance to compensate for the introduced bias [DBB06] [N11].

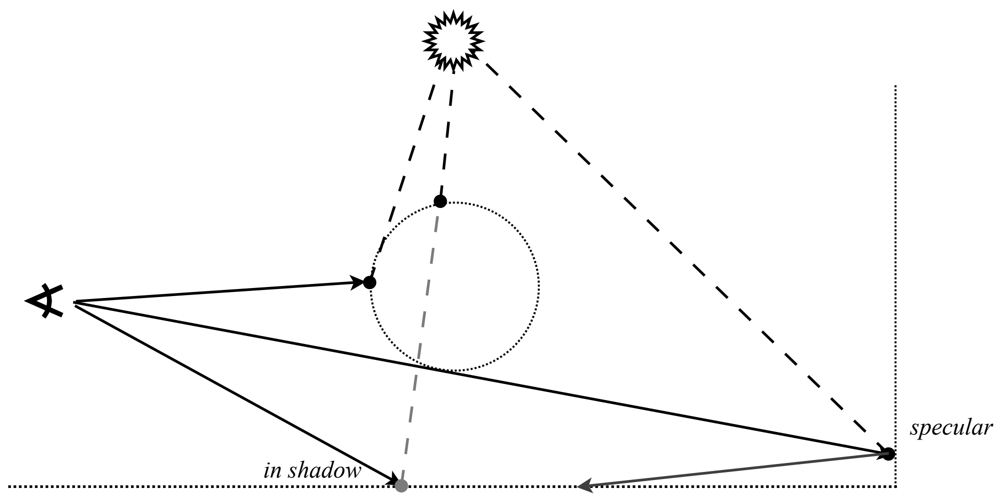
2.6.3. Classic Ray-Tracing

Classic Ray-Tracing was first introduced by Whitted in 1980 [W80], in form of a recursive ray-tracing algorithm. It can be used to render shadows and specular surfaces.

The concept of ray-tracing calculates the light reaching the observer by tracing it backwards from the viewer to the light source. The goal is to compute the incoming radiance at every intersection hit by rays being traced from every pixel. Each ray can be represented in the form

$$\underbrace{r(x, w) = x + d * w}_{ray\ description}$$

where *x* is the origin and *w* is the direction and *d* is the distance travelled along the ray. For computing the radiance we take the object that is nearest (smallest *d*). The illumination of the object at each intersection, from each light source, can then be calculated by using the surface normal at this point. Additionally, shadow rays can be used to compute shadows. For this we trace a ray in direction of the light source, and if an object intersected is closer than the light source, the light does not reach the original intersection directly.



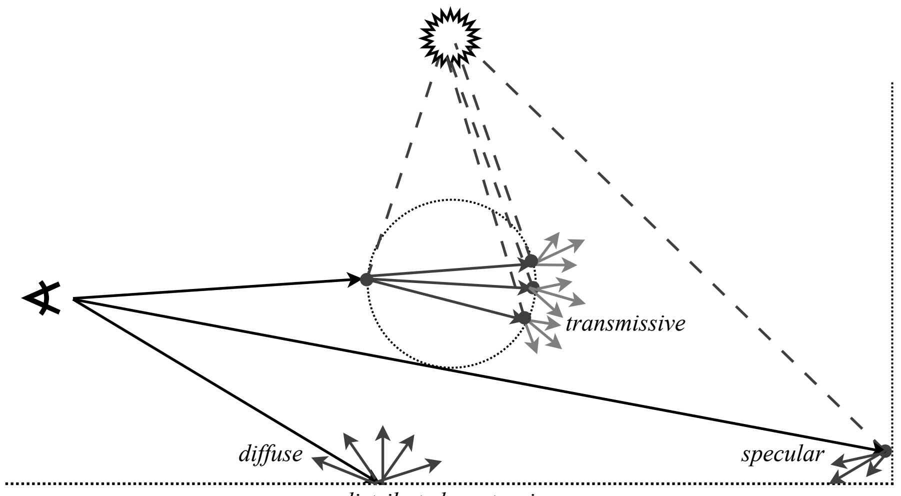
classic ray tracing

Should the material be specular, the specular reflection is determined by tracing a ray in the mirror direction. This tracing is handled in the same manner as the previous ray, hence the recursive nature of this algorithm.

This classic ray-tracing algorithm does not compute the full global illumination, since it does not handle indirect lighting of diffuse material. Also, it cannot compute other materials than perfect specular (mirror) materials. In this manner only perfect diffuse, specular or transmissive materials can be rendered, and not mixed materials. For this we need to use Monte-Carlo ray tracing which this algorithm does not, as no stochastic evaluation of the rendering equation is computed.

2.6.4. Distributed Ray-Tracing

Distributed ray-tracing is an extension of classic ray-tracing [CPC84], which uses multiple reflected rays at every intersection on diffuse specular and transparent materials to compute the radiance. The number of reflected rays is dependent on the amount of light coming from the direction of the light source.

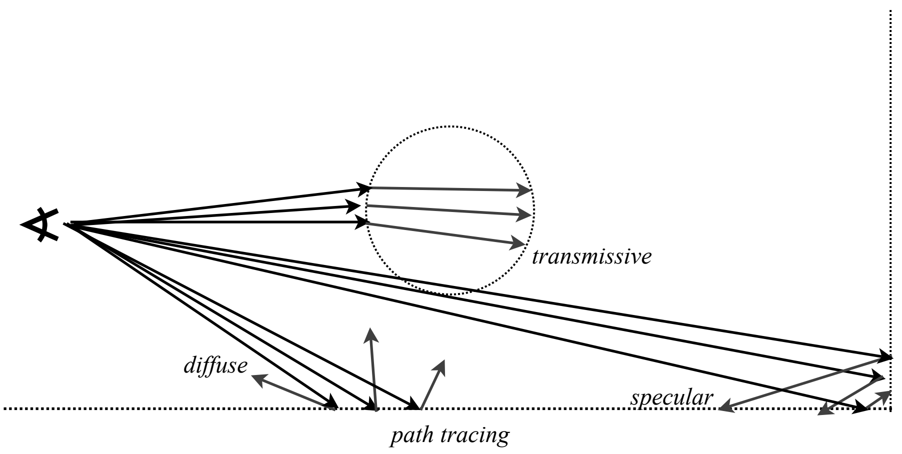


distributed ray tracing

In this approach, shadows are computed in a similar manner. Dependent on the brightness of the light sources, a number of shadow rays are shot in the direction of light. This permits, for example, soft shadows, partial obscuring of light and the proper integration of area light for shadow calculation.

2.6.5. Path Tracing

Path Tracing is another extension of the ray tracing concept, introduced in [K86]. Rather than tracing the complete tree of paths which is spawned for every pixel at the intersections of each ray, as is done in distributed ray tracing, in path tracing only a single path is followed. To be able to compute a complete result, multiple rays are shot through the scene per pixel, which enables a fairer distribution of importance per surface hit. Each ray shot from a pixel follows a different random path, determined by the BRDF of the material at every intersection.



Distributed ray tracing only samples the first intersected surface once per pixel and uses multiple rays for each following intersection, which means that the importance of the color of the first intersections is diminished. The multiple rays per pixel used in path tracing enable this approach to distribute the importance for better results.

More detail on Path Tracing can be found in [K86] and [J01].

2.6.6. Russian Roulette

Russian roulette is a Monte Carlo technique, used to improve the efficiency of particle physics computations and introduced to graphics in [AK90].

One of the main problems with the two previous ray tracing techniques is that the path tree is followed until we reach the light source. Depending on the scene complexity this can, in the worst case scenario, not be computed in a finite amount of time. Usually the tracing is aborted when a certain threshold value is reached. This threshold value is dependent on the number of reflections and the probability of each. Also an abortion of the tracing follows when the allocated storage reaches its limit.

Through the early termination of tracing a bias is introduced into the process, increased by a large number of aborted rays. To compensate for this bias and still get a similar result to rendering with an infinite amount of reflections, Russian roulette is used.

This Monte Carlo technique enables the process to be focussed on important rays and terminate the tracing for rays of lesser importance to the final rendering. The calculated radiance estimate L_n is then dependent on the probability p of another radiance estimate being calculated:

$$L_n = \begin{cases} \xi < p & \frac{L}{p} \\ 0 & 0 \end{cases}$$

randiance estimate

Here *L_n* is then computed by tracing another ray in the outgoing direction, we can see that this gives us the correct result by computing the expected value of the estimator for *L_n*:

$$E(L_n) = (1-p) * 0 + p * \frac{E\{L\}}{p} = E\{L\}$$

radiance estimator equation

This gives us the right unbiased estimate of L, which means that, using Russian roulette, we can use a sufficiently high number of bounces and still get the a close approximation of the result in comparison to using an infinite number of bounces. Russian roulette does increase the variance and thus introduces noise in the rendering, which can be diminished by using more rays per pixel, i.e. more photons. [DBB06][J01]

2.7. Photon Mapping

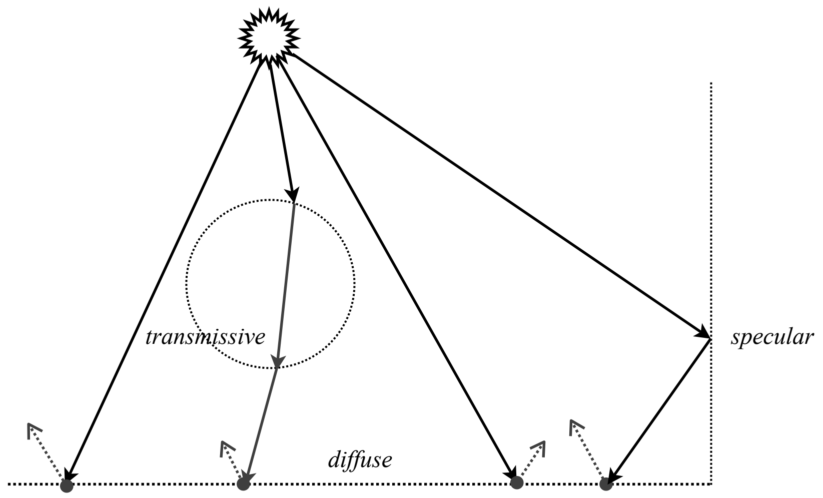
Photon Mapping is an alternative to the previously described Monte-Carlo techniques, with the goal of having the same advantages, i.e. all global illumination effects can be computed, arbitrary geometry is handled and a correct result is delivered except for variance, which is visible as noise, but without the disadvantage of high-frequency noise.

The name of the technique derives from the use of a photon map, which is a data structure storing the incoming light properties (usually direction and power) as points in a data structure decoupled from the geometry. This enables this technique to handle arbitrary geometry in complex scenes.

The photon mapping technique consists of a two-pass algorithm. The first pass is photon tracing, which creates the photon map by tracing the photons from the lights through the scene, and the second pass is the rendering pass or radiance estimate were the resulting lighting information is rendered using the photon map.

2.7.1. Photon Tracing

During the photon tracing step, photons are emitted from each light source and traced through the scenes as rays. Each photon carries a fraction of the total amount of energy of the light source, i.e. the sum of the energy of all photons equals the sum of energy of all light sources in the scene.



photon tracing

After the photon is emitted from the light source it is traced trough the scene. This works similar to normal ray tracing, except that photons propagate flux instead of gathering radiance. This means that the interaction of a photon with a material can be different, e.g. the radiance or energy of a photon is usually not changed on refraction.

When the traced photons intersect with geometry, the reflection or refraction is calculated depending on the material and the photon data saved to the photon map for diffuse surfaces and surfaces without total reflection or refraction (e.g. mirrors, glass). To calculate the reflection or refraction, the BRDF of the material of the intersected object with Russian roulette is used to decide the type of reflection depending on their probabilities. This can be described as:

$$\begin{aligned}
\xi \in [0, p_d) &\longrightarrow \text{diffuse reflection} \\
\xi \in [p_d, p_d + p_s) &\longrightarrow \text{specular reflection} \\
\xi \in [p_d + p_s, 1] &\longrightarrow \text{absorption}
\end{aligned}$$

reflection probabilities

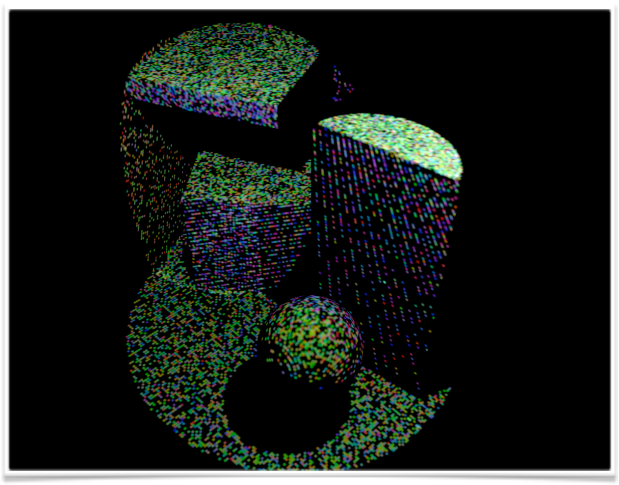
The use of Russian roulette means that if, for example, the probability of diffuse reflection is 50%, only half of the incoming photons with full energy have to be traced further, instead of all incoming photons with half their energy. The remaining photons are absorbed by the material. For calculating the radiance estimate in the second step it is preferable to have photons of equal energy levels, because then the same importance can be assumed for every photon.

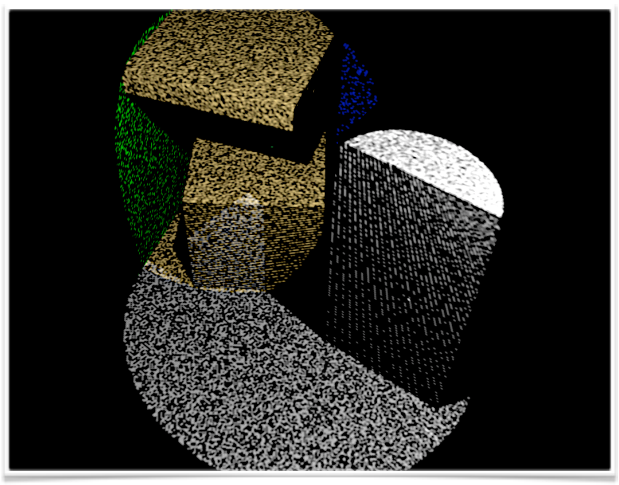
The specular and diffuse reflection directions are calculated in the same manner as ray tracing, as described in [2.4 Light Scattering].

The photon data, i.e. incoming direction and flux, is only stored in the photon map if the material reflects it diffusely or it is absorbed. This also means that the data of the same photon can be stored in the photon map multiple times along its path through the scene. For specular materials the photon is reflected, and not saved to the photon map as this usually does not give any useful information, since the probability that the stored photon data matches the particular incoming light direction needed to calculate the mirrored light for a particular viewing angle is small. In [J01], using standard ray tracing to trace a ray in the mirrored direction is proposed. If the photon hits a transmissive material it is either reflected or refracted into the material (see [2.4 Light Scattering]).

2.7.2. Photon Map (Data Structure)

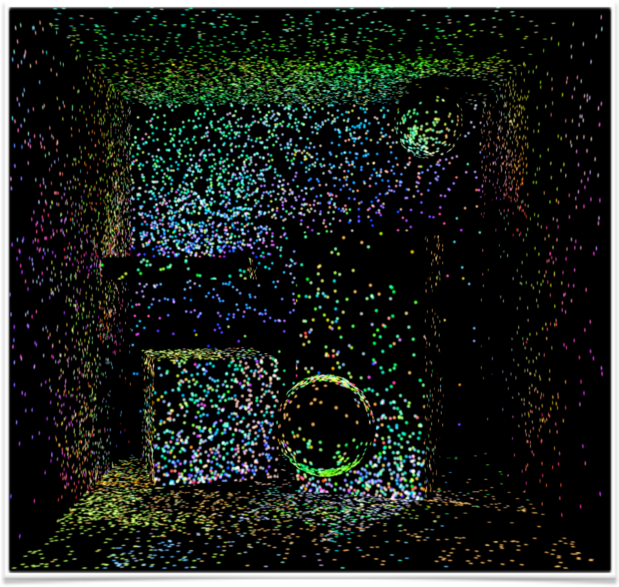
The photon map is the data structure in which the photon data is saved when a diffuse material is hit. As already noted, each photon in the photon map represents the incoming direction and flux. Thus the photon map contains the distribution of incoming flux, independent of the scene geometry. With this data the final reflected radiance is calculated in the second pass (ray tracing). In the following you can see a simple visualization of the initial bounce photon map, in which the outgoing power and the absolute values of the outgoing direction are shown as RGB color values.

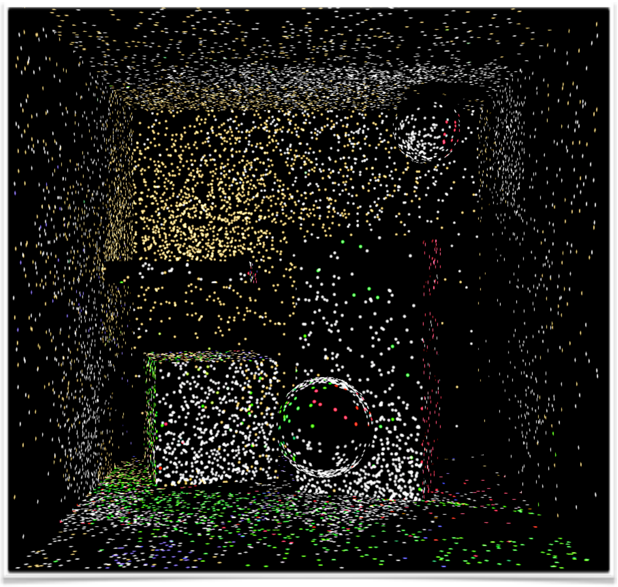

initial bounce photon map: outgoing direction as rgb


initial bounce photon map: outgoing power

(images have been brightened for better visibility)

A visualization of the photon map at a later stage, here after calculating three bounces, with 20000 photons, (without the initial bounce) is shown below.


photon map (3 bounces): outgoing direction as rgb


photon map (3 bounces): outgoing power

(images have been brightened for better visibility)

The specific data structure used for the photon map needs to allow for efficient and fast Nearest-Neighbor search for every point in the scene. It also needs to be able to save millions of non-uniformly distributed photons for each scene.

In [J01], a kd-tree is used as the data structure for the photon map. A kd-tree is a multi-dimensional (k-dimensional) binary search tree. If balanced, the kd-tree allows Nearest-Neighbor searches for an arbitrary point in *O(log N)*, where *N* is the number of photons contained in it. For further information on kd-trees, please see [J01].

The kd-tree representing the photon map is unbalanced after the creation in the photon tracing step. In an unbalanced kd-tree the Nearest-Neighbor search can take up to *O(N)* for the worst case scenario. Since the photon map is not changed after this pass, it can be advantageous to balance the kd-tree directly after the photon tracing step. In [J01], the photon map is created as an array in the photon tracing step, from which a balanced kd-tree is created before the rendering pass.

2.7.3. Ray-Tracing and Radiance Estimate

In the second step the rendering of the scene takes place. Since this step uses classic ray tracing, it is also know as the ray tracing step. Here rays are shot from the camera for every pixel of the final image and are traced through the scene until they hit a diffuse surface. To calculate the radiance at every intersection, the density of the photons in the surrounding area is estimated. Finally the color of the pixel is calculated with the estimated radiance.

$$L_i(x, w') = \frac{d^2\Phi_i(x, w')}{(n_x * w')dw'dA_i}$$

reflected radiance estimate

The calculation of the reflected radiance *L*ᵣ at point *x* in direction *w* is described in [2.4 Light Scattering]. The incoming radiance L_i is estimated from the incoming flux saved in the photon map. Using the formula for radiance, which relates the radiance to flux, we can derive:

$$L_r(x, w) = \int_{\Omega_x} f_r(x, w', w) \frac{d^2\Phi_i(x, w')}{(n_x * w')dw'_i dA_i}(n_x * w')dw'$$

$$= \int_{\Omega_x} f_r(x, w', w) \frac{d^2\Phi_i(x, w')}{dA_i}$$

radiance estimate

The incoming flux Φ_i is calculated from the photons found by Nearest-Neighbor search for point *x* in the photon map. The energy for each photon Φ_p is calculated by dividing the total energy of the light source by the number of photons emitted. With this we further derive:

$$L_r(x, w) \approx \sum_{p=1}^{n} f_r(x, w_p, w) \frac{\Delta\Phi_p(x, w_p)}{\Delta A}$$

radiance estimate

Here *A* is the surface area on which the photons are positioned. This procedure can be understood as searching for the photons in a sphere with the radius *r* around point *x*. It can get very complex to calculate the correct surface for each intersection, since the photon map is independent of the scene geometry. For this reason it is approximated that the surface is flat around *x* to be able to calculate the area as the intersection between the sphere around *x* and the surface:

$$\Delta A = \pi r^2$$

intersection area

With this, the calculation for estimating the reflected radiance for every point is:

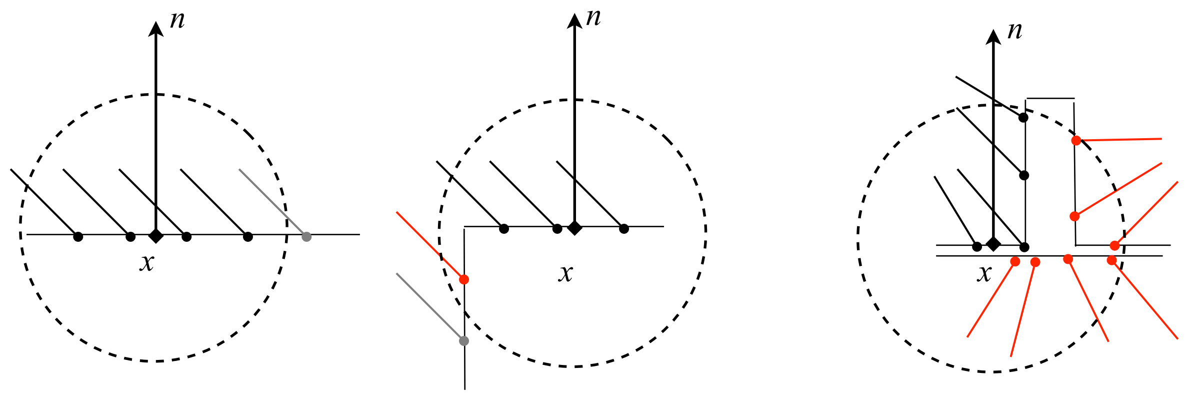
$$L_r(x, w) \approx \frac{1}{\pi r^2} \sum_{p=1}^{N} f_r(x, w_p, w) \Delta\Phi_p(x, w_p)$$

final radiance estimation

A number of artifacts can result from these approximations. The main reason for the complexity of solving these artifact problems is the independence of the photon map from the geometry in the scene. For this reason it is harder to use information from the underlying geometry to diminish the artifacts.

Common artifacts resulting from the approximations used are an overestimation of the surface at geometry edges due to the sphere radius overlapping with the geometry edge. On curved surfaces the area can in turn be underestimated. Light leaks can also appear in regions with thin geometry leading to the sphere collecting photons on the other side of an object.

These artifacts can be diminished by using more complex search geometries, like ellipsoids. This leads to a more complex calculation of the intersected area and surrounding photons. A light leak can in some cases be easily avoided by checking the direction of the incoming light against the normal of the surface at point *x*.



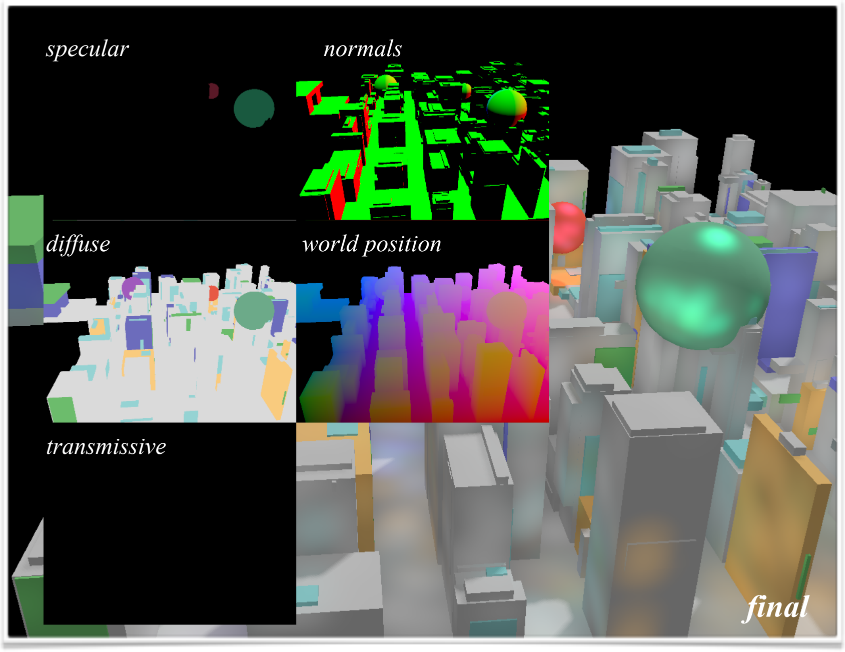
radiance estimate (artifact sources)

To eliminate artifacts due to a too large search radius of the sphere around *x*, [J01] proposes a weighting of photons according to their distance from the intersection position. This can minimize problems with blurred corners or caustics and improve the rendering of sharp corners, as well as allowing a smaller search radius.

Another approach for rendering the resulting photon map is splatting [DS06]. This has the advantage of not needing to search for the nearest photons and as a result not requiring a complex data structure, while still delivering satisfying results, depending on the photon count and splat radius. Splatting can be thought of as a backwards gathering, as the radius of the photon splat ‘collects’ pixels instead of collecting nearest photons in a radius around each pixel. Thus it is a good solution for real-time rendering of photon maps. A drawback is that the photons need to be drawn at a large radius to still light the scene in a similar manner. As a result, multiple photon splats overlap each other and cause the hardware to slow down, due to reaching a high fill-rate per pixel. For this reason, the radius of the photon splats has to be carefully adapted to still reach enough pixels without too much overdraw. Further approaches to solving these fill-rate problems are described in the implementation and evaluation chapter.

2.8. G-Buffer Creation

The G-buffer is created by a process usually known as deferred rendering or shading, which is used to separate geometry and lighting information. This process can be implemented with various optimizations and adaptions depending on the architecture used and needs of the rendering system. The G-buffer concept was first introduced in [ST90] (*deferred shading* was not yet used in that paper as a term). A system is described which renders all information of the geometry needed for later processing, such as light computation, in extra buffers. These are usually in the form of 2D render targets. The following screenshot shows an example of possible contents of a G-buffer, here shown as textures rendered directly on screen and 3D position information (XYZ) rendered as RGB values.



G-buffer of viewer camera

This enables multiple complex lighting and post-effect computations only needing one complete render pass, while all other render effects can be calculated in screen space in a pixel shader, with all necessary information available from texture resources bound to the appropriate shader stage. This also ensures that all complex computations are only done for visible geometry and not wasted on objects which are hidden. Drawbacks of this method include the more complex handling of transparent objects, which can be rendered in an extra pass after all other computation is finished, or using 'screen-door' transparency [AHH08]. Another difficulty stems from more advanced techniques which need multiple G-buffers from different camera viewpoints to eliminate view-dependent artifacts. More information on this can be found in [AHH08], [ST90].

2.9. Software and Libraries

[OPTIX], a GPU-based Ray-Tracing engine by Nvidia was used as well as [DIRECTX]. This Engine is highly optimized for ray tracing functions, fits the requirements of my approach perfectly and was chosen for photon tracing on the graphics hardware. In addition to ray-tracing operations and handling, it also provides several data structures like KD-Tree and median bounding volume hierarchy (BVH) [OPTIX][PDB10]. A wrapper for C++, as well as interoperability functions for Direct3D and OpenGL are provided by the Optix libraries. The main method of transferring data between Direct3D and OptiX is to use specially set up buffer pointers. On the GPU, OptiX and Direct3D use the same buffers but access to these has to be specially set up for each one. OptiX provides functions for automatically creating these access pointers from Direct3D buffers.

OptiX still has a few limitations: For example, in the documentation it is said that CPU access to data is always needed. Other inconsistencies and bugs still exist, like the seeming inability to properly release memory or handling already released memory on shutdown. Even with these drawbacks, OptiX remained a useful system for implementing the prototype for this thesis.

3.Previous Work

3.1. Previous Approaches

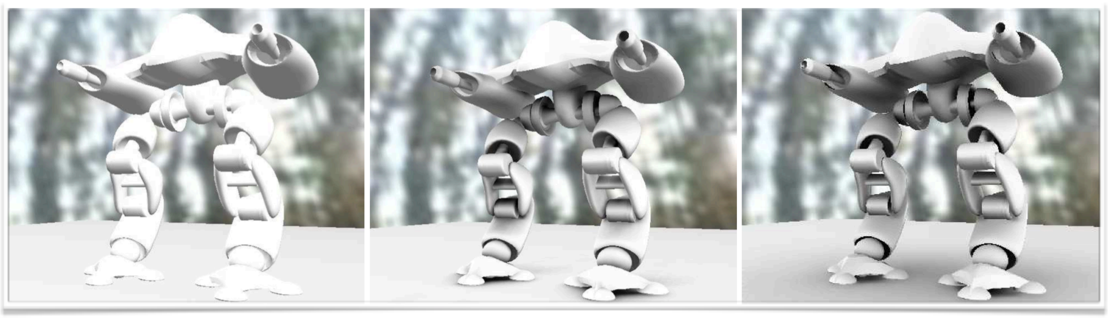
There are many previous approaches to achieving global illumination or global illumination-like effects in real-time environments such as games. The following are short descriptions of concepts representing approaches relevant to my thesis. I have selected these to give a broad enough comparison while still focussed on the main goals and problems of my particular topic.

3.1.1. Direct Light and Ambient Occlusion

Ambient Occlusion is an approach where shading is decoupled from visibility or light impact. It imitates shadows in areas where less light is present due to the underlying geometry. For example, two objects intersecting each other will be shaded at the intersection edge with ambient occlusion. It roughly approximates shading due to ambient light, which illuminates evenly from all directions and casts very soft shadows. Since there is little to no direct light in some areas, objects or geometry might appear flat with their form being harder to distinguish, when using only direct lighting without ambient occlusion. [AHH08]

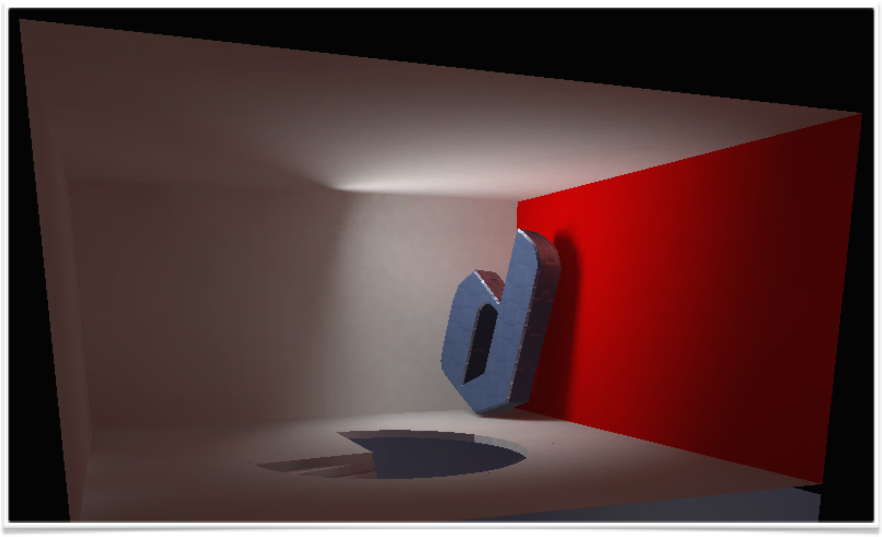
There are several methods to compute ambient occlusion. As this thesis handles real-time environments, I will only focus on real-time methods for calculating ambient occlusion. This can be done in either object or screen space. Object space implementations have the drawback of being dependent on the scene complexity and usually being limited to static geometry, whereas screen-space methods usually have to deal with more approximations, which still can be visually convincing. Ambient occlusion in general does not compute shadows cast by objects. In addition, lighting phenomenon like color bleeding or caustics are not handled by basic ambient occlusion, but can be approximated through further extension, as described in [RGS09], which includes an implementation to handle color bleeding using multiple views, computed in screen space.

Most ambient occlusion techniques use rough approximations, which can introduce additional artifacts like halos or improperly shaded areas. [DBB06][AHH08]


left: gouraud shading, middle: with Ambient Occlusion (blended), right: AO by distributed ray tracing [MMA07]

3.1.2. Precomputed Light Maps

As noted before, some lighting systems rely heavily on precomputing the relevant data and saving it in light maps. These textures allow for a very high quality shading solution for static light and geometry. The lighting data is rendered and saved in light maps which are later used to render the scene. Depending on the complexity of these systems, these light maps can already contain the complete shading. Most systems save the lighting and other data like diffuse color, specular and normal maps separately for greater flexibility (comparable to deferred shading/lighting). A prominent example of this approach is [BEAST].

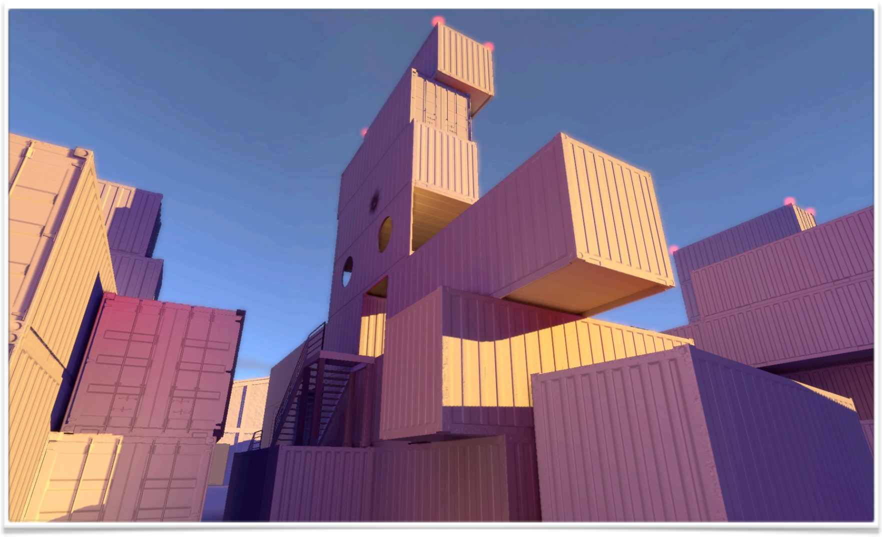

precomputed lighting [BEAST]

Since the lighting information is precomputed, dynamic objects will either be lit only by direct light or with the help of light probes. Light probes are distributed uniformly throughout the scene and hold the information of the light an object would receive at this position from multiple directions, saved to a environment map, which is then projected on the basis of spherical harmonics. The lighting for dynamic objects is then calculated by interpolating between the light probes and, in most cases, rendering simple direct light.

light probes [BEAST]

3.1.3. Enlighten Engine

The current version of the Enlighten engine by Geomerics [ME10] uses a similar approach, but updates and synchronizes the lighting data at regular intervals. The lighting data, which includes only indirect effects, is calculated and then updated about every five frames. The idea behind this is that the minimal change in indirect lighting effects does not have a great impact on the final rendered image and as such is not easily noticeable by the observer. As with precomputed light maps, dynamic objects are rendered with the help of light probes, the asynchronous updates are only used for dynamic light sources. Thus they do not interact with the indirect lighting effects, as this would make the lack of change of indirect light effects visible.



global illumination using Enlighten engine [ME10]

Additionally, the approach used in the Enlighten engine uses simplified geometry to calculate the lighting. The final lighting information is applied to the more complex geometry using the same UV coordinates for the light maps.

3.1.4. Cascaded Light Propagation Volumes

This real-time approach for rendering global illumination was introduced in [KD10] and uses lattices and spherical harmonics to compute the low-frequency portion of the lighting calculation. This means that only diffuse indirect light is computed with this method. Furthermore, the approach described in [KD10] includes only the first bounce of light through the scene. A proposal for including multiple bounces is included at the end of the paper.

The approach works by calculating the light propagation through the scene by dividing it into cells and computing it for each. This is then stored using two grids to represent the lighting information of the scene. The first grid contains the intensity of the indirect light reflected by the objects' surfaces in the scene and the second grid stores a volumetric approximation of the scene geometry. Both of these store their data by using spherical functions.

light propagation volumes [KD10]

The scene is then divided into multiple light propagation volumes, in each the lighting is computed and the flux propagated to the neighboring cells along the faces of the volume. Each cell, as a cube, has six neighbors to which to the calculated flux is propagated.

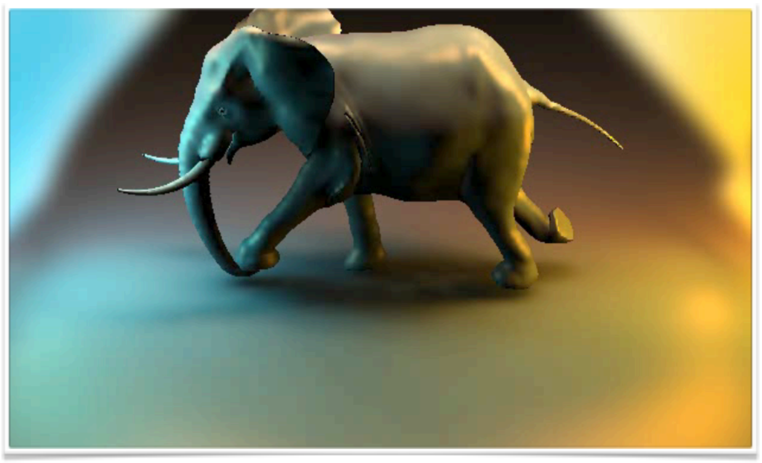
The flux propagation information stored in the two grids is then used to compute the indirect lighting for the final rendering. Due to its nature, the lighting information is saved at a low resolution and depends on the number of cells used. A cascaded approach raising the number of cells used near the viewer and thus improving the resolution of the grid helps with this problem. Still, the indirect lighting can still appear blurred, which is usually not a noticeable problem for low-frequency light information.

Drawbacks of this approach include being unable to properly render objects smaller than the grid's cell size, esp. dynamic objects, causing incoherent indirect lighting across frames. Because it can only deal with low-frequency lighting, this approach also fails to properly handle lighting effects caused by specular and transmissive surfaces such as caustics, because only diffuse materials are handled while calculating light propagation. Artifacts such as light bleeding through walls can also appear due to low resolution of the grid. Further details can be found in [KD10].

3.1.5. Imperfect Shadow Maps

Imperfect Shadow Maps is an approach for computing indirect illumination for large dynamic scenes introduced in [RGK08]. As the indirect lighting part of global illumination usually masks smaller errors due to its inherently smooth gradients, a rough point-based representation of the scene can be used to render satisfying results while using virtual point lights (VPL). The imperfect shadow maps are used to achieve a valid approximation of visibility to improve the performance of global illumination simulation.

In detail imperfect shadow maps are used to further optimize the speed of calculating visibility tests between pairs of 3D points, this is achieved by computing very rough low-resolution shadow maps for the VPLs to drastically improve performance.


imperfect shadow maps [RGK08]

Even though this approach is not directly dependent on the geometry in the scene, large and complex scenes can require an increased amount of point samples. Also, detailed indirect shadows from small geometry cannot be retained with the low-resolution shadow maps. The number of VPLs also needs to be increased to prevent flickering and inter-frame coherence problems. Light leaking due to shadow biasing can appear as well, if the number of VPLs and point samples is not high enough or the shadow bias parameter is not adjusted correctly. Further details on this can be found in [RGK08].

3.1.6. Hardware-Accelerated Global Illumination by Image Space Photon Mapping

The idea of the approach used as the basis for this thesis changes the classic photon mapping approach described above by rendering the first photon bounce in image space and using photon splatting for rendering the final result [ML09]. The photon map is only used for indirect light, direct light is rendered in a more traditional way.

The first photon bounce is rendered in the pixel shader stage and saved into a bounce map. In this map every pixel represents a photon. Each photon contains the outgoing direction and power already calculated in the pixel shader using Russian roulette and the appropriate diffuse, specular or transmissive values for each material. With this data available, the rays for the tracing of the photon paths can be created directly without further computation.

Image Space Photon Mapping

In the paper and provided source code, spotlights are used in every rendering. Spotlights are represented as a normal camera, whose frustum represents the light cone of the spotlight. An additional falloff handles the spotlight falloff inside the shader. With these simple changes the first bounce can be rendered in the pixel shader in one pass, with the bounce map representing the photons.

These bounce maps are then transferred to the CPU where a photon buffer for later photon tracing on the CPU using a multicore approach is generated. While the CPU does the photon tracing, the GPU renders the direct lighting and direct light effects, like direct light bent by transmissive materials or reflected off specular surfaces. In the final step, the photon buffer is again transferred to the GPU and, using instancing, every photon is rendered as 3D photon volume (*splat*), in the shape of a sphere reduced in size along the surface normal to create an ellipsoid. For every photon volume the indirect illumination is rendered with the appropriate falloff and added onto the existing direct light rendering. The radius of the spheres is influenced by the path density.

The path density is a rough estimate of how many photons are lying in a similar path. For diffuse reflection, photons are scattered very broadly and have a low path density, while for caustics, photons have a very high path density as the light is focused on one region. This path density is a approximation which is made esp. roughly to “enforce” its impact on the calculation. [ML09]

Judging by the results provided in [ML09], most of the frame-time is consumed when transferring data between CPU and GPU and when splatting the photons. Tracing photons on the CPU also takes a considerable amount of time, as it is only tested on a system with eight cores. In this respect the GPU is a much more suitable candidate for the photon tracing as it is optimized for highly parallel tasks. Since the photons do not interact with each other, the photon tracing step is perfect for a GPGPU implementation. The idea for this has also been named in the paper and further described for implementing in [PBD10].

My approach is based on image space based photon mapping as this includes even complex indirect effects, can be transferred to the GPU to achieve higher frame rates and is highly optimizable. It can also be adapted to be able to handle directional lighting, e.g. for sunlight, as will be shown in the following chapters.

3.1.7. Other Approaches

It should be noted that most global illumination systems used in interactive environments such as games, which are required to run at interactive frame-rates, focus on color bleeding and transfer and rarely implement other light effects like caustics or subsurface scattering. If the latter are implemented at all, they are mostly achieved through different means, like fake caustics with the use of textures for water reflections.

Slower global illumination methods are available, as, for example, described in [2. Prerequisites and Theory], but they only reach speeds between multiple hours and a few hundred milliseconds for more optimized implementations. Of course, these speeds are not acceptable for real-time applications such as games. On the other hand, most of these techniques achieve a much higher image quality and handle more advanced effects like caustics and transmissive materials (glass, volume scattering).

4.Strategy

The following is a description of how I planned to adapt and extend the approach described in [ML09]. I will include an overview of the light computation and rendering process including my planned adaptions where applicable.

4.1. Strategy

The main goal of this thesis was to create a photon mapping system which could be used for indoor and outdoor lighting seamlessly. The basic idea was to adapt the approach in [ML09] for spotlight rendering to be used for directional lights. It describes an approach which makes real-time photon mapping possible for spotlights (and point lights approximated as six spotlights). The approach still left room for improvement as it was not optimized and 'only' ran on the CPU. The implementation described in this thesis aims to solve some of the problems and to optimize the approach further by moving the photon mapping completely to the GPU. Further changes and optimizations are described in the implementation section.

The first prototype was very close to [ML09] in terms of the implementation, with the only major difference being the API used, which was Direct3D instead of OpenGL with the G3D Engine [G3D]. As not all of the necessary code was provided, some of the functionality was implemented differently.

4.1.1. G-buffer and Light Source Representation

The first system to implement was the creation of the G-buffer, making it just a simple straight-forward solution, creating a G-buffer with all the information I needed for the further calculations. This just required a simple shader and some structuring of cameras for the viewer and the lights.

The plan was to implement a simple deferred shading system for creating the needed G-buffers. As described in the paper [ML09], every light-source is treated as a camera for calculating the first photon bounce in screen space and saved to a bounce map. For this to work, each light creates a G-buffer to be used for creating the bounce map. The bounce map saves a photon per pixel, which is described via the outgoing direction and outgoing power. These are calculated using a Russian roulette based system, saving the right amount of photons according to diffuse, specular and transmissive material properties.

The spotlights have a simple projective camera attached to them, where the opening angle of the spotlight corresponds to the field-of-view of the camera, and the falloff distance corresponds to the far plane. The camera frustum needed to be created so that it encloses the spotlight cone completely.

A point light is just a set of six cameras, treated in a similar way to the spotlight approach, with wider field-of-view to render the whole scene visible in all directions from the point light. This follows the description of spotlights and point lights in [ML09].

To render directional light, i.e. parallel light rays, a projective camera could not be used. But the camera model itself still could be used with an orthographic projection. The only matter needing to be solved was what part of the scene the frustum of this camera encloses. Depending on the scene the frustum could either include the whole scene, which would result in a very low resolution per region, or the camera could be moved so that it always rendered at least the visible part of the scene. The last solution could degrade visual quality due to inter-frame coherence problems. Another possible solution is to render multiple separate regions of the scene and find a solution to handle the transition between them, similar to cascaded approaches in shadow mapping [F05].

For the prototype implementation I chose to render the complete scene and planned to use methods from different shadow mapping techniques to improve the visual quality.

4.1.2. Initial Bounce Calculation

I decided to change the initial bounce calculation to be written directly into to a stream output buffer, using a geometry shader. This made it possible to keep the data on the GPU for the complete process, without needing to create a new buffer from the bounce map render targets for use in OptiX. This not only made the use and integration easier, but also decreased the time needed, as less data has to be transferred from GPU to CPU memory.

For rendering the initial bounce buffer I created a regular vertex grid which is always aligned to the camera. This vertex grid is generated procedurally on the CPU at the initialization of the system, with the same number of vertices as maximum photons allowed in the scene. This has the advantage of being able to use the vertices as direct input for the geometry shader writing to the initial photon bounce buffer. The geometry shader then uses this vertex grid as input and writes a new photon to a photon buffer, instead of a photon map, as stream out.

The first bounce is only used to create the first rays to be traced in OptiX, it is not used for the final image as the direct light rendering can be achieved better and faster through more traditional means, using the G-buffer. The initial bounce buffer is also inappropriate for final rendering purposes as it is not in the same format as the photon result buffer and does not carry the required information for rendering the final image.

4.1.3. OptiX Integration and Tracing

As OptiX communicates with Direct3D through the same buffers I had already written to, integration into the planned system was easily possible. OptiX itself would then write to a result buffer, still residing on the GPU, after OptiX is finished and ready for the final splatting part of the system.

After calculating the first bounce, the data is used in OptiX to create the first rays for the photon tracing. Whenever an intersection occurs, a hit program for every material gets executed, which decides, using Russian roulette and the material properties, if and how the photons are killed or reflected or refracted. This happens in accordance to the described light scattering theory. If the photon would leave light information behind (on diffuse and specular materials) it saves its information into the photon result buffer. This includes the incoming photon direction and power. Every photon is killed either because of the material properties or after the set maximum number of bounces, whichever comes first.

4.1.4. Splatting

After the tracing is complete, the photon result buffer is splatted directly onto the direct light, which is rendered using the G-buffer of the viewer camera. In [ML09] the splat radii are estimated by using the path density value. I planned to find a better estimation based on the traveling distance of each photon. The less the photon travels after the initial bounce, the more likely is a dense collection of other photon splats nearby.

5.Implementation

5.1. Tools, APIs, etc.

To implement my solution I used C++ with the current DirectX SDK (June 2010) [DIRECTX] with the later addition of OptiX [OPTIX]. I programmed it in Visual Studio 2010 [VS2010], as this environment provides many features I have come to rely upon. Other environments do deliver similar features in terms of debugging and syntax highlighting, but are not as integrated with DirectX and Windows development. There are also two important plugins available for Visual Studio, namely NShader [NSHADER] for shader code syntax highlighting and VisualAssistX [VAX] which provides improved syntax highlighting and other coding functionality for C++.

For debugging purposes, as well as the Visual Studio debugger, I used PIX for debugging shader code. This program is provided with the DirectX SDK and lets the user look at every draw call and resource on the GPU as well as setup steps for every frame. If the HLSL code is compiled with the debug flag, it lets the user step through the source code and look at the change of values of the variables, much like the breakpoint and step-functionality in modern debuggers of CPU code.

The drawback of using the Direct3D API was that the source code provided alongside [ML09] needed to be reimplemented, but since it did not contain any eccentric functions it was no real hindrance. Most of the code was heavily adapted and changed to fit my needs later on, so the reimplementation made a reorganization possible, which improved later development.

Other than that, Direct3D and OpenGL are similar enough to not have any serious drawbacks in either case. For my implementation I used a few Direct3D 10/11 specific functions for further improvement.

Another drawback was that the current version of PIX seems to have problems with debugging geometry shaders with stream output. But since these special kinds of shaders were only used for creating the initial photon bounce buffer, I just used more traditional ways of debugging GPU code , like rendering resulting buffer information as color output to a render target. PIX was used for debugging all other GPU code.

OptiX provides a complete ray tracing solution complete with handling of different geometry groups, their transforms and different acceleration structures for different purposes. For my implementation I used a median BVH, as it created rather fast while still providing good performance.

The OptiX engine has a few drawbacks in error handling and integration. For error handling, the information given for any problem can be very generic, which has the the programmer revert to debugging methods as used when no debugger is available. It also is not consistent with the information provided in the programming guide by Nvidia. For example: OptiX usually requires a staging buffer with CPU Access for the interaction with the graphic API. For my implementation, however, I am using a normal stream output and vertex buffer, without needing to set any special flags, which has caused no problems. OptiX also only runs on graphics cards which support CUDA.

Even though OptiX still has a few drawbacks, it provides a fast way for easily creating systems that rely on ray-tracing and is still the best choice for this prototype.

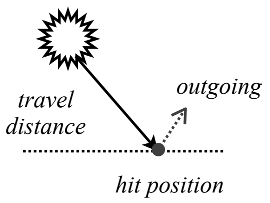
5.2. Data Structures and Concepts

This section describes the different data structures and concepts used in the implementation. The data structures where mostly used according to features needed, and adapted for performance purposes.

5.2.1. Photon Data

The photons are represented in different formats for the different steps of the global illumination calculation process. They are changed according to the data needed in each step.

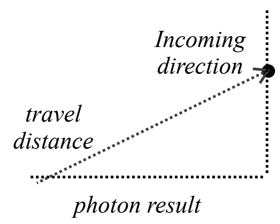
5.2.1.a. Initial Photon Bounce Data



inital bounce

For the initial bounce each photon contains the hit position, the outgoing direction, describing the direction in which the photon is reflected or refracted, the outgoing power, describing the changed power based upon the material the photon hit. In addition it also contains the travel distance, which is set to zero for the initial bounce, as the distribution of photons is still rather uniform and these photons are not used for the final splatting, meaning their distribution does not directly impact the final image in terms of splat radius.

5.2.1.b. Photon Result Data



The photon result describes the flux saved for the photons bouncing off diffuse or specular surfaces. (No photon information is saved for transmissive materials.) This result contains the photon hit position, the incoming photon direction, the incoming photon power and the travel distance, which is increased by the length of the ray. The incoming direction and power is saved instead of the outgoing power, since this allows greater flexibility in rendering the light effects in the rendering pass.

On each hit it is determined using Russian roulette if the photon is killed, or reflected diffusely, specularly or refracted into a transmissive object. The probability for each of them is determined using the diffuse, specular and transmissive values saved in the G-buffer.

5.2.2. OptiX Ray Data

The ray in OptiX is represented through a data structure which contains all the information needed for the ray casting, as well as the photon information which is carried along for further computation. This information is changed at every hit to represent the changes to the photon due to reflection or refraction. For reflections the power is updated accordingly, for every interaction the new direction and origin is stored.

5.2.3. Direct3D11 Buffers

For saving and reusing information on the GPU and in OptiX, Direct3D11 buffers were used. These buffers can be custom sized and made to fit custom data structs. With special binding flags, they can be used as vertex input buffers for the different shaders or for creating photon rays in OptiX. Buffers can also be set to receive data from the stream output of a geometry shader or from the tracing done in OptiX where each hit result is stored in an output buffer.

5.2.4. Render Targets

The implementation described in [ML09] uses render targets to save the initial bounce information and creates a photon array from these to compute the photon tracing on the CPU. From this a vertex buffer is formed to use for the splatting. This has the drawback of needing additional time to create an input buffer for the following calculations. Even though render targets could have been used as input buffers for most of the GPU computations, it is far easier and more optimized to use vertex buffers and does not require special unpacking or adaption of the code. As a result, render targets were only used for creating the G-buffer for the viewer and light cameras.

5.2.5. Stream Output

The stream output stage is a concept introduced in shader model 4.0. It allows data to be written directly into a stream output buffer, which can have a similar format to the vertex input buffer. When using the stream output, the rest of the render pipeline can be skipped after the geometry shader, i.e. the stream output stage, allowing for relatively fast computations, e.g. in particle systems running exclusively on the GPU. This enables improved harnessing of the parallel nature of GPU computations. Additionally, the geometry shader can add and omit complete primitives or single vertices if needed. This means that the output buffer needs to be appropriately sized to be able to hold the maximum number of output primitives.

The number of primitives written to the output buffer is saved and can be queried after the GPU is finished with the stream output stage. If this value is queried by the CPU the whole system usually waits for the GPU to respond, though. This could be solved by using an extra thread on the CPU side and giving the GPU additional instructions and draw calls, before querying, to use the time spent on waiting. For my implementation I did not query this value, instead I cleared the output buffer every frame and implemented an additional check for unset photon data. In a more 'traditional' implementation, i.e. when using only DirectX, one could also use the 'DrawAuto' function provided in the API. This function automatically processes written to the stream output buffer. Since I used OptiX, based on CUDA, as a ray tracing engine, I could not use this function for this purpose.

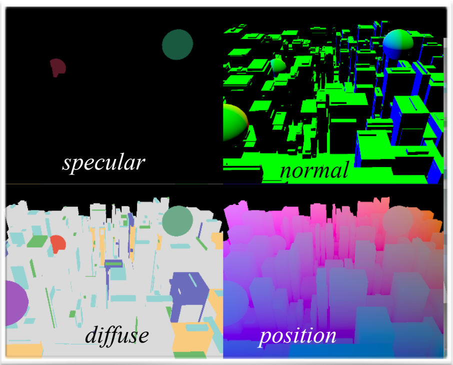
To render the photon hits saved into the photon result buffer by OptiX, splatting was used instead of gathering. This means that the photon hits are rendered and added onto the direct light computation. The photon splats are usually based on a sphere to impact the surrounding area with a falloff. The radius of the splats is chosen to include all necessary pixels, larger if there are fewer splats and smaller in a region with high photon density. The splat itself is not drawn, but rather the photon information combined with the underlying surface is rendered. Further details in [2. Prerequisites and Theory].

5.3. Implementation Overview

The following is an overview of the complete system that was implemented, including the interaction between different parts. The complete process goes through the following steps:

1. G-buffer creation
2. First bounce
3. Direct light
4. OptiX tracing
5. Splatting and tone mapping

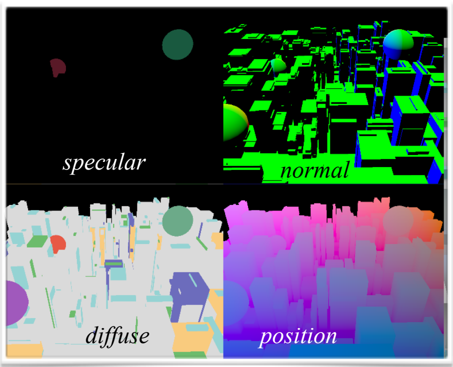
5.3.1. G-Buffer Creation



***Input**: Geometry and material data*
***Output**: G-buffer as render targets*

At first, the G-buffer is created for the viewer camera and then for each camera attached to a light source. The G-buffer contains maps for the diffuse color, specular color, transmissive color, world position and normals, An example of this can be seen to the right. The G-buffer of each light camera is used to calculate the initial bounce for each light source. Additionally, the G-buffer of the viewer camera and the position buffer of the lights are used to calculate direct lighting with shadow mapping.

5.3.2. First Bounce



***Input**: G-buffer of light camera as textures*

The initial photon bounce, which is calculated in the geometry shader, relies on Russian roulette and the power values of each scattering option to decide the death or scattering of every photon. It uses only the G-buffer and vertices from the created vertex grid as input. Each vertex in the vertex grid already contains the appropriate texture coordinate for accessing the G-buffer, an example of which can be seen to the left. The power values of diffuse, specular and transmissive are calculated as follows:

The diffuse value is taken directly from the diffuse color map and used as the reflective power value. The specular power is computed by calculating Fresnel reflection with the color from the specular map and the specular exponent which is saved in the alpha value. The transmissive value is taken directly from the transmissive map, where the alpha value represents the refractive index of the material.

These values are then used in a Russian roulette to determine in which way the photon is scattered (i.e. diffuse or specular) or if it is eliminated. They are checked sequentially in order of diffuse, specular and transmissive. The outgoing direction for diffuse materials is calculated based on the equations described in [2.4 Light Scattering], using the following algorithm:

```
float3 CosHemi(float3 normal, float randomE1, float randomE2)
{
    const float sin_theta = sqrt(1.0f - randomE1);
    const float cos_theta = sqrt(randomE1);
    const float phi = ( PI * 2 ) * randomE2;

    float x = cos(phi) * sin_theta;
    float y = sin(phi) * sin_theta;
    float z = cos_theta;

    float3 tangent = float3(1,0,0);
    float3 biTangent = float3(0,1,0);
    GetTangents(normal, tangent, biTangent);

    return (x * tangent + y * biTangent + z * normal);
};
```

diffuse light scattering

For calculating the reflection of specular materials, a slightly adapted method is used to form a more focussed scattering. The specular color is combined with the photon power and the material color. For calculating the specular reflection, the algorithm only needs to be slightly adapted to produce the resulting phong lobe typical for specular reflection. The variable *glossIndex* represents the material roughness and instead of the normal the outgoing direction (*reflected*) of a total reflection is used, as can be seen below:

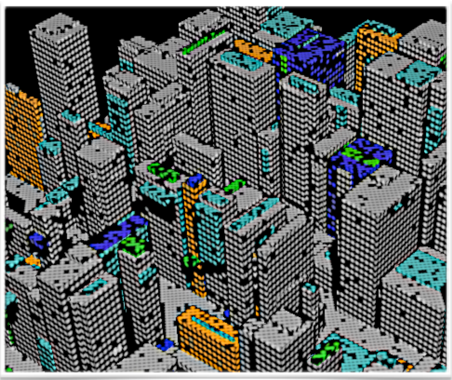
```
float3 CosPowHemi(float3 reflected, float glossIndex, float randomE1, float randomE2)
{
    const float cos_theta = pow(abs(randomE1), 1.0f / (glossIndex + 1.0f));
    const float sin_theta = sqrt(1.0f - cos_theta * cos_theta);
    const float phi = ( PI * 2 ) * randomE2;

    float x = cos(phi) * sin_theta;
    float y = sin(phi) * sin_theta;
    float z = cos_theta;

    float3 tangent = float3(1,0,0);
    float3 biTangent = float3(0,1,0);
    GetTangents(reflected, tangent, biTangent);

    return (x * tangent + y * biTangent + z * normal);
};
```

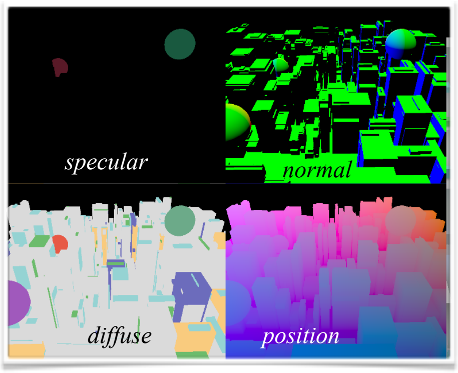
specular reflection


***Output**: Initial bounce buffer*

For transmissive values, the outgoing direction is changed according to the refractive index, i.e. the photon is refracted into the object. The outgoing power is calculated using the transmissive value, to determine how much of the light or photon power for each color are let through.

The photon hit position, outgoing direction, outgoing power as well as traveling distance of the photon (distance from the photon hit position to the light source) are saved in the initial photon bounce buffer, a sample rendering of the outgoing power of each photon saved at its position can be seen on the left.

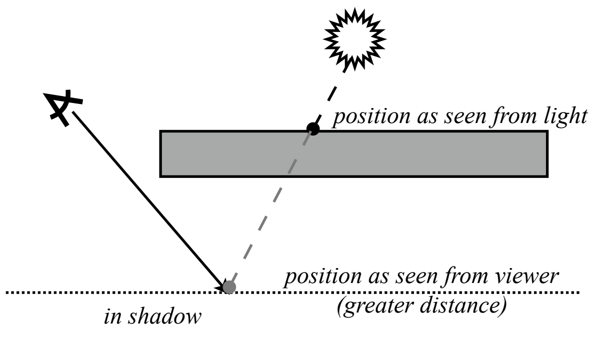
5.3.3. Direct Light



Input: *G-buffer as textures*

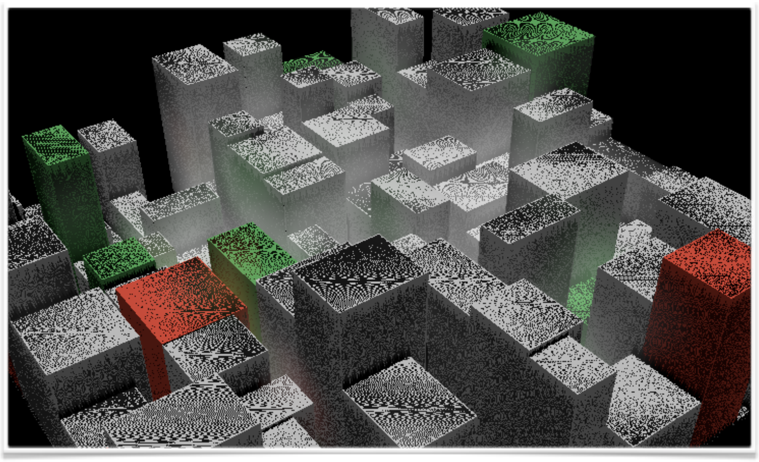
Before OptiX is sent the initial photon bounce buffer for emitting photon rays and tracing them through the scene, the GPU is tasked with calculating the direct lighting, to use the time which is spent by the CPU to set up OptiX for the tracing step.

The direct lighting is calculated in a very simple manner with the addition of computing shadows, using the G-buffer information (as seen on the left) for simple shadow mapping. The direct light is calculated by combining the diffuse and specular material properties with the illumination values in the form of the regular rendering equation, in this case using simple phong lighting.

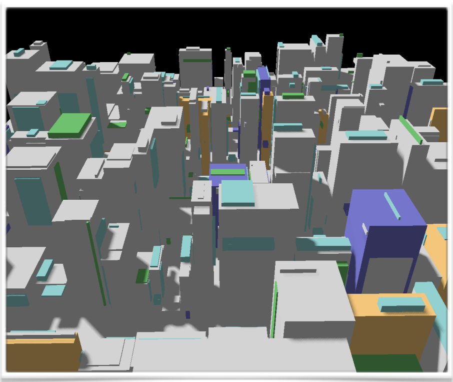


shadow mapping

The shadows, on the other hand, are calculated by using the G-buffer, esp. the position and distance information. Basically, the position stored in the position map of the viewer camera G-buffer for the texture coordinates of the current pixel is compared to the position saved in the G-buffer of the light camera (or the shadow map). If the position stored in the viewer camera G-buffer is further from the light than the position saved to the light camera G-buffer, the pixel is not reached by direct light. To handle artifacts due to the render target's inaccuracy, an additional bias is added onto the distance from the light. If no bias is used this can result in the following artifacts:



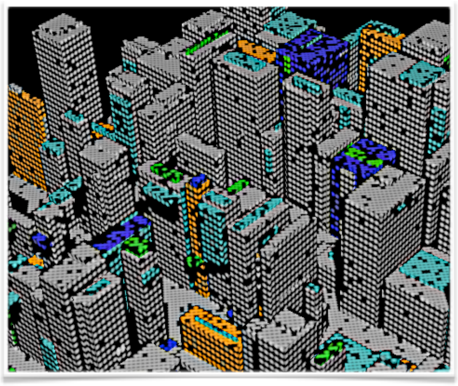
shadow artifacts without correct bias


Output: *Direct light as render target*

More on the topic of shadow mapping and its artifacts can be found in [G10], [EAS09] and [AHH08].

These calculations are then combined and saved in a render target (as shown on the left), to be used by the splatting shader and the tone mapping at a later stage.

5.3.4. OptiX Setup

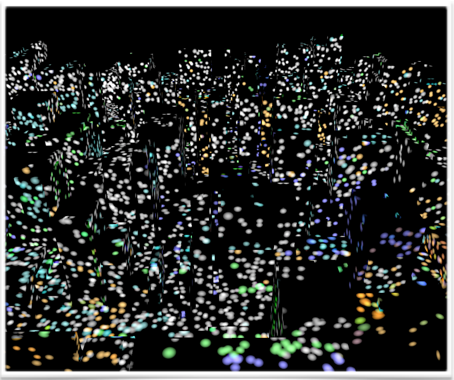

Input: *Initial bounce buffer*

After the photon bounce buffer has been filled (an example rendering of outgoing power saved in the buffer is shown on the left), it is used as the input buffer for the photon emitter program, which creates a ray for each saved photon, with the origin and direction given by the photon position and outgoing direction. These rays are then further used for the tracing steps in the OptiX raytracing engine.

5.3.5. OptiX Tracing

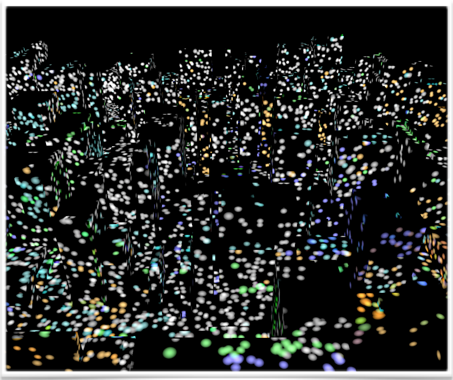
From this point onwards, OptiX handles all the ray tracing and interaction and executes the ray hit program for the material it hits. These hit programs have to be written for each custom material setup when creating the scene. In its current state I have written a program for pure diffuse materials, mainly for debugging purposes and an '*omni material*' which includes all calculations for any type of material (diffuse, specular or transmissive). Deciding if the photon stays alive and in which way it is scattered is calculated analogously to the bounce buffer shader. Again I use Russian roulette to decide the photon's 'fate' and calculate outgoing direction and power in the same manner as described previously.

The main difference to the initial bounce calculation is that the incoming power and direction for the splatting calculation is saved to the result buffer instead of the outgoing power and direction as in the bounce buffer shader. A sample rendering of the incoming power of every photon saved in the buffer as small photon splats can be seen on the right.


Output*: Photon result buffer*

After every ray is traversed and every photon is killed, either after the maximum number of bounces or when absorbed by the material, i.e. not reflected or refracted, the tracing of the photons is finished and the system moves on to the next step.

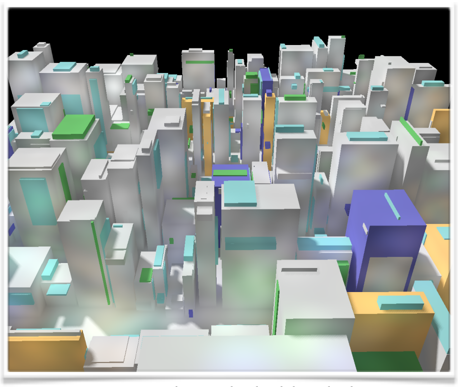
5.3.6. Splatting and Final Rendering


Input*: Photon result buffer*

The splatting shader uses the photon result buffer from OptiX as input vertex buffer. For every photon the geometry shader creates a cube, with each side being translated by the radius of the photon sphere in their particular direction, along the normal or tangents of the surface. Each cube represents a single sphere-shaped photon splat with the radius being determined by the travel distance. The further the photon has travelled, the bigger the radius of its splat. The minimum and maximum distance and splat radius are set separately for each scene. In practice, the shader only creates the three visible sides of the cube and not all six, for optimization purposes. The vertices of the created cube are then moved to screen space so that the pixel shader can render the remaining photon splat.

The pixel shader then discards all pixels outside the sphere radius and all pixels where the G-buffer contains a world position where the distance to the photon position (the sphere's center) is greater than the sphere radius.

It then calculates the color by combining the values from the G-buffer to get the appropriate diffuse and specular values to multiply them with the photon's incoming power, according to the equations described in [2. Prerequisites and Theory] to render them into the scene. Additive blending is then used to combine it with the previously rendered direct light.


Output: *Indirect light blended onto direct light additively*

As a final step, the final rendering is run through a simple tone mapper, which is further detailed in[5.4 Additional Implementations] to prevent the image from being too bright and to produce a pleasing result.

5.4. Additional Implementations

The following are descriptions of implementations that were used, were not essential to the thesis prototype, but were nonetheless important for getting the results I achieved.

5.4.1. Vertex Grid

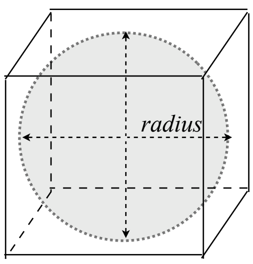
Since the geometry shader, which outputs the photon initial bounce buffer, does not work in screen space and only handles vertices, I had to find an alternative approach which would still work in a similar manner to the approach used in [ML09]. My approach was to create a regular vertex grid always aligned to the light camera's screen space currently rendering the bounce buffer. Each vertex in the grid represents a photon being shot into the scene. Since all the rendering information is available in the G-buffer and the vertex position is already in screen space, the information from the G-Buffer can be directly used.

With this information available, the remaining calculation of the photon hit and reflection or refraction is computed in much the same way as it is in the pixel shader stage in [ML09].

5.4.2. Traveling Distance

Every photon carries its travel distance along with it, which is saved to the result buffer at every photon ray intersection to use later for determining the size of the photon splats. It is updated with the distance the ray travelled with every photon hit. The probability of photons being dense diminishes the further they travel. This works well enough, but needs to be combined with the path density to be used for caustics, as these will focus the photons and bundle them in a smaller region, even if they have travelled a long distance. In addition, to calculate correct caustics many photons should be shot in the direction of the transmissive geometry.

The splatting process usually requires sphere geometry to be instanced across the scene for every photon. In my implementation I used the geometry shader to create geometry for every photon on the fly. For my implementation, cubes were sufficient, as I calculated a sphere falloff in the pixel shader. But the algorithm could be extended to use spheres or ellipsoids. Ellipsoids could be approximated either with squashed spheres or using the appropriate formula for finding a point inside an ellipsoid in the pixel shader. The created cubes have the photon hit position as their center and fully encompass the virtual sphere surrounding the photon.



photon splat cube

5.4.3. Scene Management

Every scene is built by the scene builder class and can have very different combinations of of light sources, light positions and geometry. This requires a management structure which makes it easy to set up different values for different scenes. Aside from the geometry and light setup, this includes the photon count, the minimum and maximum traveling distance as well as the radius. This has to be chosen carefully to get the right results. In a professional environment (e.g. game development), these values would be set by artists and designers for every scene. In some cases it could make sense to provide the means to set different values for different parts of the scene, especially in the case of transitions from small, enclosed spaces to large outdoor scenes. This can be easily achieved by using a few small adaptions in the scene builder to make the system use different regions for the rendering and for setting the global illumination values per region.

5.4.4. Box City Generator

The box city generator is a special kind of scene builder I built to easily generate a variety of testing scenarios. It generates an abstract city of boxes, complete with city blocks, streets and back alleys. The 'city' size can be set to any value. The box 'buildings' have in turn smaller randomized box structures attached to them to achieve more variety among the 'buildings' themselves.

The maximum and minimum sizes of buildings, streets and back alleys can also be set to different values. The smaller box additions to every building also allow for a better testing scenario of finer rendering details in comparison to larger geometry.

5.4.5. Tone Mapping

The tone mapping is the last step of rendering each frame. For my prototype I used a very simple tone mapping approach, which changes the final render image to achieve a better contrast. A more dynamic tone mapping could help to improve the transition between enclosed spaces and outdoor scenes. This post-effect can be written in such a way that it does not have a great effect on computation time and resources, even if it is dynamic.

```
float luminance = 0.3f * result.r + 0.6f * result.g + 0.1f * result.b;

float mappedLuminance = clamp(0,1,luminance / luminanceMax);
mappedLuminance = pow(mappedLuminance, GAMMA);

result.rgb = (mappedLuminance/luminance) * result.rgb;
```

tone mapping pixel shader

The gamma and maximum luminance values for the tone mapping computation have to be set for each scene, according to complexity, number and power of lights and number of photons.

Tone mapping is required, since a large number of photons in addition to the direct light make the final rendering appear over-exposed, i.e. the contrast has to be improved. For this, the tone mapping algorithm uses a maximum luminance value and a gamma constant. More on tone mapping can be found in [AHH08].

6.Evaluation

In this chapter I will compare the performance and quality of my system to the goals I had, and to previous approaches. Some of the results of previous approaches might not be directly comparable as they rely on very different concepts and/or hardware.

6.1. Testing Environment

As the OptiX ray tracing engine is based on CUDA, which only runs on Nvidia cards, I could only test the system on these. The computer I used for evaluation consists of the following components: an Intel i3 2.x Ghz CPU, and a Nvidia Geforce 560 Ti GTX (slightly overclocked). All other components should not have a significant impact on the performance of the computation. The operating system used is Windows 7 (64bit) with the latest Nvidia graphics driver (v. 285.62) and DirectX 11 installed. If not otherwise noted, all results were measured at a resolution of 1280*1024.

My evaluation focusses on the following criteria: Obviously the system needed to be able to run in real time and I wanted to see if my adaptions improved the performance. In addition, the quality of the rendering needed to be evaluated with the further conditions of being able to render dynamic geometry and not needing any pre-computation. The dynamic geometry needed to be fully integrated into the lighting calculation, i.e. also influence the indirect light.

I mainly compared my results to the results of the paper this work is based on, but also to comparable techniques used in modern real-time environments if applicable. Most modern global illumination techniques which are used still only include static geometry in the calculation of indirect light.

6.2. Results

The goals described above were achieved, even though they did result in new problems. In the following I shall describe the results of my prototype when using this approach in different test scenes. The artifacts and problems that appeared will also be shown with possible solutions given in [7. Conclusion and Future Work], if not already implemented.

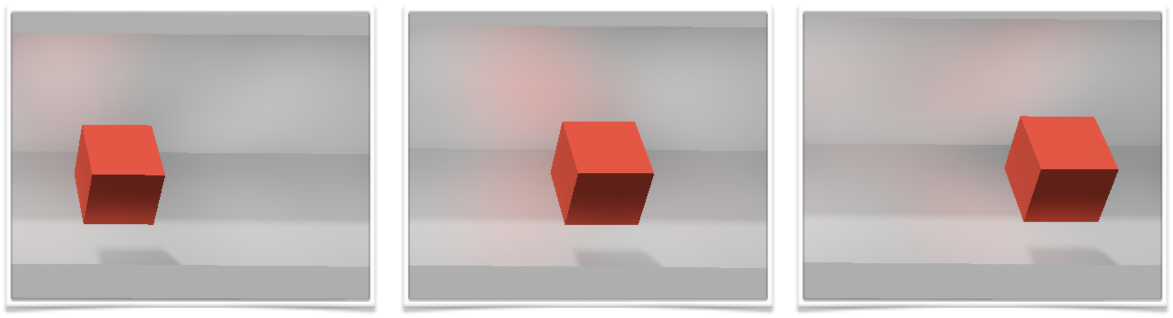
The following times are all in milliseconds (ms), using the box city scene, with 50000 photons and a maximum of three bounces.

I first tried to implement my approach in different graphics and game engines without much success, either due to lacking flexibility and features or to time constraints. For the final prototype, I implemented the basic system myself and only used the DirectX API and OptiX. The first implementation was the basic deferred shading system for creating the G-buffer for every light source as well as the viewer camera. The system manages to render all the necessary information, including full position and normal data as well as diffuse, specular and transmissive values in about 3 ms. The G-buffer data then allows the initial bounce buffer creation and direct lighting steps to be independent of the complexity of the geometry in the scene. The bounce buffer step, which renders the data directly into a stream output buffer, takes about 0.1 ms. Due to the ability to use the stream output buffer directly as input for the tracing steps, time that otherwise would bespent on building the photon buffer and transferring it between CPU and GPU is saved.

Most of the time needed for each frame is spent on tracing and splatting photons. The tracing in OptiX, complete with setup and launching of the photons, takes about 10-30ms on the GPU and between 20 and 50ms on the CPU, in general fluctuating with the visible complexity of the scene. Regrettably, OptiX does not provide any means for an in-depth analysis of these numbers.

The splatting of the photons is another very costly step of the whole approach, mostly due to the high fill-rates caused by the splats needing to overlap each other to form a satisfying result. This step largely depends on the number of photons used and the number of overlapping splats. Multiple steps were made to improve the performance, namely rendering the photon splats in multiple passes to reduce the fill rates in each pass, which improved performance by about 8 ms per frame. I also implemented a different system for determining the photon splat radius by using the distance travelled by the photon. This further improved the performance by about 30-50 ms, when the minimum and maximum radius for the photon splats were optimally chosen for the scene. This improvement is especially noticeable for transitions between indoors and outdoors or vice versa.

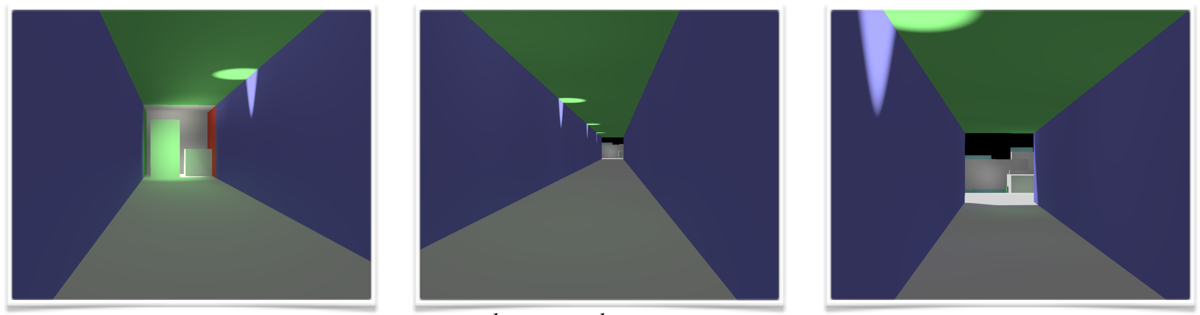
The total time spent on each frame is about 40-50 ms (about 20-25 FPS), which means that I have successfully reached real-time speeds in the range found in high-end interactive environments like games. The main speed improvement comes from eliminating the transfer steps between CPU and GPU and by tracing the photons on the GPU. This performance was reached without using any form of sophisticated culling or *level of detail* optimizations. The system also still does not require any form of pre-computation and thus allows the integration of dynamic light sources and geometry in the scene. This, of course, introduces a new problem of inter-frame coherence, i.e. 'flickering'. Possible solutions to this problem are described in the following chapter. You can see an example of this behavior in the images below. Note the changing appearance of the red color bleeding on the wall and floor as the red cube moves from left to right.



dynamic geometry (please note changing indirect illumination)

The first goal, successfully adapting the approach in [ML09] to be able to render outdoor scenes, i.e. directional light, and indoor scenes, i.e. spotlights and point lights, was achieved successfully by using the described implementation. The orthographic camera is automatically adapted to include the whole scene and can be easily adapted to render larger scenes at the same resolution by using multiple cameras to render parts of the scene. As this information is only used for the photon tracing, problems with seams generally do not appear, as the data is not used for direct rendering.

The transition between in- and outdoor scenes, i.e. between spotlights and direct light, is also easily done, as they can be rendered at the same time with the same system. In my prototype I made the number of photons used per light source adaptable. In the 'box city' scene, for example, 10% of the total numbers of photons are allocated for each spotlight, as fewer photons are usually needed for enclosed spaces. Most photons are used for the directional light, since many are lost due to reflection back out of the scene.



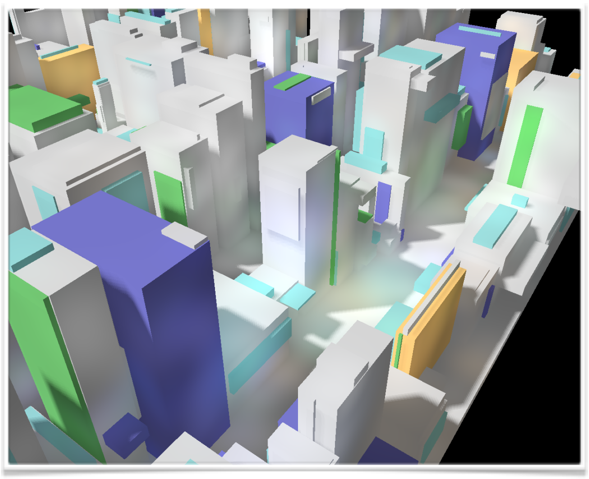
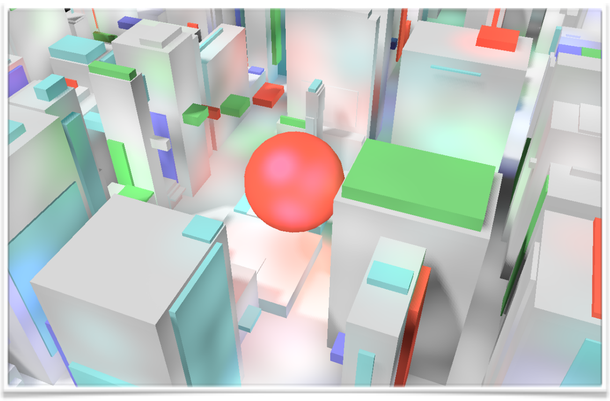
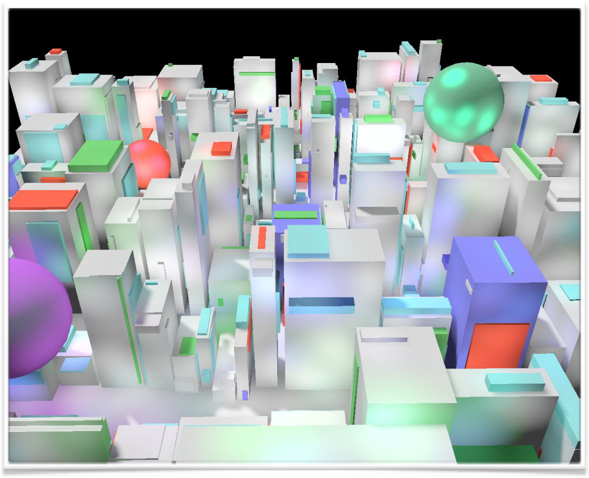
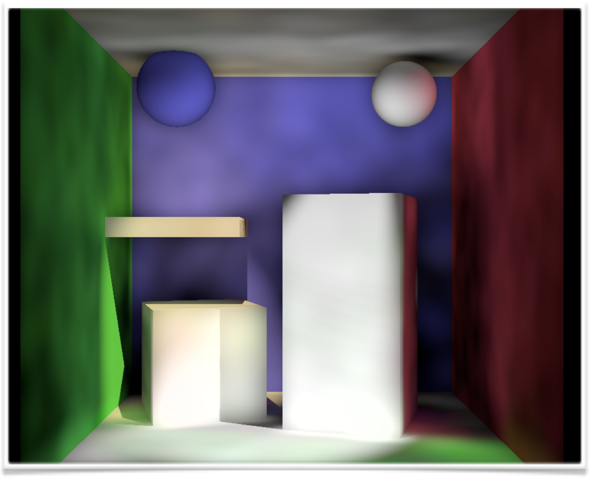
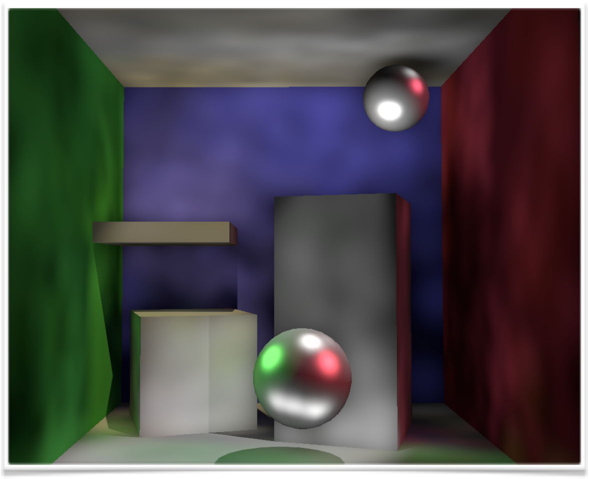
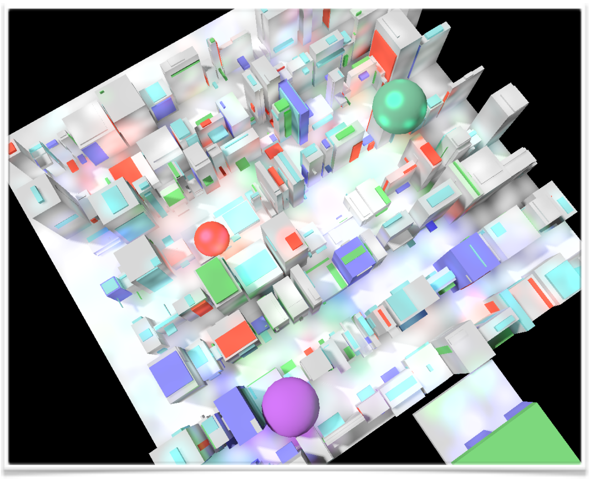
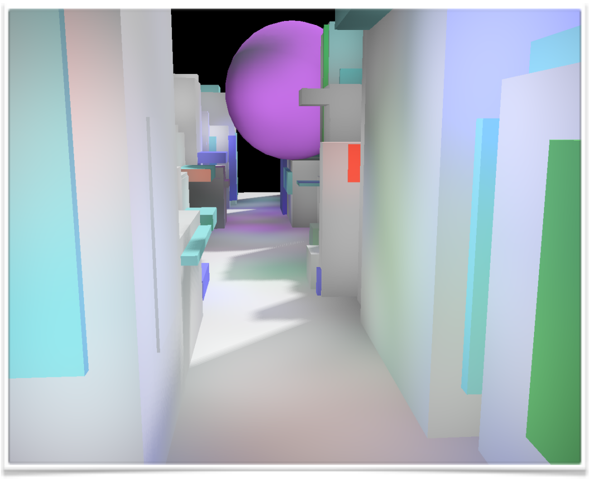
indoor to outdoor transition

In comparison to the previous approaches, my approach has a few advantages and disadvantages. The visual quality of the approach is generally on par with most other approaches.

Most approaches used in games have far superior performance but also a few drawbacks my system does not have. Precomputed light maps, due to their static nature, cannot handle dynamic light sources and or objects easily, and exempt dynamic objects from interaction with the indirect light completely. Even if light probes are used to light dynamic objects, they will not interact with the indirect light in the scene, only casting shadows from direct lighting. My system does provide a way of fully including dynamic geometry in the complete light calculation, including color bleeding.

Even though only diffuse and specular reflection are shown, the system can easily handle caustics caused by transmissive materials, providing a complete system for calculating all of these indirect lighting effects. The system provides all the necessary elements to compute the correct result, including even highly detailed effects, while still providing real-time frame rates. It is also easily combined with existing lighting systems and effects.

In the following, screenshots showing additional results of the implemented prototype are included.

Additional results, rendered at 1280*1024 at a minimum of 20FPS.

7.Conclusion and Future Work

7.1. Conclusion

The prototype implementation does reach all my planned goals [1. Introduction and Goals], even if some of them are not achieved in the form I had hoped for. The working approach shown here correctly calculates and renders indirect light due to diffuse and specular reflection at real-time speeds while running completely on the GPU.

It also manages to render in- and outdoor scenes and easily combines them with seamless transition. This includes the successful calculation and rendering of directional light using a orthogonal camera.

The visual quality is also satisfying and can compete with other real-time global illumination approaches.

Even if the approach was successfully implemented and shown in the prototype I created, it does lack a few features due to time constraints. One major one is the handling of transmissive materials. The photon tracing already handles transmissive materials and can calculate caustics, but is not completely debugged and tested to be finished. I also did not have enough time to implement the direct lighting for transparent materials, resulting in an insufficient final rendering, which would not provide an appropriate visual representation for caustics caused by transparent materials. Nonetheless the system is able to handle these effects and would only need further adaptions.

Even though still very computation intensive, the approach already reaches good speeds and a high rendering quality. This could be further improved by simple culling, level-of-detail techniques and subsampling to improve rendering time. Though it could not be used in a game in its current state, it does provide a viable approach for future implementations as it handles the complete calculation of indirect illumination and is easily integrated into an existing system.

7.2. Future Work

Aside from finishing the implementation of handling transmissive materials and rendering caustics, the main bottleneck and problem of this approach is still the splatting of the photons. A few concepts have been proposed to further improve this step, as in [DS06], which further adapts the photon splat radius and form to better fill the scene with less overlap. Another viable approach would be to cache previous photon results and change the splatting to be handled progressively, distributing it over several frames. As described, I tested rendering the photon splats in multiple draw calls instead of one, and even this already improves performance.

The implementation of [ML09] also uses a subsampling approach for the splatting of the photons to achieve better performance at a lower resolution and an additional blurring of the final information for the final rendering, resulting in a smoother but less accurate image. This could also be easily used to improve this approach in the same way. The drawback being that finer details are smoothed out and will not be rendered correctly. This can become a problem for specular reflections or refractions when rendering caustics.

As OptiX does not provide an easy way to handle a dynamic change of the maximum number of photons or photon bounces and only communicates with the graphics API through an extra abstraction level, i.e. buffers and settings given from the CPU, one could be better off by implementing a specialized system using the same graphics API for the whole process, i.e. implementing a new custom pipeline for handling the photon tracing. This could be done by implementing a GPU ray-tracing system using Compute Shaders, which would mean a consistent system with fewer requirements, which would be even easier to integrate and optimize for rendering global illumination in real-time with photon mapping.

Using Compute shaders to build a custom pipeline would also mean that there needed to be even less CPU interaction, meaning that the implementation could directly use data already provided on the GPU to improve the rendering step. This would include the number of photons drawn to the stream output or number of photon results saved as well as performance queries. These could then be used to dynamically adapt the photon count and number of calculated bounces, and would make an implementation using caching and progressive photon mapping far easier.

In addition, setting up a specialized system would mean having full control which could help eliminate further bottlenecks, which are otherwise ‘hidden’ inside the OptiX ray-tracing engine. Any bottlenecks could then be improved upon with better debugging options and a wider range of suitable profiling tools.

Building a custom system for the photon tracing step would also mean being able to make it compatible with non-Nvidia hardware, which would be a basic requirement for using it in a professional application or game.

Another problem with the current system is that the way the random numbers are generated, using a linear congruential generator [G03], is not sophisticated enough to be reliably stable and uniform.

More sophisticated random number generators, e.g. the Mersenne Twister [MN98], could be implemented on the GPU and would further improve the calculation due to better and more stable random numbers.

To solve the inter-frame coherence problems caused by dynamic lights and objects one would also have to further improve the generation and use of uniform random numbers. To ensure inter-frame coherence one would need to make sure that photons are always reflected in the same manner so as to use the same photon path in each frame, which would mean a consistent distribution of the random numbers used to calculate the reflection dependent on the objects' local positions. This could be solved by making sure that the random numbers for the reflection are dependent on the relative location on the surfaces of the object (in object space) and do not change along its surface or for predetermined regions. One could also try to implement a gradient for the random numbers to provide a smooth transition between reflection changes. With this approach a uniform distribution of random numbers could not be maintained easily, if at all.

For ensuring inter-frame coherence for dynamic lights an analogue approach could be used, making the random numbers for the calculation of the reflection dependent on the location on a object's surface. In any case a way must be found to make sure that the photon path stays similar across frames and does not change rapidly.

Still more improvements could be made by using optimization approaches from other related areas such as the fields of shadow mapping, progressive photon mapping, improved culling of lights to be rendered (including photons) and a level of detail approach for splatting and tracing steps.

Acknowledgments

I would like to especially thank my supervisor Jun.-Prof. Dr. Thorsten Grosch, who spent a great deal of time and energy supervising the writing of this thesis. Also of note is the great help of Tobias Günther, without which I would have needed far longer to understand the details of the OptiX ray tracing engine and some of the more obscure Direct3D errors. I would also like to thank Prof. Dr.-Ing. habil. Holger Theisel for the additional supervision on short notice.

Many thanks also go to all the proofreaders for investing their time to help me find errors in my thesis.

Last but definitely not least a huge thanks goes to my family, esp. my parents, for their support not only during my thesis but my whole time at the university. Without them I probably would not have had the motivation to study the subject that seemed the most challenging and would never have continued with it those times when it seemed impossible, or at least impossibly hard.

Appendix

A. References

[AHH08] AKENINE-MÖLLER T., HAINES E., HOFFMAN N.: “Real-Time Rendering”, Published by *A K Peters, Ltd.* (2008) Third Edition.

[AK90] ARVO J., KIRK D.: “Particle Transport and Image Synthesis”, In *SIGGRAPH ’90 Proceedings* (1990)

[B05] BUNNEL M.: “Dynamic Ambient Occlusion and Indirect Lighting”, In *GPU Gems 2* published by *Nvidia* (2005), Chapter 14

[BEAST] AUTODESK INC.: “Beast”, (2010-2011) http://gameware.autodesk.com/beast

[CPP05] CEREZO E., PEREZ-CAZORLA F., PUEYO, X., SERON, F., SILLICON, F.: “A Survey on Participating Media Rendering Techniques” In *The Visual Computer* (2005).

[CPC84] COOK R., PORTER T., CARPENTER L.: “Distributed Ray Tracing”, In *SIGGRAPH ’84 Proceedings* (1984)

[CUDA] NVIDIA CORPORATION: “CUDA API Reference Manual”, Published by *Nvidia* (2011) v. 4.0.

[DBB06] DUTRÉ P., BALA K., BEKAERT P.: “Advanced Global Illumination”, Published by *A K Peters, Ltd.* (2006) Second Edition.

[DIRECTX] MICROSOFT CORPORATION: “DirectX (SDK)” (2004-2010).

[DS06] DACHSBACHER C., STAMMINGER M.: “Splatting Indirect Illumination”, In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2006)

[EAS09] EISEMANN E., ASSARSSON U., SCHWARZ M., WIMMERL M.: “Casting Shadows in Real Time”, In *ACM SIGGRAPH Asia 2009 Courses* (2009)

[F05] FORSYTH T.: “Making Shadow Buffers Robust Using Multiple Dynamic Frustums”, In *Shader X^4*, Published by *Charles River Media* (2005)

[G03] GENTLE J.: “Random Number Generation and Monte Carlo Methods”, Published by *Springer Science+Business Media* (2003) Second Edition.

[G10] GROSCH T.: “Photorealistische Computergrafik”, Lecture at Otto-Von-Guericke Universität (2010)

[G3D] MCGUIRE M., (OPEN SOURCE) “G3D Innovation Engine” (2010), v.8.01, http://g3d.sourceforge.net/

[HW] HERIOT WATT UNIVERSITY “Dissertation Guide”

[J01] JENSEN H.: “Realistic Image Synthesis Using Photon Mapping”, Published by *A K Peters, Ltd.* (2001).

[K86] KAJIYA J.: “The rendering equation”, In *SIGGRAPH Computer Graphics 20* (1986), Nr. 4

[KD10] KAPLANYAN A., DACHSBACHER C.: “Cascaded Light Propagation Volumes for Real-Time Indirect Illumination”, In *ACM SIGGRAPH symposium on Interactive 3D Graphics and Games, proceedings* (2010)

[M10] MCLAUGHLIN J.: “ Realistic Atmospheric Rendering and Integration into an Interactive Real-Time Environment”, At *Otto-von-Guericke Universität Magdeburg* (2010), Studienarbeit

[ME10] MARTIN S., EINARSSON P.: “A Real-Time Radiosity Architecture for Video Games”, At *SIGGRAPH* (2010)

[ML09] MCGUIRE M., LUEBKE D.: “Hardware-Accelerated Global Illumination by Image Space Photon Mapping”, In *ACM SIGGRAPH/EuroGraphics High Performance Graphics* (2009)

[MMA07] MALMER M., MALMER F., ASSARSSON U., HOLZSCHUCH N.: “Fast Precomputed Ambient Occlusion for Proximity Shadows”, In *journal of graphics, gpu, and game tools* (2007)

[MN98] MATSUMOTO M., NISHIMURA T.: “Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator”, In *ACM Tans. on Modeling and Computer Simulation Vol. 8* (1998)

[N11] NISSEN C.: “Progressive Photon Mapping für komplexe Szenen”, At *Otto-von-Guericke Universität Magdeburg* (2011), Diplomarbeit

[NSHADER] MUTEL, A.: “NShader”, (2009-2011) v. 1.2.

[OPTIX] NVIDIA CORPORATION: “Nvidia OptiX Ray Tracing Engine Programming Guide”, Published by *Nvidia* (2011) v. 2.1.

[PDB10] PARKER S., BIGLER J., DIETRICH A., FRIEDRICH H., HOBEROCK J., LUEBKE D., MCALLISTER D., MCGUIRE M., MORLEY K., ROBISON A., STICH M.: "OptiX: A General Purpose Ray Tracing Engine", In *ACM Transactions on Graphics, SIGGRAPH Proceedings* (2010)

[RGK08] RITSCHEL T., GROSCH T., KIM M., SEIDEL H., DACHSBACHER C., KAUTZ J.: "Imperfect Shadow Maps for Efficient Computation of Indirect Illumination", In *ACM Transactions on Graphics, SIGGRAPH Asia Proceedings* (2008)

[RGS09] RITSCHEL T., GROSCH T., SEIDEL H.: "Approximating Dynamic Global Illumination in Image Space", In *ACM SIGGRAPH symposium on Interactive 3D Graphics and Games, proceedings* (2009)

[RKS07] RAGAN-KELLEY J., KILPATRICK C., SMITH B., EPPS D., GREEN P., HERY C., DURAND F.: "The Lightspeed Automatic Interactive Lighting Preview System", In *ACM Transactions on Graphics* (2007)

[S93] SCHLICK C.: "A Customizable Reflectance Model for Everyday Rendering", In *Fourth Eurographics Workshop on Rendeing* (1993)

[ST90] SAITO T., TAKAHASHI, T.: "Comprehensible Rendering of 3-D Shapes", In *Computer Graphics, SIGGRAPH Proceedings* (1990)

[STEAM] VALVE CORPORATION: "Steam Hardware & Software Survey November 2011", Conducted and Published by *Valve Corporation.* (2011) http://store.steampowered.com/hwsurvey

[VAX] WHOLE TOMATO: "Visual Assist X" (2011)

[VS2010] MICROSOFT CORPORATION: "Visual Studio 2010" (2010).

[W80] WHITTED T.: "An Improved Illumination Model For Shaded Display", In *Communications of the ACM 23* (1980), Nr. 6

B. Statement of Authorship

I, John McLaughlin, confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g. ideas, equations, figures, text, tables, programs) are properly acknowledged at the point of their use. A full list of the references employed has been included. [HW]

Signed: ....   Date: ....