České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačové grafiky a interakce

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Petr Šefčík**

Studijní program: Otevřená informatika
Obor: Počítačová grafika a interakce

Název tématu: **Výpočet globálního osvětlování v reálném čase pomocí CLVP**

Pokyny pro vypracování:

Prostudujte problematiku výpočtu globálního osvětlení v reálném čase a konkrétní algoritmy efektivní pro dynamické scény pri použití paralelní architektury GPU. Navrhněte a implementujte algoritmus "Cascaded Light Propagation Volumes" včetně aspektů podporující lesklé materiály, zastínění a několikanásobné odrazy světla. Prozkoumejte artefakty typické pro tuto metodu a možnosti jejich odstranění. Otestujte implementaci na sadě alespoň pěti scén a srovnejte výsledky metody s kanonickým řešením získaným nestrannou metodou a případně s výsledky alternativních metod pro výpočet globálního osvětlení v reálném čase.

Seznam odborné literatury:

[1] Kaplanyan, Anton, and Carsten Dachsbacher. Cascaded light propagation volumes for real-time indirect illumination. Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games. ACM, 2010.
[2] Kaplanyan, Anton. Light propagation volumes in CryEngine 3. ACM SIGGRAPH Courses 7 (2009)

Vedoucí: doc.Ing. Vlastimil Havran, Ph.D.

Platnost zadání: do konce letního semestru 2015/2016

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE Fakulta elektrotechnická -4- L.S.

[illegible]
prof. Ing. Jiří Žára, CSc.
vedoucí katedry

[illegible]
prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 25. 3. 2015

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Master's Thesis

Computing global illumination in real-time using Cascaded Light Propagation Volumes

Bc. Petr Šefčík

Supervisor: doc. Ing. Vlastimil Havran, Ph.D.

Study Programme: Open Informatics

Field of Study: Computer Graphics and Interaction

January 11, 2016

Aknowledgements

I would like to express my appreciation to my supervisor doc. Ing. Vlastimil Havran, Ph.D. for his guidance and encouragement. I am also grateful to my family and girlfriend for their support and providing me with an ideal work environment.

Declaration

Abstract

Computing global illumination is essential for synthesis of realistic looking images. Global illumination algorithms are however very computationally expensive and up until recently unfeasible to realize in real time.

The subject of this thesis is implementation of Cascaded Light Propagation Volumes. Introduced by Kaplanyan and Dachsbacher in 2010, this algorithm takes a fluid-simulation-inspired approach to approximating global illumination in fully dynamic environments. The implementation is incorporated into a rendering package and tested on several scenes. It produces visually pleasing results at interactive to real-time frame rates.

Keywords: global illumination, indirect illumination, cascaded light propagation volumes, real-time rendering

Abstrakt

Výpočet globálního osvětlení je nezbytný k syntéze realisticky vypadajících obrazů. Algoritmy globálního osvětlení jsou ale velmi výpočetně náročné a ještě v nedávné době nerealizovatelné v reálném čase.

Náplní této práce je implementace Cascaded Light Propagation Volumes. Tento algoritmus, představený Kaplanyanem a Dachsbacherem v roce 2010, používá iterativní schéma inspirované technikami pro simulaci tekutin k věrohodné aproximaci globálního osvětlení v plně dynamických prostředích. Implementace algoritmu je zabudována do zobrazovací aplikace a otestována na několika scénách, kde podává vizuálně uspokojivé výsledky s malými časovými nároky.

Klíčová slova: globální osvětlení, nepřímé osvětlení, cascaded light propagation volumes, rendering v reálném čase

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer graphics, the creation of images using computers, is a field with a broad impact. In manufacturing industry, computer aided design tools are used to bring new products to life with speed. Visualization using computer generated imagery is essential in today's science, medicine and engineering. Virtual and mixed reality systems, digital art and education also benefit from advances in this field. Finally, computer graphics is leveraged to a great extent in the film and video-game industry.

1.1 Motivation

Computer generated imagery can be applied in various ways but individual applications pose different constraints on underlying algorithms for image synthesis. Generating photo-realistic images ultimately comes down to simulating physical processes that take place when light interacts with matter. The physics of light are understood well enough but the processes involved are often too complex to be practical to simulate on current hardware. Depending on the application, concessions have to be made.

If there is ample time, an algorithm that produces high fidelity results by accounting for majority of physical phenomena can be employed without a problem, even if it takes tens of minutes to synthesize a single image. On the other hand, if time is of the essence, an entirely different array of techniques using a coarser approximation of reality must be used. Real-time rendering is a subfield of computer graphics which explores such methods. Most common real-time rendering applications are video-games and various virtual or mixed reality simulations. In these scenarios, images must be generated fast enough to allow the user a comfortable interaction with the environment (time budgets up to roughly 33 ms).

Global illumination poses a major obstacle in real-time rendering algorithm design. In this context, global illumination refers to the capacity of a method to account for both direct and indirect illumination. Direct illumination accounts for up to one light-surface interaction before light reaches an observer. Computing direct illumination is relatively simple, but light may only ever reach surfaces directly visible from some light source; the rest is not lit at all. Indirect illumination accounts for many subsequent light-surface interactions. After scattering multiple times, light may reach surfaces that are not lit directly. Indirect illumination is difficult to evaluate but is responsible for many real life phenomena such

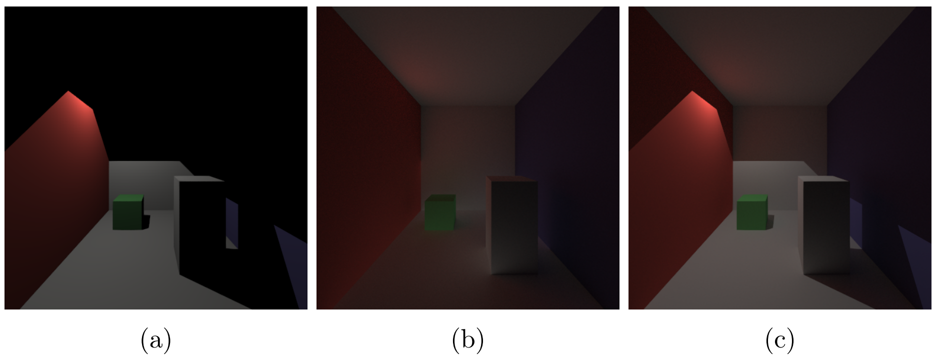


Figure 1.1: Global illumination: indirect illumination substantially improves visual fidelity of an image. (a) Direct illumination. (b) The first bounce of indirect illumination. (c) Direct and indirect illumination combined.

as soft shadows, color bleeding and caustics. Figure 1.1 illustrates the impact of indirect illumination on the visual the fidelity of an image.

Historically, in real-time rendering, indirect illumination was either precomputed or crudely approximated at runtime. Where viable, precomputation is a natural solution to the problem of constrained time budget, because it moves the complex computation step before the application is run; the precomputed solution only needs to be fetched and applied during runtime. One disadvantage of this approach is, that it only yields correct results for static scenes and the complete scene layout must be known at the precomputation time. For dynamic scenes, precomputing indirect illumination is not an option. Fortunately, recent rendering hardware generations provide enough power to support techniques previously considered infeasible. Also, with high demand for dynamic environments in video games, novel algorithms emerge, designed precisely to meet this particular set of requirements. Nonetheless, these approaches are still coarse approximations of reality and their robust implementation, maximizing performance and minimizing artifacts, remains a non-trivial effort.

1.2 Goals

The goal of this thesis is to survey algorithms suitable for computing indirect illumination on current hardware at real-time speeds and implement a modern rendering package integrating one such algorithm: Cascaded Light Propagation Volumes. Additionally, performance of the resulting implementation is to be thoroughly evaluated and compared to the ground truth.

1.3 Structure

The thesis is structured in a way to allow an initiated reader to skip sections on familiar material, while a novice is still presented with the bare minimum theory to appreciate techniques explored later. The chapter 2 explores the basics of image synthesis: it deals with light, light-matter interaction and computation of illumination. The following chapter 3 introduces a number of approaches for dealing with indirect illumination in real-time with special focus on the Cascaded Light Propagation Volumes algorithm. The chapter 4 elaborates on the rendering application design choices. Implementation details are explored in chapters 5 and 6, the former focused on the core rendering system in detail, the latter dedicated to the rest. Performance characteristics of the implemented algorithm and visual fidelity of images it produces are evaluated in the penultimate chapter 7. The final chapter 8 then concludes the thesis with a summary and suggestions of future work.

Chapter 2

Image synthesis

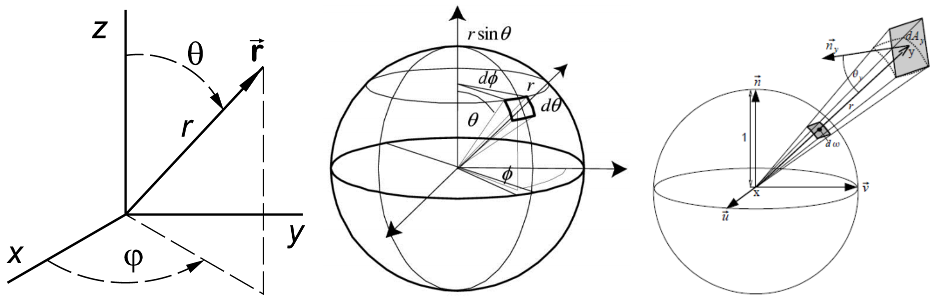
Photo-realistic images are generated by simulating real, physics-based interactions that occur between light and matter. While graphics algorithms use a suitably simplified physics model, having a grasp of the underlying theory is useful, because it makes it easier to see where the simplifications come from and why they work when they do. This chapter reviews the relevant rudiments. A quick introduction of useful geometry constructs is followed by the basics of light-matter interaction and how it is abstracted in computer graphics. The final part deals with computation of illumination.

2.1 Spherical coordinates

In image synthesis, we often deal with functions whose parameters are directions. A common way to represent a 3D direction is with a unit vector in Cartesian coordinates. Other useful representations for directions exist, which are sometimes more advantageous to use because they allow for easier simplification of expressions and introduce fewer degrees of freedom when used as function parameters.

Spherical coordinates map directions to points on a sphere whose surface is parametrized with *polar angle* θ and *azimuth* φ (figure 2.1a). The relationship between the Cartesian ($\vec{r}_C$) and spherical ($\vec{r}_S$) coordinates of a direction $\vec{r}$ is expressed in the following equation:

$$\begin{aligned}
&\vec{r}_C = (x, y, z) && x, y, z \in [0,1], x^2 + y^2 + z^2 = 1 \\
&\vec{r}_S = (\theta, \varphi) && \theta \in \left[0, \frac{\pi}{2}\right], \varphi \in [0, 2\pi] \\
\\
&\vec{r}_C \to \vec{r}_S : && \vec{r}_S \to \vec{r}_C : \\
&\quad \theta = \arccos z && x = \sin\theta\cos\varphi \\
&\quad \varphi = \arctan\frac{y}{x} && y = \sin\theta\sin\varphi \\
& && z = \cos\theta
\end{aligned} \tag{2.1}$$



(a) (b) (c)

Figure 2.1: (a) Relationship between spherical ($[\theta, \phi]$) and Cartesian ($[x, y, z]$) coordinates of a direction $\vec{r}$. (b) A differential solid angle. (c) Relationship between a differential solid angle and a differential surface from the equation 2.3.

2.2 Solid angle

An *angle* denotes the length of an arc on a unit circle. Similarly, a *solid angle* denotes the size of an area on a unit sphere. A solid angle is measured in *steradians* [sr], a dimensionless unit. Let A be the area of a projection of an object onto a sphere with the radius r, then the solid angle Ω is defined as:

$$\Omega = \frac{A}{r^2}. \tag{2.2}$$

A *differential solid angle* $d\vec{\omega}$ is another useful quantity. It represents an infinitesimally small solid angle around a direction. The direction of $d\vec{\omega}$ points to the center of projection of the differential surface on a unit sphere; the size of $d\vec{\omega}$ is equal to the area $dA = \sin\theta d\theta d\varphi$ of the differential surface on a unit sphere (figure 2.1b). Note the term $\sin\theta$, which comes from the $[\theta, \varphi]$ parametrization and accounts for the fact that sphere surface patches have smaller area closer to poles. Finally, a differential solid angle can be computed from a differential surface as follows:

$$d\vec{\omega} = \frac{\cos\theta dA}{r^2}, \tag{2.3}$$

where dA is the differential area of the differential surface, and θ and r are its orientation and distance as shown in the figure 2.1c.

2.3 Light transport

Light transport deals with processes that occur when energy transfers between media that affect visibility. This section is dedicated to exploring these processes and how they are commonly modeled in computer graphics and real-time rendering specifically.

2.3.1 Light

In general, light refers to electromagnetic radiation. More precisely, visible light is a name for the subset of electromagnetic radiation with wavelengths of length from roughly 400 nm to 700 nm. Radiation in this portion of the electromagnetic spectrum is visible to a human eye.

Light exhibits characteristics of both waves and particles, a property called the wave-particle duality. Depending on the context, light is better described using concepts appropriate to waves or particles, but is exactly neither. The elementary light particle, the quantum of light, is called a photon. The speed of light in vacuum c is precisely $299\,792\,458\,\mathrm{m\,s^{-1}}$; the value is precise because the meter is defined in terms of c.

In computer graphics, a highly abstracted model of light based on ray optics is commonly used where light propagates rectilinearly along rays through homogeneous media. At interfaces between two dissimilar media, a light ray may reflect, transmit or get absorbed. Finally, the speed of light is ignored.

Two terminologies are available when discussing light and energy transport. The first one, *radiometry*, describes optical radiation. Optical radiation is electromagnetic radiation with wavelengths between 0.01 µm and 1000 µm subdivided into regions corresponding to ultraviolet, visible and infra-red light. The other terminology is *photometry*, which is only concerned with visible light. Photometric quantities correspond to radiometric quantities but are additionally weighted by the spectral response of the human eye. Of the two, radiometry is the terminology more prevalent in computer graphics.

Due to the wave nature of light, all radiometric quantities are some function of wavelength. Definitions presented below are for a single wavelength. To compute the values for some real light, which is rarely composed of radiation of only a single wavelength, one needs to integrate over the range of wavelengths of interest. In practical applications, a discrete subset of wavelengths is selected (e.g. red, green and blue) and computation carried out for each element individually.

Radiant energy

$$Q = ne[\mathrm{J}], \tag{2.4}$$

is the total energy of all photons inside a volume. n is the number of photons inside the volume and e is the energy carried by a single photon. *Radiant flux*

$$\Phi = \frac{dQ}{dt}[\mathrm{W}], \tag{2.5}$$

is the rate of change of energy per time unit. *Irradiance*

$$E(x) = \frac{d\Phi}{dA}[\mathrm{W\,m^{-2}}], \tag{2.6}$$

is the total energy incoming from all directions at a point x per time unit. Note that E depends on the surface normal as it only makes sense to consider directions over the upper hemisphere. *Radiosity*

$$B(x) = \frac{d\Phi}{dA}[\mathrm{W\,m^{-2}}], \tag{2.7}$$

is the total energy emitted in all directions at a point x per time unit. Like irradiance, radiosity depends on the surface normal. *Radiant intensity*

$$I(\vec{\omega}) = \frac{d\Phi}{d\vec{\omega}}[\mathrm{W\,sr^{-1}}], \tag{2.8}$$

is the total energy emitted in a given direction per time unit. *Radiance*

$$L(x, \vec{\omega}) = \frac{d^2\Phi}{\cos\theta dA d\vec{\omega}} [\mathrm{W\,m^{-2}\,sr^{-1}}], \tag{2.9}$$

is the total energy arriving from the direction $\vec{\omega}$ at a point x. Due to the normalization by $\cos\theta$, radiance is invariant to the surface orientation at x. Given the absence of participating media (in vacuum), radiance stays constant along a ray, which makes it very useful quantity in light transport computation.

To give an example of a photometric quantity, *luminous flux* for the wavelength λ is defined as:

$$\Phi_V(\lambda) = \Phi(\lambda)V(\lambda)[\mathrm{lm}], \tag{2.10}$$

where $V(\lambda)$ denotes the spectral response of the human eye.

2.3.2 Light-matter interaction

When light encounters matter, it scatters in many directions. In the real world, light is reflected and transmitted depending on a number of factors like wavelength, time and the incident and outgoing direction. Nature of this interaction may also vary across the surface and depend on the incoming and even outgoing position. If a position on the surface is expressed using two parameters, accounting for all aforementioned factors yields a ten dimensional function. While comprehensive, such function is not practical for use in simulation on current hardware due to its high dimensionality. To get a usable abstraction, some parameters of the general scattering interaction are neglected.

Time and wavelength domains, necessary for modeling phosphorescence and fluorescence, are usually ignored because materials exhibiting such properties are rare or handled specially. *Subsurface scattering* refers to a phenomenon, where light scatters below the surface of an object and then leaves at a position possibly different from the one at which it entered. Examples of real world materials with such properties are skin, wax, milk and marble. If correct handling of these materials is required, spatial variation can be disregarded, leading to a six dimensional *Bidirectional Scattering Surface Distribution Function* (BSSRDF). Conversely, ignoring the phenomenon yields a six dimensional *Bidirectional Texture Function* (BTF), where the incoming and outgoing position of the scattering event are considered to be identical.

There are two simple ways to eliminate the two spatial degrees of freedom present in BTFs. The first is to subdivide the surface into parts over which the function is spatially constant. The second is to encode the spatially varying factors (e.g. albedo) into a two dimensional texture. The four dimensional representation resulting from this approach is called a *Bidirectional scattering distribution* function (BSDF). BSDF can be further decomposed into a *Bidirectional Reflectance Distribution Function* (BRDF) and a *Bidirectional Transmittance Distribution Function* (BTDF). Only BRDFs are important to the methods explored in this thesis.

2.3.3 BRDF

Generally speaking, a BRDF describes reflective properties of a material for incident and outgoing direction at a point. According to the formal definition:

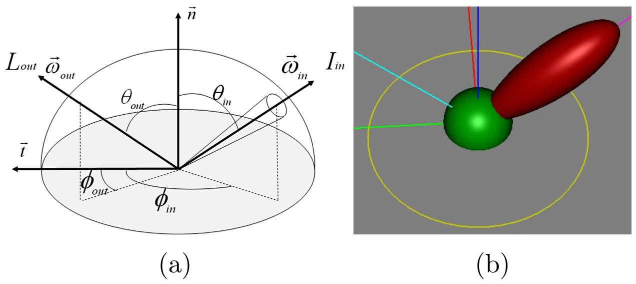


Figure 2.2: (a) Bidirectional Reflectance Distribution Function. (b) Visualization of the Blinn-Phong BRDF diffuse (green) and glossy (red) terms for a selected incident direction (cyan). Obtained using BRDF Explorer [Dis15].

$$f_r(x, \vec{\omega}_i, \vec{\omega}_o) = \frac{dL_o(x, \vec{\omega}_o)}{dE(x, \vec{\omega}_i)} = \frac{dL_o(x, \vec{\omega}_o)}{L_i(x, \vec{\omega}_i) \cos\theta_i d\vec{\omega}_o}, \tag{2.11}$$

a BRDF states what fraction of energy coming from the incident direction $\vec{\omega}_i$ (over the differential solid angle $d\vec{\omega}_i$) at point x is reflected in the outgoing direction $\vec{\omega}_o$. The figure 2.2a above relates the BRDF arguments visually.

The integrand over the hemisphere:

$$\rho(x, \vec{\omega}_o) = a(x, \vec{\omega}_o) = \int\limits_{\vec{\omega}_i \in H(x)} f_r(x, \vec{\omega}_i, \vec{\omega}_o) \cos\theta_i d\vec{\omega}_i, \tag{2.12}$$

is a quantity called *albedo* or *directional hemispherical reflectance* and states how much light is reflected in the direction $\vec{\omega}_o$ given uniform unit incoming radiance over the hemisphere. Adhering to the law of conservation of energy, physically correct BRDFs have albedo in the range $[0, 1]$, where zero and one represent materials that absorb and reflect all light respectively. All physically correct BRDFs also obey the Hemholtz law of reciprocity:

$$f_r(x, \vec{\omega}_i, \vec{\omega}_o) = f_r(x, \vec{\omega}_o, \vec{\omega}_i), \tag{2.13}$$

meaning they are invariant to swapping the incident and the outgoing direction. Additionally, the range of a BRDF is the set of all non-negative values. For example, the BRDF for a perfect mirror would yield infinity in the principal reflection direction and zero everywhere else.

Under certain assumptions, dimensionality of a BRDF can be reduced further. Materials whose BRDFs are invariant to rotation around normal at the scattering point are called isotropic. A three dimensional BRDF:

$$f_r(x, \vec{\omega}_i, \vec{\omega}_o) = f_r(x, \theta_i, \theta_o, (\varphi_i - \varphi_o)), \tag{2.14}$$

is sufficient for isotropic materials. Materials like human hair or polished aluminum which do not exhibit this invariance are called anisotropic. Finally, diffuse (i.e. Lambertian) materials reflect light into all directions equally. Their BRDFs are invariant with respect to both the incident and outgoing direction, resulting in a constant function:

$$f_r(x, \vec{\omega}_i, \vec{\omega}_o) = \frac{\rho_d}{\pi}, \tag{2.15}$$

where ρ_d denotes the diffuse reflectance property of the material. Facades, walls and paper are examples of close-to-ideal diffuse reflectors.

Historically, most BRDFs employed in real-time rendering have been of the empirical origin. *Empirical BRDFs* are arbitrary functions without any grounding in the underlying physics of light transport. They are motivated by the need for fast evaluation and tend to model some small subset of materials reasonably well. The Blinn-Phong BRDF [Bli77] is perhaps the most well known empirical BRDF used in real-time rendering. It is defined as:

$$f_r(x, \vec{\omega}_i, \vec{\omega}_o) = \rho_d + \rho_s \frac{\cos^s \theta_h}{\cos \theta_i}, \tag{2.16}$$

where θ_h is the angle between the incident direction and the normal at x, $\vec{h} = (\vec{\omega}_i + \vec{\omega}_o)/2$ is the half-vector and s is glossiness. It combines two elementary interactions: diffuse reflections via the ρ_d element and glossy reflections via the cosine lobe part (figure 2.2b). It models plastic materials reasonably well.

In recent years, as the power of consumer hardware increased, there has been a strong shift towards more physically motivated BRDFs even in real-time rendering. In contrast to empirical BRDFs, physically motivated BRDFs obey energy conservation and Hemholtz reciprocity laws. While the model for diffuse reflections usually stays the same (Lambertian), glossy reflections are modeled using *micro-facet theory*. In micro-facet theory, surfaces are composed of many tiny micro-facets, each of which only reflects light in a single direction according to its normal. Micro-facet BRDFs typically have following form:

$$f_r(x, \vec{\omega}_i, \vec{\omega}_o) = \frac{F(\vec{\omega}_i, \vec{h}) G(\vec{\omega}_i, \vec{v}, \vec{h}) D(\vec{h})}{4(\vec{n} \cdot \vec{\omega}_i)(\vec{n} \cdot \vec{v})}, \tag{2.17}$$

where F models the Fresnel effect, G models the visibility and D models the directional distribution of micro-facets. Physical fidelity and evaluation complexity of the BRDF depends on particular choice of these terms. Working with physically motivated BRDFs carries several benefits: control parameters are more intuitive, the results are more predictable with less surprising edge cases and the overall visual quality is higher in the common case. An example of a physically motivated BRDF currently in use in production real-time rendering is the Disney principled BRDF [BS12].

2.4 Computing illumination

To compute global illumination in a scene, it is instructive to first start with local reflections. Let x be a scene surface point and $\vec{\omega}_o$ an arbitrary outgoing direction. The total radiance reflected at x in $\vec{\omega}_o$ can be expressed using the reflectance equation:

$$L_o(x, \vec{\omega}_o) = L_e(x, \vec{\omega}_o) + \int\limits_{\vec{\omega}_i \in \Omega} L_i(x, \vec{\omega}_i) f_r(x, \vec{\omega}_i, \vec{\omega}_o) \cos \theta_i d\vec{\omega}_i, \tag{2.18}$$

where $L_i(x, \vec{\omega}_i)$ is the radiance incoming at x from the incident direction $\vec{\omega}_i$. The total radiance reflected in $\vec{\omega}_o$ is thus equal to the sum of radiance emitted in $\vec{\omega}_o$ and radiance reflected from all possible incident directions in $\vec{\omega}_o$.

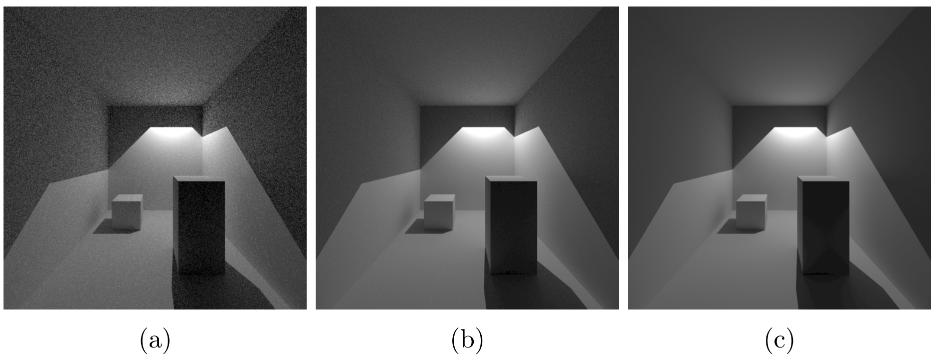


Figure 2.3: Image obtained via path-tracing using (a) 25, (b) 450 and (c) 7000 samples per pixel.

2.4.1 The rendering equation

Because radiance is constant along the ray $(x, \vec{\omega}_i)$, it follows, that for some point y on the ray

$$L_i(x, \vec{\omega}_i) = L_o(y, -\vec{\omega}_i), \tag{2.19}$$

must hold. The point y is intuitively the nearest intersection of the ray $(x, \vec{\omega}_i)$ with the scene. Let $r(x, \vec{\omega})$ be a function returning such intersection. Assuming that cases where no intersection exists are handled smoothly, the equation 2.18 can be rewritten as follows:

$$L_o(x, \vec{\omega}_o) = L_e(x, \vec{\omega}_o) + \int\limits_{\vec{\omega}_i \in \Omega} L_o(r(x, \vec{\omega}_i), -\vec{\omega}_i) f_r(x, \vec{\omega}_i, \vec{\omega}_o) \cos\theta_i d\vec{\omega}_i. \tag{2.20}$$

All instances of the incoming radiance have been eliminated, expressed recursively as the outgoing radiance in the opposite direction at some other scene point.

Rewriting the equation 2.20 one last time without radiance subscripts yields the *rendering equation* [Kaj86]:

$$L(x, \vec{\omega}_o) = L_e(x, \vec{\omega}_o) + \int\limits_{\vec{\omega}_i \in \Omega} L(r(x, \vec{\omega}_i), -\vec{\omega}_i) f_r(x, \vec{\omega}_i, \vec{\omega}_o) \cos\theta_i d\vec{\omega}_i. \tag{2.21}$$

It describes a stable state of energy distribution within the scene. It states, that the illumination of a given scene point, as observable from an arbitrary position, depends on the scene as a whole. Algorithms that try to find a stable energy distribution within a scene with respect to the rendering equation are called global illumination algorithms.

2.4.2 Evaluating the rendering equation

Given a virtual camera, synthesizing an image amounts to computing $L(x, \vec{\omega}_{xToCamera})$ for all scene points x visible from the camera according to the equation 2.21. However, the presence of the recursive integral makes direct evaluation infeasible for all but the most trivial cases.

Path-tracing computes an unbiased estimate of the solution using the Monte Carlo estimator:

$$L(x,\vec{\omega}_o) = L_e(x,\vec{\omega}_o) + \frac{1}{N}\sum_{n=1}^{N}\frac{L(r(x,\vec{\Psi}_n),-\vec{\Psi}_n)f_r(x,\vec{\Psi}_n,\vec{\omega}_o)\cos\theta_i}{p(\vec{\Psi}_n)}, \tag{2.22}$$

where N is the number of trials (samples) and $\vec{\Psi}_n$ is the incident direction vector generated from some hemispherical probability distribution function p. Russian roulette, where the albedo is taken as the absorption probability, is used to terminate the recursion. The solution may be inaccurate due to variance, which can be observed in resulting image as a high-frequency noise. To reduce the variance and thus obtain visually acceptable images, the number of trials must be very high. The figure 2.3 shows the impact the number of samples has on the image quality in a simple scene. Path-tracing is a fundamental algorithm, an interested reader should consult [PH10] for a thorough introduction.

Traditional, rasterization-based, real-time rendering focuses on evaluation of the *direct illumination* part of the rendering equation. Direct illumination refers to only the first bounce of light — only the first expansion of the integral. To signify this, recursion in the equation 2.21 is eliminated:

$$L(x,\vec{\omega}_o) = L_e(x,\vec{\omega}_o) + \int\limits_{\vec{\omega}_i\in\Omega} L_e(r(x,\vec{\omega}_i),-\vec{\omega}_i)f_r(x,\vec{\omega}_i,\vec{\omega}_o)\cos\theta_i d\vec{\omega}_i. \tag{2.23}$$

Additionally, because of the usually small number of constrained dynamic light sources employed, it makes more sense to express the integral as a sum over contributions from individual light sources:

$$L(x,\vec{\omega}_o) = L_e(x,\vec{\omega}_o) + \sum_{s\in S} V_s(x)L_s(x)f_r(x,\vec{\omega}_i,\vec{\omega}_o)\cos\theta_i, \tag{2.24}$$

where s is a light source from the set of all light sources S, $V_s(x)$ is the visibility function for s and $L_s(x)$ is the radiance emitted by s in the direction of x. While direct illumination can be computed efficiently, it only reveals surfaces that are lit directly or have emissive properties. Some approximation of the omitted indirect illumination component is still necessary to prevent regions of a scene from appearing incorrectly black in the synthesized image. The following chapter 3 surveys concrete indirect illumination approximation methods employed in real-time rendering.

Chapter 3

Real-time indirect illumination

Due to a limited time budget, indirect illumination in real-time rendering is typically not computed very accurately. Believability is often prized higher than correctness, especially if it comes with lower performance impact. Although many techniques approximating indirect illumination in real-time have been developed over the years, there is no silver bullet. Selection of an appropriate method is a compromise between performance impact, visual fidelity and the level of scene dynamism it must support. This chapter presents a non-exhaustive overview of real-time indirect illumination techniques.

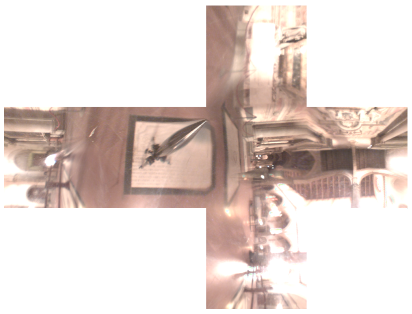
3.1 Ambient light

Ambient light is the crudest approximation of indirect illumination. Light that was scattered many times is modeled using a scene-wide ambient light source of fixed a intensity I_a. When computing illumination, each point is additionally brightened by $\rho_d I_a$, where ρ_d is the diffuse reflectance at the point. The result is flat, uniform, direction-less illumination across a surface. Ambient light is cheap to compute, trivial to implement but very lacking in fidelity.

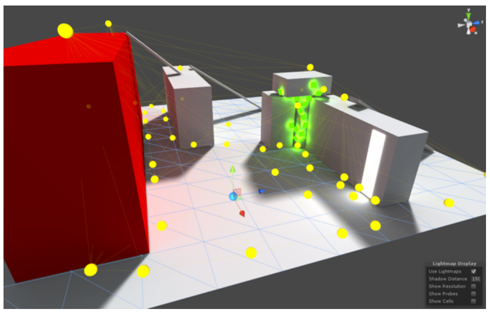
3.2 Light maps

Indirect illumination in scenes with few moving parts is often best precomputed. Before the application runs, an arbitrarily slow algorithm such as radiosity or photon-mapping is used to compute the view-independent part of indirect illumination for all surfaces in a scene. The solution is then stored into textures called *light maps.* Applying the stored illumination at runtime amounts to fetching a value from a texture and multiplying it with the diffuse reflectance property of the surface material.

The biggest advantage offered by light maps is the high level of visual fidelity. Only the objects that were present during the precomputation stage may be lit using light maps and only as long as they remain static. Since the light map data cannot be applied to moving objects, dynamic scene elements often stand out in an unpleasant way because of the discontinuity in illumination. Long computation times also mean slow iteration when modifying scene lighting parameters.



(a)



(b)

Figure 3.1: Light probes: (a) A real world light probe arranged in a cube map. (b) Positioning light probes (yellow spheres) in a scene for precomputation in the Unity 5 game engine. Images courtesy of Paul Devebec and Unity Technologies.

3.3 Light probes

One major disadvantage of the light mapping solution from the section 3.2 is that it does not work with moving objects. *Light probes*, also known as ambient cubemaps, attempt to rectify this shortcoming. A light probe stores incoming indirect illumination for each direction (figure 3.1a). When computing illumination at a point, the light probe is simply sampled in the direction of the surface normal.

To better capture detail, in the precomputation phase, light probes can be captured in strategic locations throughout the scene to enclose the areas reachable by dynamic objects as shown in the figure 3.1a. To compute indirect illumination at a point, four light probes forming an enclosing tetrahedron are selected. Each probe is sampled then using the approach described earlier and the final value computed using tetrahedral interpolation via barycentric coordinates.

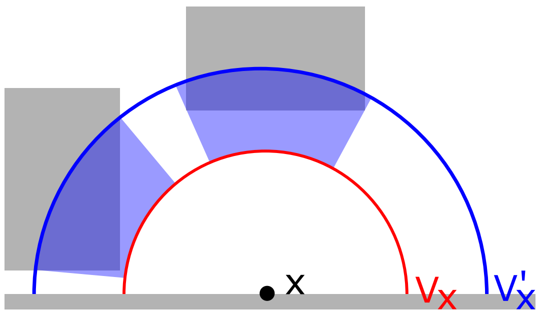
3.4 Ambient occlusion

Ambient occlusion (AO) refers to how exposed a point is to ambient light. The amount of ambient occlusion at a point x is an integral over the upper hemisphere:

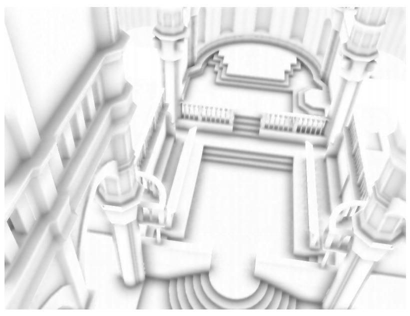
$$\mathrm{AO}(x) = \frac{1}{\pi} \int\limits_{\vec{\omega} \in \Omega} \mathrm{V_x}(\vec{\omega}) \cos\theta d\vec{\omega}, \tag{3.1}$$

where $\mathrm{V_x}$ is the visibility function at x and θ is the angle between the normal at x and $\vec{\omega}$. Once $\mathrm{AO}(x)$ is available, the accessibility factor $\alpha = 1 - \mathrm{AO}(x)$ can be used to modulate indirect illumination received at x. Visualization of visibility factors is depicted in the figure 3.2b.

One approach to evaluating the integral in equation 3.1 is Monte Carlo ray-casting. Rays are shot over the hemisphere at x; intersections closer than a selected visibility function radius r signify occluders. It follows that the resulting AO value is dependent on the choice of r as illustrated in the figure 3.2a.

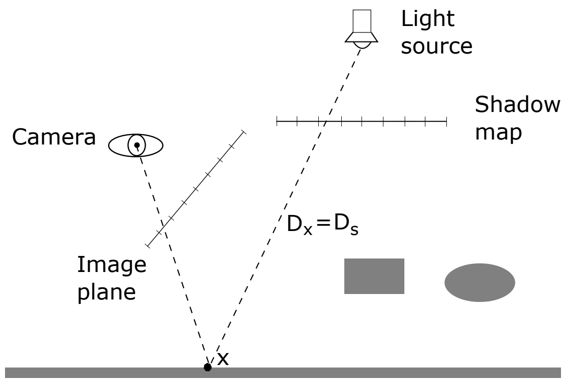


(a)

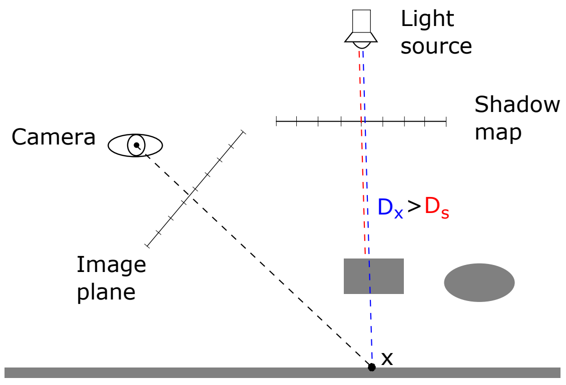


(b)

Figure 3.2: Ambient occlusion: (a) Impact of the visibility function radius on computed AO factor. Of the two visibility functions V_x and V'_x only the latter reports any occluders. (b) Visibility factors for scene surface obtained using SSAO in real-time; image courtesy of [BS08].



(a)



(b)

Figure 3.3: Shadow mapping: (a) Distance to light equals the value in shadow map, x is lit. (b) Distance to light is greater than the value in shadow map, x is in shadow.

In real-time rendering, ambient occlusion is either precomputed in an approach similar to light mapping or computed at runtime. *Screen Space Ambient Occlusion* (SSAO) [BS08] is an algorithm that leverages data stored in the depth buffer to approximate per pixel AO every frame. The SSAO and its variations are commonly deployed in real-time rendering applications thanks to their ability to seamlessly handle dynamic objects, independence on scene complexity and acceptable performance impact.

3.5 Shadow maps

Shadow mapping [Wil78] is a technique for adding hard shadows to spot and directional lights. A *shadow map* is an image of the scene from the point of view of the light. Each shadowmap texel stores the distance to the nearest surface along its corresponding view ray. To determine whether a point x is lit or in shadow, the distance D_s stored in the shadow map texel corresponding to the view ray passing through x is first retrieved. The distance

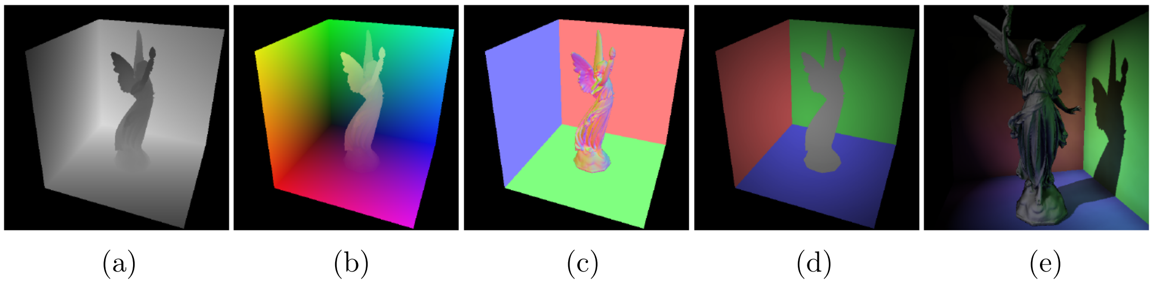


Figure 3.4: Reflective shadow maps: (a) depth, (b) world space position, (c) normal, (d) reflected flux, (e) resulting image with approximate indirect illumination. Image courtesy of [DS05].

between x and the light source D_x is then computed. If D_x is larger than D_s it means that some surface along the same view ray is closer the to light source and x is in shadow (figure 3.3b); otherwise, x is lit (figure 3.3a).

Although the basic algorithm is simple, production grade implementations must address a number of problems stemming from the limited precision of stored depth values, limited resolution of the shadowmap and other issues. There are many extensions improving on some aspect of the original algorithm. For example *cascaded shadow maps* [Dim07] use multiple shadow maps to provide better shadow resolution in areas near the camera; *percentage closer filtering* [RSC87] can mitigate aliasing artifacts along shadow edges; and *percentage closer soft shadows* [Fer05] can believably approximate soft shadows—to name just a few.

3.6 Reflective shadow maps

Reflective shadow maps, an algorithm introduced by Dachsbacher et al. [DS05], uses an extended shadow map structure to simulate a single bounce of indirect illumination. A *reflective shadow map* (RSM) is a set of textures rendered from the light's point of view storing depth, position, normal and reflected flux (figure 3.4a-d). Each reflective shadow map texel (depth, normal, position and flux) represents a hemispherical *virtual point light* (VPL). When computing illumination for a point, the first bounce of indirect illumination is approximated by illuminating the point with a randomly selected subset of VPLs (figure 3.4e). The position of the shaded point is projected to the reflective shadow map and VPLs are chosen from texels within its neighborhood. This does not guarantee that the point and the VPLs are also close to each other within the scene, but the selection doesn't need to be very accurate. The point-VPL visibility is ignored.

3.7 Instant radiosity

Instant radiosity [Kel97] is a technique designed to compute diffuse indirect illumination at interactive to real time speeds. It operates by generating a number of VPLs throughout the scene and using them to approximate indirect illumination.

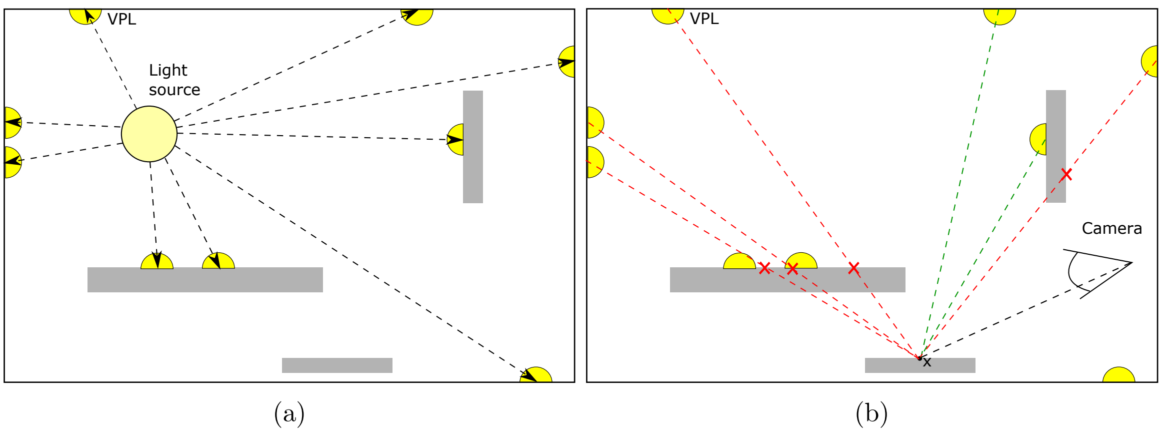


Figure 3.5: Instant radiosity stages: (a) VPLs are created by shooting light particles from light sources (no bounces are shown). (b) Diffuse indirect illumination at x is estimated by evaluating contributions from VPLs.

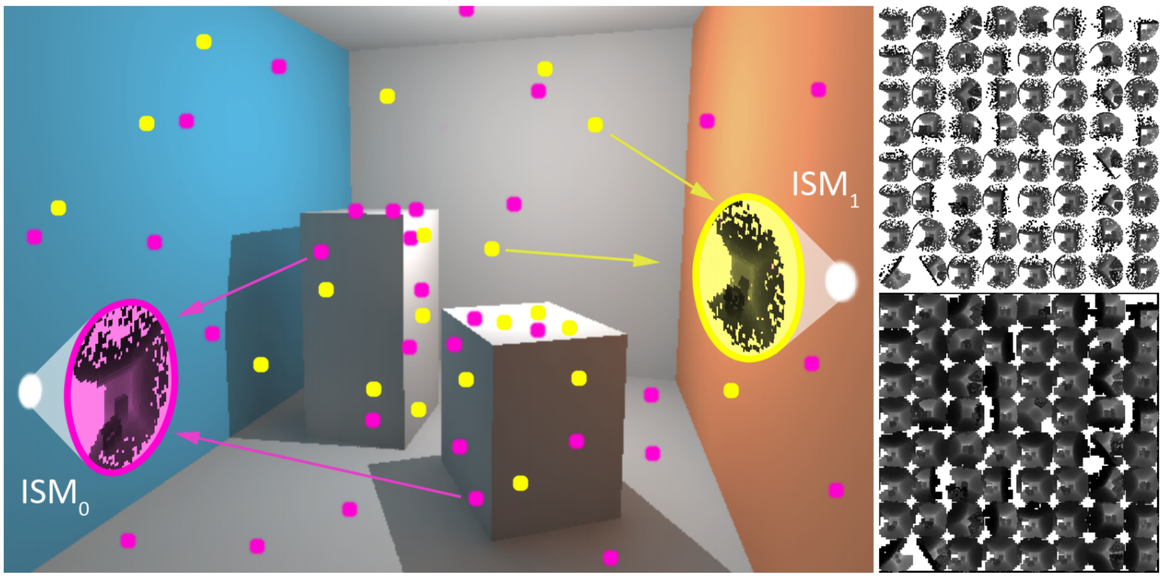


Figure 3.6: Instant radiosity via imperfect shadow maps: A scene point cloud is distributed among the set of VPLs and rendered into an imperfect shadow map atlas (top right). A pull-push algorithm is employed to fill holes present in the shadow maps due to the coarse point representation. Image courtesy of [RGK+08].

3.7.1 VPL generation

In the first stage, VPLs are distributed throughout the scene using a quasi random walk. Initially, each light source is sampled for starting position and direction of light particles to be shot. The shooting itself is done via ray-casting. VPLs are generated at ray-scene (particle-scene) intersections and the corresponding light particles are either absorbed or continue to bounce. The absorption probability, as well as the number of particles to shoot, is determined using the mean scene reflectivity. The figure 3.5a illustrates this process.

3.7.2 Indirect illumination accumulation

In the second stage, diffuse indirect illumination for each visible point is computed by summing up illumination from generated VPLs; shadows maps are used to resolve visibility. The process is illustrated in the figure 3.5b.

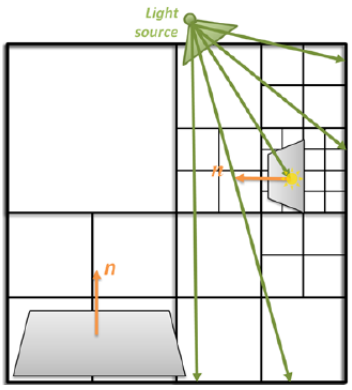
3.7.3 Achieving real-time performance

To achieve real-time performance, the original instant radiosity algorithm needs to be adjusted in several places. Instead of using a quasi-random walk to distribute the VPLs, a reflective shadow map is rendered for each light source. VPLs are then sampled from its texels either randomly or using some heuristic. To resolve visibility, the original algorithm renders a full shadow map for each virtual light. Given the amount of lights needed to get stable results, this approach is not viable in real-time. Instead, the *imperfect shadow maps* [RGK+08] technique can be employed. A low resolution parabolic shadow map is created for all VPLs at once by rendering a scene point cloud. Points of the cloud are distributed among individual shadow maps, each of which receives only a subset of points (figure 3.6). As a result, the generation is fast, but resulting shadow maps contain holes. The holes are filled with the help of a pull-push algorithm [MKC07]. When computing indirect illumination, only a small subset of VPLs is evaluated for each point due to performance constraints. Finally, geometry aware blur filter is applied to the indirect illumination image to reduce noise.

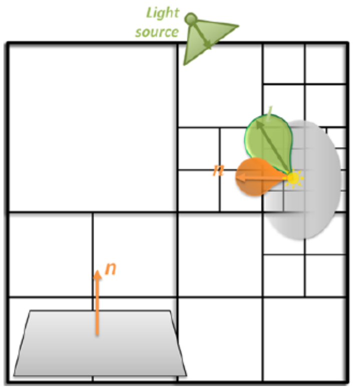
Instant radiosity with imperfect shadow maps can be used to approximate the first bounce of diffuse indirect illumination at real-time speeds in dynamic scenes. Its downside is, that the quality of the resulting images is very sensitive to algorithm parameters. Additionally, it is not free from artifacts such as light leaking through walls due to the approximate nature of information stored in imperfect shadow maps.

3.8 Voxel cone tracing

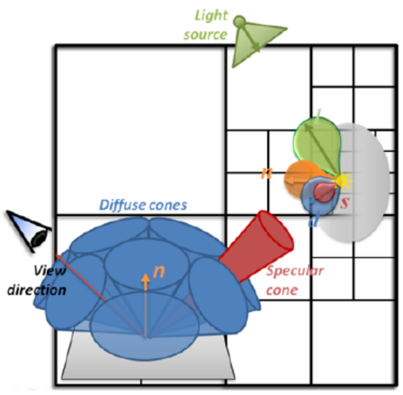
Voxel cone tracing [CNS+11] is a voxel based algorithm aiming to approximate diffuse and glossy indirect illumination at real-time speeds. The algorithm works with a voxel representation of the scene stored in an octree. Direct illumination information is injected into the octree with the help of a structure similar to a reflective shadow map. To compute indirect illumination at a point, cones are traced in several directions to sample light intensity values stored in the octree.



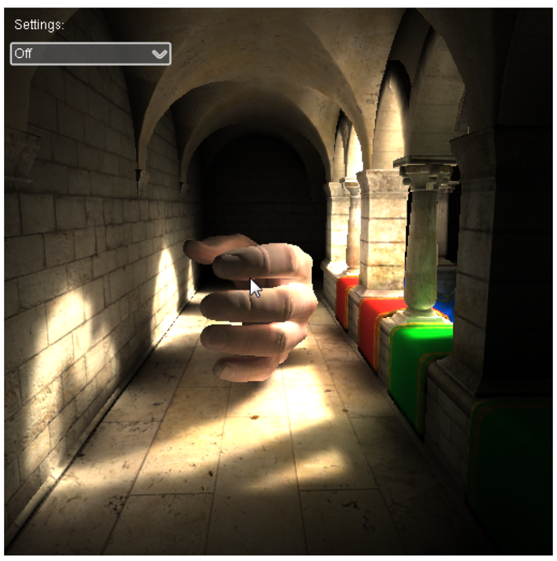
(a)



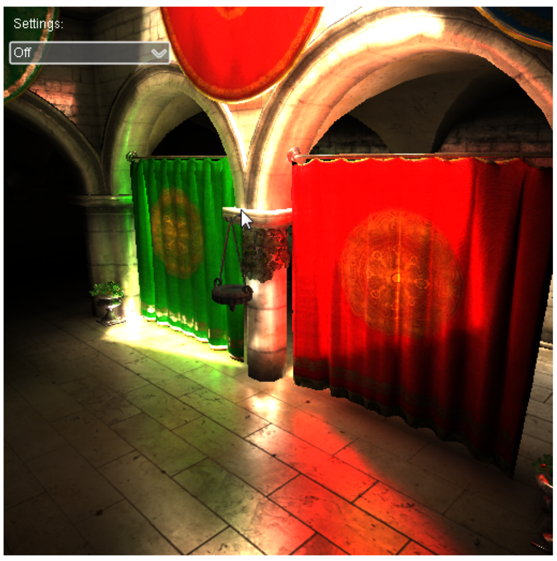
(b)



(c)



(d)



(e)

Figure 3.7: Voxel cone tracing: (a) Injecting light into the octree. (b) Irradiance mipmapping inside the octree. (c) Collecting diffuse and glossy indirect illumination via cone tracing. (d) Diffuse indirect illumination. (e) Glossy indirect illumination. Images courtesy of [CNS$^+$11].

3.8.1 Scene voxelization

To create the scene voxel octree representation, meshes are rasterized three times, once along each main axis. Each shader fragment thread generated by rasterization subdivides the octree top to bottom as necessary and writes fragment data into a leaf. Surface normal is represented using Isotropic Gaussian lobes while reflectance is stored as a scalar per color channel.

Once all meshes are voxelized, data stored in octree leaves is mip-mapped into inner nodes. Static scene elements need to undergo the voxelization process only once, dynamic elements are re-voxelized every frame to account for the change of position or material properties. Octree data for dynamic elements is stored at the end of the structure so that it can be updated efficiently after each re-voxelization.

3.8.2 Capturing direct illumination

Direct illumination information is captured using *light-view maps.* A light-view map is created for every light source in much the same way as one would create a reflective shadow map (section 3.6). The light-view map information is then injected into octree leaves using a fragment or compute shader. Incoming light direction is represented using Gaussian lobes; incoming light intensity as an rgb three-tuple (figure 3.7a). Since multiple texels may land in the same voxel (octree leaf), atomic operations are used to maintain consistency.

Once direct illumination from all lights has been injected into leaves, the data is filtered into inner nodes to create a mip-map hierarchy (figure 3.7b). Direct illumination information needs to be re-injected after every change to light sources or to the structure of the scene octree.

3.8.3 Computing indirect illumination

Indirect illumination at a point is collected by approximate cone tracing. Depending on the material, several cones covering the entire hemisphere are traced. Diffuse reflective properties are modeled by wide cones spaced across the hemisphere; glossiness, on the other hand, is represented by a long tight cone in the reflection direction.

An arbitrary BRDF can be approximated using the right combination of cones. Illumination from a single cone is computed by marching along the principal cone direction ray and aggregating contributions from individual steps using the classical emission-absorbtion model [EHK+04].

At each step, the octree level (node) to sample is selected depending on the cone span (the larger the span the higher the level). Sampled surface normal and incoming light direction Gaussian lobes are convolved with the a Gaussian lobe representing the cone span at the current step. Resulting lobe is evaluated in the direction towards the surface point, yielding the fraction of intensity reflected in that direction. Multiplying this factor with the sampled node reflectance and incoming light intensity gives the final radiant intensity reflected towards the surface point.

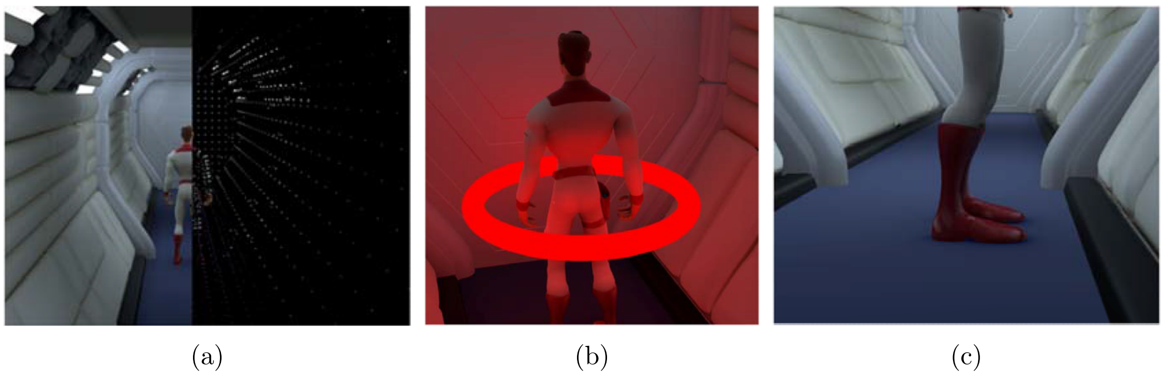


Figure 3.8: Point-based global illumination: (a) Scene and associated point cloud. (b) Emissive materials as light sources. (c) Per pixel GI resolve produces contact shadows. Images courtesy of [KFC+10].

3.8.4 Performance and quality

Voxel cone tracing can approximate both diffuse and glossy indirect illumination. It can deal with dynamic scenes by re-voxelizing elements and re-injecting light on the fly. Majority of the scene should still remain static to achieve real-time speeds.

While the algorithm yields smooth a believable results, they are not free from artifacts due to the discretization inherent in voxel based approaches, light leaking through thin objects being the most common one. Examples results for diffuse and glossy materials are shown in the figure 3.7d and 3.7e.

Performance wise, the algorithm achieves near real-time speeds on high end hardware. The main disadvantage of the octree approach is, that current rendering hardware is not particularly suited to the heavy pointer chasing inherent in navigating the octree throughout the voxelization, injection, filtering and cone tracing stages. A cascade of several three dimensional textures moving with the camera could be used in place of the octree to alleviate this problem, but larger voxels would lead to more pronounced light leaking.

3.9 Point-based global illumination

Point-based global illumination (PBGI) [KFC+10] is a real-time technique for computing both diffuse and glossy indirect illumination inspired by the Point-based approximate color bleeding [Chr08] algorithm used in the film industry.

(PBGI) operates as follows: First, a cloud of sample points is obtained by rendering the scene into a vertex stream (figure 3.8a). The sample cloud does not need to be especially dense, so using high resolution meshes is impractical. Instead, it is better to either use tessellation control meshes or level-of-detail meshes, depending on the overall application architecture. Generated samples are cached and only regenerated if necessary. With world position, normal, tangent and material information available for each sample a set of equations is solved:

$$\begin{aligned} L_{out}[i] &= L_{in}[i]\rho \\ A[i] &= -L_{in}[i]S[i] \\ L_{in}[i] &= \sum_{j \in P, j \neq i} L_{out}[j]F[j,i] + A[j]F_a[j,i], \end{aligned}$$

where

$L_{out}[i]$	:	light emitted forward from sample i
$L_{in}[i]$	:	light received by sample i
$A[i]$	:	light emitted backward by sample i (antiradiance from [DSDD07])
$S[i]$	:	shadow factor for sample i
ρ	:	surface reflectance
P	:	set of all point samples
$F[i,j]$	:	disk to disk radiance transfer forward form factor for samples i-j.
$F_a[i,j]$	:	disk to disk radiance transfer backward form factor for samples i-j.

In an approach similar to [McT04], three irradiance values aiming in directions spaced around the sample normal are computed. Convenient extrapolation of irradiance values for any direction is needed to facilitate normal mapping and is also used to simulate glossy reflections. Once the per sample irradiance values are available, they are upsampled (for level-of-detail meshes) or tessellated (for subdivision surfaces) into a vertex stream and the scene is rendered. At a point, diffuse indirect illumination is computed by converting tangent space normal into coefficients for the three precomputed irradiance values. Glossy indirect illumination is approximated by treating the three irradiance values as intensities of a directional lights oriented around the normal and summing up their contributions.

The PBGI algorithm computes coarse approximation of diffuse and specular indirect illumination. The solution it provides suffers mainly from light leaking related artifacts and tone shifts as a result of using anti-radiance to solve visibility instead of an explicit method. One way to ameliorate this is by grouping the samples into clusters and manually specifying inter-cluster dependency factors.

3.10 Cascaded light propagation volumes

Cascaded light propagation volumes [KD10] is a grid based, fluid simulation inspired technique for approximating diffuse and glossy indirect illumination in fully dynamic scenes at real-time speeds. The algorithm works with two grids: a geometry volume and a light volume. During initialization, reflective shadow maps are used to fill light volume cells with virtual point lights and geometry volume cells with blocker information. Light is then iteratively propagated through the light volume; geometry volume data may optionally be used to occlude light and generate additional light bounces. To retrieve indirect illumination at a point, the light volume is evaluated for a given position and surface normal.

3.10.1 Light and geometry volumes

Light volume is an axis aligned Cartesian grid. Located at the center of each light volume cell is a virtual point light with the directional intensity distribution $\mathrm{I}(\vec{\omega})$ encoded in spherical harmonics (SH). *Geometry volume* is a also a Cartesian grid; its resolution and cell size are identical to those of the light volume. Geometry volume cells may contain blockers, which occlude light. The amount and orientation of blockers inside a cell is represented by the directional distribution $\mathrm{B}(\vec{\omega})$ encoded in SH. Each geometry volume cell also stores diffuse reflectance of blockers inside the cell. The reflectance value is used for simulation of additional light scattering events; it is also useful for visualization of the geometry volume contents.

Geometry volume position is shifted exactly half a cell size relative to the light volume position. This way, light volume cell corners line up with geometry volume cell centers, which will later enable efficient retrieval of data stored inside the geometry volume.

3.10.2 Light and blocker injection

The purpose of this step is to fill light volume with initial light intensity distribution and store volumetric representation of the scene inside the geometry volume; both are needed for the next step.

Every frame, light and geometry volumes are cleared. Reflective shadowmap (RSM) is rendered for each light source. RSM texels represent surface elements (surfels), each with their own depth (D_p), normal ($\vec{n}_p$), diffuse reflectance (ρ_p) and diffuse reflected intensity (Φ_p). Surfels are used to populate both the light and the geometry volume.

To inject a surfel to light or geometry volume its position in the volume P_p is first reconstructed from D_p and light camera parameters used to render RSM. Surfel weight factor

$$W_p = A_s A_c^{-1} \tag{3.2}$$

is also computed from surfel area A_s and volume cell face area A_c. W_p is used to scale surfel contribution to volume data in order to make the injection process independent of the volume cell size and RSM resolution.

When injecting a surfel into the light volume, its position P_p is first shifted slightly along the surfel normal and surfel to-light direction to mitigate self-lighting artifacts. Next, the light volume destination cell is determined from the shifted P_p. A virtual point light corresponding to the surfel is then created with directional intensity distribution

$$\mathrm{I_p}(\vec{\omega}) = W_p \Phi_p \langle \vec{n}_p \cdot \vec{\omega} \rangle_+ . \tag{3.3}$$

$\mathrm{I_p}(\vec{\omega})$ is projected to low order spherical harmonics and stored in the destination cell. Contributions from all surfels falling into the same light volume cell are summed.

Geometry volume population is done similarly to light volume population: Destination volume cell is computed from surfel position P_p and surfel blocker distribution function

$$\mathrm{B_p}(\vec{\omega}) = W_p \langle \vec{n}_s \cdot \vec{\omega} \rangle_+ \tag{3.4}$$

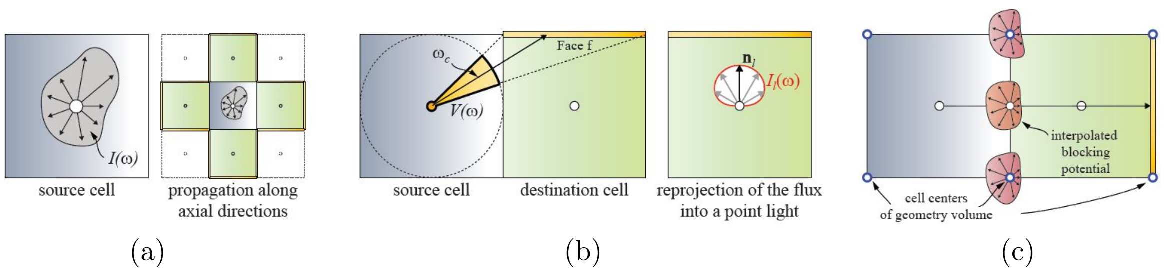

I(ω)
source cell
propagation along axial directions
ω_c
Face f
V(ω)
source cell
destination cell
n_l
I_l(ω)
reprojection of the flux into a point light
interpolated blocking potential
cell centers of geometry volume
(a)
(b)
(c)

Figure 3.9: Light propagation 2D view: (a) Light intensity from the source cell is propagated to its neighbors along main axial directions. (b) Propagation is done by computing the flux onto a destination cell face reprojecting it to a point light and accumulating the result. (c) Propagated light is occluded using the volumetric scene representation of the scene in the geometry volume. Thanks to the light and geometry volume positions being shifted with respect to each other, geometry volume data can be retrieved efficiently using hardware texture filtering. Image courtesy of [KD10].

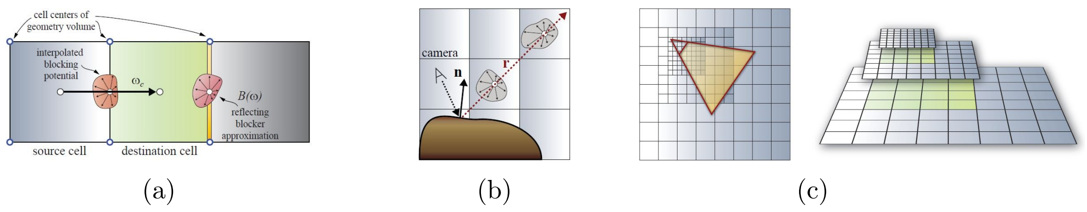

cell centers of geometry volume
interpolated blocking potential
ω_c
B(ω)
reflecting blocker approximation
source cell
destination cell
camera
n
r
(a)
(b)
(c)

Figure 3.10: Cascaded Light Propagation Volume features: (a) Generating more light bounces by reflecting light distribution inside the light volume off of the blockers in the geometry volume. (b) Glossy indirect illumination can be approximated by marching the light volume along the reflected view ray. (c) Multiple light propagation volumes are used in a cascade configuration to efficiently cover large areas while maintaining detail close to camera. Images courtesy of [KD10].

is projected to SH. Both ρ_p and $\mathrm{B_p}(\vec{\omega})$ are stored in the destination geometry volume cell. Blocker distribution contributions from surfels falling into the same cell are summed, while ρ_p contributions are aggregated using max operator. Because the same surface may be captured by multiple different RSMs, surfels from each distinct RSM are first injected into a separate intermediate geometry volume. The intermediate geometry volumes are then aggregated into a final geometry volume using the max operator. Note that the surfels injected onto the geometry volume do not need to come only from RSMs. For example when doing deferred shading, surfels from the g-buffer can be injected too.

3.10.3 Light propagation

In this step, light intensity injected into the light volume from RSMs is now propagated to simulate light bouncing through the scene. The approximate volumetric representation if the scene stored in the geometry volume can be used to occlude propagated light and simulate more light bounces.

The propagation is carried out using successive local iteration steps. A light volume serves both as an input and an output of each iteration: iteration output is fed as input to the following iteration. The input of the first iteration is the light volume obtained by RMS injection. The light volume containing the final distribution of indirect illumination corresponds to the sum of outputs from all iterations and the initial injected intensity.

In every iteration, light in each cell is propagated along grid's main axial directions (figure 3.9a). Given a source cell, a neighbor destination cell and a face of the destination cell, a single propagation step is computed as follows (figure 3.9b): First, the flux Φ_f reaching the face from the source cell is approximated as:

$$\Phi_f = \frac{\Delta\omega_f}{4\pi} I_s(\vec{\omega}_{sf}), \tag{3.5}$$

where $\Delta\omega_f$ is the solid angle subtended by the face as seen from the center of the source cell, $I_s(\vec{\omega})$ is directional intensity distribution of light inside the source cell and $\vec{\omega}_{sf}$ is the direction from the center of of the source cell to the center of the face. Essentially, the intensity in the direction $\vec{\omega}_{sf}$ is interpreted as the average intensity over the solid angle. Next, a virtual light is created at the center of the destination cell to model Φ_f. The directional intensity distribution of this new light corresponds to:

$$\mathrm{I_l}(\vec{\omega}) = \frac{\Phi_f}{\pi} \langle \vec{f} \cdot \vec{\omega} \rangle_+, \tag{3.6}$$

where $\vec{f}$ is the direction from the center of the destination cell to the center of the face. Finally, $\mathrm{I_l}(\vec{\omega})$ is projected to SH and stored inside the destination cell. The afore-described elementary propagation step is executed once for every face of every six-neighborhood neighbor of every cell.

The propagation algorithm can be extended by accounting for blocking geometry (figure 3.9c). Let f_{sd} denote the center of the face shared by the source and the destination cell. The blocker distribution at the interface of the source and the destination cell $B_{sd}(\vec{\omega})$ is approximated by sampling the geometry volume at f_{sd}. Because the geometry volume is shifted by half the cell size, this amounts only to averaging values of cells located in vertices of the shared face. During the elementary propagation step, the blocking potential β in the propagation direction is evaluated as $\beta = \mathrm{B_{sd}}(-\vec{\omega}_{sf})$ and Φ_f, the flux reaching the destination face, is attenuated using the factor $1 - \beta$.

More light bounces can also be generated during the propagation (figure 3.10a). For a source-destination cell pair, let $\mathrm{B_r}(\vec{\omega})$ be the blocker distribution and ρ_r the reflectance at the center of the destination face furthest from the source cell. During the elementary propagation step, diffuse illumination of the blocker geometry can be estimated as $\mathrm{B_r}(-\vec{\omega}_{sf})$. This can be used to model the reflected diffuse illumination as a virtual light located at the center of the destination cell with directional intensity distribution:

$$\mathrm{I_l}(\vec{\omega}) = \rho_r \, \mathrm{B_r}(\vec{\omega}) \, \mathrm{B_r}(-\vec{\omega}_{sf}), \tag{3.7}$$

which is then projected to SH and accumulated in the destination cell.

3.10.4 Relighting

To compute diffuse indirect illumination at a point x, spherical harmonics coefficients of the indirect illumination distribution $\mathrm{I_x}(\vec{\omega})$ are first retrieved from the final light volume using trilinear interpolation. $\mathrm{I_x}(\vec{\omega})$ is then evaluated in the negative direction of the surface normal $\vec{n}_x$.

To mitigate self-illumination and light leaking artifacts stemming from the low frequency nature of the approximation a dampening factor can be applied to the value computed in the previous paragraph. The dampening factor is based on directional derivative of the spherical harmonics coefficients. Let c stand for the vector of spherical harmonic coefficients of the intensity distribution and $\nabla_n c$ be the vector of coefficients of the directional gradient. The dampening is proportional to the magnitude of c and the deviation between c and $\nabla_n c$.

Glossy indirect illumination approximation can be obtained by ray marching along the reflection direction (figure 3.10b). The approach is to march along the reflection direction at a point with steps equal to the cell size. At each step, the intensity distribution is retrieved from the light volume and evaluated in the negative reflection direction. The approximate glossy indirect illumination is obtained as the average of step contributions weighted by inverse square distance from the point.

3.10.5 Solution stabilization

The propagation volume may either be placed at fixed position or, especially in larger scenes, move with the camera. In the latter case, the volume is centered at the camera and then translated slightly along the view direction. This results in a scheme, where majority of the volume is always in front of the viewer but some space is reserved in the area behind the camera so that the indirect illumination may come from behind. To avoid flickering, the volume does not move smoothly with the camera but is instead snapped to multiples of the cell size.

While high resolution volumes are required to capture detail, the number of propagation iterations necessary to distribute the indirect illumination throughout the scene rises with the decreasing cell size. Rather than using one massive, high resolution volume, a large area is more efficiently covered with a set of volumes of increasing size organized in a cascade (figure 3.10c). Cascade levels all have the same dimensions and each moves with the camera in the afore-described fashion. All operations except for relighting are done in each cascade level separately. To avoid jarring transitions, indirect illumination coming from areas near a cascade level border is faded out.

Chapter 4

Application design

Having reviewed the theory and surveyed related techniques in previous chapters, it is now time to design the demo application — a rudimentary game engine with focus on the rendering system. This chapter discusses the design of individual application components. Starting with this chapter, `this font` is used to denote programming language classes and methods.

4.1 Interfacing with rendering hardware

Practical real-time rendering applications use dedicated hardware, a graphics processing unit (GPU), to accelerate computation. Initially, GPUs exposed only fixed functionality. A user would tweak a limited set of parameters and pass in set a vertex attributes for rendering. The GPU would then rasterize the primitives, be it triangles, lines or points, and compute resulting color for covered image pixels according to a built in illumination model. As the time progressed, GPUs became increasingly programmable, allowing to replace portions of the previously fixed function pipeline with custom behavior specified in user-written programs called shaders. Finally, besides the programmable functionality of the rasterization pipeline, GPUs nowadays also expose completely decoupled, general purpose compute functionality operating on arbitrary pieces of data.

GPU functionality is accessed through a dedicated application programming interface (API). The API allows client applications to perform common tasks like uploading and downloading data to and from the GPU, setting fixed functionality parameters and dispatching rendering and compute tasks. Two types of GPU APIs exist: The first are graphics APIs, which understand the notion of submitting primitives for rasterization and use specialized abstractions for GPU memory suited to rendering tasks (e.g. textures, vertex attribute buffers). The other are compute APIs, which focus on conveniently exposing general purpose computation facilities. A programming language used to write GPU programs is typically specific to a given API. It is possible to combine certain compute and graphics APIs in a single application and share resources between them at the cost of extra synchronization between the two API drivers. To avoid this performance penalty, newer versions of graphics APIs expose the general compute functionality through their own interfaces.

The landscape of available graphics APIs is varied. On the personal computing class of hardware, OpenGL [Khr15a] is the only cross-platform option. A caveat with OpenGL is

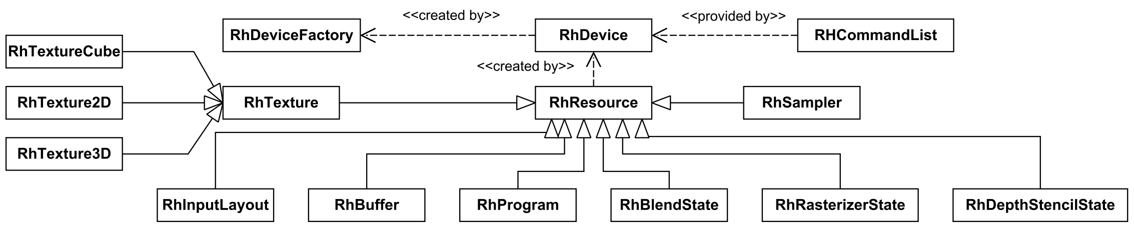


Figure 4.1: Rendering hardware abstraction layer components.

that not all platforms implement the most recent version of the API; moreover, the same code paths may have different performance characteristics depending on the implementation. On Windows operating systems, the proprietary Microsoft DirectX [Mic15] graphics API is available and well supported. On mobile hardware, both Android and iOS systems use OpenGL ES [Khr15b] with newer iOS systems using Metal [App15]. In the video game console world, Xbox systems use a modified version of DirectX while PlayStation exposes its very own set of APIs.

The preceding non-exhaustive overview makes it clear that an application targeting multiple platforms needs a rendering hardware interface layer of its own. Even in the rare single platform scenario, crafting an abstraction over the underlying graphics API is the sensible decision from the software engineering point of view: only the parts of the API actually relevant to the application are exposed and the abstraction layer may provide arbitrary extra functionality such as profiling or validation.

In the demo application, rendering system will not communicate with a specific graphics API directly, instead, in the spirit of the previous paragraph, it will only work with a generic `Rh*` abstraction layer consisting of several interfaces. Concrete implementations of those interfaces will pass the commands to a specific graphics API. `Rh*` abstraction layer components are depicted in the figure 4.1. At the core of the system lies a `RhDevice`, which is used to create GPU memory resources in the form of `RhTextures` and `RhBuffers`; graphics and compute shaders encapsulated by `RhProgram` objects; and graphics pipeline fixed function state descriptors `RhRasterizerState`, `RhBlendState` and `RhDepthStencilState`. The `RhDevice` also provides access to a `RhCommandList` instance, which represents a stream of commands to the GPU. The `RhCommandList` can be used to upload and download GPU data, bind resources, set pipeline states, draw primitives and dispatch compute shaders.

4.2 Rendering

It is the sole responsibility of the rendering subsystem to synthesize the final image from the current scene data. The demo application only uses a simple scene representation composed of primitives and lights. To render a scene, all primitives are drawn and illumination contribution from all lights is iteratively accumulated.

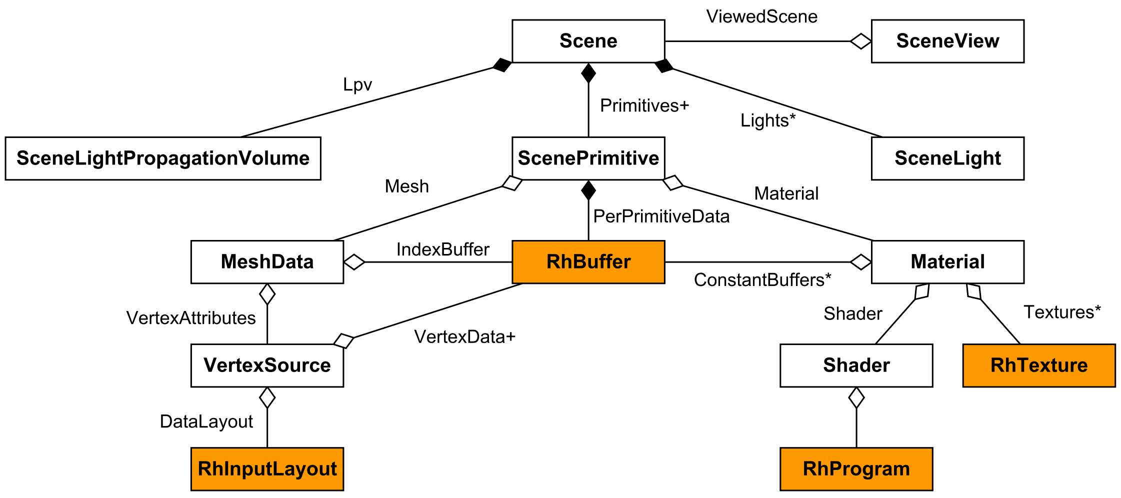


Figure 4.2: Scene representation hierarchy and how it uses `Rh*` components in orange.

4.2.1 Scene representation

The world as seen by the rendering subsystem is encapsulated in a `Scene`. Figure 4.2 depicts relationships of classes used to describe a `Scene`.

A `ScenePrimitive` represents an instance of renderable geometry. It consists of a reference to `MeshData` and `Material` objects and a transformation, which is unique to the instance. A `MeshData` object corresponds to a drawable mesh. It specifies mesh vertex attribute data and their layout in the form of a `VertexSource` and also stores mesh topology, index data and object space bounds. Information on how to shade a primitive is contained inside of a `Material`, which is just a reference to a `RhProgram` and rendering resources, like `RhTexture` and `RhBuffer`, to use with it.

Lights in a `Scene` map to `SceneLight` objects. Three light source types are supported: Directional lights, modeling distant lights with high intensity; point lights — infinitesimally small, omni-directional lights; and spot lights, which are essentially point lights, but only effective within a specified cone or pyramid around a principal direction. All information related to a single light is stored inside a `SceneLight` structure. In addition to its type `SceneLight` also stores the light's transform, shadow configuration info, intensity, falloff and light propagation volume interaction data.

Also included in a `Scene` is a reference to a single `SceneLightPropagationVolume`, which describes parameters of the light propagation volume in the scene, if there is any.

Primitive and light data within a `Scene` is organized in flat arrays, so that caches are well utilized when submitting primitives for rendering. No part of the rendering subsystems is allowed to modify the scene data in any way. Instead, agents from higher level subsystems are responsible for adding, updating and removing primitives and lights through methods of the `Scene` class.

Instructions on how to render an image of a scene are encapsulated in the `SceneView` class. It specifies a `Scene` to render, parameters of a virtual camera and resolution of the

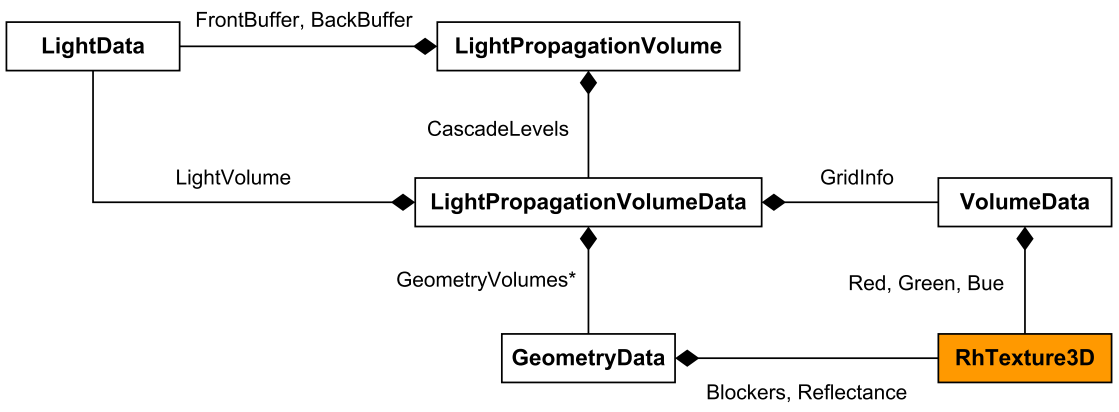


Figure 4.3: Representation of the light propagation volume structures in the application; `Rh*` components in orange.

final image.

4.2.2 Light propagation volume representation

When describing the Cascaded Light Propagation Volumes algorithm in section 3.10, we mentioned that the light and blocker directional distribution functions are stored encoded into spherical harmonics, or more precisely real spherical harmonics.

Real spherical harmonics use normalized Associated Legendre Polynomials to define an orthonormal basis over a sphere, allowing for a memory efficient, approximate representation of directional functions. The family of Associated Legendre Polynomials consists of band functions defined by two integer parameters. An n-th order SH approximation of a function uses n first bands of polynomials to represent the function. The more bands are used the more accurate the approximation but also the higher the number of coefficients required to store the approximation. n^2 real coefficients are needed to store n-th order SH approximation of a function. For more background on SH, their evaluation and usage, interested reader should consult [Slo08] or [Gre03].

In the demo application, second order SH approximation is used. The corresponding four coefficients fit nicely into four channels of a rendering hardware texture. Also, the math associated with evaluating the approximation is computationally inexpensive.

Figure 4.3 illustrates how a light propagation volume cascade is represented inside the application: A `VolumeData` structure defines a Cartesian grid using grid cell size, resolution and position. Inside a `LightData` object, three 3D `RhTextures` store the per-grid-cell the light directional distribution — each color channel is stored in a separate texture. Similarly, a `GeometryData` instance contains two 3D `RhTextures`: one for the blocker directional distribution function and one for diffuse reflectance. `VolumeData`, `LightData` and `GeometryData` together form a `LightPropagationVolumeData` object. While only one `LightData` instance is stored per `LightPropagationVolumeData` instance, multiple `GeometryData` instances may be necessary to correctly aggregate blockers from independent sources. Finally, one to three

`LightPropagationVolumeData`s are encapsulated in a `LightPropagationVolume`, which represents a complete light propagation volume cascade.

4.2.3 Renderer architecture

There are many ways to structure rendering logic. Perhaps the most intuitive approach is to render scene objects one by one. To draw an object, activate the GPU program corresponding to its BRDF, set per object program parameters and finally submit triangle vertex data to the rendering API. On the GPU side, illumination for each rasterized fragment is evaluated and resulting color stored in the corresponding frame buffer pixel. This is called *forward rendering.*

One disadvantage of forward rendering is that, under certain circumstances, it leads to a lot of unnecessary computation. For example, when multiple fragments corresponding to the same pixel are evaluated in the back-to-front order, only the computation for the fragment with position closest to camera is useful (unless multiple fragment contributions are used to composite the final color). Additionally, evaluating contributions from all lights for every fragment can be wasteful because point and spot lights actually affect only a limited sphere or cone shaped area.

In complex scenes with many lights, where performance caveats of forward rendering can prove prohibitive, a deferred approach may work better. In *deferred rendering*, scene objects are first rendered into a geometry buffer (g-buffer). The *g-buffer* is a set of textures, where each texture stores some piece of information necessary to evaluate illumination (normal, reflectance, etc). Once the g-buffer is ready, illumination contribution from lights is accumulated in individual rendering passes. For each light, illumination is only evaluated in the relevant parts of the scene by fetching data from corresponding parts of the g-buffer.

The deferred rendering approach is not without its own shortcomings. First, since the g-buffer data used for illumination information is the same for all pixels, all scene object are forced to use the same BRDF. This limitation can be worked around somewhat by storing a material id in the g-buffer, which decides what BRDF to use. Second, the g-buffer only gives information on on the first surface a given camera ray encounters, which makes implementing effects such as transparency solely through simple deferred shading impossible. Finally, the memory bandwidth of storing and fetching g-buffer data may itself prove to be a bottleneck; high resolution g-buffer will also leave a large memory footprint.

Deferred and forward rendering are just two of the many possible approaches to designing a renderer, some other, slightly more advanced techniques include tile-based deferred rendering [Lau10], deferred lighting [Lau10] and Forward+ [HMY12].

In the demo application, scenes are rendered by implementations of the `ISceneRenderer` interface. `DeferredShadingRenderer` implements `ISceneRenderer` in terms of the deferred approach discussed earlier. To obtain an image of a scene, the renderer is supplied a virtual camera in the form of a `SceneView` and a `RhCommandList`, through which it interfaces with rendering hardware.

In the first step, all `ScenePrimitive`s are rendered into a g-buffer, encapsulated in a `GBuffer` structure. `LightPropagationVolume` is then cleared and injected with blocker information from the g-buffer. Subsequently, scene lights are processed. For each light, a shadow map is generated. Stencil pre-pass is then executed to mark pixels of the g-buffer that are

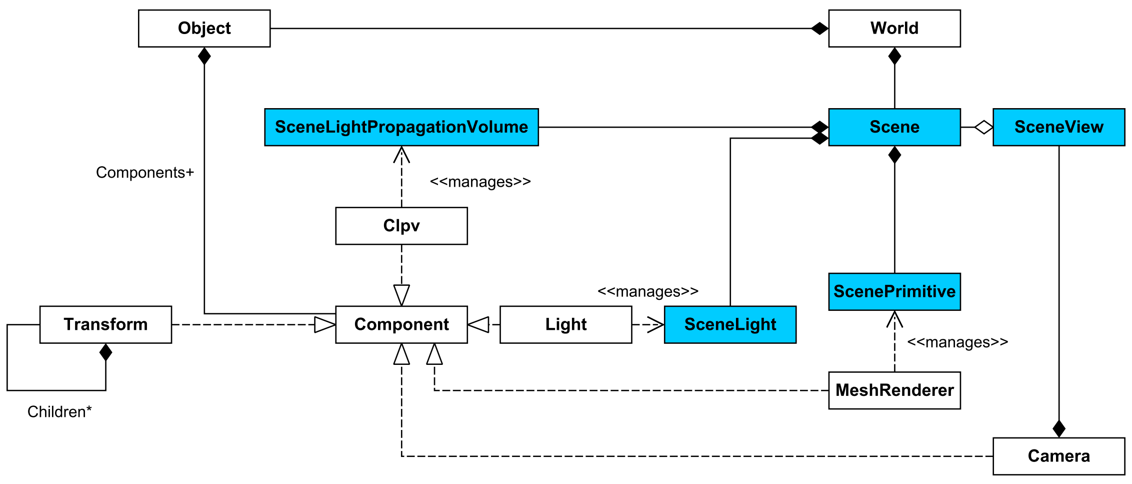


Figure 4.4: Demo application object hierarchy; rendering subsystem components in blue.

affected by the light, after which the illumination contribution is computed for marked pixels; the shadow map generated earlier is used to resolve visibility. Independently, a reflective shadow map is also generated for the light and injected into the `LightPropagationVolume` to provide both direct illumination and blocker information. Once all lights have been processed, several propagation passes are run on the `LightPropagationVolume`. Indirect illumination is accumulated for all g-buffer pixels in a final full-screen pass by sampling the propagated light volume data.

4.3 Object and component model

While a `Scene` is a good representation of the world as seen by the rendering system, it is far too rigid and specialized to be usable on an application-wide level. Object-component hierarchies are a good solution to the problem of integrating multiple orthogonal subsystems such as sound, graphics, animation, physics and so on. The idea is to use an object as a representative for a world entity with a unique identity. Besides identification, objects do not provide any functionality on their own, but they are instead used to group components. Specialized components define the object by plugging into aforementioned subsystems or adding custom functionality. Hierarchical parent-children relationships can be defined either between objects themselves or with the help of components. The particulars on how to realize such system differ but the general object-component approach is commonly seen in game frameworks such as Unity [Tec15] or Unreal [Gam15].

The demo application's take on the object-component pattern is summarized in the figure 4.4. The `World` structure functions as a top level container for `Objects`. Once an `Object` is added to a `World`, its `Components` can be queried and new attached through the interface it exposes. By default, an `Object` only contains the `Transform` component, which specifies position, orientation and scale relative to its parent `Transform`. Hierarchical relationships can be established by setting one `Transform` as a child of another. The `MeshRenderer` and `Light` components serve to interface with the rendering subsystem. It is their responsibility to add, remove and update `ScenePrimitives` and `SceneLights` in the `Scene` attached to the active `World`. Finally, the `Camera` component defines a virtual camera which can be used to render the scene and the `Clpv` component manages a `SceneLightPropagationVolume`.

4.4 Asset management

In this context, asset is used to refer to arbitrary data used by application. Examples include models, sounds or animation tracks. Asset management encompasses loading, caching and release of assets, ideally in a manner that minimizes memory consumption and media storage access.

The demo application utilizes three types of assets: `TextureAsset`, `MaterialAsset` and `MeshAsset`, which are used to store rendering texture, material properties and triangle vertex data respectively. Assets share a common `Asset` interface, through which they are stored in an `AssetManager`, a cache for loaded assets. Assets can be added to the manager manually, but are automatically added when loading models from files through the `LoadModelAsObjectHieararchy` method of `AssetManager`. When loading a model, the actual work of parsing the file and loading associated textures is delegated to dedicated third party libraries. The third party loaded data is then converted to the object-component hierarchy representation used the application.

4.5 Implementation technologies and third party libraries

The C(++) languages are the traditional go to option when looking to do performance intensive tasks close to the metal. The CPU side of the demo application is implemented in C++. In this particular case, the decision is motivated mainly by personal preference and not performance reasons — most intensive computations are likely to take place on the GPU.

The application relies on third party libraries to carry out several tasks. Image loading functionality is handled by the ResIL [Res15] library. ResIL has a simple C interface and support for a wide variety of image file formats. Besides images, the application needs to access model and material definitions. This responsibility is delegated to the Open Asset Import Library (Assimp) [Ass15], a powerful, open source scene file loader. The Assimp can parse almost any 3D file format and present its contents to the client application through a set of common structures. In order to facilitate dynamic adjustment of scene and algorithm parameters, the application relies on the ImGui [Oma15] library to draw simple immediate mode GUIs directly in code. Finally, the OpenGL Mathematics [Chr15] library is used for low level matrix algebra computations.

The demo application comes with a single implementation of the `Rh` rendering hardware abstraction layer. OpenGL was selected as the backing graphics API because of the free cross-platform portability it offers. OpenGL context creation and extension management are not concepts incorporated into the core API because they depend on the actual platform operating system. Fortunately, several third party libraries exist that abstract the platform dependent tasks under a common interface. To query and load OpenGL extensions available on particular hardware, the application leverages the OpenGL Extension Wrangler Library

(GLEW) [GLE15]. OpenGL context creation is handled by the GLFW [GLF15] library, which is also used to access other platform dependent functionality such as window creation, CPU timing and input handling.

Chapter 5

Rendering system implementation

The DeferredShadingRenderer class constitutes the bulk of the rendering subsystem. High level overview of the data flow within the system is depicted in the figure 5.1. Following sections contain a breakdown of individual steps taken to synthesize an image.

5.1 G-buffer generation

In the FillGBuffer method, a g-buffer is generated by rendering all ScenePrimitives into the GBuffer render target collection. The GeometryPass shader is used to output diffuse and specular reflectance, glossiness, normal and depth values per fragment.

5.2 G-buffer data injection

GBuffer data can be used to populate the light propagation volume with blocker information. Because it is inefficient and unnecessary to inject a full resolution GBuffer, diffuse reflectance, normal and depth textures are first downsampled several times. In the demo application, textures are repeatedly downsampled to half their resolution until their larger dimension is less or equal to 128. The data is then injected into the LightPropagationVolume using approach described in the section 5.3.5. GBuffer downsampling and injection is implemented in the LpvInjectGBuffer method; the LpvDownsampleGbuffer shader handles the downsampling process GPU-side.

5.3 Processing lights

Once the GBuffer has been initialized, all SceneLights in the Scene are passed to the ProcessLight method, where the following sequence of steps is executed.

5.3.1 Shadowmap generation

If the light is marked to have a shadow, a shadowmap is generated. First, a projection transform is created for the light. For directional lights, parameters for orthographic projection

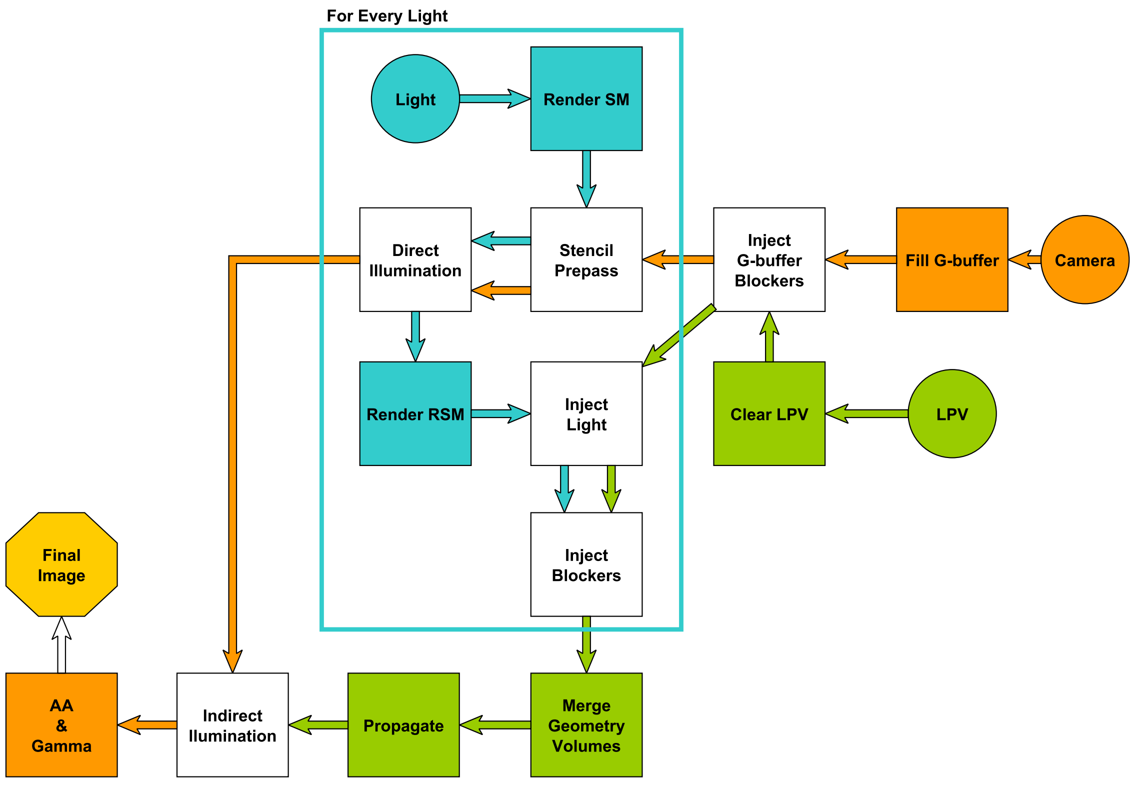


Figure 5.1: Steps taken to generate an image — high level overview.

that tightly fits the scene to the shadowmap viewport are computed from light orientation and scene bounds. For spotlights, perspective projection transform is created from light's cutoff angle. In both cases, light intensity, orientation and scene bounds are used to compute near and far plane locations that tightly fit the scene in order to maximize the depth buffer precision. Once a projection transform for the light has been computed, all `ScenePrimitive`s marked as shadow casters are rendered using the `DepthOnly` shader, which only writes depth.

Point lights are handled by splitting them into six separate spot lights with a 45° cutoff angle. The shadowmap is stored in a cubemap texture and six passes are used to populate all cubemap faces with depth data one by one.

5.3.2 Direct illumination

The `LightPass` shader computes direct illumination of a `GBuffer` texel by a given light source. It retrieves `GBuffer` material data for texel and evaluates shading for the normalized Phong BRDF. Light visibility is optionally resolved using the previously generated shadowmap. Results are written into the resolve part of the `GBuffer`. Additive blending is used to accumulate contributions from all light sources.

The above computation is optimized to only process `GBuffer` texels actually affected by the light source. First, the area in which the light has any effect is approximated with a simple object: a sphere or a box for a point light; a pyramid for a spotlight. Exact scale, position and orientation of the light area object are computed from the intensity and transform of the light. This light area object is then rendered with following pipeline stencil state: for front facing triangles, decrement stencil if depth test fails; for back facing triangles, increment stencil if depth test fails. After this pass, only `GBuffer` texels inside the area affected by the light are flagged with non-zero stencil value. The light area object is finally rendered using the `LightPass` shader described above; the stencil test is set to only pass for non-zero stencil values.

5.3.3 Reflective shadowmap generation

If the light is marked to interact with the light propagation volume, a reflective shadowmap is generated at this point. The generation process is similar to shadowmap generation described in the section 5.3.1: `ScenePrimitives` are rendered from the point of view of the light. Depth value, surface normal, diffuse reflectance and reflected radiance from diffuse light-surface interaction are stored. The `LpvReflectiveShadowmap` shader implements this step.

Point lights are handled by splitting them into six separate spotlights and executing steps in sections 5.3.3 through 5.3.5 for each spotlight individually.

5.3.4 Light injection

Light injection is initiated by invoking the `LightPropagationVolumeData`'s `InjectLight` method with radiance, normal and depth RSM textures. Layered rendering is used to perform the injection. `LightData` textures are first bound to the rendering pipeline as layered render targets, each z slice of the 3D light volume texture corresponding to one layer. The `LpvInject` shader is then activated and number of points equal to the number of texels in the RSM is submitted for rendering.

Inside the vertex shader, the built-in `gl_VertexID` attribute is first converted to a 2D texel coordinate. The coordinate is then used to retrieve corresponding depth, intensity and normal values from bound RSM textures. World position of the surfel is reconstructed from its depth and transformed to volume texture coordinates. The coordinates are then shifted slightly towards the light and along the surfel normal to minimize self-lighting. Surfel weight factor is computed from view space depth. The data is then passed to the geometry shader. There, the render target layer to store the final fragment data is selected depending on the z element of the volume texture coordinates.

In the fragment shader, the SH representation of the virtual point light is computed in the following manner: Cosine lobe oriented in the direction of the surfel normal is projected to SH and resulting SH coefficients are scaled by surfel intensity and weight factor. Three SH coefficient four-tuples are written — one for each color channel. Additive blending is used to accumulate contributions from multiple virtual lights landing into the same volume cell.

Note that if the light propagation volume contains multiple cascade levels, light is injected into each individual cascade level separately

5.3.5 Blocker injection

To initiate blocker injection, the `LightPropagationVolumeData`'s `InjectBlockers` is invoked with desired diffuse reflectance, normal and depth RSM textures. When injecting blockers, new `GeometryData` instance is used to accumulate contributions from separate sources (section 3.10.2). Other than that, the process itself is almost identical to light injection and is implemented in the same `LpvInject` shader.

In the vertex shader, surfel depth, diffuse reflectance and normal are retrieved from respective textures. Surfel position is reconstructed, volume texture coordinates and surface weight factor are computed. Destination render target layer is selected in the geometry shader from the volume texture z coordinate. In the fragment shader, blocker distribution SH is computed as a projection of a cosine lobe oriented in the direction of the surfel normal and scaled by the surfel weight. SH coefficients and surfel diffuse reflectance are stored. To aggregate data from surfels landing in the same volume cell, additive blending is used for SH coefficients and max blending for diffuse reflectance values.

5.4 Merging blocker information

If blocker information was injected from multiple sources, more than one `GeometryData` (geometry volume) instance is present for each `LightPropagationVolumeData` (cascade level). In such case, data from intermediate volumes is iteratively merged to form one final geometry volume.

To merge two volumes, resources from two `GeometryData` instances are bound to the rendering pipeline and the `LpvMergeGvs` compute shader is executed. GPU side, blocker and reflectance data for a given grid cell is fetched from both volumes. The data is then aggregated using the max operator and written back into one of the source volumes designated as accumulator.

Merging N volumes is accomplished by designating the volume 1 as an accumulator and executing the previously described merge step once for each of the remaining $N-1$ volumes.

5.5 Light propagation

In the section 3.10.3, the propagation process was originally explained in terms of a scatter operation: For each source cell, compute the amount of light propagated to each of its six neighbors. Implementing the scatter scheme on the GPU, however, would lead to a very undesirable memory access pattern of one read and six atomic add operations per volume cell. Alternatively, the propagation can be expressed as a gather operation: for each cell, compute the amount of light it receives from each of its six neighbors. In this form, the algorithm can be realized using much more reasonable six reads and one write per cell.

A single propagation step is implemented as a gather operation in the `LpvPropagate` compute shader. The algorithm from section 3.10.3 is realized in GLSL code with support for both the indirect illumination occlusion and extra light bounce simulation features. Inside the shader, precomputed constants and hardware trilinear interpolation are used to optimize performance. Besides outputting the propagation results, which will serve as an input for the subsequent propagation iteration, the per iteration results are also added to an accumulator, which will at the end store the final indirect illumination solution.

Light propagation in a single cascade level is initiated by invoking the `Propagate` method of the `LightPropagationVolumeData` class with the desired number of iteration steps. Double buffered temporary `LightData` are used to store inputs and outputs of intermediate propagation iterations; the front and the back buffer are exchanged after each iteration. The local `LightPropagationVolumeData`'s `LightData` instance serves to accumulate the result. Similarly to injection, if the light propagation volume cascade contains multiple levels, each individual level must be propagated separately.

5.6 Indirect illumination

To retrieve the indirect illumination from a light propagation volume, clients invoke the `DeferredRelight` method of the `LightPropagationVolume` class. This routine realizes a deferred additive pass over the `GBuffer`, in which the surface is lit with previously propagated light.

The GPU side computation is implemented in the `LpvRelight` shader: Surface attributes are first retrieved for a given `GBuffer` surfel. Volume texture coordinates are computed from surfel world position, which is reconstructed from the surfel depth and camera parameters. SH coefficients representing the per color channel light directional distribution are retrieved from three 3D light volume textures using the computed volume texture coordinates. The incident radiance is then computed by integrating the product of the light intensity distribution and cosine lobe oriented in the direction of the surfel normal over the hemisphere. Thanks to the SH representation of both functions, this amounts to a dot product.

To mitigate light leaking and self illumination due to large cell sizes, final illumination can be dampened. The dampening factor is derived from central difference of the SH coefficients computed at small offsets along the surfel normal per color channel. Maximum over square magnitude of the difference vectors is taken as the dampening factor.

The `MarchGlossIntensity` shader routine implements the Glossy indirect illumination approximation feature by marching through the light volume along the refection direction ray. Longer rays with more steps are used for glossier surfaces. Step contributions are weighted by inverse distance from origin.

To facilitate smooth transitions between cascade levels and areas completely outside of the light propagation volume, contribution from a each cascade level is faded out in locations close to its border. To do this, an inlier factor of a cascade level is computed from the 3D texture coordinate. The factor decreases from one to zero in the border area of the 3D texture. As mentioned in the section 3.10.5, when moving with camera, the grid positions of individual cascade levels are snapped to their respective cell sizes to reduce flickering artifacts. For inlier factor computation, hoverer, coordinates into a virtual unsnapped volume grid are used. This way, the inlier factors for world position vary smoothly with each camera movement, and not only when the volume grid is resnapped.

If more than one propagation volume cascade is available, illumination coming from individual cascade levels is evaluated separately, each with its own inlier factor. The final illumination is then computed as an average over all cascade levels. Note that glossy illumination is only computed for the finest cascade level.

5.7 Post-processing

Once all previous steps have completed, the resulting image is processed by *Mophological anti-aliasing* (MLAA) [Res09] algorithm to smoothen jagged edges. MLAA works in three passes: In the first pass, edge elements in the image are detected using color, depth or normal based thresholding. Per pixel blend weights are then computed by analyzing edges reconstructed from edge elements obtained in the previous step. Finally, blend weights are used to smoothen edge texels by blending their neighborhood texel values into them. The `MlaaDetectEdges`, `MlaaComputeBlendWeights` and `MlaaBlendResult` compute shaders implement the three stages of the algorithm.

The anti-aliased image of the scene is then the gamma-corrected using the `GammaCorrect` shader, after which it is ready to be presented.

Chapter 6

Auxiliary systems implementation

6.1 Rendering hardware interface

One implementation of the `Rh*` abstraction layer is provided, backed by the OpenGL version 4.3 API. The overall process of converting the functionality expressed in the `Rh*` interface is fairly straightforward: `OpenGlRhDevice` implements both the `RhDevice` and `RhCommandList` interfaces. Opaque OpenGL handles are wrapped using corresponding `RhBuffer`, `RhTexture` and `RhProgram` implementations. Vertex array object configuration, i.e. how vertex buffer data are interpreted inside a shader, is encompassed in a single `RhInputLayout`. Pipeline state objects map to a set of OpenGL state values and so on.

To minimize the number of state changes propagated to the actual driver, active OpenGL state is tracked on the CPU side in the `OpenGlRhState` class. State setting (via OpenGL routines) is delayed until a command to draw primitives or dispatch a compute operation arrives. When that happens, all changed resource binding a pipeline state is committed to the driver through appropriate API invocations. This way, the number of state changes is further minimized by only committing state used to perform some work.

To make switching between different pipeline states slightly more convenient, state object singletons can be accessed through template classes. Typically, with an object such as `RhDepthStencilState`, only one instance per combination of parameters (depth test on/off, depth compare function, stencil operation, etc.) ever needs to be created. With C++ templates, the one singleton per parameter combination design can be easily achieved by using a static variable inside a static method inside a template class. This approach is realized for example in the `RhStaticDepthStencilState` class.

6.2 Objects and components

At the very top of the object-component hierarchy designed in the section 4.3 resides the `World` class, which serves as a container for root instances of the `Object` class. `Object`s are never allocated directly, but rather immediately created within a specified `World` using its `CreateObject` method. Inside an `Object`, its components are stored by their runtime type in a hash table. The implementation leverages the standard C++ runtime type information system, namely the `typeid` operator, to create a per type unique `TypeToken` value. Similarly to `Objects`, instances of `Component` subclasses are never allocated directly. They are instead added to an object by invoking the `AddComponent` method with a specified `TypeToken`.

Both `Objects` and `Components` can be marked for destruction using their `Destroy` methods. The destruction itself is postponed until the very end of the frame to minimize the likelihood of following a dead object or component reference. Clients may choose to respond to predefined events in the component's life-cycle by overriding dedicated virtual methods. Events interceptable this way include per frame update, transform change, attachment to an object or destruction. Finally, replication of entire object hierarchy subtrees is supported in the `Object`'s `Clone` method. The method performs a deep copy of the object including all its children in the transform hierarchy.

6.3 Asset loading

External 3D models are loaded into the demo application's runtime via `AssetManager`'s `LoadModelAsObjectHieararchy` method. First, the Assimp library is used to parse the specified 3D file format, returning an instance of the `aiScene` structure. The model directory is then searched for <model-name.ext>.ppinfo file. In this file, a user can specify additional operations to execute for the imported model such as computing flat or smooth normals, flipping triangle winding and ignoring certain objects. Once all custom operations are executed, meshes contained in the imported scene are classified according to their vertex attribute layout. Vertex attribute buffers, one per vertex attribute layout, are then uploaded to the GPU in interleaved format yielding a `VertexSource` object. `MeshData` objects containing necessary draw information are then created for distinct scene meshes. Finally, the node hierarchy inside the `aiScene` structure is traversed and reconstructed as a tree of `Objects`, each with a corresponding `Transform` and `MeshRenderer` components. During this process, referenced materials are converted into instances of the `Material` class. If a particular material references any textures, images are loaded into memory using the ResIL library and uploaded onto the GPU as `RhTextures`.

Object hierarchy snippets can be stored inside .prefab files and reused throughout multiple scenes. Individual demo test scenes are defined inside .scene files, which contain definition of the demo `World` and additional meta parameters like camera move speed, near/far planes and optional skybox. Both .prefab and .scene files use the same JSON-inspired format to describe object hierarchies. A recursive top down parser for .scene and .prefab files is implemented in the `InitializeFromConfig` method of the `ClpvDemo` class.

6.4 Application life-cycle

When the application is run, the first step is to create a system `Window` and initialize platform message handlers. This task is delegated to the GLFW library. Once the initialization is complete, the application enters the `MainLoop::Start` method which it doesn't leave until the client requests the application to quit.

During one main loop iteration, platform messages indicating events such as cursor move, window resize or a key press are first processed. Afterwards, all `Components` currently active in the application have their `Update` method executed. Updates are batched per component type. Once all components are updated, the scene is rendered. Debug visualizations and GUI are then drawn over the scene image and the result is presented. At the end of the frame, the queue of objects and components that were marked for destruction during the frame is processed until no remain.

After leaving the main loop, all `Objects` are destroyed and cached assets are released before returning from the `main` function.

6.5 Profiler

GPU and CPU side timing responsibilities are handled by the `Profiler` class. Statistics for a single frame are delimited by calls to `Start` and `EndFrame` methods. Between these two calls, clients signal start and end of named CPU or GPU timespan using a dedicated set of methods. The timespans may be arbitrarily nested and chained, yielding a tree structure. For each frame, `Profiler` additionally keeps an instance of `RhStats` structure, containing stats on the number of draw and compute commands executed during the frame.

For CPU side timing `Profiler` relies on the precision platform timer exposed by the GLFW library; GPU timing is supported using the `RhTimer` interface, a GPU stopwatch abstraction built using graphics API timestamping functionality. CPU and GPU statistics for last 500 frames are stored in a circular buffer. Clients may either read raw information from the timing trees or the call `Profiler::ShowBreakdown` method to display a visualization depicted in the figure C.2.

6.6 Rendering system support

6.6.1 GPU program utilities

Resources used in a `RhProgram` are referred to using logical indices, which must be queried with the shader-side name of the parameter. While a good design, the indices need to be required after each shader recompilation with a lot of repetitious code. For this reason, the framework provides the `GlobalShaderDefinition` and `ShaderResourceBlockBase` classes.

A shader resource block layout inheriting from `ShaderResourceBlockBase` can be defined using a set of macros to specify each individual program resource. `GlobalShaderDefinition` then associates a shader source file with a runtime `Shader` object and a particular subclass of `ShaderResourceBlockBase`. Defining a GPU program this way conveys two main advantages: Easy access to the compiled program singleton and typesafe binding of program resources via associated resource block type.

Recompilation of programs defined using `GlobalShaderDefinition` can be manually forced through debug GUI, which is very useful during development. This is taken one step further on the Windows OS version of the application, where the recompilation is triggered automatically in response to a shader source file change.

6.6.2 Render target reuse

Many temporary render targets are used while rendering the final frame. Render target pooling is a technique which reduces GPU memory footprint by encouraging render target reuse. Whenever a texture render target is needed somewhere in code, a request is made to the `RenderTargetCache` with a `RenderTargetDesc` specifying the properties of the texture to retrieve. The caller then receives a handle to a `CachedRenderTarget`, which it may use for an arbitrary amount of time. Once the user relinquishes the ownership of the acquired render target, it returns to the `RenderTargetCache` and may be freely reused by others. `CachedRenderTargets` that haven't been used for a specified number of frames are automatically evicted from the cache and associated GPU resources are destroyed.

6.7 Reference path-tracer

A rudimentary path-tracer is implemented in the `PathTracer` class to serve as the reference global illumination algorithm. Path tracing is accelerated by a `KdTree` containing all scene object triangles. Three position, normal and texture coordinate vectors are stored for each triangle along with a material id. The structure is built on the CPU using the midpoint-split strategy.

Path-tracing itself takes place on the GPU, realized in the `PathTrace` compute shader. Ray-triangle intersections are resolved using the supplied kd-tree data. Both push-down and short-stack [HSHH07] kd-tree traversal techniques are implemented, but short-stack is not actually used as it only seemed to decrease performance on tested GPUs. Normalized Phong BRDF and light types used in the main application (point/spot/directional) are supported. Russian roulette and a max depth parameter are used to terminate path generation.

6.8 Debug visualization

To ease the development process, the framework exposes several Debug visualization facilities in the `debug` namespace. Support for drawing debug scene-depth-aware lines is provided through a set `Draw*` calls. This functionality is useful when displaying object bounding boxes, light effect volumes, camera positions and similar. During the frame, lines are accumulated in a CPU side buffer, which, at the end of the frame, gets uploaded onto the GPU and used to draw all lines in one draw call. For line sets that are prohibitively large to submit every frame, a `LineBatch` can be created once and then submitted efficiently multiple frames. The demo application uses this functionality to display kd-tree bounds.

To verify content of `CachedRenderTargets`, the `OverlayRenderTarget` method can be used overlay texture contents over the final frame image. Basic layout algorithm ensures that multiple displayed textures don't overlap each other. The debug GUI supports dynamically displaying and hiding of render targets which have been tagged in code using the `ExposeRenderTarget` call.

GPU programs are supplied for visualization of light propagation volume data. Cascade coverage visualization mode shows how individual light propagation volume cascade levels affect the scene. Blocker and light distribution SH in geometry and light volumes is visualized using spheres colored from red to green depending on the function value in the given direction. Instanced rendering is used to generate spheres for all volume grid cells in the `VisualizeSHGrid` shader. The voxelized scene representation stored in the final geometry volume can also displayed, each non-empty voxel visualized as a cube with the stored voxel diffuse reflectance. This visualization option is implemented in the `VisualizeVolume` shader, which uses the geometry shader stage to generate a cube for every voxel on the fly.

6.9 GUI

The ImGui library is used to render the demo application GUI. The GUI consists of four main windows depicted in figure C.1. Most important is the object hierarchy inspector, which can be used to view and edit `Object Components`. Also useful is the debug info window, which shows latest frame timing, GPU stats and contains a list of `CachedRenderTarget`s available for visualization. Remaining two windows are used to control path-tracing mode and display control hotkeys. `Profiler` per frame timing breakdown is also visualized using ImGui.

Chapter 7

Evaluation and verification

In this chapter, the implemented Cascaded Light Propagation Volumes is tested on several scenes of varying geometrical and lighting complexity. Algorithm performance and quality of the resulting image is analyzed and compared with a reference solution computed via path-tracing.

7.1 Scenes

The testing dataset consists of six scenes. 3D models used in the scenes were acquired from the McGuire Graphics Data repository [Mor15] and various other free online model repositories [tur15, t3d15]. Table 7.1 contains detailed scene parameters. Individual scene thumbnails are shown figure 7.1 .

The first scene is a simple Cornell box setup with a single moving spotlight. It is designed to illustrate color bleeding and inspect stability of the solution for a moving light source. Light propagation volume position is fixed in this scene, only a single cascade level is used.

The second scene, titled Bleeding and Gloss (B&G), features several rotating cubes with differently colored, high contrast faces. Illumination is provided by a single directional light source. Purpose of this scene is to stress test the stability of the solution in a scene with dynamic objects. Scene floor is also assigned material with varying glossiness to test the glossy indirect illumination ray-marching. Light propagation volume in this scene is also a single fixed cascade level.

The third scene features a slightly modified version of the Conference Room model by Anat Grynberg and Greg Ward. Blinders covering windows were removed and the originally one-sided walls and ceiling were fixed using simple box meshes. The scene is lit by a single directional light source which enters the scene by the newly uncovered windows. A single moving model of a dwarf is used to obstruct the incoming light. This is the first of the more complex scenes and the last one to use a single fixed light propagation volume.

The fourth scene, Dabrovic, is set in the Marko Dabrovic's version of the Sponza palace atrium. It is additionally filled with a high number of static scenery objects and a few moving animal models. Lighting simulates a night time scenario: three moving spotlights are used to illuminate the scene. Light propagation volume cascade with three levels attached to camera is used to compute indirect illumination.

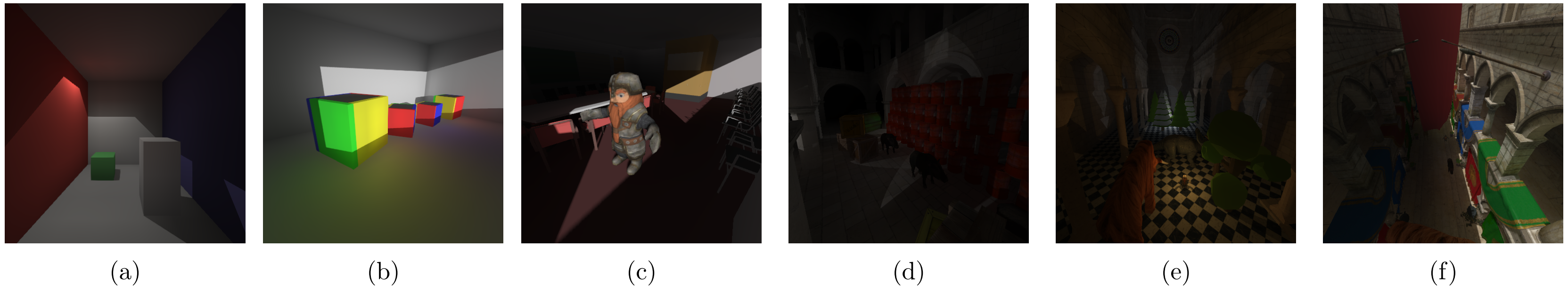


Figure 7.1: Testing scene thumbnails: (a) Cornell. (b) B&G. (c) Conference. (d) Dabrovic. (e) Sibenik. (f) Sponza.

	Cornell	B&G	Conference	Dabrovic	Sibenik	Sponza
#Triangles	84	360	188780	180320	91300	296873
Light 1 [RSM resolution]	Dir [128]	Dir [128]	Dir [128]	Spot [128]	Point[128]	Dir [256]
Light 2 [RSM resolution]	N/A	N/A	N/A	Spot [128]	Point[128]	Point [128]
Light 3 [RSM resolution]	N/A	N/A	N/A	Spot [128]	N/A	N/A
#LPV cascade levels	1	1	1	3	3	3
LPV resolution	$8 \times 8 \times 7$	$32 \times 15 \times 27$	$32 \times 8 \times 21$	$32 \times 32 \times 32$	$32 \times 32 \times 32$	$32 \times 32 \times 32$
GV occlusion [Inject Gbuffer]	Yes[No]	Yes[No]	Yes[Yes]	Yes[No]	Yes[Yes]	Yes[Yes]
#Propagation steps	16	64	16	12	12	16
Total RSMs triangles	84	360	188780	540960	1095600	2078111
Total elementary injections	32786	32786	49152	294912	1228800	1032192
Total EPS	7167	829440	86016	1179648	1179648	1572864

Table 7.1: Testing scene parameters. *Total RSMs triangles*: $\#Triangles \times \#RenderedRSMs$ *Total elementary injections*: number of RSM/g-buffer surfels injected into LV/GV (g-buffer is estimated as 128^2 texels). *Total EPS (elementary propagation steps)*: $\#TotalLpvCells \times \#PropagationSteps$.

The fifth scene tests the algorithm with point lights and moving objects. Two point light sources are placed inside the Sibenik cathedral interior and each is surrounded by a circle of rotating objects. This scene also uses the three level light propagation volume cascade.

The final, sixth scene is set in the Crytek Sponza palace atrium model. Several moving objects are added to the atrium and first floor. Illumination is provided by a directional light modeling sun and a point light circling around the atrium ground floor. Light propagation volume with three cascade levels and moving with camera is used in this scene.

7.2 Testing environment

The demo application was compiled with the Microsoft Visual C++ Compiler 2013 in release mode as a 32bit executable. The tests were run on two different machines: powerful desktop machine (GTX980) and a lower mid range laptop (930M). Spec details of each hardware setup are listed in the table 7.2.

	GTX980	930M
GPU	GeForce GTX980	GeForce 930M
CPU	Core i7-2600K	Core i5-4210M
RAM	16GB	8GB
OS	Windows 7 64b	Windows 7 64b
Driver	v355.98	v355.98

Table 7.2: Testing hardware setup details

7.3 Performance evaluation

This section discusses performance characteristics of our Cascaded Light Propagation Volumes implementation. All timings presented are from the GPU, measured with timestamp queries. The values are averages over samples from 30 frames. Measurements were taken in windowed mode with the application set to run as fast as possible (no fps cap). Tables 7.3 and 7.4 list test scene timings for 512×512 and 1024×1024 resolutions respectively.

7.3.1 G-buffer blocker injection

There are two factors affecting the time complexity of the g-buffer blocker injection (IGB) stage: g-buffer resolution and number of cascade levels. The g-buffer is always downsampled to roughly 128×128 resolution before injection. The downsampling is realized by successive halving, so an extra downsampling iteration is necessitated roughly each time the resolution doubles. This is however only a minor cost.

The major factor in g-buffer injection is the number of cascade levels because blocker information is injected into each level individually. This is clearly illustrated in the measured data: The IBG step is two to three times faster in the Conference scene than in the Sibenik and Sponza scenes because the latter two have a light propagation volume with three cascade levels, whereas Conference only has only one cascade level.

Scene	IGB	GRSM	IL	IB	PRP	REL	TOTAL
GTX980, 512 × 512 *px*							
Cornell	N/A	0.01	0.19	0.10	0.36	0.03	0.71
B&G	N/A	0.01	0.03	0.03	2.16	0.09	2.34
Conference	0.06	0.14	0.02	0.02	0.42	0.05	0.73
Dabrovic	N/A	0.24	0.48	0.42	2.23	0.05	3.42
Sibenik	0.17	17.66	2.22	2.27	2.96	0.06	25.34
Sponza	0.16	1.69	1.17	1.23	3.02	0.05	7.32
930M, 512 × 512 *px*							
Cornell	N/A	0.04	0.43	0.30	0.37	0.18	1.32
B&G	N/A	0.04	0.19	0.16	7.21	0.57	8.17
Conference	0.43	1.55	0.16	0.13	0.80	0.29	3.36
Dabrovic	N/A	2.37	2.21	1.84	11.24	0.35	18.01
Sibenik	0.91	11.04	9.29	7.52	12.44	0.43	41.63
Sponza	1.02	12.68	6.48	5.28	15.42	0.44	41.32

Table 7.3: Scene timings for 512×512 resolution. IGB: G-buffer downsampling and injection. GRSM: Reflective shadowmap generation. IL: Injecting RSM texels to light volume. IB: Injecting RSM texels to geometry volume. PRP: propagation. TOTAL: sum of all steps. All values are in ms.

7.3.2 Reflective shadowmap generation

Complexity of the reflective shadowmap generation stage (GRSM) depends on the number of reflective shadowmaps generated, their resolution and the number of triangle primitives rendered. One RSM suffices for each spot and directional light. Point lights are much more expensive in this context, since they require six separate RSMs to be rendered. Essentially the *Total RSMs triangles* statistic in the table 7.1 should constitute a reasonable relative measure of complexity of this step. And it actually does, except for one outlier: the Sibenik scene on the GTX980 hardware, whose GRSM step takes an order of magnitude longer to execute than the same step for the theoretically more complex Sibenik scene. A brief inspection reveals, that `ScenePrimitives` in the Sibenik scene are very granular and over 1200 draw calls are made to render a single RSM, which is a lot for a scene of this size. But the ordinarily vastly inferior 930M hardware handles GRSM for Sibenik just fine, relatively to other scenes. Essentially, the main problem here is that the primitive submission logic in the demo application is extremely primitive: no frustum culling is performed so all primitives are always rendered; and the primitives aren't sorted by their material, shader or distance from camera at all. Submitting so many primitives in arbitrary order can lead to a lot of resource switching and GPU pipeline state invalidation, which could have more pronounced effect on the more advanced GPU. So to find out what really causes the horrible slowdown, one would start with the primitive submission optimization and see what happens.

7.3.3 Light and blocker injection

Inject light (IL) and inject blockers (IB) should scale linearly with the number of RSM texels times the number of cascade levels. We would expect the (IL) stage to take slightly longer

Scene	IGB	GRSM	IL	IB	PRP	REL	TOTAL
GTX980, 1024×1024 *px*							
Cornell	N/A	0.01	0.17	0.16	0.36	0.08	0.78
B&G	N/A	0.01	0.03	0.03	2.14	0.32	2.53
Conference	0.10	0.14	0.02	0.03	0.42	0.15	0.86
Dabrovic	N/A	0.24	0.59	0.46	2.23	0.15	3.68
Sibenik	0.20	15.99	2.78	2.02	2.58	0.14	23.71
Sponza	0.22	1.16	1.65	1.21	3.03	0.16	7.44
930M, 1024×1024 *px*							
Cornell	N/A	0.05	0.46	0.32	0.38	0.66	1.86
B&G	N/A	0.04	0.19	0.16	7.24	2.16	9.78
Conference	0.78	1.56	0.16	0.13	0.81	1.04	4.46
Dabrovic	N/A	2.37	2.45	1.98	11.41	1.27	19.47
Sibenik	1.74	11.04	9.25	7.49	12.61	1.43	43.55
Sponza	1.88	12.75	8.14	6.20	15.63	1.55	46.15

Table 7.4: Scene timings for 1024×1024 resolution. See table 7.3 for legend.

because the shader does extra computation shifting VPLs to light and writes two times more data than the (IB) stage. While the GTX980 hardware is likely too fast to care, the data measured using the 930M shows that the (IB) stage is consistently faster. Change in display resolution should not affect IL and IB stages at all. Indeed the data does not show any consistent pattern, any fluctuations likely be due to variance.

7.3.4 Propagation

In the propagation step (PRP), each volume cascade level is processed separately, indicating linear scaling with the number of cascade levels. Before the propagation begins, geometry volumes are merged. The merging is iterative, one iteration executed for each intermediate geometry volume beyond the first. Once the merging is done, the propagation itself begins. A preset number of iterations is executed and every iteration, all light propagation volume cells are processed. This step this expectedly scales linearly with three factors: the aforementioned number of cascade levels; number of cells per cascade; and the number of propagation steps. The *Total EPS* statistic from the table 7.1 should roughly correspond to the time spent in the PRP stage. Note the timing for the PRP stage of the scenes Dabrovic and Sibenik in any timing set. Both have the same number of cascades, number of propagation iterations and number of cells per cascade, yet PRP is consistently slower for Sibenik. The difference is time spent merging geometry volumes: for Dabrovic scene, only 6 (3 cascades $\times$ (3 RSMs - 1)) merge steps must be executed, whereas 36 (3 cascades $\times$ (12 RSMs + G-buffer - 1)) are needed for Sibenik.

7.3.5 Relighting

The relight (REL) step is a full-screen g-buffer pass, so the timings are definitely expected to scale linearly with the number of texels in the g-buffer. We can confirm this easily by comparing the timing datasets for 512×512 and 1024×1024 resolution. The performance of the relight shader additionally depends on the number of cascade levels, and whether gradient dampening and glossiness ray-marching features are enabled. Resolution of the light propagation volume also plays a role since smaller 3D textures fit better into GPU memory caches.

The Cornell scene serves as a baseline: it has one cascade level and REL stage has no extra features enabled. It is consistently the fastest. The B&G scene still uses only one cascade level but it has much finer resolution and also uses the ray-march gloss feature. We can observe a slowdown from the Cornell to the B&G by the factor of three to four in all timing datasets. Toggling the ray-march gloss feature on-and off reveals, that it is solely responsible for the extra time. The conference scene also uses only one cascade level and the gradient dampening feature, which turns out inexpensive compared to ray-marched gloss. If we compare the number of texture fetches executed by the ray-march and dampening techniques, we see that it is expected: gradient dampening uses 6 texture fetches, whereas ray-march in the B&G scene may perform up to and over 24 texture fetches depending on the exact number of steps. Dabrovic, Sibenik and Sponza scenes use no extra features during the relight step but data is fetched from three cascade levels. On GTX980, this is performance-wise roughly as expensive as turning on gradient dampening for a single cascade level, on 930M even more so.

7.4 Quality analysis

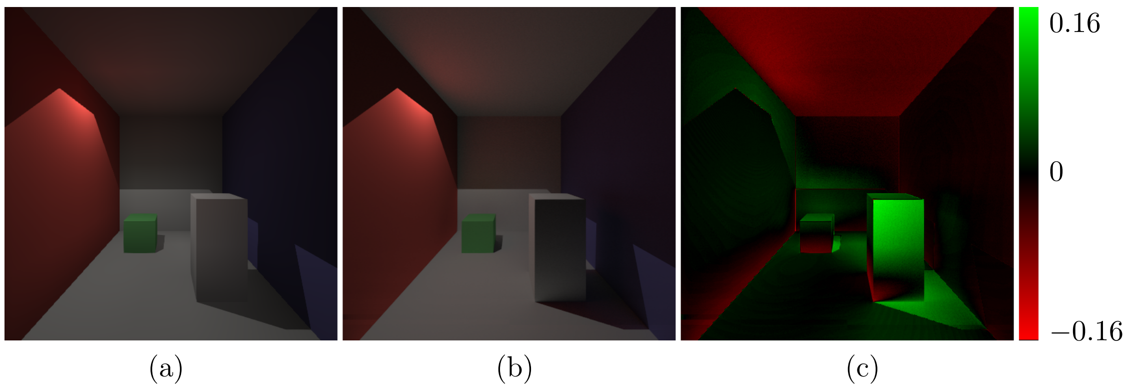


Figure 7.2: The Cornell scene: (a) CLPV (b) Path-traced reference (c) difference a - b.

In this section we analyze the correctness of indirect illumination computed using CLPV a inspect how algorithm parameters affect the quality of the solution. Stability of the solution under motion is also evaluated.

Figures 7.2 to 7.7 compare an image generated using the CLPV algorithm to a path traced reference solution for test scenes from table 7.1. Path-tracer was set to terminate path-generation after the second scattering event to match the CLPV possibilities.

Light leaking through thin geometry is visible in the majority if scenes, most notably the front vertical block in the Cornell scene (figure 7.2), tree group in the Sibenik scene (figure 7.6) and point light illumination leaking through the floor on the left side of the Sponza scene

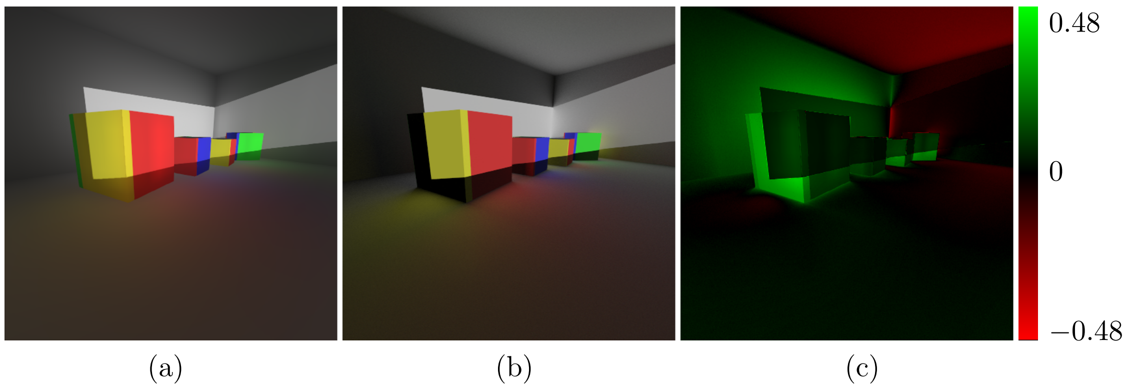


Figure 7.3: The B&G scene: (a) CLPV (b) Path-traced reference (c) difference a - b.

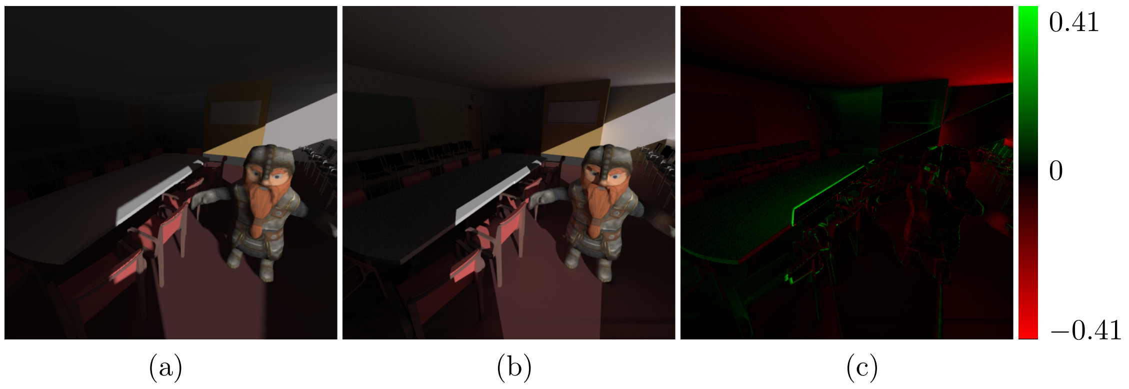


Figure 7.4: The Conference scene: (a) CLPV (b) Path-traced reference (c) difference a - b.

(figure 7.7). This is inherent weakness of the discretized representation of the scene. Using occlusion in the propagation process and gradient dampening during relighting mitigates the issue in some cases, but not every time. Figure 7.8 shows how in the Conference scene, the attenuation factor helps to get rid of almost all light leaking through the wall; in the B&G scene, it however introduces major artifacts.

The other problem we encountered is, that the propagation scheme smears the illumination distribution in all directions heavily just after a few iterations. One result of this is that higher frequency details are lost — floor areas that receive color from the lit cubes in figure 7.3 are well defined in the reference image but smeared in the CLPV solution. Other, more visible, result of this smearing is self illumination, also best illustrated in the B&G scene on the over-brightened back wall and lit cubes.

To account for light traveling over longer distances, the CLPV algorithm must either use a grid with large cells or execute many propagation iterations. The figure 7.9 illustrates how the number of propagation steps affects the solution in a simple scenario. We can see the indirect illumination reaches more distant areas of the scene (with respect to the the initial injection point) with more propagation steps. The related case of varying cell sizes and leaving the number of propagation steps fixed is shown in the figure 7.10. We observe, that the effective distance the light travels expectedly decreases with smaller cell sizes, additionally using too fine a grid causes artifacts in the areas receiving direct illumination, likely due to

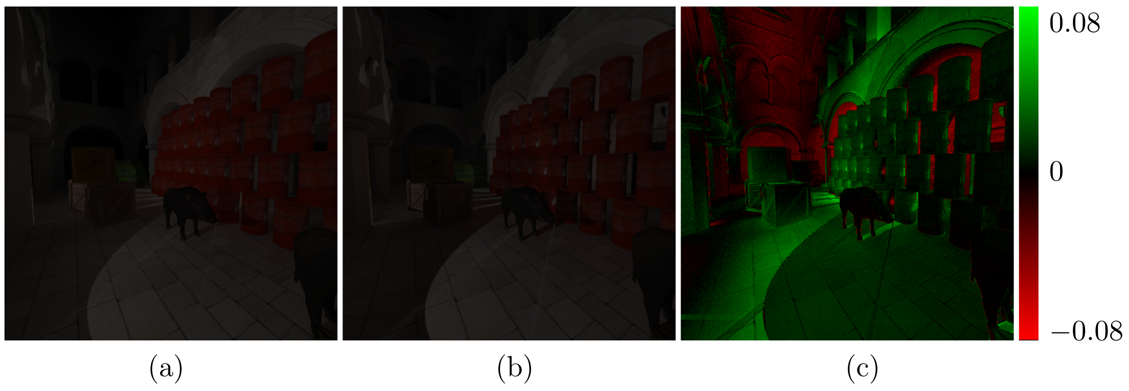


Figure 7.5: The Dabrovic scene: (a) CLPV (b) Path-traced reference (c) difference a - b.

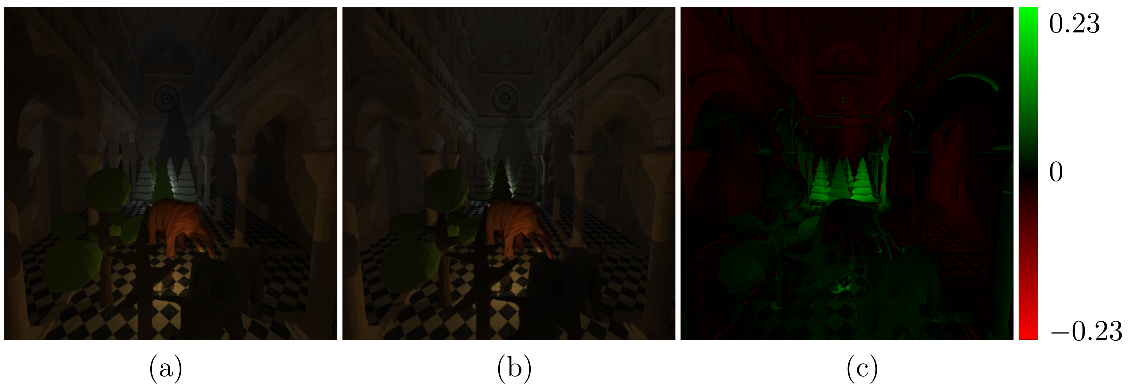
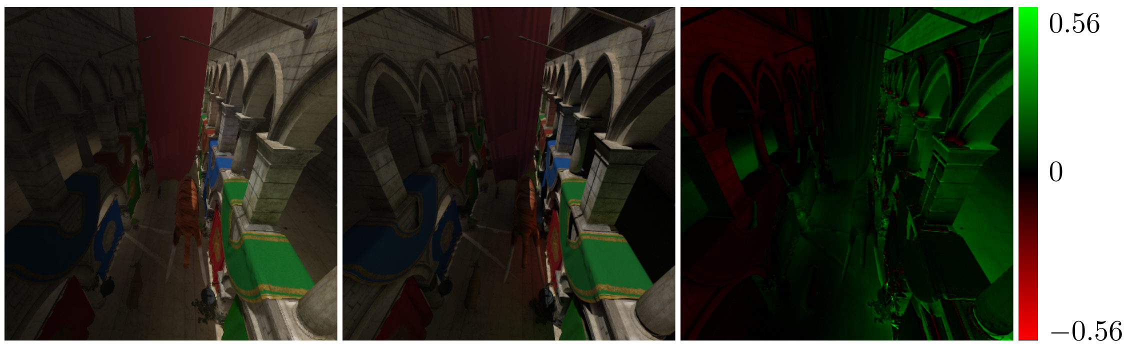


Figure 7.6: The Sibenik scene: (a) CLPV (b) Path-traced reference (c) difference a - b.

self illumination.

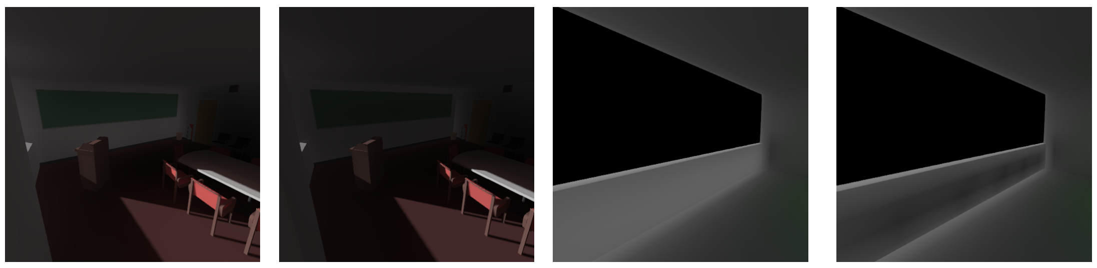
One of the problems we observed with our implementation of the CLPV algorithm is that no single LPV grid configuration fits ideally all scenes. We finally settled on three cascade levels, each with $32 \times 32 \times 32$ resolution. Cell size in the finest grid is rougly 30cm and doubles with each following cascade level. 12 to 16 iterations are used to propagate illumination through each level. This configuration seems to work reasonable well in the Dabrovic, Sibenik and Sponza scenes.

The algorithm is stable under camera motion as can be seen in the Sibenik, Dabrovic and Sponza scenes. Remaining scenes use a fixed LPV grid, so camera motion is a non-issue. The algorithm is however sensitive to sudden high-contrast changes in the inject VPL colored intensity. The propagated indirect illumination then flickers in areas near the LPV grid cells, in which the sudden injection fluctuation took place. This issue is immediately visible in the B&G scene in motion. As differently colored faces of the rotating cube receive illumination and their surfels are injected into different LPV cells, visible flicker occurs on the scene floor.



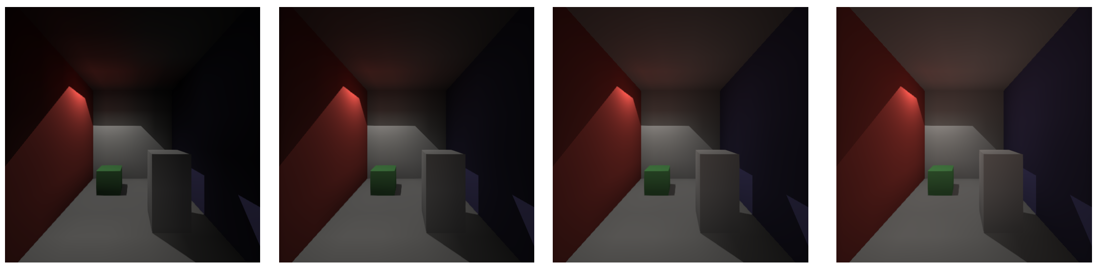
(a) (b) (c)

Figure 7.7: The Sponza scene: (a) CLPV (b) Path-traced reference (c) difference a - b.



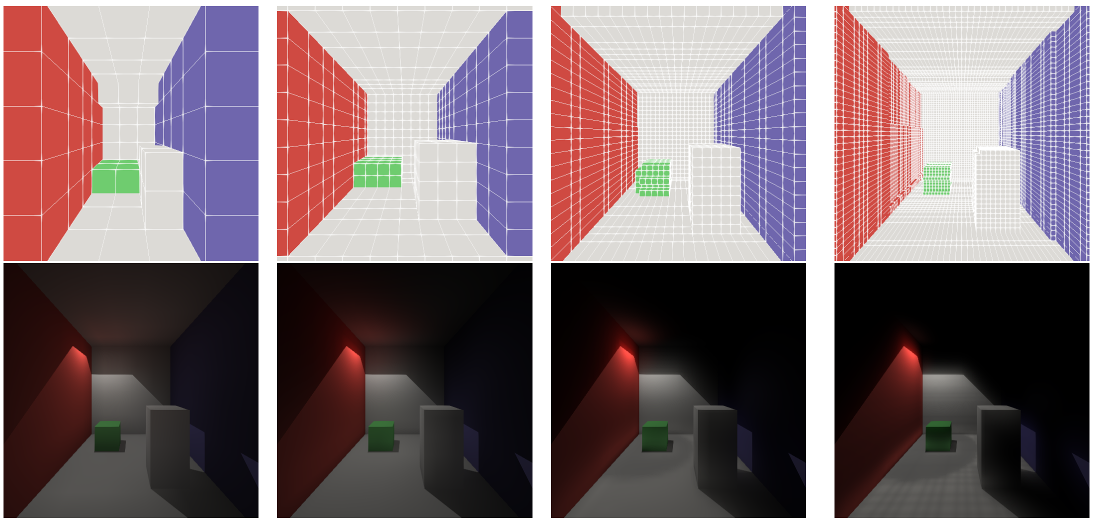
(a) (b) (c) (d)

Figure 7.8: Gradient based attenuation: (a) Conference: disabled. (b) Conference: enabled. (c) B&G: disabled. (d) B&G: enabled.



(a) 4 steps (b) 8 steps (c) 16 steps (d) 32 steps

Figure 7.9: Cornell scene: Fixed 8×8×7 LPV grid resolution, varying number of propagation steps.



(a) $8 \times 8 \times 7$ (b) $15 \times 16 \times 15$ (c) $31 \times 32 \times 29$ (d) $59 \times 62 \times 57$

Figure 7.10: Cornell scene: Fixed 16 propagation steps, varying LPV grid resolution. Top image row shows voxelized scene from geometry volume to provide cell size reference. Note that the grid is regular, a but some visualization lines did not pass the depth test when rendering.

Chapter 8

Conclusion

Global illumination algorithms are essential for generating high fidelity images. They account for indirect illumination generated by multiple light scattering events, which is responsible for visual phenomena the human eye is trained to expect from the real-world. In real-time rendering, for largely static scenes, the best solution to computing indirect illumination is to compute it before the application runs. This is however not possible in dynamic scenes, where geometry, surface properties, lighting or camera position might change every frame. With recent rendering hardware advances; it has become possible to efficiently approximate indirect illumination in dynamic scenes.

8.1 Summary

The reviewed real-time indirect illumination algorithms usually presented a trade-off between visual fidelity of the solution and time necessary to compute it. They also differed in the level of scene dynamism they supported. In this thesis, we focused on implementing the Cascaded Light Propagation [KD10] algorithm. CLPV works by injecting direct illumination into a volume grid and them propagating it in an approach similar to grid based fluid simulation techniques.

Our implementation of the CLPV technique achieves interactive frame rates on lower mid range and real time speeds on higher end rendering hardware. The algorithm is fully GPU-bound as that is where all the work takes place. The rendering time dedicated to the algorithm can be controlled by adjusting the number of cascades level, grid resolution, number of injected VPLs, propagation steps as well as by toggling the glossiness ray-march, gradient attenuation and GV occlusion features. Fully dynamic scenes are supported, requiring no preprocessed information. Moving camera, geometry and light sources are handled seamlessly.

The technique is unfortunately not free from artifacts. Light leaking through thin objects and self-illumination are an inherent result the coarse discretization of the scene due to relatively large cell sizes and the lack of complete representation due to limited area covered by the g-buffer and reflective shadow maps. Additionally, the low order spherical harmonic representation of illumination distribution function as well as the coarse 6-axial propagation scheme make it hard to capture any higher frequency and local detail. However, perhaps the most inconvenient issue is, that the same parameters used to control the algorithm performance described in the previous paragraph also affect the results in a hard to predict manner. It is therefore often hard to come up one parameter configuration that would works equally well in multiple scenes. Qualitatively, the algorithm is able to produce visually pleasing results with impressively pronounced light bleeding. The results themselves do not approach the quality of the path traced solution, as is expected since the CLPV algorithm only aims for a fast, visually pleasing approximation.

8.2 Future work

Several enhancements come to mind that could readily improve the quality of images generated by the current implementation. Since the light propagation volumes are not suitable for capturing local lighting events, algorithm authors themselves recommend supplementing the indirect illumination obtained from light propagation volume with a screen space light transfer solution. Either basic screen space ambient occlusion [BS08] or more advanced techniques [BSD08, RGS09] could be leveraged to add the sought after local detail.

The other relatively simple extension would be adding support for area lights and light emitting particles. Supporting area light sources would only amount to sampling the source surface for virtual point lights and injecting them into the light propagation volume. Similarly, light emitting particles could be injected as virtual point lights directly.

Performance-wise the RSM rendering could benefit from frustum culling and better primitive submission ordering as mentioned earlier. Resulting RSM could also be importance sampled for VPLs instead of blindly injecting each texel. All shaders could also take advantage of actually optimizing the instructions required to accomplish their task. Finally, relighting step could be sped up by only executing it on g-buffer texels inside the light propagation volume. An approach identical to the one described in the section 5.3.2 could be used to stencil out texels that need processing.

Bibliography

[App15] Apple Inc. Metal for Developers, 2015. <`https://developer.apple.com/metal/`>.

[Ass15] Assimp Development Team. Open asset import library, 2015. <`http://assimp.sourceforge.net/`>.

[Bli77] James F Blinn. Models of light reflection for computer synthesized pictures. In *ACM SIGGRAPH Computer Graphics*, volume 11, pages 192–198. ACM, 1977.

[BS08] Louis Bavoil and Miguel Sainz. Screen space ambient occlusion. *NVIDIA developer information: http://developers.nvidia.com*, 6, 2008.

[BS12] Brent Burley and Walt Disney Animation Studios. Physically-based shading at disney. In *ACM SIGGRAPH*, pages 1–7, 2012.

[BSD08] Louis Bavoil, Miguel Sainz, and Rouslan Dimitrov. Image-space horizon-based ambient occlusion. In *ACM SIGGRAPH 2008 talks*, page 22. ACM, 2008.

[Chr08] P Christensen. Point-based approximate color bleeding. *Pixar Technical Notes*, 2(5):6, 2008.

[Chr15] Christophe Riccio. GLM - OpenGL Mathematics, 2015. <`http://glm.g-truc.net/`>.

[CNS$^+$11] Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, and Elmar Eisemann. Interactive indirect illumination using voxel cone tracing. In *Computer Graphics Forum*, volume 30, pages 1921–1930. Wiley Online Library, 2011.

[Dim07] Rouslan Dimitrov. Cascaded shadow maps. *Developer Documentation, NVIDIA Corp*, 2007.

[Dis15] Disney Enterprises. BRDF Explorer, 2015. <`http://www.disneyanimation.com/technology/brdf.html`>.

[DS05] Carsten Dachsbacher and Marc Stamminger. Reflective shadow maps. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games*, I3D '05, pages 203–231, New York, NY, USA, 2005. ACM.

[DSDD07] Carsten Dachsbacher, Marc Stamminger, George Drettakis, and Frédo Durand. Implicit visibility and antiradiance for interactive global illumination. In *ACM SIGGRAPH 2007 papers*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.

[EHK$^+$04] Klaus Engel, Markus Hadwiger, Joe M Kniss, Aaron E Lefohn, Christof Rezk Salama, and Daniel Weiskopf. Real-time volume graphics. In *ACM SIGGRAPH 2004 Course Notes*, page 29. ACM, 2004.

[Fer05] Randima Fernando. Percentage-closer soft shadows. In *ACM SIGGRAPH 2005 Sketches*, page 35. ACM, 2005.

[Gam15] Epic Games. Unreal engine 4, 2015. <`www.unrealengine.com`>.

[GLE15] GLEW Development Team. The opengl extension wrangler library, 2015. <`http://glew.sourceforge.net/`>.

[GLF15] GLFW Development Team. GLFW, 2015. <`http://www.glfw.org/`>.

[Gre03] Robin Green. Spherical harmonic lighting: The gritty details. In *Archives of the Game Developers Conference*, volume 56, 2003.

[HMY12] Takahiro Harada, Jay McKee, and Jason C Yang. Forward+: Bringing deferred lighting to the next level. 2012.

[HSHH07] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive k-d tree gpu raytracing. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, I3D '07, pages 167–174, New York, NY, USA, 2007. ACM.

[Kaj86] James T Kajiya. The rendering equation. In *ACM SIGGRAPH Computer Graphics*, volume 20, pages 143–150. ACM, 1986.

[KD10] Anton Kaplanyan and Carsten Dachsbacher. Cascaded light propagation volumes for real-time indirect illumination. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, I3D '10, pages 99–107, New York, NY, USA, 2010. ACM.

[Kel97] Alexander Keller. Instant radiosity. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '97, pages 49–56, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[KFC$^+$10] Jaroslav Křivánek, Marcos Fajardo, Per H Christensen, Eric Tabellion, Michael Bunnell, David Larsson, Anton Kaplanyan, B Levy, and RH Zhang. Global illumination across industries. *SIGGRAPH Courses*, 2010.

[Khr15a] Khronos Group. OpenGL, 2015. <`http://www.opengl.org`>.

[Khr15b] Khronos Group. OpenGL ES, 2015. <`http://www.khronos.org/opengles/`>.

[Lau10] Andrew Lauritzen. Deferred rendering for current and future rendering pipelines. *SIGGRAPH Course: Beyond Programmable Shading*, pages 1–34, 2010.

[McT04] Gary McTaggart. Half-life 2/valve source shading. *Valve Corporation*, 97, 2004.

[Mic15] Microsoft. DirectX graphics api, 2015. <`msdn.microsoft.com/en-us/library/windows/desktop/hh309466`>.

[MKC07] Ricardo Marroquim, Martin Kraus, and Paulo Roma Cavalcanti. Efficient point-based rendering using image reconstruction. In *PBG'07: Proceedings of the Eurographics Symposium on Point-Based Graphics*, pages 101–108, September 2007.

[Mor15] Morgan McGuire. Mcguire graphics data, 2015. <`http://graphics.cs.williams.edu/data`>.

[Oma15] Omar Cornut. ImGui - immediate mode gui library, 2015. <`https://github.com/ocornut/imgui`>.

[PH10] Matt Pharr and Greg Humphreys. *Physically based rendering: From theory to implementation.* Morgan Kaufmann, 2010.

[Res09] Alexander Reshetov. Morphological antialiasing. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 109–116. ACM, 2009.

[Res15] ResIL Developers. Resilient Image Library - successor to DevIL, 2015. <`http://sourceforge.net/projects/resil/`>.

[RGK$^+$08] Tobias Ritschel, Thorsten Grosch, Min H. Kim, Hans-Peter Seidel, Carsten Dachsbacher, and Jan Kautz. Imperfect Shadow Maps for Efficient Computation of Indirect Illumination. *ACM Trans. Graph. (Proc. of SIGGRAPH ASIA 2008)*, 27(5), 2008.

[RGS09] Tobias Ritschel, Thorsten Grosch, and Hans-Peter Seidel. Approximating dynamic global illumination in image space. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 75–82. ACM, 2009.

[RSC87] William T Reeves, David H Salesin, and Robert L Cook. Rendering antialiased shadows with depth maps. In *ACM SIGGRAPH Computer Graphics*, volume 21, pages 283–291. ACM, 1987.

[Slo08] Peter-Pike Sloan. Stupid spherical harmonics (sh) tricks. In *Game developers conference*, volume 9, 2008.

[t3d15] Tf3dm model repository, 2015. <`http://tf3dm.com/`>.

[Tec15] Unity Technologies. Unity 5, 2015. <`https://unity3d.com`>.

[tur15] TurboSquid model repository, 2015. <`http://www.turbosquid.com/`>.

[Wil78] Lance Williams. Casting curved shadows on curved surfaces. In *ACM Siggraph Computer Graphics*, volume 12, pages 270–274. ACM, 1978.

Appendix A

List of Abbreviations

API	Application Programming Interface
BRDF	Bidirectional Reflectance Distribution Function
CLPV	Cascaded Light Propagation Volumes
CPU	Central Processing Unit
g-buffer	Geometry Buffer
GPU	Graphics Processing Unit
GUI	Graphical User Interface
LPV	Light Propagation Volume
RSM	Reflective shadowmap
SH	Spherical harmonics
SM	Shadowmap
VPL	Virtual Point Light

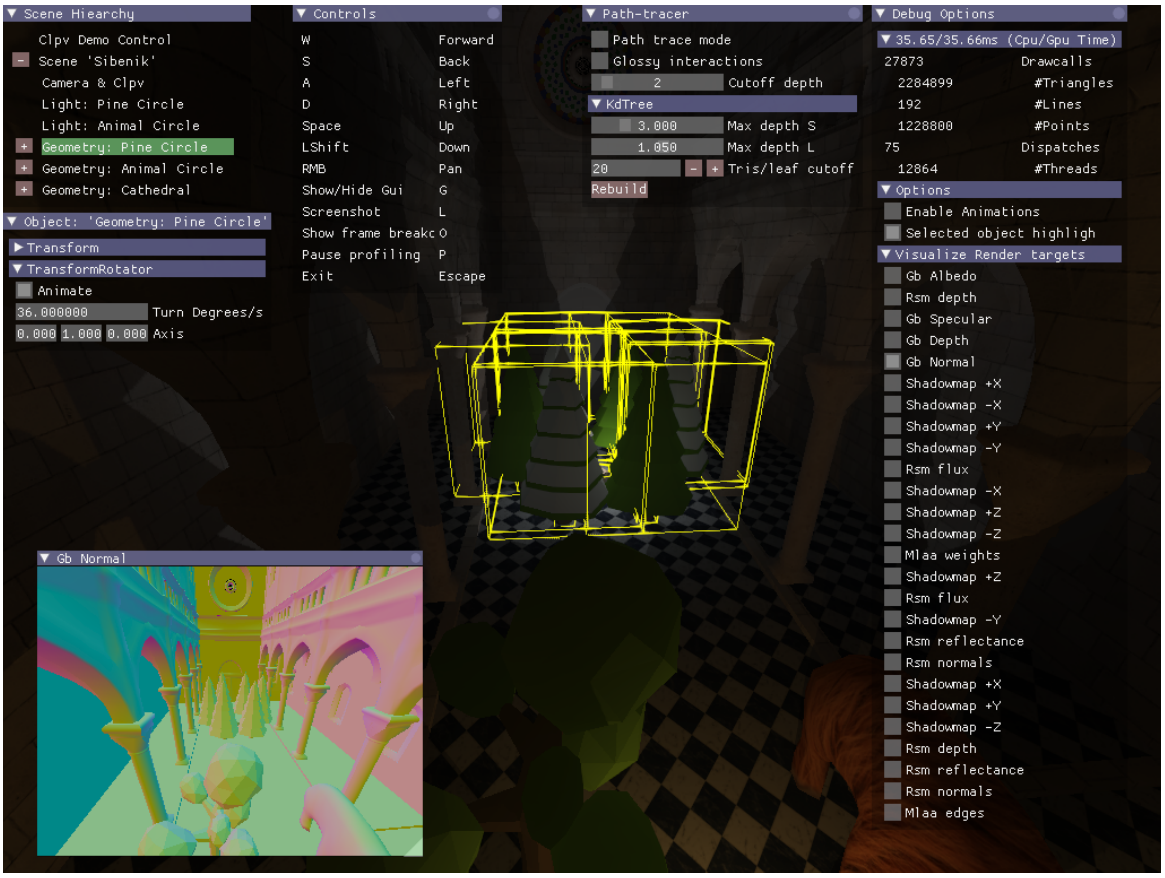
Appendix B

Nomenclature

$\vec{\omega}$	arbitrary direction
φ	azimuth
f_r	Bidirectional Reflection Distribution Function
$d\vec{\omega}$	differential solid angle
θ_i	incident angle
$\vec{\omega}_i$	incident direction
$\vec{\omega}_o$	outgoing direction
θ	polar angle
`Type`	programming language type system element or method
ρ	reflectance, albedo

Appendix C

Image Gallery



```
▼ Scene Hiearchy
  Clpv Demo Control
- Scene 'Sibenik'
   Camera & Clpv
   Light: Pine Circle
   Light: Animal Circle
+  Geometry: Pine Circle
+  Geometry: Animal Circle
+  Geometry: Cathedral

▼ Object: 'Geometry: Pine Circle'
▶ Transform
▼ TransformRotator
  Animate
  36.000000            Turn Degrees/s
  0.000 1.000 0.000    Axis
```

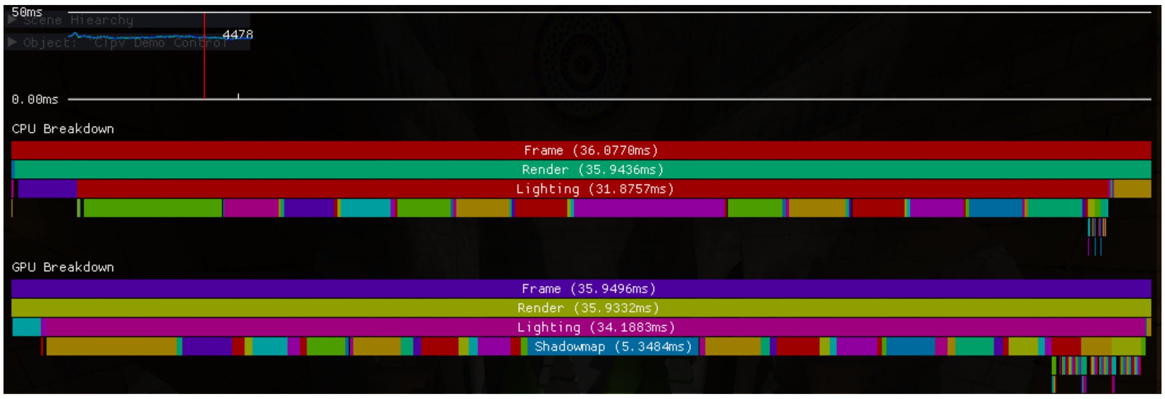
```
▼ Controls
W                 Forward
S                 Back
A                 Left
D                 Right
Space             Up
LShift            Down
RMB               Pan
Show/Hide Gui     G
Screenshot        L
Show frame breakc O
Pause profiling   P
Exit              Escape
```

```
▼ Path-tracer
  Path trace mode
  Glossy interactions
       2          Cutoff depth
▼ KdTree
     3.000        Max depth S
     1.050        Max depth L
  20        - +   Tris/leaf cutoff
Rebuild
```

```
▼ Debug Options
▼ 35.65/35.66ms (Cpu/Gpu Time)
 27873             Drawcalls
  2284899          #Triangles
  192              #Lines
  1228800          #Points
 75                Dispatches
  12864            #Threads
▼ Options
  Enable Animations
  Selected object highligh
▼ Visualize Render targets
  Gb Albedo
  Rsm depth
  Gb Specular
  Gb Depth
  Gb Normal
  Shadowmap +X
  Shadowmap -X
  Shadowmap +Y
  Shadowmap -Y
  Rsm flux
  Shadowmap -X
  Shadowmap +Z
  Shadowmap -Z
  Mlaa weights
  Shadowmap +Z
  Rsm flux
  Shadowmap -Y
  Rsm reflectance
  Rsm normals
  Shadowmap +X
  Shadowmap +Y
  Shadowmap -Z
  Rsm depth
  Rsm reflectance
  Rsm normals
  Mlaa edges
```

```
▼ Gb Normal
```

Figure C.1: The application GUI and debug features. Scene hierarchy inspector and active object detail window (top-left). Controls and Path-tracer windows (top-center). Debug info window with latest frame time measurements and render target visualization panel (top-right). Selected scene object bounds rendered with wireframe (center). Visualized g-buffer normal channel render target (bottom-left). See the manual.pdf document enclosed with the project for details.



```
50ms
        4478
0.00ms

CPU Breakdown
Frame (36.0770ms)
Render (35.9436ms)
Lighting (31.8757ms)

GPU Breakdown
Frame (35.9496ms)
Render (35.9332ms)
Lighting (34.1883ms)
Shadowmap (5.3484ms)
```

Figure C.2: CPU and GPU time measurements can be displayed in a hierarchical visualization.

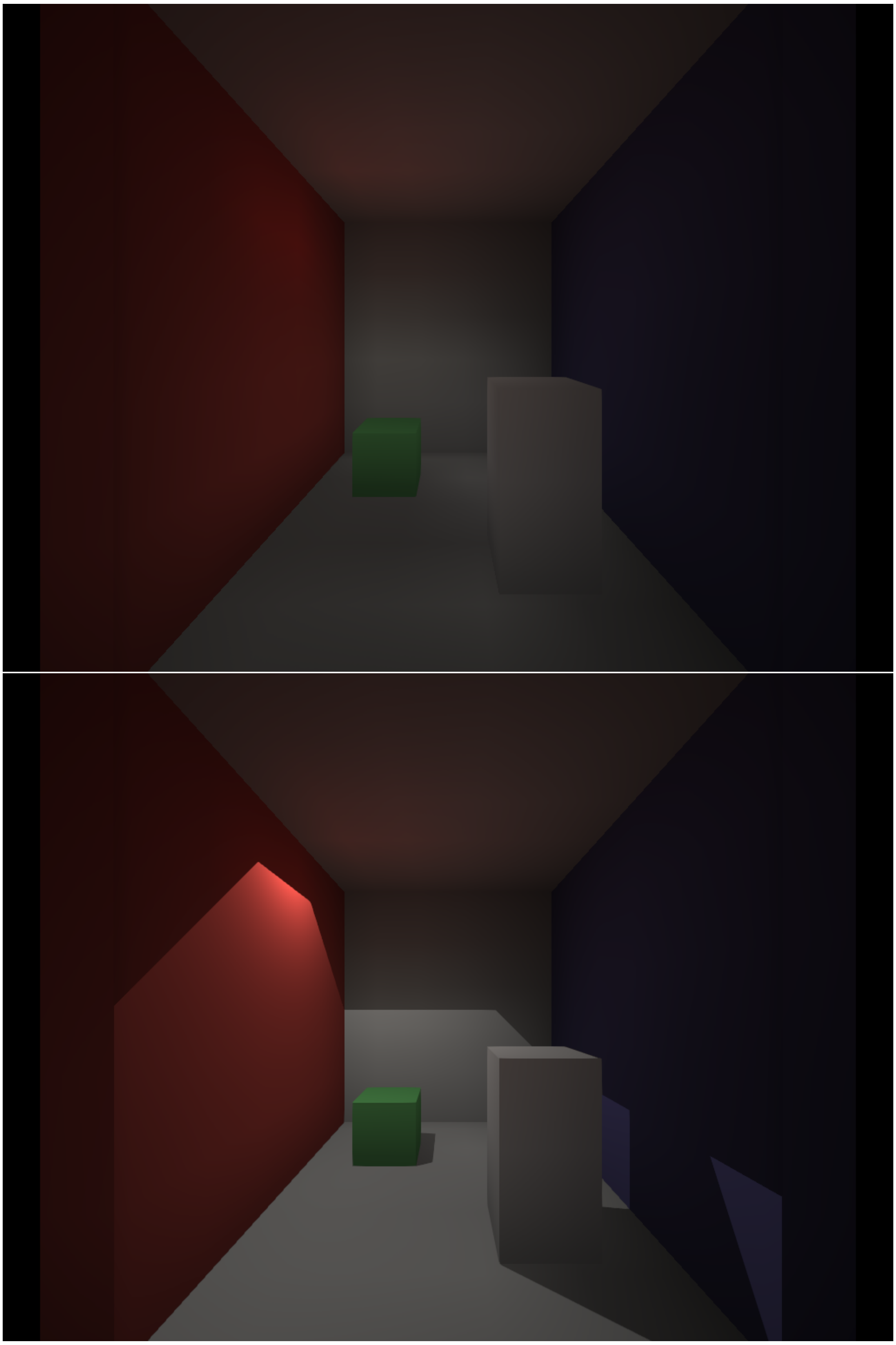


Figure C.3: Cornell scene global illumination via CLPV. (top) Indirect illumination. (bottom) Direct and indirect illumination.

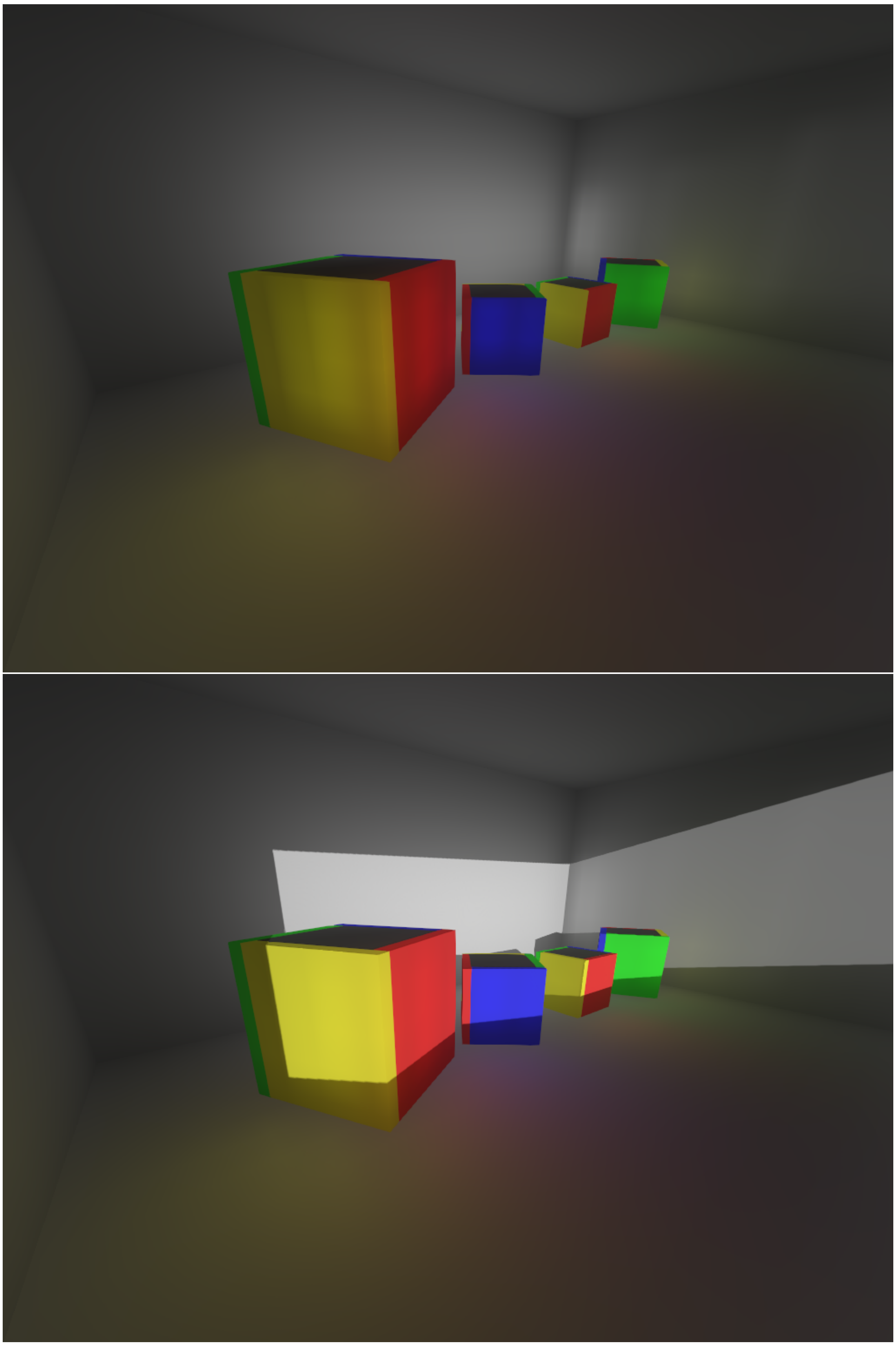


Figure C.4: B&G scene global illumination via CLPV. (top) Indirect illumination. (bottom) Direct and indirect illumination.

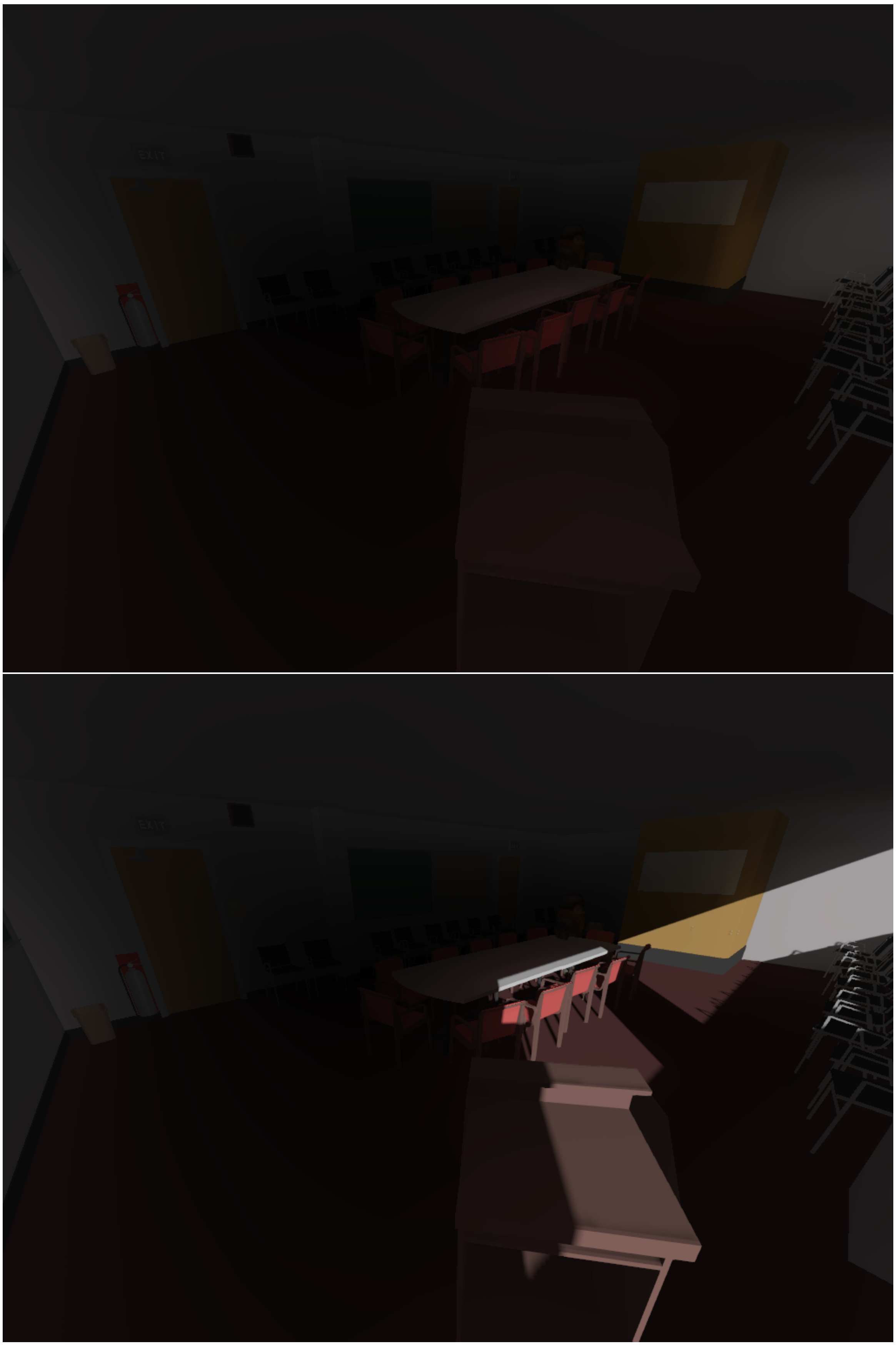


Figure C.5: Conference scene global illumination via CLPV. (top) Indirect illumination. (bottom) Direct and indirect illumination.

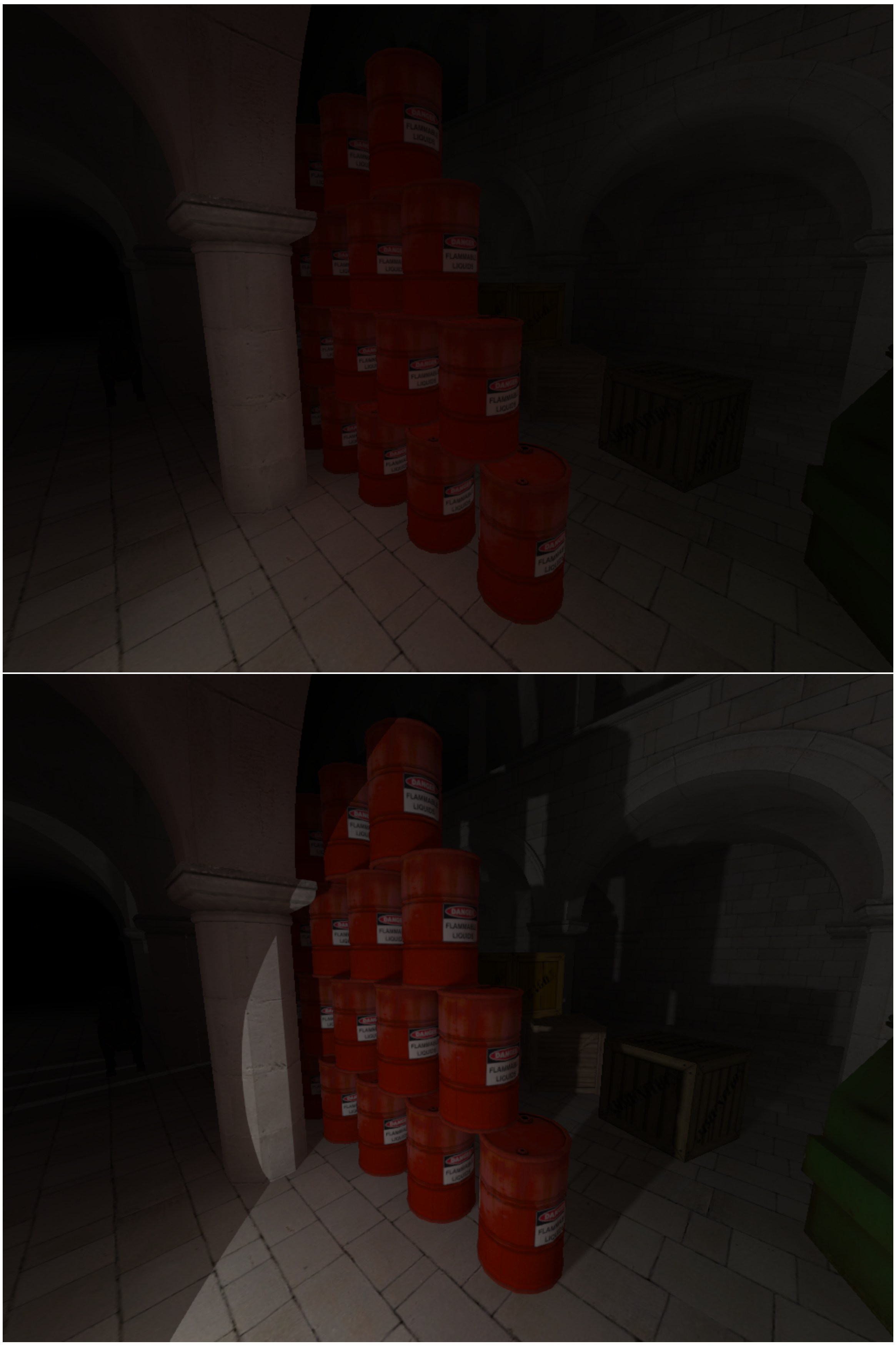


Figure C.6: Dabrovic scene global illumination via CLPV. (top) Indirect illumination. (bottom) Direct and indirect illumination.

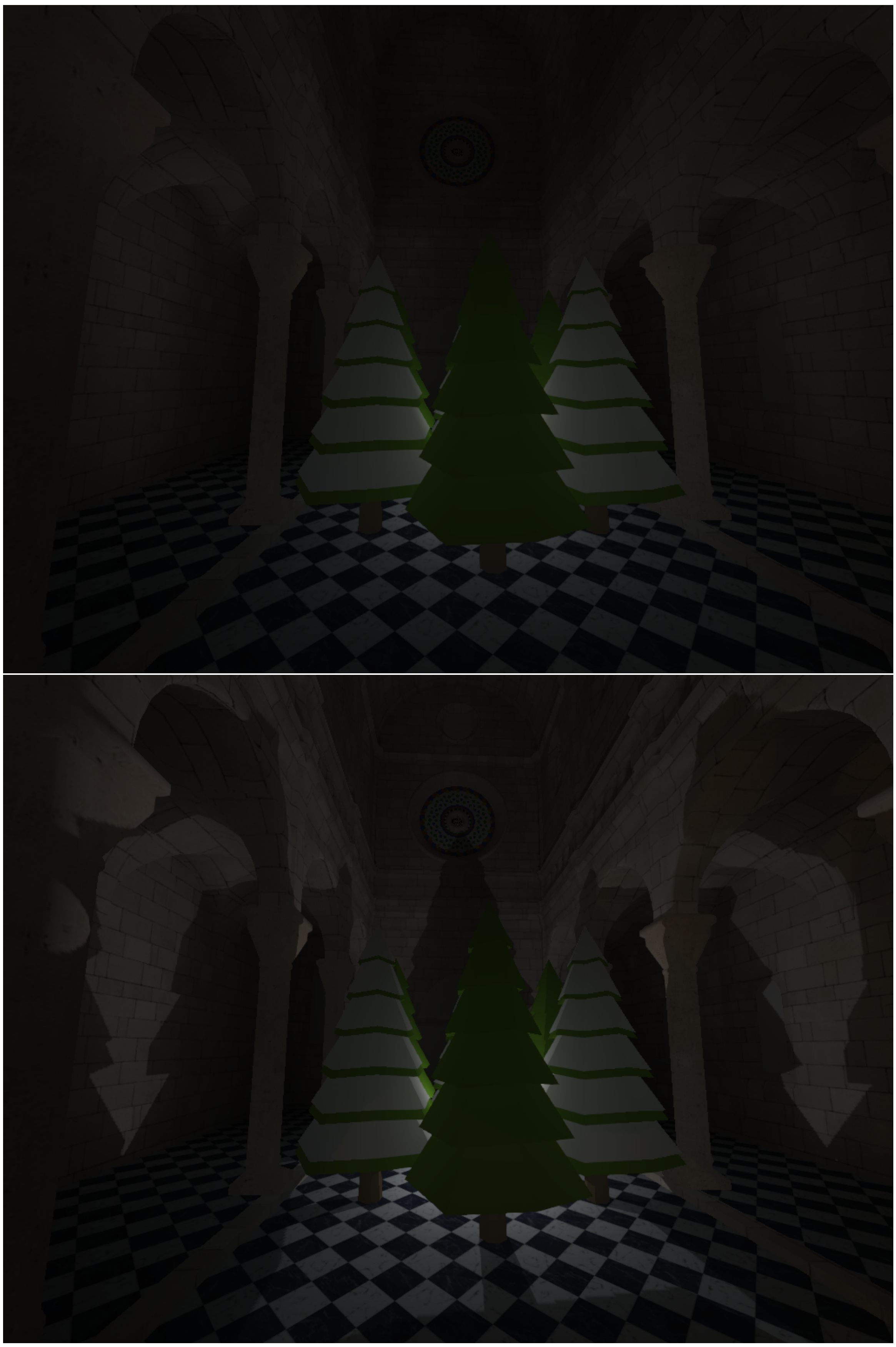


Figure C.7: Sibenik scene global illumination via CLPV. (top) Indirect illumination. (bottom) Direct and indirect illumination.

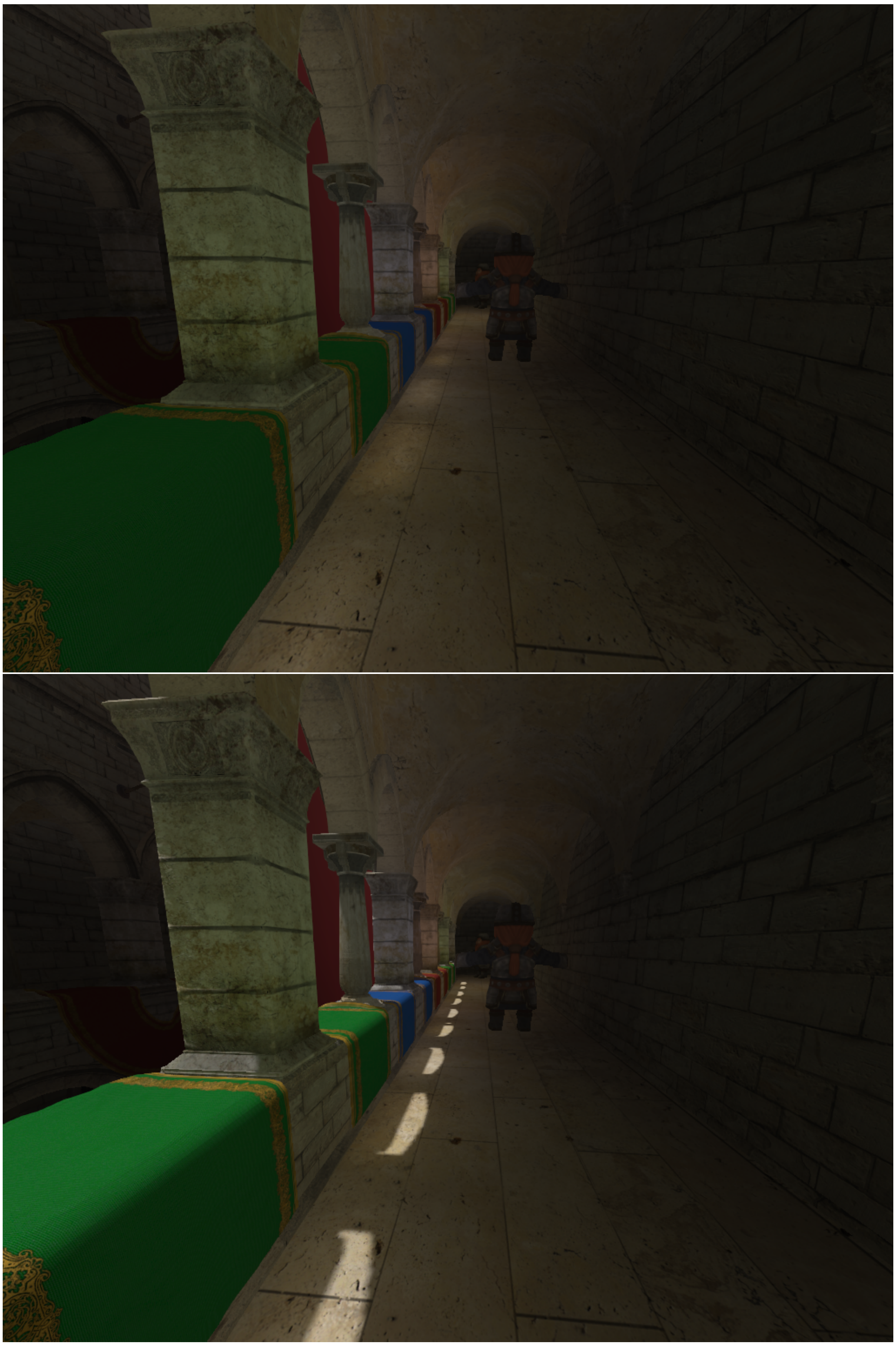


Figure C.8: Sponza scene corridor global illumination via CLPV. (top) Indirect illumination. (bottom) Direct and indirect illumination.

Appendix D

Contents of Attached CD

```
sefcipe2_thesis/         - project directory
|--data/                 - application resources
|  |--models/            - 3D model database
|  |--prefabs/           - definitions of common scene objects
|  |--scenes/            - definitions of testing scenes
|  |--shaders/           - GPU program sources
|  `--textutes/          - scene skybox textures
|--doc/html/             - Doxygen-generated code documentation
|--msvs/                 - Microsoft Visual Studio 2013 project resources
|--src/                  - source code
|  |--clpvdemo/          - demo application specific source code
|  `--tse/               - reusable framework source code
|--thesis/               - pdf and latex sources of this thesis
|--win32release/         - Win32 binaries of the demo application
|--clpvdemo.bat          - launch script for enclosed Win32 binaries
|--contents.txt          - contents of the project directory
`--manual.pdf            - demo application manual
```