AN ABSTRACT OF THE THESIS OF

Randall Rauwendaal for the degree of Doctor of Philosophy in Computer Science presented on August 26, 2013.

Title: Voxel Based Indirect Illumination using Spherical Harmonics

Abstract approved: _____

Michael J. Bailey

Realistic (ideally photorealistic) real-time rendering has remained an elusive goal in computer graphics. While photorealistic rendering has certainly been achieved at the expense of tremendous computational resources and corresponding rendering times; real-time rendering typically must accept a great number of compromises to achieve adequate performance, such as aliasing artifacts, the absence of secondary illumination effects such as diffuse inter-reflection and realistic specular reflections, and a lack of geometric detail. This dissertation demonstrates solutions which reduce the computational cost of solving the rendering equation through a series of strategic approximations which are well suited to the massively parallel nature of current consumer GPUs and their integrated filtering hardware. Firstly, we discretize scene geometry, using a novel and highly efficient voxelization technique. From the voxelization, we efficiently generate a hierarchical representation of scene geometry. We then use this hierarchical representation as a proxy for computation

of indirect illumination using a technique called *Voxel Cone Tracing*. Finally we explore the storage of both isotropic and anisotropic functions within our hierarchical scene proxy, and evaluate the usage of low order spherical harmonics as a more suitable approximation of radiance.

Voxel Based Indirect Illumination using Spherical Harmonics

by

Randall Rauwendaal

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented August 26, 2013
Commencement June 2014

Doctor of Philosophy thesis of Randall Rauwendaal presented on August 26, 2013.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Randall Rauwendaal, Author

ACKNOWLEDGEMENTS

I would like to thank Intel Corporation's Visual Computing Academic Program for their generous funding. My advisor, Mike Bailey, for finding funding for me throughout my graduate career. My old neighbor, Geoff Dalgas, of StackOverflow, for donating an old, yet still powerful, computer to the cause. And most of all I would like to thank my wife, Leslie Rauwendaal, for her infinite patience and valuable input, without which, this never would have been possible.

Much of the decision to attend graduate school stemmed from the wonderful time I spent as an undergraduate researcher at the Institute for Data Analysis and Visualization at UC Davis. It was a place of open collaboration and illuminating discussions. Unfortunately, my graduate career proved to be a much different experience; it was ever more isolating and ever more trying. The fact that this dissertation exists at all is a testament to an unrelenting determination that I inherited from my wonderful parents, who would both stubbornly claim that such a trait came from their side of the family. It is but one of the many positive traits I owe them, along with a sometimes inconvenient tendency towards forthrightness and honesty.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

LIST OF TABLES

LIST OF APPENDIX FIGURES

Chapter 1: Introduction

We are confronted with a world rich in visual information. Everyday we perceive complex and beautiful interactions of light and materials played out in their expression of intricate optical phenomena. It is our aim to conduct research towards the simulation of these effects using physical models to not only produce convincing results, but to do so at interactive rates. Rendering algorithms have long been able to produce realistic images, see Figure 1.1, and with the steady increase in computational resources, have been able to render scenes with greater detail and complexity. But real-time simulation of complex, fully dynamic scenes, has remained an elusive goal.

1.1 Motivation

Our motivation is present in the beauty of the world around us. We are compelled to attempt to synthesize all the complex interactions demonstrated in Figure 1.2 that nature computes implicitly. To this end we must simulate both *direct* and *indirect* light interactions. That is, the light that falls directly on a point in the scene and the light that may have scattered off of any other object anywhere else in the scene to arrive at the same point. The sum of both direct and indirect light interactions results in Global Illumination.

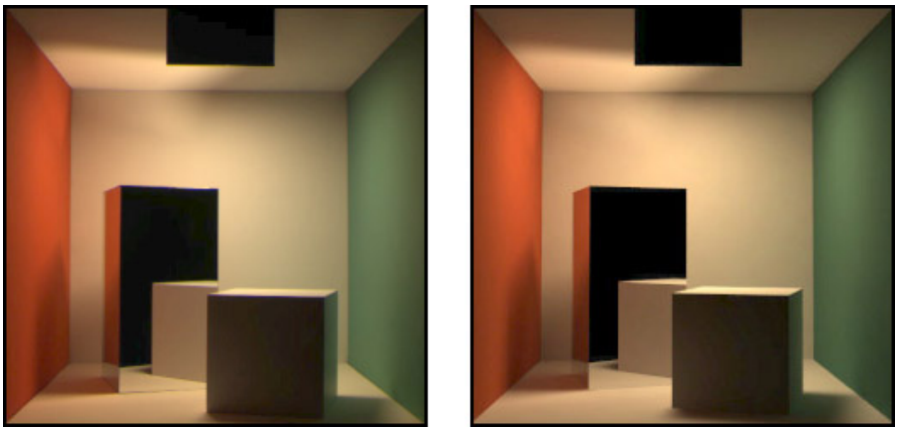


Figure 1.1: On the left a photograph of the Cornell Box, and on the right a simulated rendering of the same box. (Image courtesy of Cornell University)

The difficulty in computing global illumination comes from the fact that light can bounce off virtually all objects, thus to compute the color at a point in the scene we must integrate over all incident illumination. This is the approach as formulated by the Rendering Equation by Kajiya [Kaj86], which has come to be the defining equation for physically based rendering and Global Illumination, indeed all physically based rendering research is generally about solving this single equation. However, the Rendering Equation is *extremely* hard to solve, and likely impossible for many scenes. That the Rendering Equation is difficult to solve is due to several reasons; first, it requires global information about the scene (all geometry, materials, and light information), second, it is a recurrence relation describing a potentially infinite recursion, and third, it is an integral over a continuous space. Thus, it follows that most attempts to "solve" the rendering equation are really attempts to find an approximate solution, and that these approximate solutions are often based upon the discretization of continuous domains, such as lighting,

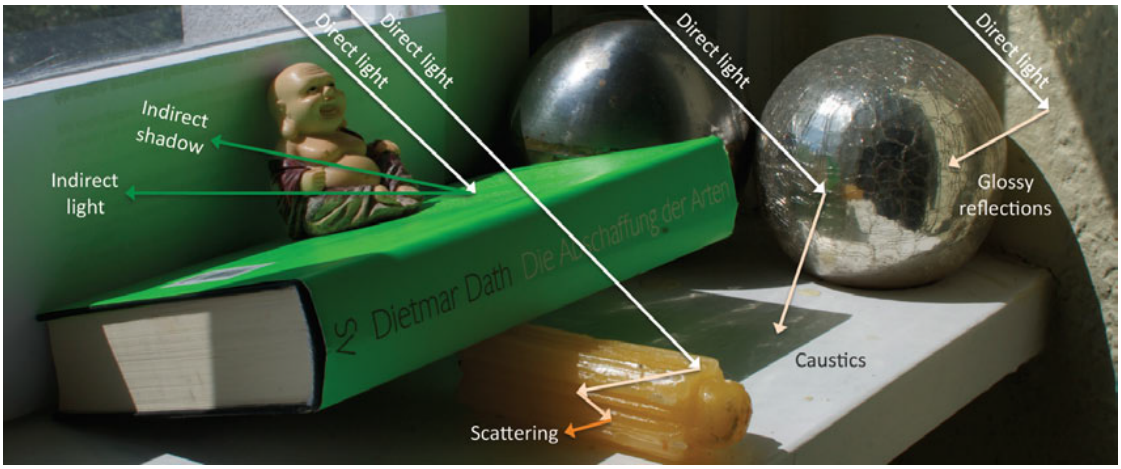


Figure 1.2: Image demonstrating several global illumination effects; multiple diffuse and specular bounces, caustics and scattering. (Image courtesy of Tobias Ritschel [RDGK12]).

scene geometry, and material properties.

The costs associated with global illumination have long prevented it from being employed in the context of real-time rendering. Recently, however, as the power of modern GPUs has continued to increase at a rate that defies Moore's law, and as rendering algorithms are adapted to their massively parallel architectures, we are beginning to see real-time simulation of effects previously relegated to slow, offline renderers. These effects, such as soft shadows, diffuse interreflection (color bleeding), caustics, and refractions all greatly enhance the visual perception and realism of a scene. Our work has focused on rendering techniques which enable these indirect illumination techniques while maintaining framerates suitable for real-time rendering.

1.2 Contributions

We make contributions in the usage of spherical harmonics as a representation for the exitant radiance of discretized scene geometry, and also detail methods of constructing an accurate hierarchy of approximate radiance values suitable for efficient sampling using cone tracing methods. Additionally, we build upon our recently developed hybrid computational voxelization techniques (see [Rau12]), and detail their extension towards the inclusion of appropriate voxel attributes. We detail a new technique for mipmapping a hierarchy of spherical harmonic values. We detail two methods for performing voxel cone tracing, each suitable to a particular component of indirect illumination. Further, we construct an efficient rendering-pipeline which enables high-quality rendering with indirect illumination of fully dynamic scenes, including non-static geometry, lighting, and materials.

1.3 Outline

This dissertation is organized as follows. Background information concerning the theory of light transport is described in Chapter 2. We detail the components of the rendering equation, (see equation 2.10), and their application towards the accurate simulation of the interaction of light and materials. Additionally, we review useful properties of Spherical Harmonics (Section 2.2), and discuss the evolution of GPU hardware in Section 2.3. We review related real-time global illumination work in Chapter 3, as well as related work in voxelization (Section 3.7)

In Chapter 4, we describe the process of efficiently discretizing scene geom-

etry, using our novel voxelization approach. Following voxelization, we discuss voxel storage formats, voxel sampling techniques, and voxel mipmapping strategies (Chapter 5). Chapter 5 is divided into a discussion of: the necessary voxel attributes to enable both isotropic and anisotropic radiance storage, and the adaptations required to enable these in our voxelization approach (Section 5.1); appropriate sampling strategies for each voxel storage format (Section 5.2); and a description of an optimized mipmapping approach and how to create a hierarchy of radiance values (Section 5.3).

Chapter 6 provides a straightforward introduction to the geometry of voxel cone tracing, followed by its application to several notable illumination effects. These include soft shadows (Section 6.2), ambient occlusion(Section 6.3), diffuse interreflection (Section 6.4), and glossy specular reflections (Section 6.5).

Chapter 7 pulls together all the effects from Chapter 6 and discusses the process used to create a final voxel based rendering pipeline. Timings and results for the final image generation are also presented in this chapter, while timings for individual effects are presented in the associated sections of Chapter 6. Additional images are also listed in Appendix A to further corroborate presented results, while snippets of useful shader code are listed in Appendix B.

Finally, in Chapter 8, we present our conclusions, discuss several directions for further optimizations and numerous potential avenues for future research based upon the foundation of this work.

Chapter 2: Background

In this Chapter we introduce the background materials that form the building blocks for this thesis. In Section 2.1 we discuss light transport theory as it applies to global illumination rendering. In Section 2.1.3 we discuss the reflectance models used to define materials. In Section 2.1.4 we discuss the rendering equation, from which virtually all rendering research springs. In Section 2.2 we review spherical harmonics as they are an integral component of the research conducted in this dissertation. Finally, in Section 2.3 we discuss the evolution of GPU architecture along with the APIs that enable massively parallel graphics computation.

2.1 Light Transport Theory

The goal of rendering algorithms is to produce synthetic images of virtual scenes described by three-dimensional geometry, coupled with lighting and material information. Global illumination rendering algorithms create these images via a physically accurate simulation of how light propagates through the virtual world. This simulation is described by *light transport* theory which details how light, or *energy*, is emitted from light sources, scattered by elements in the scene, and ultimately arrives at the viewer, or "camera," all of which is neatly encompassed by the rendering equation, (see Section 2.1.4). In this section, we first explore

the necessary background and develop the terminology needed to understand light transport and the rendering equation, and discuss the representation of materials using the *bidirectional scattering distribution function*.

2.1.1 The Nature of Light

In computer graphics we make many simplifying assumptions about the nature of light. For instance, while the *wavelength* of light determines its perceived color, typically we concern ourselves only with the wavelengths corresponding to the primary colors red, green, and blue (RGB). This simplification precludes the physically accurate simulation of certain effects such as *dispersion* and *fluorescence*. Additionally, we typically rely upon the simplest model of light, that of ray optics; there also exists models in increasing order of complexity: wave optics, electromagnetic optics, and quantum optics [ST07]. The model of ray optics comes with the simplifying assumption that light traverses space instantaneously, which dictates that light can only be emitted, reflected, and transmitted. Despite this, we can still accurately reproduce a wide range of physical phenomena.

2.1.2 Radiometry

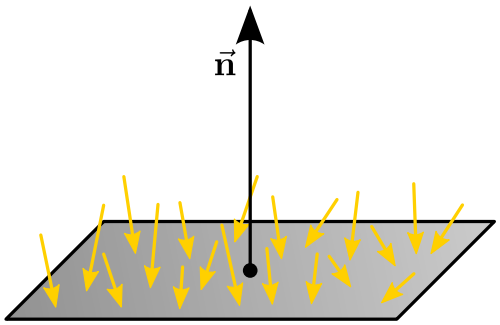


Figure 2.1: *Flux*, shown above, measures the amount of light falling on a surface area from all directions and is measured in watts [W], while *radiant exitance* (radiosity) measures the amount of light (flux) leaving a point in all directions and is measured in watts per meter squared[$\mathrm{W} \cdot \mathrm{m}^{-2}$].

Radiometry describes the physical measurement of electromagnetic radiation. In the context of this dissertation we use the term, *light*, to describe the visible spectrum of electromagnetic waves. Radiometric units are useful in describing global illumination algorithms in that they define a common terminology for the physical quantities of light. In this section we review the basic radiometric quantities.

The first radiometric quantity, *radiant energy*, denoted Q, describes the energy of light in joules [J]. Differentiating radiant energy in time leads us to *radiant flux* (see Figure 2.1), denoted $\Phi = \frac{dQ}{dt}$, which expresses the amount of energy emitted by a surface over time in watts [W]. When buying a light bulb, its listed wattage also describes its radiant flux [$\mathrm{W} = \mathrm{J} \cdot \mathrm{s}^{-1}$]. Integrating radiant flux over time leads back to radiant energy, whereas differentiating radiant flux in area leads to *irradiance*, and differentiating in solid angle leads to *intensity*.

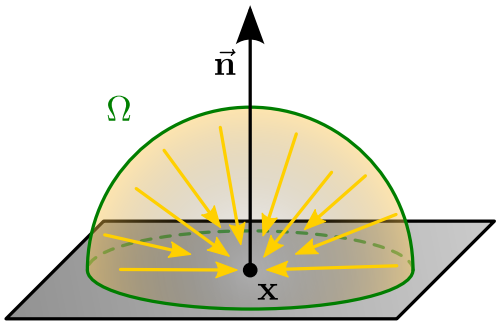


Figure 2.2: Irradiance integrates the total incident light over the hemisphere Ω and is measured in watts per meter squared $[\mathrm{W} \cdot \mathrm{m}^{-2}]$.

Irradiance, denoted E, expresses the amount of incident power hitting a surface per unit area. Hence, it has units of $[\mathrm{W} \cdot \mathrm{m}^{-2}]$. Integrating radiance over area leads back to flux, while differentiating irradiance in direction will lead to *radiance*. The term irradiance implies a measure of the flux *arriving* at a surface location $\mathbf{x}$. Conversely, the terms *radiant exitance* (M) or *radiosity* (B) are used instead if the flux is leaving a surface. More explicitly:

$$E(\mathbf{x}) = \frac{d\Phi_{in}(\mathbf{x})}{dA(\mathbf{x})} \qquad \left[\frac{\mathrm{W}}{\mathrm{m}^2}\right] \tag{2.1}$$

$$M(\mathbf{x}) = B(\mathbf{x}) = \frac{d\Phi_{out}(\mathbf{x})}{dA(\mathbf{x})} \qquad \left[\frac{\mathrm{W}}{\mathrm{m}^2}\right] \tag{2.2}$$

Radiance, denoted L, is perhaps the most important radiometric quantity for global illumination. It is closest to what we commonly conceive of as "light." Radiance expresses how much light arrives from a differential direction $d\vec{\omega}$ onto a

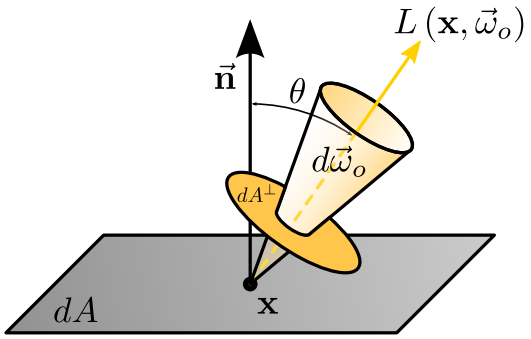


Figure 2.3: Radiance expresses the amount of light arriving at a point from a differential solid angle and has units $[\mathrm{W} \cdot \mathrm{sr}^{-1} \cdot \mathrm{m}^{-2}]$.

hypothetical differential area *perpendicular* to that direction $dA^\perp$. Radiance has units of $[\mathrm{W} \cdot \mathrm{sr}^{-1} \cdot \mathrm{m}^{-2}]$. This five-dimensional function of position and direction can be expressed as:

$$L(\mathbf{x}, \vec{\omega}) = \frac{d^2\Phi(\mathbf{x}, \vec{\omega})}{d\vec{\omega} dA^\perp(\mathbf{x})} = \frac{d^2\Phi(\mathbf{x}, \vec{\omega})}{(\vec{\mathbf{n}} \cdot \vec{\omega}) dA(\mathbf{x})} = \frac{d^2\Phi(\mathbf{x}, \vec{\omega})}{\cos\theta d\vec{\omega} dA(\mathbf{x})} \qquad \frac{\mathrm{W}}{\mathrm{m}^2\mathrm{sr}} \tag{2.3}$$

In this dissertation we employ the notation $L(\mathbf{x} \leftarrow \vec{\omega})$ for incident radiance that arrives at a point $\mathbf{x}$ from direction $\vec{\omega}$, and similarly the notation $L(\mathbf{x} \rightarrow \vec{\omega})$ for exitant radiance leaving $\mathbf{x}$ in direction $\vec{\omega}$. The radiance invariance law states that radiance does not change along a ray in a vacuum, that is $L(\mathbf{x} \leftarrow \vec{\omega}) = L(\mathbf{x} \rightarrow \vec{\omega})$. We can express the previously defined terms, irradiance and radiosity (radiant exitance) in terms of radiance as:

$$E(x) = \int_\Omega L(\mathbf{x} \leftarrow \vec{\omega})(\vec{\mathbf{n}} \cdot \vec{\omega}) d\vec{\omega} = \int_\Omega L(\mathbf{x} \leftarrow \vec{\omega}) \cos\theta d\vec{\omega} \tag{2.4}$$

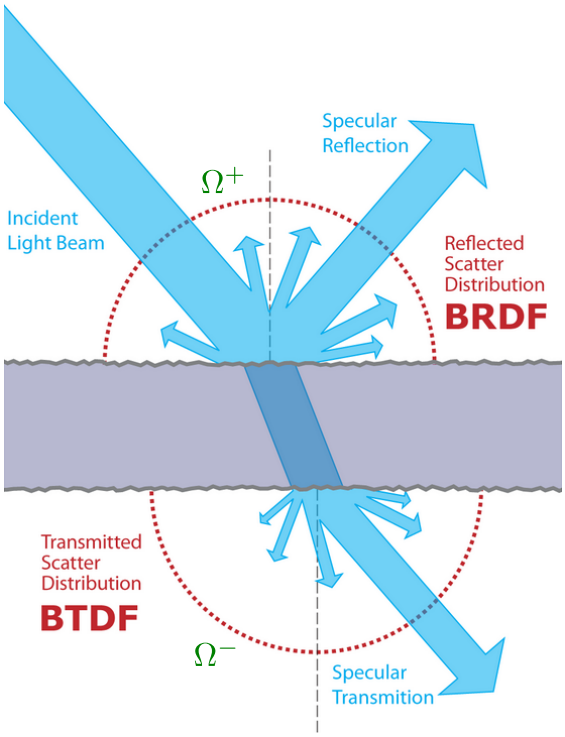


Figure 2.4: The different forms of the *bidirectional scattering distribution function*. Above, the *bidirectional reflection distribution function* (BRDF) describes the light reflected at a point of the surface, while below, the *bidirectional transmission distribution function* (BTDF) describes the light transmitted through the material (Original image courtesy of Wikipedia).

$$M(x) = B(x) = \int_{\Omega} L(\mathbf{x} \to \vec{\omega})(\vec{\mathbf{n}} \cdot \vec{\omega})\, d\vec{\omega} = \int_{\Omega} L(\mathbf{x} \to \vec{\omega}) \cos\theta d\vec{\omega} \tag{2.5}$$

Where Ω is the visible hemisphere, intuitively we are integrating over all incident radiance adjusted for projected area.

2.1.3 Materials

The interaction of lights and surfaces is a crucial component of rendering. The materials which make up these surfaces determine the manner in which the light is "scattered." This "scattering function," denoted by f_s, is referred to as the *bidirectional scattering distribution function*, or BSDF, which generalizes the *bidirectional reflection distribution function* (BRDF f_r) introduced by Nicodemus et al. [NN77], and the *bidirectional transmission distribution function,* BTDF f_t, (cf. Figure 2.4). The main distinction between the BRDF and the BTDF is which hemisphere they are integrated over, the positive hemisphere Ω^+ for the BRDF, and the negative hemisphere Ω^- for the BTDF. Additionally, the *bidirectional subsurface scattering reflection distribution function* (BSSRDF, f_{ss}) introduced by Jensen et al. [JMLH01] describes light that enters a surface at one point, $\mathbf{x}_i$, and exits at another point, $\mathbf{x}_o$, after traveling beneath the surface of the material, hence *subsurface*. The BSDF is defined as the ratio of scattered exitant radiance to incident irradiance. More formally, the BSDF describes the appearance of a surface at a point $\mathbf{x}$ when viewed from a direction $\vec{\omega}_o$ while being illuminated by a light from direction $\vec{\omega}_i$:

$$f_s\left(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_o\right) \equiv \frac{dL\left(\mathbf{x} \rightarrow \vec{\omega}_o\right)}{dE\left(\mathbf{x} \leftarrow \vec{\omega}_i\right)} = \frac{dL\left(\mathbf{x} \rightarrow \vec{\omega}_o\right)}{L\left(\mathbf{x} \leftarrow \vec{\omega}_i\right)\left(\vec{\mathbf{n}} \cdot \vec{\omega}_i\right) d\vec{\omega}_i} \qquad \left[\frac{1}{sr}\right] \tag{2.6}$$

BSDFs provide a more physically plausible model of reflectance as they are constrained by the laws of thermodynamics [Vea98]. This means that they are non-

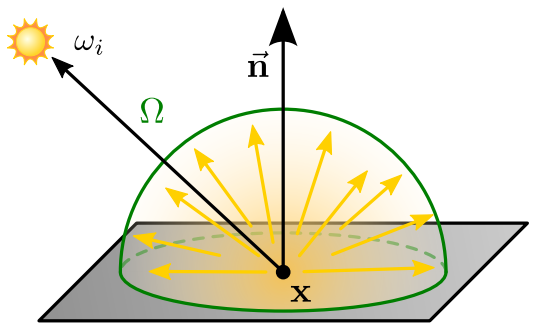


Figure 2.5: A perfectly diffuse, or *Lambertian*, BRDF.

negative, as a surface cannot absorb more light than falls on it:

$$f_s\left(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_o\right) \geq 0 \tag{2.7}$$

Additionally the BSDF is *energy conserving*, that is a surface cannot reflect more light than it receives:

$$\int_{\Omega} f_s\left(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_o\right)\left(\vec{\mathbf{n}} \cdot \vec{\omega}_o\right) d\vec{\omega}_o \leq 1, \qquad \forall \omega_i \tag{2.8}$$

Lastly, the BSDF obeys the *Helmholtz reciprocity* principle, which means that the value of the BSDF stays the same when the incident and outgoing directions are swapped:

$$f_s\left(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_o\right) = f_s\left(\mathbf{x}, \vec{\omega}_o, \vec{\omega}_i\right) \tag{2.9}$$

This allows graphics algorithms to follows rays in either direction, either from the viewer towards the light, or from the light towards the viewer.

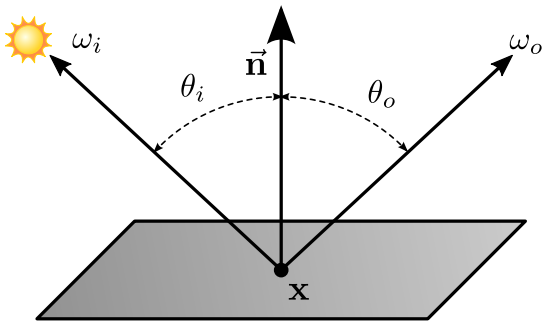


Figure 2.6: A perfectly specular BRDF, or "mirror."

There are many classes of BSDF. *Isotropic* BSDFs scatter light evenly in all directions, and thus depend on only one input direction, $\vec{\omega}_i$, see Figure 2.5, this is also known as a *Lambertian* BRDF, and commonly used for modeling diffuse surfaces. The opposite of a perfectly diffuse surface is a perfectly specular surface, that is, a surface where $\vec{\omega}_o$ is a perfect reflection of $\vec{\omega}_i$ about the normal $\vec{\mathbf{n}}$, this is what we commonly refer to as a "mirror," see Figure 2.6.

Any surface that is not perfectly diffuse or a mirror is "glossy," and a representative BRDF exists somewhere between specular and Lambertian, see Figure 2.7.

2.1.4 The Rendering Equation

The rendering equation as formulated by Kajiya in 1986 [Kaj86] was derived from previous research into radiative heat transfer [HSM10]. Fundamentally, it attempts to find the illumination, or more intuitively, "brightness," at each point by eval-

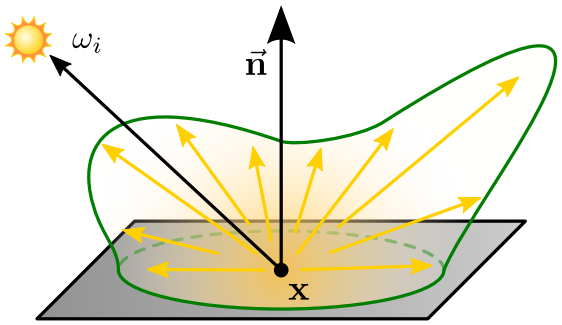


Figure 2.7: Most materials exist somewhere between perfectly diffuse and perfectly specular, we refer to such as "glossy" BRDFs.

uating everything that can be seen from that point. The illumination of these visible surfaces is in turn evaluated in the same manner, leading to a recursive formulation for the evaluation of illumination.

2.1.4.1 Hemispherical Formulation

In its simplest form, the rendering equation merely expresses the outgoing radiance, $L(\mathbf{x} \to \vec{\omega}_o)$, of any point $\mathbf{x}$ in a scene as the sum of the emitted radiance, L_e, at the point $\mathbf{x}$, and the reflected radiance, L_r, at the point:

$$\underbrace{L(\mathbf{x} \to \vec{\omega}_o)}_{\text{outgoing}} = \underbrace{L_e(\mathbf{x} \to \vec{\omega}_o)}_{\text{emitted}} + \underbrace{L_r(\mathbf{x} \to \vec{\omega}_o)}_{\text{reflected}} \tag{2.10}$$

Expanding upon the reflected term in Equation 2.10 with surface geometry and reflectance functions (described in section 2.1.3) results in the hemispherical form

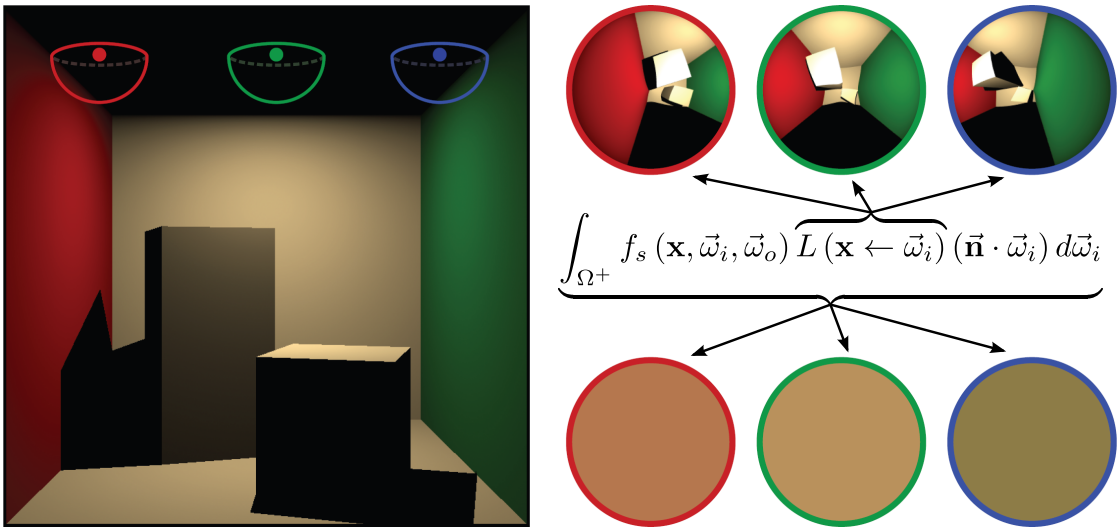


Figure 2.8: The exitant radiance at a point on a surface depends on the incident radiance over the hemisphere. The incident radiance field resembles a "fisheye" view from the point, while the exitant radiance is the integral of this value of the entire hemisphere. (Original image courtesy of Wojiech Jarosz [Jar08]).

of the rendering equation:

$$\underbrace{L(\mathbf{x} \rightarrow \vec{\omega}_o)}_{\text{outgoing}} = \underbrace{L_e(\mathbf{x} \rightarrow \vec{\omega}_o)}_{\text{emitted}} + \underbrace{\int_{\Omega^+} L(\mathbf{x} \leftarrow \vec{\omega}_i) f_s(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_o) (\vec{\mathbf{n}} \cdot \vec{\omega}_i) d\vec{\omega}_i}_{\text{reflected}} \tag{2.11}$$

Since outgoing radiance at one point is dependent on the outgoing radiance at all other visible points in the scene, this means that the radiance, L, is defined in terms of its own integral, which we observe by noting that it appears on both sides of equation 2.11. We illustrate this distinction in Figure 2.8. This form of recursive integral is known as a Fredholm integral of the second kind. Save in only the most trivial cases, an analytic solution of such an integral is impossible.

Making this an even more challenging problem is the fact that the rendering

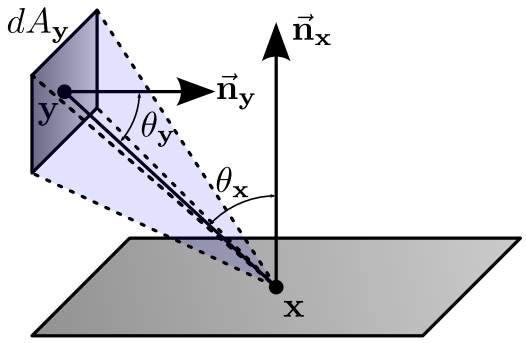


Figure 2.9: The geometry term in the area formulation of the rendering equation describes the relation of energy transfer between differential surfaces, $dA_{\mathbf{x}}$ and $dA_{\mathbf{y}}$, based on their relative angle and distance.

equation performs an integration of a continuous domain. To fully appreciate this, one must consider the fact that most computer science problems are considered "hard" when finding a solution takes exponential time on an input n. However, this assumes a discrete problem domain; for a continuous domain, n isn't even *finite* since the domain itself requires considering an infinite amount of input. Thus, if solving the rendering equation were merely exponentially hard it would be infinitely easier than solving the rendering equation in its current form [McG12].

2.1.4.2 Area Formulation

Equation 2.11 represents the *hemispherical formulation* of the rendering equation. It can be useful to express the rendering equation as an integration over other surfaces in the scene rather than over the hemisphere. To this end, we must express the relationship between the differential solid angle, $d\vec{\omega}$, and the differential area

at another point $\mathbf{y}$, by considering the solid angle subtended by $dA_{\mathbf{y}}$ at the point $\mathbf{x}$ as illustrated in Figure 2.9 and given by:

$$d\vec{\omega} = \frac{(\vec{\mathbf{n}}_{\mathbf{y}} \cdot -\vec{\omega})}{\|\mathbf{x}-\mathbf{y}\|^2} dA_{\mathbf{y}} = \frac{\cos\theta_{\mathbf{y}}}{\|\mathbf{x}-\mathbf{y}\|^2} dA_{\mathbf{y}} \tag{2.12}$$

In order to change the integration from the hemisphere to surface area we must take into account the visibility between surface points. To accomplish this we introduce a binary visibility function, V, which determines the mutual visibility between two points:

$$V(\mathbf{x},\mathbf{y}) = \begin{cases} 1 & \text{if } \mathbf{x} \text{ and } \mathbf{y} \text{ are mutually visible,} \\ 0 & \text{otherwise} \end{cases} \tag{2.13}$$

We can now transform Equation 2.11 into an integral over all surfaces, A, in the scene, as follows:

$$L(\mathbf{x} \to \vec{\omega}_o) = L_e(\mathbf{x} \to \vec{\omega}_o) + \int_{y \in A} f_r(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_o)\, L(\mathbf{x} \leftarrow \mathbf{y})\, V(\mathbf{x},\mathbf{y})\, \frac{(\vec{\mathbf{n}}_{\mathbf{x}} \cdot \vec{\omega}_i)(\vec{\mathbf{n}}_{\mathbf{y}} \cdot -\vec{\omega}_i)}{\|\mathbf{x}-\mathbf{y}\|^2} dA_{\mathbf{y}} \tag{2.14}$$

To further simplify this expression we define the *geometry term*, G, as follows:

$$G(\mathbf{x},\mathbf{y}) = \frac{(\vec{\mathbf{n}}_{\mathbf{x}} \cdot \vec{\omega}_i)(\vec{\mathbf{n}}_{\mathbf{y}} \cdot -\vec{\omega}_i)}{\|\mathbf{x}-\mathbf{y}\|^2} = \frac{\cos\theta_{\mathbf{x}} \cos\theta_{\mathbf{y}}}{\|\mathbf{x}-\mathbf{y}\|^2} \tag{2.15}$$

The rendering equation can now be expressed as:

$$L\left(\mathbf{x} \rightarrow \vec{\omega}_o\right) = L_e\left(\mathbf{x} \rightarrow \vec{\omega}_o\right) + \int_{y \in A} f_s\left(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_o\right) L\left(\mathbf{x} \leftarrow \mathbf{y}\right) V\left(\mathbf{x}, \mathbf{y}\right) G\left(\mathbf{x}, \mathbf{y}\right) dA_{\mathbf{y}} \tag{2.16}$$

This formulation allows us to integrate over the area of the other visible surfaces in the scene directly, rather than over the hemisphere Ω^+.

2.1.4.3 Direct and Indirect Illumination

By reformulating the rendering equation as an integral over area, it allows us to directly integrate over the surfaces of light sources. This allows us to separate out the components of global illumination into *direct* illumination, i.e. the light arriving directly from light sources, and *indirect* illumination, i.e. the light arriving from all other sources. This allows us to express the outgoing radiance as:

$$L\left(x \rightarrow \vec{\omega}_o\right) = L_e\left(x \rightarrow \vec{\omega}_o\right) + L_{direct}\left(x \rightarrow \vec{\omega}_o\right) + L_{indirect}\left(x \rightarrow \vec{\omega}_o\right) \tag{2.17}$$

This allows us to compute the direct and indirect contributions separately and using different techniques. We can take this approach further, as shown by Arvo et al. [ATS94]. We can think of light reflection as a convolution of incoming light, L_i, with a BSDF, f_r, producing outgoing light L_o, and rewriting in operator form, using a reflection operator $\mathbf{K}$:

$$\begin{aligned} L_o\left(x \to \vec{\omega}_o\right) &= \int_\Omega L_i\left(x \to \vec{\omega}_i\right) f_s\left(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_o\right) \\ L_o\left(x \to \vec{\omega}_o\right) &= \left(\mathbf{K}L_i\right)\left(x \to \vec{\omega}_o\right) \\ L_o &= \mathbf{K}L_i \end{aligned}$$

Next, we define a geometry operator $\mathbf{G}$:

$$\left(\mathbf{G}L\right) = L\left(x'\left(\mathbf{x}, \omega\right), \omega\right) \tag{2.18}$$

where $x'\left(\mathbf{x}, \omega\right)$ is the closest point from $\mathbf{x}$ in direction ω. This operator includes the visibility term and turns distant surface radiance into local incident radiance, allowing us to rewrite the rendering equation as:

$$L = E + \mathbf{K}\mathbf{G}L \tag{2.19}$$

$$L = E + \mathbf{T}L \tag{2.20}$$

where $\mathbf{T}$ is the *transport operator*. This equation can then be solved using an infinite Neumann series where each summand represents one bounce of light:

$$L = E + \sum_{i=0}^{\infty} \mathbf{T}^i E \tag{2.21}$$

or,

$$L = E + \underbrace{\mathbf{T}E}_{1^{\text{st}}\text{ bounce}} + \underbrace{\mathbf{T}^2E}_{2^{\text{nd}}\text{ bounce}} + \underbrace{\mathbf{T}^3E}_{3^{\text{rd}}\text{ bounce}} + \ldots \tag{2.22}$$

allowing us to categorize rendering methods by the number of bounces of illumination considered.

2.2 Spherical Harmonics

In this section we review the formulation, notation and properties of spherical harmonics as they are a critical component of research detailed in later chapters of this dissertation, we largely follow the notation used in Jarosz [Jar08].

The Legendre polynomials are at the heart of the Spherical Harmonics, a mathematical system analogous to the Fourier transform, but defined across the surface of a sphere.

Firstly, a *harmonic* is a function that satisfies Laplace's equation:

$$\nabla^2 f = 0 \tag{2.23}$$

Spherical harmonics are an infinite set of harmonic functions defined on the sphere. They are derived by solving the angular portion of Laplace's equation in spherical coordinates using separation of variables. The spherical harmonic basis functions derived in this fashion take on complex values, but a complementary, strictly real-valued, set of harmonics can also be defined. In the context of com-

puter graphics, we restrict our discussion to the real-valued basis, since we generally only deal with real-valued functions.

If we represent a direction vector $\vec{\omega}$ using the standard spherical parametrization,

$$\omega = (\sin\theta\cos\varphi, \sin\theta\sin\varphi, \cos\theta) \tag{2.24}$$

the SH function is traditionally denoted by the symbol y, where the real spherical harmonic basis functions are defined as:

$$y_l^m = \begin{cases} \sqrt{2}K_l^m \cos(m\varphi) P_l^m(cos\theta), & \text{if } m > 0 \\ K_l^0 P_l^0(\cos\theta), & \text{if } m = 0 \\ \sqrt{2}K_l^m \sin(-m\varphi) P_l^{-m}(cos\theta) & \text{if } m < 0 \end{cases} \tag{2.25}$$

where P is the associated Legendre polynomial and K is a scaling factor to normalize the functions:

$$K_l^m = \sqrt{\frac{(2l+1)}{4\pi}\frac{(l-|m|)!}{(l+|m|)!}} \tag{2.26}$$

In order to generate all the SH functions, l is a positive integer starting for 0, but m takes signed integer values from $-l$ to l:

$$y_l^m(\theta,\varphi) \qquad \text{where } l \in \mathbf{R}^+,\ -l \le m \le l \tag{2.27}$$

Sometimes it is useful to think of the SH functions occurring in a specific order so that we can flatten them into a 1D vector, so we will also define the sequence

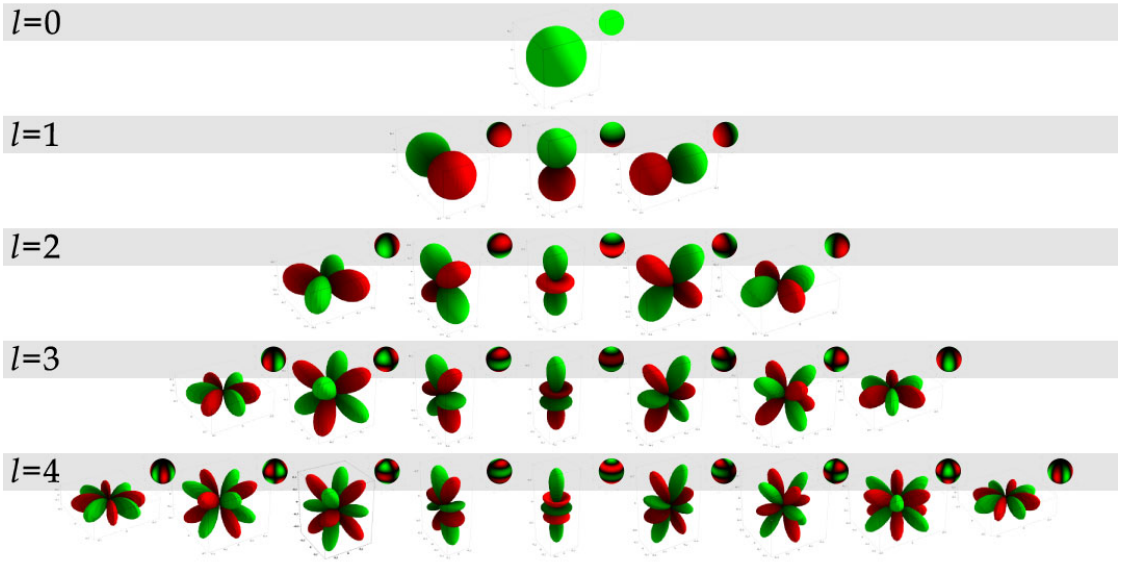


Figure 2.10: The first 5 SH bands plotted as unsigned spherical functions by distance from the origin and by colour on a unit sphere. Green (light gray) are positive values and red (dark gray) are negative. Image courtesy of [Gre03].

y_i:

$$y_l^m(\theta, \varphi) = y_i(\theta, \varphi) \qquad \text{where } i = l(l+1) + m \tag{2.28}$$

The first 5 SH bands plotted as unsigned spherical functions by distance from the origin and by color on a unit sphere can be seen in Figure 2.10, while the polynomial forms of the first 3 bands of the spherical harmonic basis functions are listed in Table 2.1. The spherical harmonic functions along the center column (i.e. $m = 0$) of Figure 2.10 are known as the as the *zonal harmonics*, and they are circular symmetric. Those functions along the edges, where $l = |m|$, are known as the *sectoral harmonics*. All other spherical harmonics are referred to as the *tesseral harmonics*.

	$m=-2$	$m=-1$	$m=0$	$m=1$	$m=2$
$l=0$			$\frac{1}{2\sqrt{\pi}}$		
$l=1$		$-\frac{\sqrt{3}y}{2\sqrt{\pi}}$	$\frac{\sqrt{3}z}{2\sqrt{\pi}}$	$-\frac{\sqrt{3}x}{2\sqrt{\pi}}$	
$l=2$	$\frac{\sqrt{15}yx}{2\sqrt{\pi}}$	$-\frac{\sqrt{15}yz}{2\sqrt{\pi}}$	$\frac{\sqrt{5}\left(3z^2-1\right)}{4\sqrt{\pi}}$	$-\frac{\sqrt{15}xz}{2\sqrt{\pi}}$	$\frac{\sqrt{15}\left(x^2-y^2\right)}{4\sqrt{\pi}}$

Table 2.1: The polynomial forms of the spherical harmonic basis functions of the first 3 bands ($m = 0 \ldots 2$ and $l = -2 \ldots 2$).

2.2.1 Projection and Expansion

As the spherical harmonics define a complete basis over the sphere, any real-valued spherical function f may be expanded as a linear combination of the basis functions:

$$f(\vec{\omega}) = \sum_{l=0}^{\infty} \sum_{m=-l}^{l} y_l^m(\vec{\omega})\, f_l^m \tag{2.29}$$

where the coefficients f_l^m are computed by *projecting* the real-valued spherical function, f, onto each basis function y_l^m:

$$f_l^m = \int_{\Omega_{4\pi}} y_l^m(\vec{\omega})\, f(\vec{\omega})\, d\vec{\omega} \tag{2.30}$$

With an infinite number of coefficients, this expansion would be exact as l goes to infinity. However, by limiting the number of bands to $l = n - 1$ we retain only the frequencies of the function up to some threshold. We can obtain an n^{th} order band-limited approximation $\tilde{f}$ of the original function f as follows:

$$\tilde{f}(\vec{\omega}) = \sum_{l=0}^{n-1} \sum_{m=-l}^{l} y_l^m(\vec{\omega})\, f_l^m \tag{2.31}$$

Just a few bands allow us to approximate low-frequency functions. Higher frequency signals require more bands, and quadratically more coefficients. It can be helpful to "flatten" the indexing scheme to use a single parameter $i = l\,(l+1)+m$. This convention make it clear that an n^{th} order approximation can be reconstructed using n^2 coefficients:

$$\tilde{f}\,(\vec{\omega}) = \sum_{l=0}^{n^2-1} y_i\,(\vec{\omega})\,f_i \tag{2.32}$$

2.2.2 Properties

There are many properties of spherical harmonics that make them particularly useful for use in computer graphics, we describe several of the most significant here.

2.2.2.1 Convolution

The spherical harmonic basis inherits a similar frequency space convolution property as a Fourier domain basis. If $h\,(z)$ is a circularly symmetric kernel, then the convolution $h \star f$ is equivalent to weighted *multiplication* in the SH domain:

$$(h \star f)_l^m = \sqrt{\frac{4\pi}{2l+1}} h_l^0 f_l^m \tag{2.33}$$

and flattened as:

$$(h \star f)_i = (h \star f)_l^m = (h \star f)_{(l(l+1)+m)} = \sqrt{\frac{4\pi}{2l+1}} h_{(l(l+1))} f_{(l(l+1)+m)} \tag{2.34}$$

The convolution property allows for efficient computation of prefiltered environment maps and irradiance environment maps [RH01]. Note, that the kernel function, $h(z)$, must be circular symmetric, as the result of a non-symmetric convolution would not be defined over the sphere.

2.2.2.2 Orthonormality

The inner product of any two distinct SH basis function is zero due to the fact that spherical harmonics are orthogonal for different l and different m. In addition the inner product of a basis function with itself is one due to the normalization constant K_l^m. This can be expressed as:

$$\int_{\Omega_{4\pi}} y_i(\vec{\omega})\, y_j(\vec{\omega})\, d\vec{\omega} = \delta_{ij} \tag{2.35}$$

where δ_{ij} is the Kronecker delta function. The orthonormal basis functions of spherical harmonics allow for the efficient projection and expansion operations described above, in addition to many other operations.

2.2.2.3 Rotational Invariance

Let us define g to be a representation of function f as rotated by some arbitrary rotation R over the unit sphere. We can define the following relationship:

$$g(\vec{\omega}) = f(R\vec{\omega}) \tag{2.36}$$

which implies that it does not matter if the function or the input has been rotated, the outcome remains the same. This *rotational invariance* means that there will be no aliasing artifacts when samples from f are collected at a rotated set of sample points. For example, rotating a light function will not cause any amplitude fluctuations.

2.2.2.4 Double Product Integral

Thanks to the orthonormality property, we can express the integrated product of two spherical harmonic functions as a simple expression. The integral product of two SH functions $\tilde{a}(\vec{\omega})$ and $\tilde{b}(\vec{\omega})$ can be expanded as:

$$\begin{aligned} \int_{\Omega_{4\pi}} \tilde{a}(\vec{\omega})\,\tilde{b}(\vec{\omega}) &= \int_{\Omega_{4\pi}} \left(\sum_i a_i y_i(\vec{\omega})\right)\left(\sum_j b_j y_j(\vec{\omega})\right) d\vec{\omega} \\ &= \sum_i \sum_j a_i b_j \underbrace{\int_{\Omega_{4\pi}} y_i(\vec{\omega})\, y_j(\vec{\omega})\, d\vec{\omega}}_{C_{ij}} \end{aligned} \tag{2.37}$$

where C_{ij} are called the **coupling coefficients**, which, due to the definition of orthonormality in Equation 2.35, are simply $C_{ij} = \delta_{ij}$. This simple form for the coupling coefficients introduces significant sparsity in the expression above, leading

to the simplification:

$$\begin{aligned}\int_{\Omega_{4\pi}} \tilde{a}\left(\vec{\omega}\right)\tilde{b}\left(\vec{\omega}\right) &= \sum_i \sum_j a_i b_i C_{ij} \\ &= \sum_i \sum_j a_i b_i \delta_{ij} \\ &= \sum_i a_i b_i\end{aligned} \tag{2.38}$$

Effectively, this means that the integrated product of two SH functions can be computed as the dot product of their coefficient vectors. That this integral can be computed by a simple dot product means that lighting can be computed very efficiently in the frequency domain. By expressing both the lighting and the cosine-weighted BRDF as spherical harmonics, the lighting integral can be evaluated using a simple dot product. Many existing precomputed rendering techniques (PRT) exploit this property [SKS02, KSS02].

2.2.2.5 Double Product Projection

Sometimes we want to compute the product of two spherical harmonic functions directly in the SH basis. We can compute the i^{th} coefficient of the SH projection

of the product $c(\vec{\omega}) = a(\vec{\omega})\, b(\vec{\omega})$ as:

$$\begin{aligned} c_i &= \int_{\Omega_{4\pi}} y_i(\vec{\omega})\, c(\vec{\omega})\, d\vec{\omega} \\ &= \int_{\Omega_{4\pi}} y_i(\vec{\omega})\, a(\vec{\omega})\, b(\vec{\omega})\, d\vec{\omega} \\ &= \int_{\Omega_{4\pi}} y_i(\vec{\omega}) \left(\sum_j a_j y_j(\vec{\omega}) \right) \left(\sum_k b_k y_k(\vec{\omega}) \right) d\vec{\omega} \\ &= \sum_j \sum_k a_j b_k \int_{\Omega_{4\pi}} y_i(\vec{\omega})\, y_j(\vec{\omega})\, y_k(\vec{\omega})\, d\vec{\omega} \\ &= \sum_j \sum_k a_j b_k C_{ijk} \end{aligned} \tag{2.39}$$

where C_{ijk} are the tripling coefficients, a sparse set of coefficients which correspond to Clebsch-Gordan coefficients, whose analytic values and properties are well studied [Tin03]. This expression states that the i^{th} coefficient of c is a linear combination of the, up to, $j \times k$ coefficients from a and b. The weighting of these terms is determined by the tripling coefficients, which are independent of the particular choice of a and b. This allows us to compute the tripling coefficients once for an efficient evaluation of product projection for many pairs of functions.

2.3 GPU Evolution

Graphics hardware evolution has far outpaced that of CPUs for several processor generations now, and the programming models that have been developed for them have come to dominate the parallel-programming landscape due to the ubiquity of graphics processors in everyday devices. As much of our work focuses on effi-

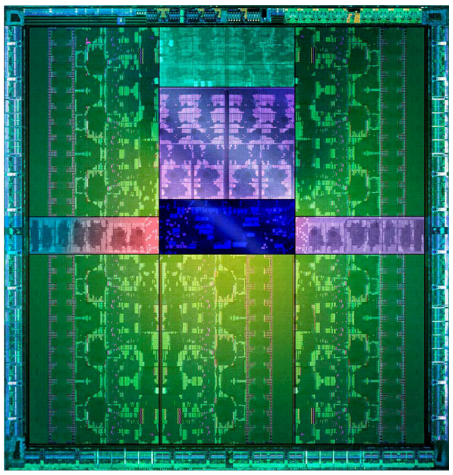


Figure 2.11: The Kepler GK110 (Titan) die. The Titan GPU has 2,688 shader cores and uses over 7.1 *billion* transistors.

cient utilization of the GPU and its resources, we will briefly describe some of the evolutionary changes that have impacted this work.

Modern GPUs are massively parallel devices (cf. Figure 2.11) whose performance scaling is predominantly dependent on the number of processors rather than clock-speed or pipeline depth (cf. Figure 2.12). This makes GPUs extremely good at executing batches of the same instructions in parallel. On appropriate workloads, this approach scales extremely well with the number of shader processors. It also allows far greater peak theoretical computational throughput as more of the chip is dedicated to computation rather than increasing processor complexity (see Figure 2.13). In order to accommodate the massive amounts of data processed by GPUs, total memory bandwidth has also scaled at a much faster rate than that of CPUs (cf. Figure 2.14).

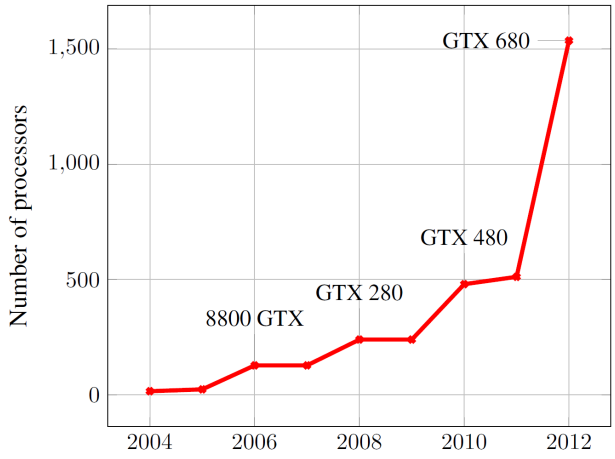


Figure 2.12: Increase in the number of "Shader Processors" in Nvidia GPUs over time. Credit: [Gai12]

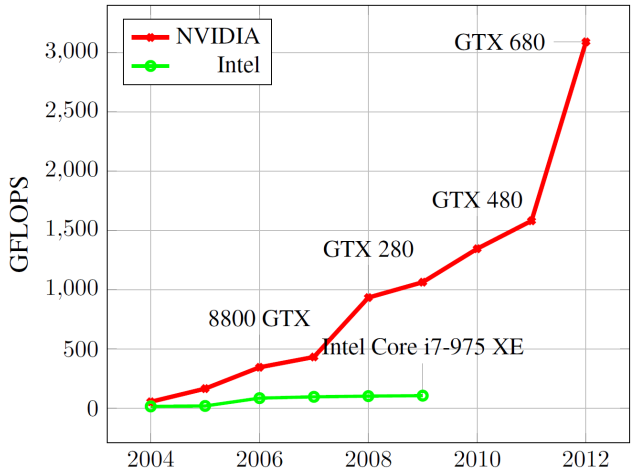


Figure 2.13: Comparison of GFLOPS available in successive iterations of Nvidia and Intel processors. Credit: [Gai12]

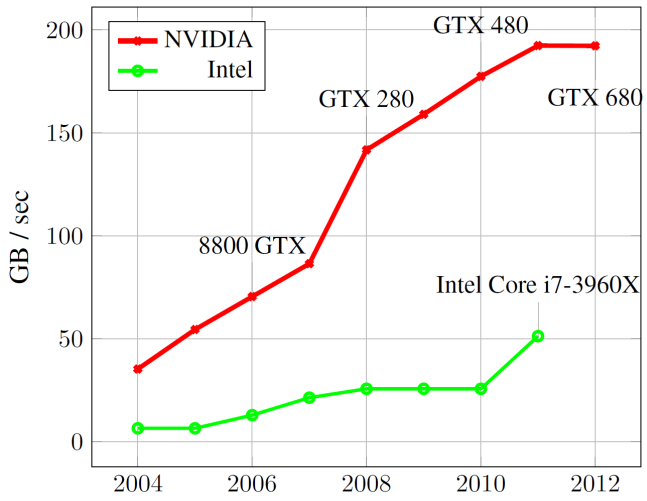


Figure 2.14: Bandwidth comparison of Intel CPUs and Nvidia GPUs. Credit: [Gai12]

2.3.1 API Evolution

For the past decade, commercial graphics hardware has been tightly intertwined with the two main APIs with which it can be programmed: that of Microsoft's DirectX API [Wik13] and that of the industry consortium Khronos, OpenGL [SA13]. The capabilities of which are largely reflected in the capabilities exposed in each new *Shader Model* (SM). For the purposes of this dissertation, we concern ourselves primarily with the features exposed by the OpenGL API. Digging up the features exposed during each Shader Model is a tedious exercise in technical writing archeology, particularly as they are often described in terms of the competing APIs evolving version numbering schemes. An attempt to correlate the shader model, API versions and capabilities can be seen in [Men12]. Suffice it to say that as the programming model has evolved from SM 1.0 to the most recent SM 5.0, more

and more flexibility and control has been added to the graphics pipeline. Initially, only vertex and fragment shaders were available as programmable pipeline stages, OpenGL 3.2 introduced programmable geometry shaders, while OpenGL 4.0 introduced tessellation shaders. Finally, OpenGL 4.3 introduced compute shaders which eschew the traditional graphics pipeline altogether and attempt to utilize the GPU as a purely general purpose massively parallel processor.

As the GPU has evolved, so too has the graphics pipeline, indeed it no longer resembles a pipeline so much as a convoluted subway map (cf. Figure 2.15). There are many ways to traverse the map as many pipeline stages are optional, and many components remain due to legacy support. However, attempting to combine bleeding edge features with legacy components can lead to extreme performance degradations. The `ARB_shader_image_load_store` [BB11] extension added a fundamentally new way to write to textures in OpenGL, by binding textures to an "image" unit, whereas previously texture writes were an operation restricted to the framebuffer. However, this bleeding edge feature introduces a number of constraints, for instance even the best hardware has only 8 image units, a crippling limitation when one considers that each mip-level of a texture must be bound to a separate "image." Additionally any texture bound to an "image" unit cannot simultaneous be bound to a "texture" unit, furthermore attempting to alternate between image writes and framebuffer writes on the same texture (i.e. mixing old and new code paths) leads to severe performance degradations.

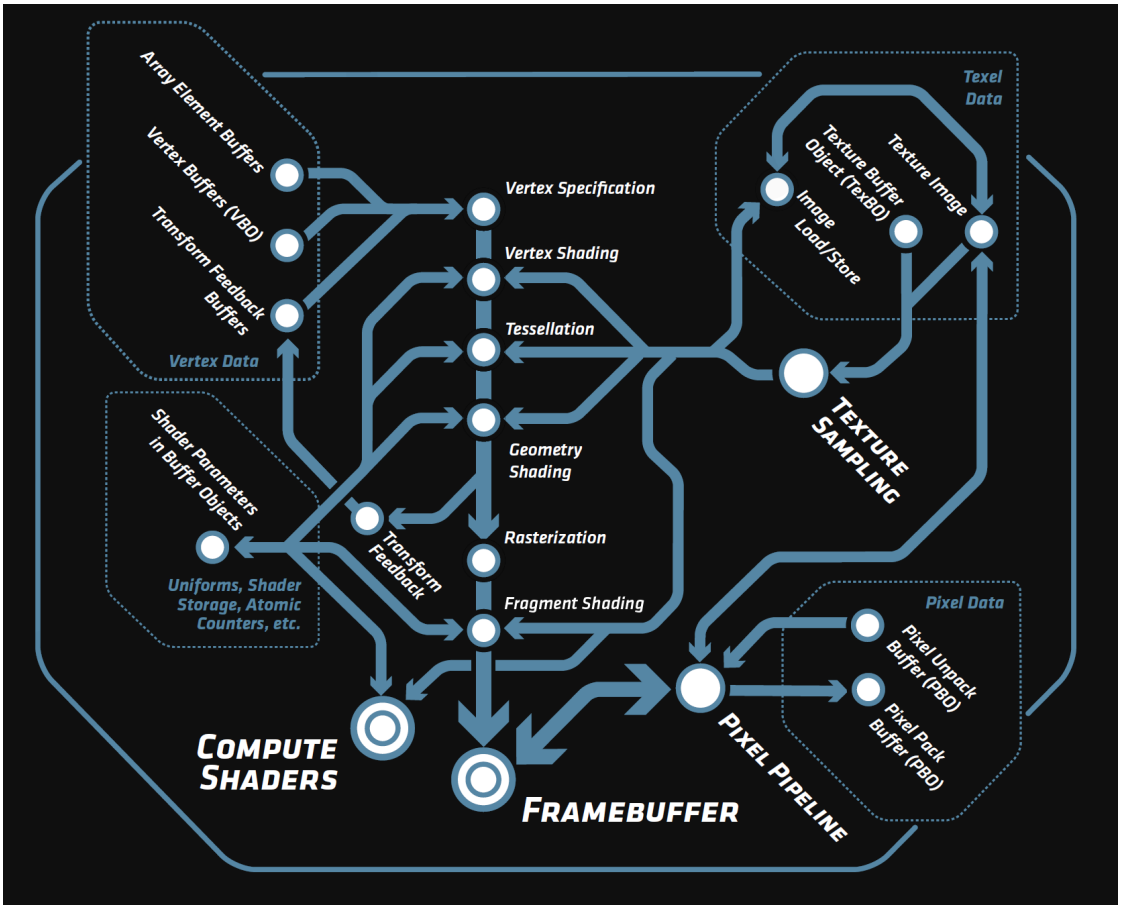


Figure 2.15: The modern graphics "pipeline," which now more closely resembles a subway system map. (Image courtesy of the Khronos Foundation)

Chapter 3: Related Work

Unquestionably the two greatest sources of influence on this work have been the Voxel Cone Tracing work by Crassin et al. [CNS*11, Cra11], and the Light Propagation Volume work by Kaplanyan et al. [KD10]. In this chapter we will discuss these influences in addition to some of the corpus of work that lead to them, and subsequently, this work. Although for an overview of the state of the art in global illumination it would be virtually impossible to surpass the quality of the survey by Ritschel et al. [RDGK12]. Finally, as much of our work is dependent upon a fast voxelization (discussed in Chapter 4), we also discuss related work in the field of voxelization (cf. Section 3.7).

3.1 Early Global Illumination

Early global illumination research was dominated by two algorithms–ray tracing and radiosity, both of which are discussed below, and revisited throughout this chapter as they are adapted to ever more sophisticated global illumination methods.

3.1.1 Ray Tracing

Ray tracing was introduced by Turner Whitted in 1980 [WH80]. The original paper used rays for determining visibility through a single pixel and also used rays to compute direct illumination, specular reflection, and refractive illumination effects. As such, this seminal paper described a major new tool in generating images.

The ray tracing algorithm has been researched and implemented extensively during the subsequent decades. Initially, much attention was on efficiency, using well-known techniques such as spatial subdivision and bounding-volumes. More and more, the focus was also on lighting effects themselves. By treating ray tracing as a tool for computing integrals, effects such as diffuse reflections and refractions, motion blur, lens effect, etc. could be computed within a single framework. Many of the global illumination algorithms discussed below employ some form of ray shooting. However, ray tracing had difficulty reproducing indirect illumination effects such as color bleeding and diffuse reflections.

3.1.2 Radiosity

A solution supporting indirect illumination effects came in the form of a finite-element method called Radiosity, introduced by Goral et al. in 1984 [GTGB84]. Radiosity is based on the calculation of energy transfer between all surface elements in a scene. This has the drawback that many costly visibility tests are required to perform an accurate computation.

In radiosity, the distribution of light is computed by subdividing the scene into

surface elements (patches) and computing for each element the correct radiometric value. Once the radiosity value for each surface element was known, the solution could be displayed with existing graphics hardware. Early radiosity research was centered around computing a faster solution for the linear system of equations that expressed the equilibrium of the light distribution in the scene.

Initially, radiosity was limited to diffuse surfaces, and the accuracy of the method was set by the choice of surface elements. Finer details in the shading at a frequency higher than the initial mesh could not be displayed.

3.2 Evolution of Global Illumination

As the sheer computational intensity of global illumination became apparent, several avenues of optimization and acceleration were explored. Traditional techniques were adapted onto the GPU, new techniques like Instant Radiosity [Kel97], Lightcuts [WFA*05], and Photon Mapping [Jen01] were developed, and hierarchical adaptations of traditional techniques were introduced.

3.2.1 GPU Ray Tracing

Early experiments with GPU ray tracing methods relied on the versatility of programmable graphics hardware and used fragment shaders to perform ray-primitive intersections [CHH02, PBMH02].

3.2.2 GPU Radiosity

Several full radiosity based global illumination algorithms tailored to GPUs were proposed. Dachsbacher et al. [DSDD07] and Dong et al. [DKTS07] demonstrated global illumination using techniques based on hierarchical radiosity, yet avoided traditional visibility computation. Only per-vertex and low-frequency lighting was supported due to directional discretization. Dachsbacher and Stamminger [DS05] introduced the idea of reflective shadow maps, where shadow map texels correspond to virtual point lights. However, no hierarchical lighting and no visibility were taken into account. Martin et al. [MPT98] computed a coarse-level hierarchical radiosity solution on the CPU, and used graphics hardware to refine the solution by texture mapping the residual. In each of these cases, graphics hardware is used to accelerate elements of the radiosity solution, but the bulk of the processing occurred on the CPU.

Among attempts to accelerate radiosity, the hemi-cube approach [CG85] used graphics hardware to identify the patches visible from a given patch in the scene, attacking form factor computation, generally considered the bottleneck of radiosity techniques. More recently [CHL04], and [CHH03] proposed methods for GPU-based radiosity. The former relied on texturing and visibility testing, whereas the latter used the GPU to process the radiosity matrix.

Figure 3.1: A comparison of direct illumination only on the left vs. global illumination on the right. Image credit: [DSDD07].

3.2.3 Hierarchical Radiosity

A large body of research on hierarchical radiosity (HR) was started by the seminal paper by Hanrahan et al. [HSA91]. These approaches essentially approximated blocks of the matrix with a constant, and used error estimation oracles to decide whether to subdivide or approximate, see Figure 3.1. Importance [SAS92] can be used to speed up the convergence of these techniques.

Hierarchical radiosity proved to be a major step forward, since the algorithm was now able to adapt its underlying solution mesh to the actual shading values found on those surfaces. Discontinuity meshing was similarly used to precompute accurate meshes that followed the discontinuity lines between umbra and penumbra regions caused by area light sources. The algorithm was also extended by subdividing the hemisphere around surfaces in a mesh as well, such that glossy surfaces could also be handled. On the other side of hierarchical radiosity, cluster-

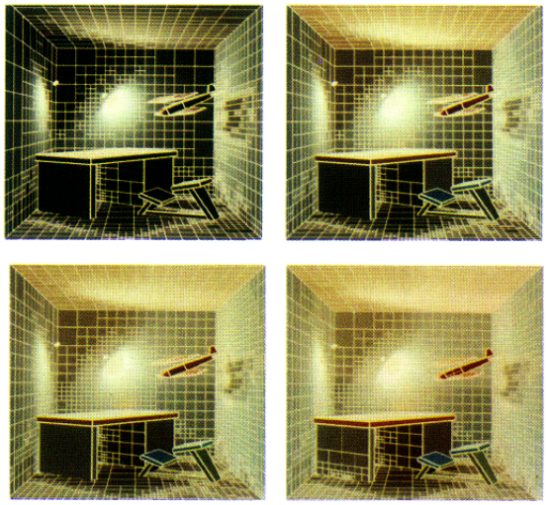


Figure 3.2: An illustration of hierarchical radiosity refinement. Credit: [HSA91].

ing algorithms were introduced to compute the illumination for disjunct objects in single clusters.

A hierarchical radiosity technique that took advantage of temporal coherence was introduced by Drettakis et al. [DS97], but it did not scale to high-complexity geometry and materials. Some promising approaches based on hierarchical radiosity have been presented that avoid the computation of visibility, notably [Bun05] and [DSDD07]. These methods provided very fast solutions; however, their drawback is that the antiradiance (i.e. light that has to be subtracted to correct for ignoring visibility) is a highly directional quantity, so a large number of directional samples were needed for each patch to maintain accuracy. This causes these methods to have difficulty scaling beyond a few thousand patches without compromising the accuracy of indirect and environment shadows.

3.2.4 Instant Radiosity

Instant Radiosity (IR), introduced by Keller in 1997 [Kel97], was one of the first techniques to exploit the graphics pipeline. The key insight of instant radiosity was that indirect illumination could be modeled as direct illumination by placing secondary light sources, referred to as virtual point lights (VPLs), in the scene on surfaces where indirect light is to be emitted, see Figure 3.3. Rendering can then be done on the GPU, with visibility handled by shadow mapping, achieving near-interactive rates. However, this approach is not without its drawbacks, the VPL placement must be found using simulated photon trajectories to discover intersections in the scene, and the resultant number of VPLs is often prohibitively large.

Instant incremental radiosity [LSKL07] is a promising adaptation for one-bounce indirect illumination that manages a set of 256 VPLs as illumination changes, without retracing new particles. However, for interactive performance, it uses sparse interleaved sampling of the image, and re-renders at most 10 shadow maps per frame, which is only correct for a static scene.

Instant radiosity techniques do not capture all kinds of light transport equally well, in particular, they have difficulty reproducing caustic effects, and they are generally limited to 1-bounce global illumination.

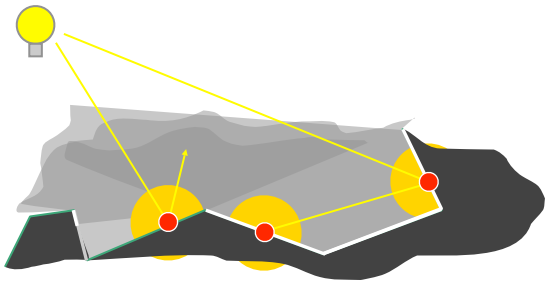


Figure 3.3: VPLs for indirect illumination using Instant Radiosity, the VPL in the middle represents second bounce indirect illumination. Image credit: [Kel97].

3.2.5 Lightcuts

The Lightcuts algorithm introduced by Walter et al. in 2005 [WFA*05], attempted to improve on the scalability issues of Instant Radiosity. Its approach is based on the premise that a clustering of lights in the scene can be used to approximate the full solution, yet no single partitioning of the lights in the scene is likely to work over the entire image. Thus, the *light tree* is introduced, essentially a binary tree used to store a hierarchical evaluation of scene irradiance represented as point light sources. Clustering is used to progressively approximate groups of lights, which are stored at nonleaf nodes in the tree, where the leaves are individual lights and the interior nodes are light clusters containing the lights below them in the tree. So we are left with a tree in which, each node has a representative light that approximates the contribution of all the lights in the node's cluster. Computing the incoming radiance at a given point requires making a *cut* through the tree in order to select a small subset of lights with an error below a given threshold. This approach is effective because it unifies the computation of both direct and indirect illumination,

permitting highly efficient irradiance interpolation using reconstruction cuts.

Multidimensional lightcuts [WABG06] extended the basic lightcuts algorithm to handle motion blur and participating media, but does not amortize over multiple frames. Both lightcuts and multidimensional lightcuts require an approximation of the lighting solution in order to perform light clustering, thus reducing performance and making lightcuts essentially a two-pass algorithm.

Arbee et al. [AWB08] introduced a single-pass importance driven variant of the Lightcuts algorithm, and extended it for subsurface scattering and translucent materials. Coherent Lightcuts [BD08], exploited pixel coherence to improve cut construction and reduce rendering times, and has the advantage of not requiring an approximate solution.

Lightcuts and its variant share the weakness that they rely on a ray-tracer to compute visibility. Though the algorithm achieves sublinear performance in the number of point lights, it achieves this using complex data structures and traversal mechanisms that are not easily parallelizable, and thus will not scale with improved parallel processors.

3.2.6 Photon Mapping

Photon mapping is another popular technique popularized by Jensen in [Jen01], which has also been given the GPU treatment. The GPU ray tracing work from [PBMH02] has been extended to photon map rendering [PDC*03]. Another photon map rendering method is presented in [MM02]. Both approaches suffered from the

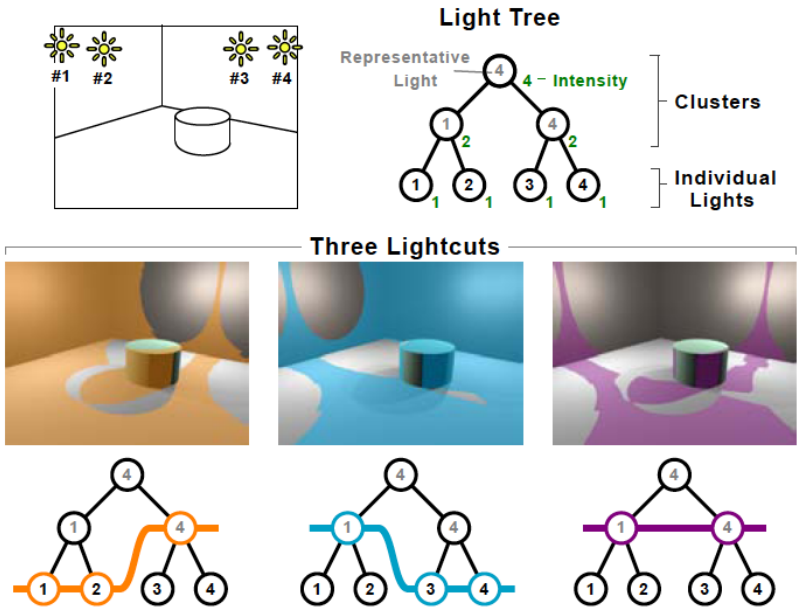


Figure 3.4: Light tree and three example cuts, the highlighted areas represent regions where error is small. Image credit: [WFA*05].

same drawback: the current GPU architecture did not allow for efficient handling of complex data structures such as trees, which are commonly used in ray tracing optimization and photon map storage. Therefore, the photon map is stored in a regular grid [PDC*03], or in a costly hash table [MM02]. The related nearest-neighbors queries were simplified to meet the data structure and GPU constraints, yielding quality or performance drops.

Other GPU-accelerated photon map rendering methods have been proposed. Larsen et al. [LC04] used graphics hardware to perform the costly final gathering: the photon map was built on the CPU using the classical method defined in [Jen01]. For each surface, an "approximate illumination map" was built using the data contained in the photon map. The GPU was used to perform final gathering and caustics filtering. The approaches presented in [SB97] and [LP03] used the GPU for irradiance reconstruction: each photon was rendered as a textured quadrilateral. The corresponding texture represented the kernel function for the photon. Although those methods showed encouraging results, they were bounded by the large number of photons required to render a high quality image.

Although photon mapping provides a solution to many difficult problems in global illumination, the algorithm by itself is often inadequate or inefficient under complex lighting conditions. There has been a great quantity of research done into improving efficiency in all areas of the algorithm to deal with these situations, including new methods of density estimation, photon propagation, and sampling.

Poor-quality results caused by inadequate under representation of illumination by the photon map is a well studied problem. A number of solutions have been

proposed, which focus on optimizing the distribution of photons prior to rendering. Visual importance sampling [KW00], [PP98] and a technique based on the Metropolis-Hastings algorithm [FCL05] have all shown to be effective at storing photons in a much more optimal distribution pattern. Conversely, unnecessary over representation in certain areas has been addressed using density control [SW00] to restrict photon storage in areas of strong incident illumination. Tawara et al. [TMS04a] introduced a novel method also based on importance sampling, which separated strong and weak diffuse illumination into two independent data sets together with a voxel grid containing information about photon density. Havran et al. [HHS05] accelerated final gathering by performing the process in reverse, computing density estimations at each photon and propagating the irradiance to nearby gather ray hits.

3.3 Advanced Global Illumination

New global illumination techniques continue to abound, as well as many variants and hybrid techniques. Several major research directions such as Precomputed Radiance Transfer [Leh04], (Ir)radiance caching [WRC88], and spherical function representation have been very actively researched.

3.3.1 Advanced Hierarchical Methods

Hierarchical data structures are widely used throughout computer graphics to store data at progressive levels of detail. When accessed according to an error metric, these structures may be used to reduce image artifacts and improve the efficiency of global illumination algorithms

The mipmapping concept was extended independently by Benson and Davis with octree textures [KLS*05] and by DeBry et al. with octexes, both of which compensated for the poor performance of 2D mipmaps on complex 3D surfaces. A volumetric GPU-friendly octree data structure based on mipmapping, called the Histopyramid, was also introduced in [DZTS07], with application to many areas.

Jensen and Buhler [JB02] use a hierarchical data structure to rapidly evaluate the BSSRDF of translucent materials. Irradiance is sampled across the surface of translucent objects and is progressively stored throughout the nodes of an octree. Using approximated samples where it is appropriate, greatly accelerated evaluation of the diffusion approximation when compared to evaluating each sample individually.

The irradiance atlas [CB04] used a sparse adaptive octree to represent photon maps that are too large to be held in memory. The irradiance stored at each photon is compiled into a hierarchical data structure called a brick map. This approach allowed irradiance data to be cached and swapped in and out of memory efficiently and made rendering scenes containing extremely detailed photon maps practical, even with limited memory. As a result of the progressive approximation,

sampling from the brick map also benefits from filtering, resulting in a reduction in noise. Yue et al. [YIDN07] employed a similar volumetric data structure to evaluate irradiance from surfaces, allowing relighting of scenes at interactive frame rates.

3.3.2 Precomputed Radiance Transfer

Precomputed Radiance Transfer (PRT) is a method to quickly compute the reflection integral for a given environment, e.g. [SKS02, KSS02, SLS05][NRH04, HPB07]. Radiance (or irradiance) is stored in some fashion, such that it can be recovered later. These techniques are based on extensive precomputation to render static or dynamic scenes under distant or indirect illumination. The classic PRT [SKS02] approach allowed static scenes under distant low-frequency lighting, visibility and BRDFs, which were extended to all frequencies in [NRH04] and other follow up work. In PRT, scenes are usually assumed to remain static. Limited dynamic scenes (rigid objects) can be supported, e.g. by Zhou et al. [ZHL*05], but indirect illumination is then very difficult to achieve [IDYN07, PLPB07]. This was generalized to deforming geometry in [RWS*06]. Recently, Akerlund et al. [AUW07] demonstrated real-time one-bounce global illumination in conjunction with local lighting. However, geometry still needed to be static due to the use of precomputed visibility.

PRT permits the efficient rendering of many illumination effects on static objects, such as soft shadows and glossy reflections, in real-time [SKS02, NRH03,

LSSS04]. The illumination solution is parametrized by the incident lighting, that is assumed to be represented by means of basis functions, such as spherical harmonics [SKS02] or wavelets [NRH03], which allowed for efficient rendering. PRT exploits the restriction to static objects by pre-computing all the visibility queries and baking them into the parametrized solution.

Generally, lighting in PRT is assumed to be distant. However, low-frequency localized lighting can be integrated [AKDS04]. When only indirect illumination is considered, local point and spot lights are possible [KAMJ05, KTHS06, HPB06].

Dynamic scenes are inherently difficult for PRT techniques, since visibility can no longer be precomputed. Mei et al. [MSW04] precomputed visibility on a per-object basis for a discrete set of directions and stored it in the form of uncompressed shadow maps. This allowed them to render multiple rigidly moving objects under low-frequency distant illumination. However, dynamic local light sources remained infeasible. Similar in spirit, Zhou et al. [ZHL*05] proposed the use of shadow fields, which allowed movement of individual rigid objects under semi-local (lights may not enter an object's bounding sphere) or distant illumination, producing correct inter-shadowing. In practice, this technique was limited to low-frequency lighting, as all frequency lighting updates took several seconds for dynamic scenes. Dynamic objects were even more difficult to handle, as no precomputation can be employed. Hemispherical rasterization [KLA04] and spherical harmonics exponentiation [RWS*06] have been proposed for small- or medium-sized scenes under low-frequency illumination.

Kristensen et al. [KAMJ05] described a technique for rendering indirect il-

lumination from omnidirectional, local, moving light sources. The results were impressive, but dynamic objects were not supported, and preprocessing costs were substantial. Kontkanen et al. [KTHS06] extended wavelet radiosity for computing a full hierarchical direct-to-indirect transport operator for a static scene. The technique supported all types of light sources, and, in principle, glossy BRDFs. However, dynamic objects could not be easily supported. Furthermore, precomputation still took tens of minutes.

Coherent Shadow Maps (CSMs) by Ritschel et al. [RGKM07] used a shadow map based data structure, but enhanced such that it could be used for physically-based real-time rendering as a PRT-like technique. CSMs exploit coherence between many shadow maps for compression, and support all-frequency lighting, dynamically moving rigid objects, local as well as distant light sources, and dynamic material properties. Nonetheless, the method is fast, supports progressive rendering for even faster updates, and is memory efficient through the use of CSMs. However, CSMs are unable to model self-shadowing on objects. Coherent Surface Shadow Maps (CSSMs) [RGKS08] rectified this deficiency, allowing for visibility tests between moving objects and a high number of lights outside their convex hulls using simple shadow mapping, see Figure 3.5.

Precomputed radiance transfer is an excellent technique to support interactive previewing of lighting and material changes in static scenes, but the cost of often several minutes to hours of precomputation and (in some cases) fixing the camera, limits its interest.

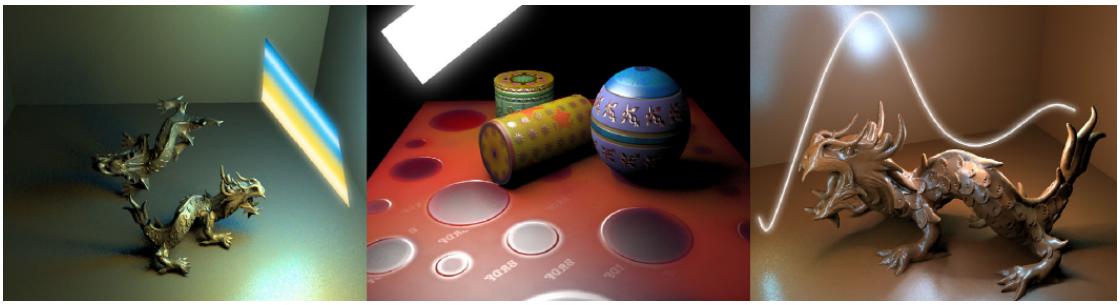


Figure 3.5: Images generated by the Coherent Shadow Map approach. Image credit: [RGKM07].

3.3.3 (Ir)Radiance Caching

Radiance and irradiance are two similar, yet distinct, and often confused quantities, hence radiance caching and irradiance caching methods are often referred to as (ir)radiance caching methods. Regardless, (ir)radiance caching was first proposed by Ward et al. [WRC88] as a means of computing indirect diffuse inter-reflections in a distributed ray tracer [War94]. The technique exploited the smoothness of the indirect illumination by sampling the irradiance sparsely over surfaces, caching the results and interpolating them.

For each ray hitting a surface, the irradiance cache is queried. If one or more irradiance records are available, the irradiance is interpolated using irradiance gradients, see Figure 3.6. Otherwise a new irradiance record is computed by sampling the hemisphere and added to the cache. In this way, the cache gets filled lazily, progressively in a view dependent manner. As it gets filled, more and more irradiance computations can be carried out by interpolation. Ward used an octree for storing the irradiance records. In [WH92] the interpolation quality is improved by

the use of irradiance gradients.

In [Nij03, NPG05], Nijasure et al. proposed a volumetric irradiance cache method for non-diffuse global illumination computation using graphics hardware. The incoming radiance function at a number of locations a priori selected was sampled and projected into the spherical harmonics basis. Then the incoming radiance at any surface point is estimated by interpolating the incoming radiance at nearby sample locations. Although the authors demonstrate real-time performance, the main drawback of this method is the choice of sample points. In [Nij03, NPG05] these points are placed on a regular grid inside the volume of the scene, therefore not adapting to the lighting complexity.

The irradiance caching concept was extended to radiance caching by Krivanek et al. [KGPB08], [KGBP05] in which incoming radiance at each sample was stored using hemispherical and spherical harmonic coefficients. This allowed for much more accurate interpolation, especially over high-frequency BRDFs.

Radiance cache splatting [GKBP05][KGPB08] presented an algorithm for one-bounce global illumination that took advantage of illumination coherence by subsampling it at a sparse set of locations. Temporal radiance caching [GBP08] accelerated computation of global illumination for image sequences by reusing samples between frames. Reflective shadow maps [DS05] and Splatting indirect illumination [DS06] provided interactive solutions for one-bounce global illumination, but neglected shadowing effects in the indirect bounces.

Radiance interpolation can be used whenever there is a certain level of smoothness in the radiometric quantity being computed. All radiosity methods use in-

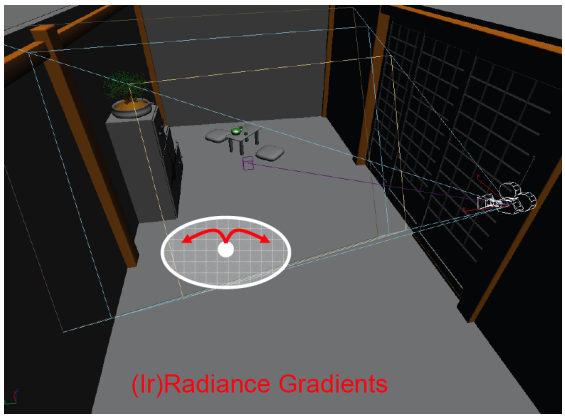


Figure 3.6: (Ir)Radiance Gradients. Image credit: [KGW*07].

terpolation in the form of surface discretization. They adapt to the irradiance smoothness by adaptive geometry subdivision, e.g. [HSA91, Hec91, LTG92].

In the context of Monte Carlo ray tracing many approaches have been proposed for screen space interpolation [Guo98], [WDP99, WDG02, BWG03]. The goal of these methods was to display an approximate solution quickly, possibly at interactive frame rates. Object space interpolation has also been used for the purpose of interactive previewing [SS00, TPWG02]. Sparse sampling and interpolation for high quality rendering was used in [BT99] and [WRC88]. The approach of Bala et al. [BT99] is suitable only for deterministic ray tracing. Ward et al. [WRC88] used interpolation only for diffuse surfaces. [KGPB08] extended this work to support caching and interpolation of the directional incoming radiance on glossy surfaces.

3.3.3.1 Spherical Function Representation

A suitable representation of functions on a (hemi)sphere is necessary for incoming radiance caching. Piecewise constant representation [SHS98, CLS97, TMS04b] is simple but prone to aliasing and usually very memory intensive.

Wavelets Wavelets are one popular solution, however, unless higher order wavelets are used, even wavelet representation [PB96, SGCH94, LF96], [SSG*00, SS95] does not remove the aliasing problems. But with higher order wavelets the mathematics becomes complicated and hence discourages their use.

Spherical Harmonics Spherical Harmonics [SAWG91], [SDS95], [CMS87, WAT92, RH02, Ram02, KSS02, SKS02, SHHS03] removed the aliasing problem and are efficient for the representation of low-frequency functions. However, the representation of sharp functions required a large number of coefficients and ringing artifacts could appear. Hemispherical harmonics [GKPB04] are better suited for representing functions on a hemisphere. Basis functions very similar to spherical harmonics are Zernike polynomials [WC92], [KDS96] and hemispherical harmonics of Makhotkin [Mak96].

3.3.4 Ambient Occlusion

Ambient occlusion is a heuristic approximation to global illumination now in common practical use. For a point being shaded, it is defined as the hemispherical

integral of either the visibility function or some suitable function of the distance to the nearest surface in each direction. The model nicely reproduces soft dark corners, an important feature of "real" global illumination solutions. Kontkanen and Laine [KL05] and Malmer et al. [MAH07] described techniques for rendering ambient occlusion to the surroundings of moving, rigid objects. Although ambient occlusion is a popular technique, it is less interesting in that it doesn't respect the rendering equation.

3.3.5 Implicit Visibility

Several new papers explore the usage of Implicit Visibility for global illumination solutions. In [DKTS07] they tackled the visibility problem by implicitly evaluating mutual visibility while constructing a hierarchical link structure between scene elements. However, their link structure did not scale well with the number of patches in the scene, and was complicated to maintain with moving geometry.

A more interesting approach to implicit visibility was presented by Dachsbacher et al. [DSDD07] as antiradiance, which reformulates the rendering equation in such a way that requires the consideration of both radiance and antiradiance, see Figure 3.7, but enabled the treatment of visibility in an implicit manner. Similar approaches such as "negative light" have been used to allow for incremental radiosity updates, where it compensates for the different visibility configuration between two frames [BF89, PSV90]. Shadow photons [JC95] are also related to the idea of negative light. However, visibility computation was still required. In

particular, methods for the efficient update of global illumination [Sha97, DS97] required searching for regions that need recomputation of visibility. The antiradiance reformulation avoids this search and does not require any explicit visibility computation. Bunnel [Bun05] used "negative light" to approximate ambient occlusion and simulated the effect of inter-reflection, but the negative light was not directional, and this heuristic did not respect the rendering equation. Ren et al. [RWS*06] proposed a different reformulation of visibility using the exponentiation of spherical harmonics in the context of soft shadows. They, however, still needed to determine the occluders between an object and the light and do not treat global illumination.

The work most related to antiradiance is by Pellegrini [Pel99] who also derived a new rendering equation where explicit visibility is avoided, but his work is purely theoretical. Antiradiance used similar ideas where negative light is transmitted through each surface to compensate for the lack of occlusion treatment.

A number of solutions for dynamic global illumination performed partial computations and used caching or reprojection of results from previous frames. An example is the Shading Cache [TPWG02]; this approach cached samples from a path tracer and used graphics hardware for interpolation. However, these approaches still required visibility to compute the sparse samples.

Since antiradiance removed occlusion testing, it is an inherently $O(n^2)$ algorithm in the number of patches. Thus it employed hierarchical radiosity methods [Sil95, SAG94], to reduce computational complexity. These approaches required sophisticated data structures and global random-access visibility computations for

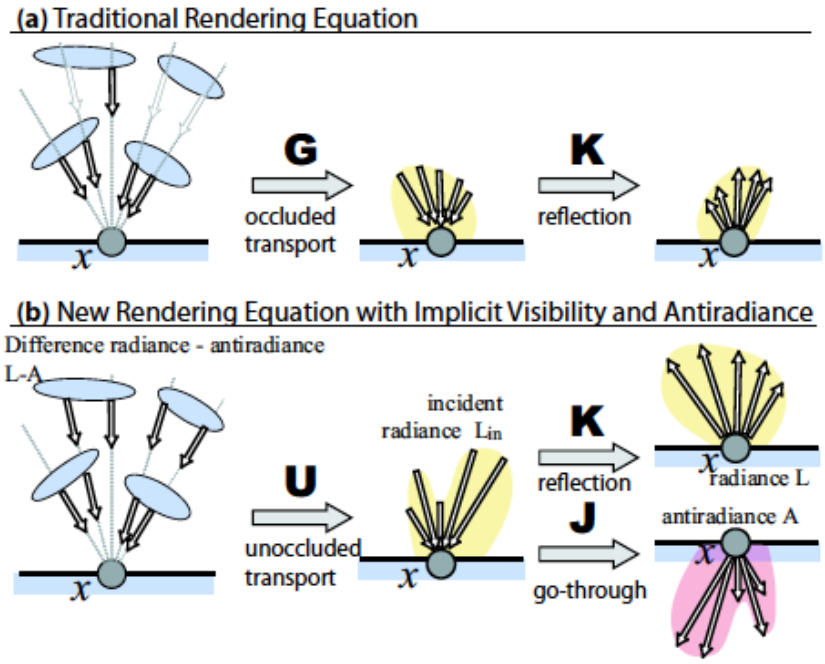


Figure 3.7: (a) Operator formalization of the rendering equation. (b) Reformulation using unoccluded transport, creating antiradiance to compensate for extraneous transport. Image credit: [DSDD07].

form-factors. Extensions to this method stored directional radiance using wavelets [SSG*00] or spherical harmonics [SDS95]. In contrast, antiradiance stored the directional information of "negative light," thus avoiding visibility computation.

3.4 Volumetric Techniques

Eikonal rendering, proposed by Ihrke et al. [IZT*07], as opposed to most ray shooting based methods, simulated the propagation of a wavefront though volumetric scene geometry. This approach enabled them to display realistic refractive

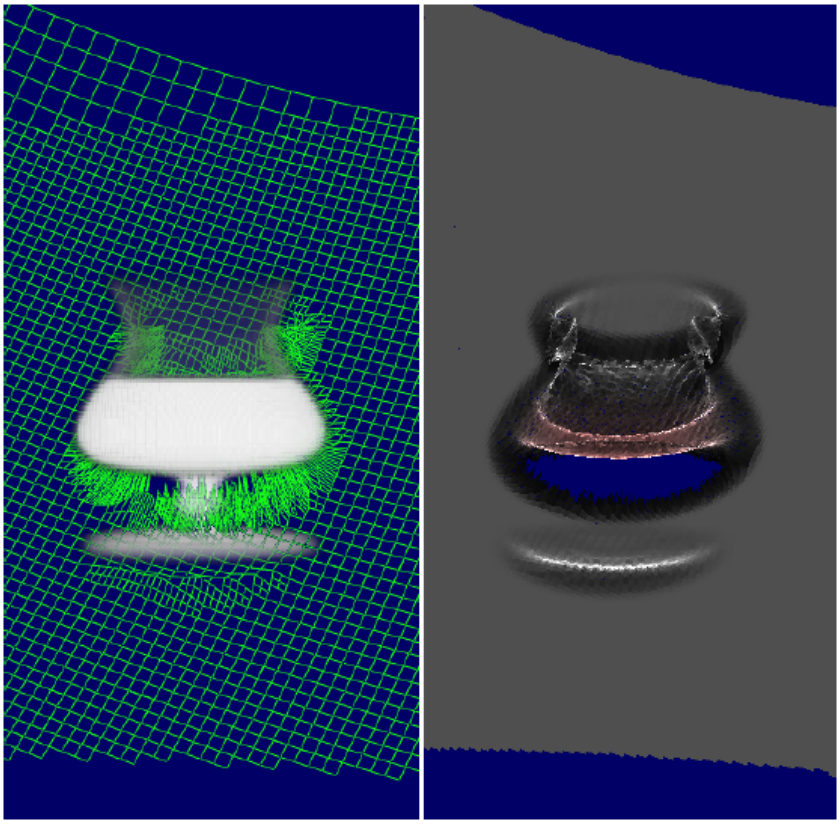


Figure 3.8: Eikonal adaptive wavefront propagation. Image credit: [IZT*07].

objects with complex material properties, such as an arbitrarily varying refractive index, inhomogeneous attenuation, as well as spatially-varying anisotropic scattering and reflectance properties. Through the wavefront propagation based on the Eikonal equation, they deposit illumination data in a refractive index volume, as seen in Figure 3.8, which once filled can be efficiently displayed using a fast ray-caster. Though rather unique, the drawback of this approach is that the wavefront propagation takes a few seconds. Thus, lighting changes cannot be interactively displayed.

Another paper, *Interactive Relighting of Dynamic Refractive Objects* by Sun et al. [SZS*08], shared several aspects in common with the Eikonal renderer, with the notable exception of Eikonal wavefront propagation. Though not advertised as such, [SZS*08] is essentially a very well parallelized photon mapper, except instead of depositing radiance values in a balanced KD-tree, radiance is stored in a volumetric texture much as in the Eikonal approach, which, similarly, can then be visualized extremely efficiently with ray-casting. [SZS*08] further extended the flexibility of their approach by performing fast object voxelization of triangulated geometry, allowing for a greater variety of inputs.

[SZS*08] deserves special mention because it embodies a concept described in [LD08] known as "time to image." This is a notion in which frames per second is deemphasized, and the total time to construct a solution, or "image," is considered more important. This is significant when one considers that this implies that [SZS*08] performed *no* precomputation, and that its entire rendering pipeline, see Figure 3.9, was fully evaluated every frame. This means that their implementation *implicitly* handled a dynamic camera, dynamic geometry, and dynamic lighting.

3.5 Light Propagation Volumes

Originally developed for CryTek's CryEngine 3, and published as *Cascaded Light Propagation Volumes* by Kaplanyan et al. [KD10], light propagation volumes is a real-time global illumination algorithm inspired by the Discrete Ordinates Method, and lattice based diffusion methods of light transport.

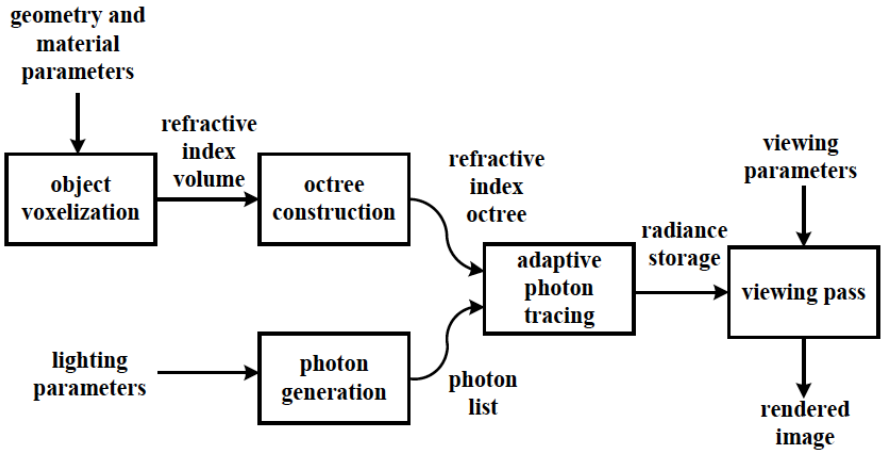


Figure 3.9: The rendering pipeline from [SZS*08].

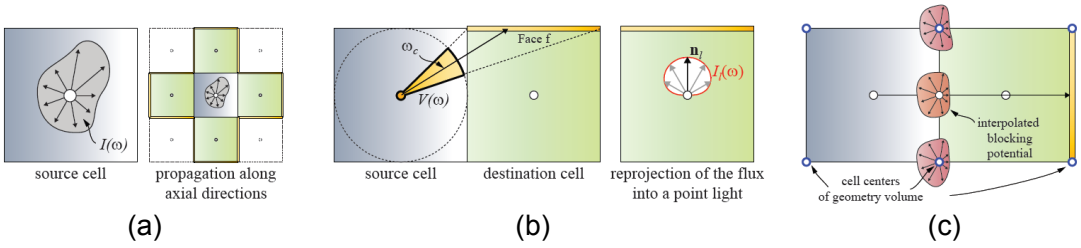


Figure 3.10: (a) Each cell (voxel) of the LPV stores the directional light intensity that is propagated to its axial neighbors; (b) incident flux is computed for each face of the destination voxel; (c) to account for occlusion a separate "geometry volume" is used, offset by half from the voxel centers. Image credit: [KD10].

After an initial radiance injection phase using Dachsbacher's reflective shadow maps [DS05] into a volumetric texture referred to as the *Light Propagation Volume* (LPV), the light is then iteratively transferred (propagated) to its neighbors. Additionally, a *Geometry Volume* is used to prevent light from leaking through walls. In the cascaded version of the algorithm, a set of nested grids is used to improve performance and lower memory consumption.

Notably, light is stored as 2 band spherical harmonics, and during propagation

the flux incident to each of the neighbor cell's faces is computed, next the incident flux of each cell is converted into outgoing intensity. Conceptually, we can consider this process as a series of VPLs, each facing one of the faces of the cell and emitting flux equal to that of the face. These per-face VPLs are then accumulated into a single VPL and stored back into the light propagation volume as spherical harmonics, see Figure 3.10. After several iterations, the light propagation volume contains an approximation of the diffuse lighting in the scene. During rendering, the incident radiance can be sampled from the trilinearly interpolated spherical harmonic functions stored in the light propagation volume.

3.6 Voxel Cone Tracing

Voxel cone tracing as introduced in *Indirect Illumination using Voxel Cone Tracing* by Crassin et al. in [CNS*11], builds a sparse voxel octree which stores a filtered representation of the scene. The voxels store many components, the diffuse color, and opacity, in addition to the light direction and intensity stored as an isotropic Gaussian lobe.

As the voxel cone tracing technique as presented in [CNS*11] employs an octree structure and still attempts to exploit hardware based quadrilinear filtering, it's octree nodes, or "bricks," must store an extra layer of redundant information which must be transferred to neighboring bricks, which incurs significant cost and complexity.

However, once the filtered sparse voxel octree is constructed, it provides an

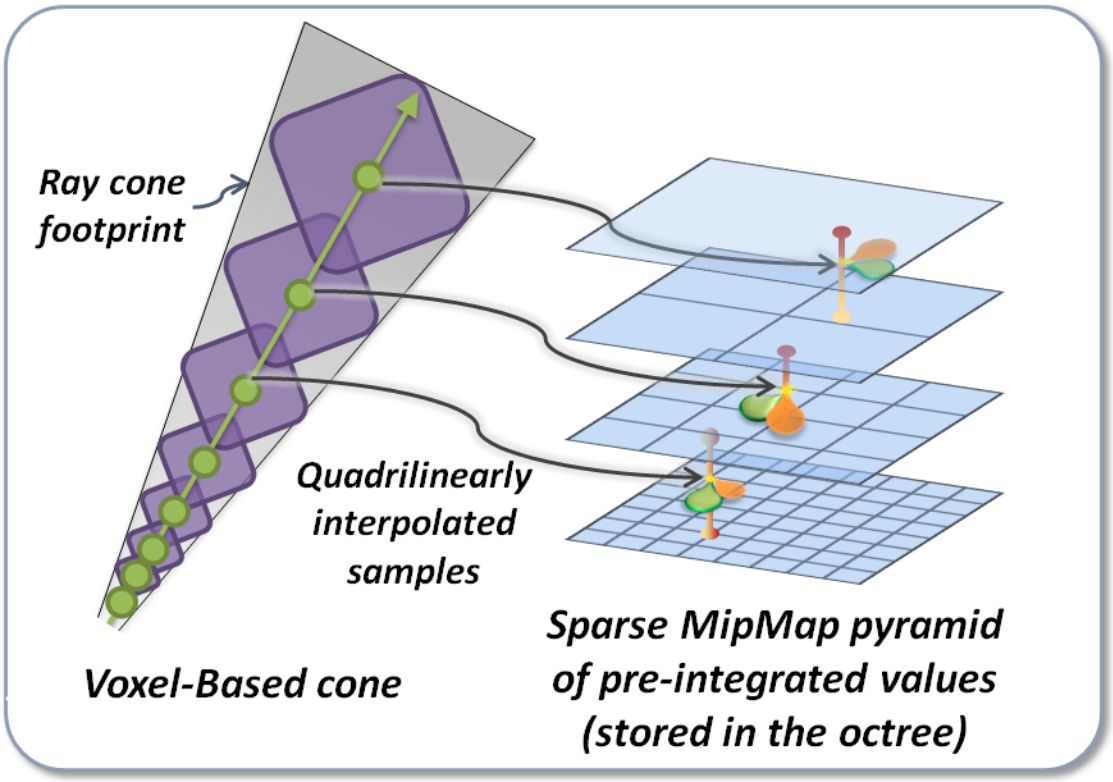


Figure 3.11: Illustration of the sampling scheme employed by voxel cone tracing, the cone radius at a sample point indicates the depth in the octree to sample from. Image credit: [CNS*11]

excellent structure from which to compute global illumination. In a deferred rendering context, for each pixel, a small set of cones approximating a BRDF is traced within the structure to gather incident radiance. As the cone expands the sampling rate decreases and samples are taken from higher levels of the filtered octree, see Figure 3.11. This fact makes voxel cone tracing extremely efficient, while the filtered voxel information allows the cone based sampling to rapidly approximate the contribution from a large amount of geometry. Varying the cone aperture and distribution allows voxel cone tracing to approximate different BRDFs, for example, a uniformly distributed set of wide cones allows for the approximation of a diffuse BRDF, while a single narrow cone traced in the direction of reflection about the normal can provide the specular component of a glossy BRDF.

3.7 Voxelization

Approaches to voxelization take many forms, and must balance several properties. One of the earlier approaches to utilize the graphics pipeline, [FC00] constructed a surface voxelization via rasterizing the geometry for each voxel slice while clamping the viewport to each slice. [LFWK05] introduced "depth peeling" which reduced the number of rendering passes by capturing 1-level of surface depth complexity per render pass. These approaches tended to miss voxels, and often must be applied once along each orthogonal plane to capture missed geometry. [DCB*04] utilized binary encoding to store voxel occupancy in separate bits of multi-channel render targets, allowing them to process multiple voxel slices in a single rendering pass.

This approach is sometimes referred to as a *slicemap* [ED06].

Approaches exist, such as conservative voxelization by [ZCEP07], which employ the conservative rasterization technique of [HAMO05]. This approach amplified single triangles to potentially nine triangles by expanding triangle vertices to pixel sized squares and outputting the convex hull of the resultant geometry. [SEA08] improved on this by ensuring that fewer triangles would be generated during triangle expansion, while [HHW09] found it was most effective to simply expand triangles by half the diagonal of a pixel and discard extra fragments in the pixel shader.

Some voxelization techniques also target solid voxelization; generally, these must restrict their input geometry to closed, watertight models, and classify voxels as either interior or exterior. As surface geometry is voxelized, entire columns of voxels are set, final classification is based on the count, or parity, of the voxel. An odd value indicates a voxel as interior, while even indicates exterior. In GPU hardware this corresponds to applying a logical XOR which is supported by the frame buffer. [FC00] presented such an approach using slice-wise rendering, while [ED08] developed a high-performance single pass approach.

Most recently, [CG12] have released an approach that operates similarly to the fragment-parallel component of our scheme, discussed in section 4.1.2, exploiting the recently exposed ability to perform random texture writes in OpenGL using the image API. By constructing an orthographic projection matrix per-triangle in the geometry shader, they were able to rely on the OpenGL rasterizer to voxelize their geometry.

More recently, approaches have been developed which take an explicitly computational approach to voxelization without utilizing fixed function hardware. [SS10] implemented a triangle parallel voxelization approach in CUDA, which achieved accurate 6- and 26-separating binary voxelization into a sparse hierarchical octree. Pantaleoni's VoxelPipe [Pan11] implementation took a similar approach while fully supporting a variety of render targets and robust blending support. Both approaches also employed a tile-based voxelization.

Chapter 4: Voxelization

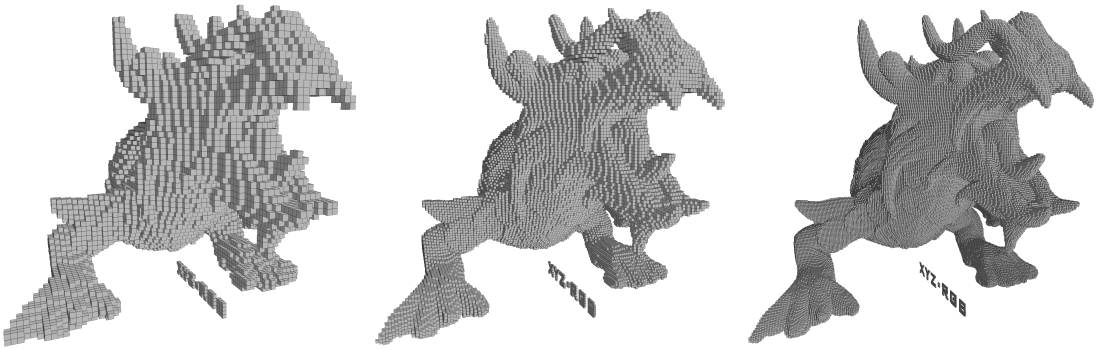


Figure 4.1: The XYZ RGB Asian Dragon voxelized at 128^3, 256^3, and 512^3 resolutions.

In this chapter we cover our work on voxelization. Related work in voxelization can be seen in Section 3.7. In Section 4.1 we discuss first our triangle-parallel and fragment-parallel approaches and how we combine them for our hybrid implementation. Additionally, we discuss several Voxel-List construction methods, and a method to correctly interpolate attributes using barycentric coordinates. This is followed by a look at the comparative performance of our method, Section 4.2, and finally a discussion of our findings with respect to voxelization, Section 4.3,

Our voxelization approach largely follows the techniques described in [RB13], with modifications and extensions suitable to the application of Global Illumination. Primarily these extensions involve methods to store appropriate attributes at voxel locations (see Chapter 5), and modifications to generate active-voxel-lists

in order to accelerate mipmapping. The mipmapping process is covered in greater detail in Section 5.3.

[RB13] proposed a hybrid voxelization pipeline which adapted previous computational approaches to the context of the graphics pipeline, and divided the voxelization workload between Triangle-Parallel and Fragment-Parallel techniques.

4.1 Voxelization

Whereas previous techniques relied exclusively on the graphics pipeline, or rejected it completely for a computational approach, we demonstrate how to find a middle ground to apply the techniques of computational voxelization approaches within the framework of the graphics pipeline. First, however, we must introduce both the triangle-parallel (section 4.1.1) and fragment-parallel (section 4.1.2) techniques which make up the primary components of our hybrid approach (section 4.1.3). Both techniques employ the same 3D extension of the [AM05] triangle/box overlap tests found in [SS10] and [Pan11]. These approaches differ from each other primarily in their factorization of the computational overlap testing, and the methods in which they try to achieve optimal parallelism.

Triangle/Voxel Overlap We can consider the exercise of finding an intersection between a triangle $\mathcal{T}$ (with vertices $\mathbf{v}_0$, $\mathbf{v}_1$, $\mathbf{v}_2$ and edges $\mathbf{e}_i = \mathbf{v}_{(i+1)\%3} - \mathbf{v}_i$) and a voxel $\mathbf{p}$ to be fundamentally an exercise in first reducing the number of triangle voxel pairs to consider, and secondly, an effort in reducing the computation

required to confirm an intersection between a triangle and a voxel. Considering initially the potential intersection between a triangle and the set of all voxels, conceptually, the process is executed in the following order.

1. Reduce the set of potential voxel intersections to only those that overlap the axis-aligned bounding volume $\mathbf{b}$ of the triangle.

2. Iterate over this reduced set of voxels (from $\mathbf{b}_{\min}$ to $\mathbf{b}_{\max}$) and discard any that do not intersect the triangle's plane.

3. If the triangle plane divides the voxels, test all three of its 2D planar projections $\left(\mathcal{T}^{\mathrm{XY}},\mathcal{T}^{\mathrm{YZ}},\mathcal{T}^{\mathrm{ZX}}\right)$ to confirm overlap.

The steps above rely heavily on point to plane, and point to line distance calculations. For instance, the plane overlap test relies on computing the signed distance to the plane from two points on opposite ends of the voxel, let us call these points $\mathbf{p}_{\min}$ and $\mathbf{p}_{\max}$. If these distances have opposite signs, i.e. $\mathbf{p}_{\min}$ and $\mathbf{p}_{\max}$ are on opposite sides of the plane, this indicates overlap. The selection of $\mathbf{p}_{\min}$ and $\mathbf{p}_{\max}$ determines the separability of the resultant voxelization, see figure 4.2.

Similarly, when testing the triangle projections $\left(\mathcal{T}^{\mathrm{XY}},\mathcal{T}^{\mathrm{YZ}},\mathcal{T}^{\mathrm{ZX}}\right)$ against their respective voxel projections $\left(\mathbf{p}^{\mathrm{XY}},\ \mathbf{p}^{\mathrm{YZ}},\ \mathbf{p}^{\mathrm{ZX}}\right)$, we use the projected inward facing edge normals ($\mathbf{n}_{\mathbf{e}_i}^{\mathrm{XY}}$, $\mathbf{n}_{\mathbf{e}_i}^{\mathrm{YZ}}$, $\mathbf{n}_{\mathbf{e}_i}^{\mathrm{ZX}}$ for $i = 0, 1, 2$) to select the "most interior" point on the box for each edge ($\mathbf{e}_i^{\mathrm{XY}}$, $\mathbf{e}_i^{\mathrm{YZ}}$, $\mathbf{e}_i^{\mathrm{ZX}}$ for $i = 0, 1, 2$), and if all projected edge to interior point distances are positive, this indicates overlap within that projection, see figure 4.3.

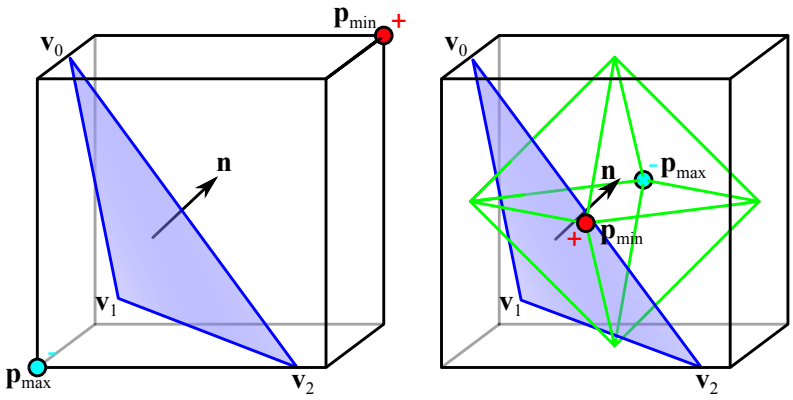


Figure 4.2: $\mathbf{p}_{\min}$ and $\mathbf{p}_{\max}$ for 26-separable voxelization on left, and for 6-separable voxelization on right. Note that for 6-separable voxelization we are actually testing for intersection of the diamond shape inscribed inside the voxel as opposed to the entire voxel in the 26-separable case.

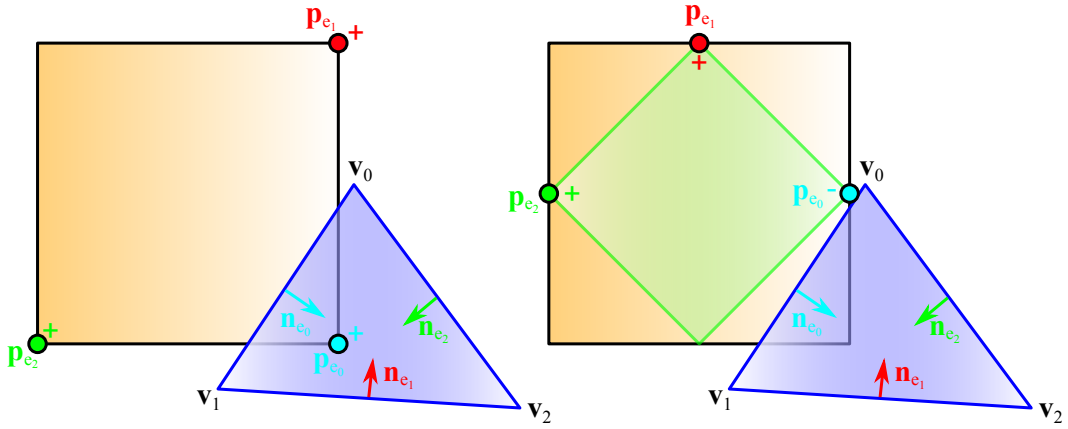


Figure 4.3: $\mathbf{p}_{\mathbf{e}_i}$ for 26-separable voxelization on the left, and for 6-separable voxelization on the right. Similar to the plane-overlap test, the 6-separable voxelization is actually testing against the diamond inscribed inside the voxel's planar projection.

Factorization As described in [SS10] and [Sch12], the points $\mathbf{p}_{\min}$ and $\mathbf{p}_{\max}$ and $\mathbf{p}_{\mathbf{e}_i}^{\mathrm{XY}}$, $\mathbf{p}_{\mathbf{e}_i}^{\mathrm{YZ}}$, $\mathbf{p}_{\mathbf{e}_i}^{\mathrm{ZX}}$ (for $i = 0, 1, 2$) are determined with the aid of an offset vector, known as a *critical point*, which is determined by the relevant normal. However, if we take the distance calculations and refactor them such that minimal computation occurs while iterating over the voxels, i.e. factor out all computations not directly dependent on the voxel coordinates of $\mathbf{p}$, we can actually simplify the expressions to the point that the critical point and the points $\mathbf{p}_{\min}$ and $\mathbf{p}_{\max}$ and $\mathbf{p}_{\mathbf{e}_i}^{\mathrm{XY}}$, $\mathbf{p}_{\mathbf{e}_i}^{\mathrm{YZ}}$, $\mathbf{p}_{\mathbf{e}_i}^{\mathrm{ZX}}$ for $i = 0, 1, 2$ need never be determined. Instead we substitute per-triangle variables $d_{\min}$, $d_{\max}$ and $d_{\mathbf{e}_i}^{\mathrm{XY}}$, $d_{\mathbf{e}_i}^{\mathrm{YZ}}$, $d_{\mathbf{e}_i}^{\mathrm{ZX}}$ (for $i = 0, 1, 2$), which represent the factored out components of the distance calculation not dependent on the voxel coordinates.

Optimization There are several ways in which we can optimize this process with an eye towards reducing the amount of computation that occurs in the innermost loops of our bounding box traversal.

1. The first involves pre-computing all per-triangle variables, which includes the triangle normal $\mathbf{n}$, the nine planar projected edge normals $\mathbf{n}_{\mathbf{e}_i}^{\mathrm{XY}}$,$\mathbf{n}_{\mathbf{e}_i}^{\mathrm{YZ}}$, $\mathbf{n}_{\mathbf{e}_i}^{\mathrm{ZX}}$ (for $i = 0, 1, 2$), and the eleven factored variables $d_{\mathbf{e}_i}^{\mathrm{XY}}$, $d_{\mathbf{e}_i}^{\mathrm{YZ}}$, $d_{\mathbf{e}_i}^{\mathrm{ZX}}$ (for $i = 0, 1, 2$), $d_{\min}$, and $d_{\max}$.

2. Determine the dominant normal direction, and use this to select the orthogonal plane of maximal projection (XY, YZ, or ZX), then iterate over the component axes of this plane first, the remaining axis we shall refer to as the depth-axis.

3. Test the 2D projected overlap with the orthogonal plane of maximal projection first.

4. Replace the plane overlap test with an intersection test along the depth-axis test to determine the minimal necessary range to iterate over (rather than the entire range of the bounding box along the depth-axis).

5. Test the remaining two planar projections for overlap.

Should all of these tests succeed, we can confirm that triangle $\mathcal{T}$ intersects voxel $\mathbf{p}$. Pseudocode for both conservative and thin voxelization routines is provided in Figures 4.4 and 4.5, respectively. For more detail on the triangle/box overlap test, the reader is referred to [SS10, Sch12] and [Pan11].

4.1.1 Triangle-parallel voxelization

The most natural approach to voxelization of an input mesh is to parallelize on the input geometry (i.e. the triangles). [Sch12] implemented such an approach in a Direct3D Compute shader as a single pass. [SS10, Pan11] implemented a multi-pass approach to improve parallelism. [SS10] improved coherence by specializing the triangle-box intersection code into nine different voxel-dependent cases; 1D bounding boxes along each axis; 2D bounding boxes in each coordinate plane; and 3D bounding boxes for three dominant normal directions. Unfortunately this requires a 2-pass approach, and while it results in high thread coherence (since kernels operate exclusively on similar triangles), it is quite complex, and exceeds

function conservativeVoxelize($\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{b}_{\min}, \mathbf{b}_{\max}, unswizzle$)
  $\mathbf{e}_i \leftarrow \mathbf{v}_{(i+1)\bmod 3} - \mathbf{v}_i$
  $\mathbf{n} \leftarrow \operatorname{cross}(\mathbf{e}_0, \mathbf{e}_1)$
  $\mathbf{n}^{\mathrm{XY}}_{\mathbf{e}_i} \leftarrow \operatorname{sign}(\mathbf{n}_{\mathrm{z}}) \cdot (-\mathbf{e}_{i,\mathrm{y}}, \mathbf{e}_{i,\mathrm{x}})^{\mathrm{T}}$
  $\mathbf{n}^{\mathrm{YZ}}_{\mathbf{e}_i} \leftarrow \operatorname{sign}(\mathbf{n}_{\mathrm{x}}) \cdot (-\mathbf{e}_{i,\mathrm{z}}, \mathbf{e}_{i,\mathrm{y}})^{\mathrm{T}}$
  $\mathbf{n}^{\mathrm{ZX}}_{\mathbf{e}_i} \leftarrow \operatorname{sign}(\mathbf{n}_{\mathrm{y}}) \cdot (-\mathbf{e}_{i,\mathrm{x}}, \mathbf{e}_{i,\mathrm{z}})^{\mathrm{T}}$
  $d^{\mathrm{XY}}_{\mathbf{e}_i} \leftarrow -\left\langle \mathbf{n}^{\mathrm{XY}}_{\mathbf{e}_i}, \mathbf{v}_{i,\mathrm{xy}} \right\rangle + \max\left(0, \mathbf{n}^{\mathrm{XY}}_{\mathbf{e}_i,\mathrm{x}}\right) + \max\left(0, \mathbf{n}^{\mathrm{XY}}_{\mathbf{e}_i,\mathrm{y}}\right)$
  $d^{\mathrm{YZ}}_{\mathbf{e}_i} \leftarrow -\left\langle \mathbf{n}^{\mathrm{YZ}}_{\mathbf{e}_i}, \mathbf{v}_{i,\mathrm{yz}} \right\rangle + \max\left(0, \mathbf{n}^{\mathrm{YZ}}_{\mathbf{e}_i,\mathrm{x}}\right) + \max\left(0, \mathbf{n}^{\mathrm{YZ}}_{\mathbf{e}_i,\mathrm{y}}\right)$
  $d^{\mathrm{ZX}}_{\mathbf{e}_i} \leftarrow -\left\langle \mathbf{n}^{\mathrm{ZX}}_{\mathbf{e}_i}, \mathbf{v}_{i,\mathrm{zx}} \right\rangle + \max\left(0, \mathbf{n}^{\mathrm{ZX}}_{\mathbf{e}_i,\mathrm{x}}\right) + \max\left(0, \mathbf{n}^{\mathrm{ZX}}_{\mathbf{e}_i,\mathrm{y}}\right)$
  $\mathbf{n} \leftarrow \operatorname{sign}(\mathbf{n}_{\mathrm{z}}) \cdot \mathbf{n}$ // ensures $z_{\min} < z_{\max}$
  $d_{\min} \leftarrow \langle \mathbf{n}, \mathbf{v}_0 \rangle - \max(0, \mathbf{n}_{\mathrm{x}}) - \max(0, \mathbf{n}_{\mathrm{y}})$
  $d_{\max} \leftarrow \langle \mathbf{n}, \mathbf{v}_0 \rangle - \min(0, \mathbf{n}_{\mathrm{x}}) - \min(0, \mathbf{n}_{\mathrm{y}})$
  for $\mathbf{p}_{\mathrm{x}} \leftarrow \mathbf{b}_{\min,\mathrm{x}}, \ldots, \mathbf{b}_{\max,\mathrm{x}}$ **do**
    for $\mathbf{p}_{\mathrm{y}} \leftarrow \mathbf{b}_{\min,\mathrm{y}}, \ldots, \mathbf{b}_{\max,\mathrm{y}}$ **do**
      if $\forall_{i=0}^{2} \left(\left\langle \mathbf{n}^{\mathrm{XY}}_{\mathbf{e}_i}, \mathbf{p}_{\mathrm{xy}} \right\rangle + d^{\mathrm{XY}}_{\mathbf{e}_i} \geq 0\right)$ **then**
        $z_{\min} \leftarrow \max\left(\mathbf{b}_{\min,\mathrm{z}}, \left\lfloor \left(-\left\langle \mathbf{n}_{\mathrm{xy}}, \mathbf{p}_{\mathrm{xy}} \right\rangle + d_{\min}\right) \frac{1}{\mathbf{n}_{\mathrm{z}}} \right\rfloor\right)$
        $z_{\max} \leftarrow \min\left(\mathbf{b}_{\max,\mathrm{z}}, \left\lceil \left(-\left\langle \mathbf{n}_{\mathrm{xy}}, \mathbf{p}_{\mathrm{xy}} \right\rangle + d_{\max}\right) \frac{1}{\mathbf{n}_{\mathrm{z}}} \right\rceil\right)$
        for $\mathbf{p}_{\mathrm{z}} \leftarrow z_{\min}, \ldots, z_{\max}$ **do**
          if $\forall_{i=0}^{2} \left(\left\langle \mathbf{n}^{\mathrm{YZ}}_{\mathbf{e}_i}, \mathbf{p}_{\mathrm{xy}} \right\rangle + d^{\mathrm{YZ}}_{\mathbf{e}_i} \geq 0 \wedge \left\langle \mathbf{n}^{\mathrm{ZX}}_{\mathbf{e}_i}, \mathbf{p}_{\mathrm{xy}} \right\rangle + d^{\mathrm{ZX}}_{\mathbf{e}_i} \geq 0\right)$ **then**
            $V\left[unswizzle \cdot \mathbf{p}\right] \leftarrow$ **true**
end function

Figure 4.4: Pseudocode for a conservative (26-separable) computational voxelization, this assumes that the inputs, $\mathbf{v}_0$, $\mathbf{v}_1$, $\mathbf{v}_2$, $\mathbf{b}_{\min}$, and $\mathbf{b}_{\max}$, are pre-swizzled, while *unswizzle* represents a permutation matrix used to get the unswizzled voxel location.

function thinVoxelize($\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{b}_{\min}, \mathbf{b}_{\max}, \mathit{unswizzle}$)
  $\mathbf{e}_i \leftarrow \mathbf{v}_{(i+1) \mod 3} - \mathbf{v}_i$
  $\mathbf{n} \leftarrow \operatorname{cross}(\mathbf{e}_0, \mathbf{e}_1)$
  $\mathbf{n}_{\mathbf{e}_i}^{\mathrm{XY}} \leftarrow \operatorname{sign}(\mathbf{n}_{\mathrm{z}}) \cdot (-\mathbf{e}_{i,\mathrm{y}}, \mathbf{e}_{i,\mathrm{x}})^{\mathrm{T}}$
  $\mathbf{n}_{\mathbf{e}_i}^{\mathrm{YZ}} \leftarrow \operatorname{sign}(\mathbf{n}_{\mathrm{x}}) \cdot (-\mathbf{e}_{i,\mathrm{z}}, \mathbf{e}_{i,\mathrm{y}})^{\mathrm{T}}$
  $\mathbf{n}_{\mathbf{e}_i}^{\mathrm{ZX}} \leftarrow \operatorname{sign}(\mathbf{n}_{\mathrm{y}}) \cdot (-\mathbf{e}_{i,\mathrm{x}}, \mathbf{e}_{i,\mathrm{z}})^{\mathrm{T}}$
  $d_{\mathbf{e}_i}^{\mathrm{XY}} \leftarrow \langle \mathbf{n}_{\mathbf{e}_i}^{\mathrm{XY}}, 0.5 - \mathbf{v}_{i,\mathrm{xy}} \rangle + 0.5 \cdot \max\left(\left|\mathbf{n}_{\mathbf{e}_i,\mathrm{x}}^{\mathrm{XY}}\right|, \left|\mathbf{n}_{\mathbf{e}_i,\mathrm{y}}^{\mathrm{XY}}\right|\right)$
  $d_{\mathbf{e}_i}^{\mathrm{YZ}} \leftarrow \langle \mathbf{n}_{\mathbf{e}_i}^{\mathrm{YZ}}, 0.5 - \mathbf{v}_{i,\mathrm{yz}} \rangle + 0.5 \cdot \max\left(\left|\mathbf{n}_{\mathbf{e}_i,\mathrm{x}}^{\mathrm{YZ}}\right|, \left|\mathbf{n}_{\mathbf{e}_i,\mathrm{y}}^{\mathrm{YZ}}\right|\right)$
  $d_{\mathbf{e}_i}^{\mathrm{ZX}} \leftarrow \langle \mathbf{n}_{\mathbf{e}_i}^{\mathrm{ZX}}, 0.5 - \mathbf{v}_{i,\mathrm{zx}} \rangle + 0.5 \cdot \max\left(\left|\mathbf{n}_{\mathbf{e}_i,\mathrm{x}}^{\mathrm{ZX}}\right|, \left|\mathbf{n}_{\mathbf{e}_i,\mathrm{y}}^{\mathrm{ZX}}\right|\right)$
  $\mathbf{n} \leftarrow \operatorname{sign}(\mathbf{n}_{\mathrm{z}}) \cdot \mathbf{n}$ // ensures $z_{\min} < z_{\max}$
  $d_{\mathrm{cen}} \leftarrow \langle \mathbf{n}, \mathbf{v}_0 \rangle - 0.5 \cdot \mathbf{n}_{\mathrm{x}} - 0.5 \cdot \mathbf{n}_{\mathrm{y}}$
  for $\mathbf{p}_{\mathrm{x}} \leftarrow \mathbf{b}_{\min,\mathrm{x}}, \ldots, \mathbf{b}_{\max,\mathrm{x}}$ **do**
    for $\mathbf{p}_{\mathrm{y}} \leftarrow \mathbf{b}_{\min,\mathrm{y}}, \ldots, \mathbf{b}_{\max,\mathrm{y}}$ **do**
      if $\forall_{i=0}^{2} \left(\langle \mathbf{n}_{\mathbf{e}_i}^{\mathrm{XY}}, \mathbf{p}_{\mathrm{xy}} \rangle + d_{\mathbf{e}_i}^{\mathrm{XY}} \geq 0\right)$ **then**
        $z_{\mathrm{int}} \leftarrow \left(-\langle \mathbf{n}_{\mathrm{xy}}, \mathbf{p}_{\mathrm{xy}} \rangle + d_{\mathrm{cen}}\right) \frac{1}{\mathbf{n}_{\mathrm{z}}}$
        $z_{\min} \leftarrow \max\left(\mathbf{b}_{\min,\mathrm{z}}, \lfloor z_{\mathrm{int}} \rfloor\right)$
        $z_{\max} \leftarrow \min\left(\mathbf{b}_{\max,\mathrm{z}}, \lceil z_{\mathrm{int}} \rceil\right)$
          for $\mathbf{p}_{\mathrm{z}} \leftarrow z_{\min}, \ldots, z_{\max}$ **do**
            if $\forall_{i=0}^{2} \left(\langle \mathbf{n}_{\mathbf{e}_i}^{\mathrm{YZ}}, \mathbf{p}_{\mathrm{xy}} \rangle + d_{\mathbf{e}_i}^{\mathrm{YZ}} \geq 0 \wedge \langle \mathbf{n}_{\mathbf{e}_i}^{\mathrm{ZX}}, \mathbf{p}_{\mathrm{xy}} \rangle + d_{\mathbf{e}_i}^{\mathrm{ZX}} \geq 0\right)$ **then**
              $V\left[\mathit{unswizzle} \cdot \mathbf{p}\right] \leftarrow$ **true**
end function

Figure 4.5: Pseudocode for a thin (6-separable) computational voxelization, this assumes that the inputs, $\mathbf{v}_0$, $\mathbf{v}_1$, $\mathbf{v}_2$, $\mathbf{b}_{\min}$, and $\mathbf{b}_{\max}$, are pre-swizzled, while *unswizzle* represents a permutation matrix used to get the unswizzled voxel location.

the number of available image units commonly available. However, we can reduce this by a factor of three, allowing all 1D, 2D, and 3D cases to be treated the same by performing a simple transformation discussed in section 4.1.2.

Input geometry is first transformed into "voxel-space," that is the space ranging from $(0, 0, 0)^{\mathrm{T}}$ to $(\mathbf{V}_{\mathrm{x}},\ \mathbf{V}_{\mathrm{y}}, \mathbf{V}_{\mathrm{z}})^{\mathrm{T}}$, in the vertex shader. Second, an intersection routine implemented in the geometry shader, as described in section 4.1, performs the voxelization, the performance of which can be seen in figure 4.6. It is readily apparent that a naïve triangle-parallel approach only performs well in scenes that exhibit certain characteristics, for instance, the evenly tessellated XYZ RGB Dragon and Stanford Bunny models, both scenes that exhibit even and regular triangulation. Any scene that contains large triangles (such as might be found on a wall) like the Crytek Sponza Atrium, the Conference Room, or even, sadistically, a single large scene-spanning triangle, the naïve triangle-parallel approach has no mechanism by which to balance the workload, and the voxelization must wait while individual threads work alone to voxelize large triangles.

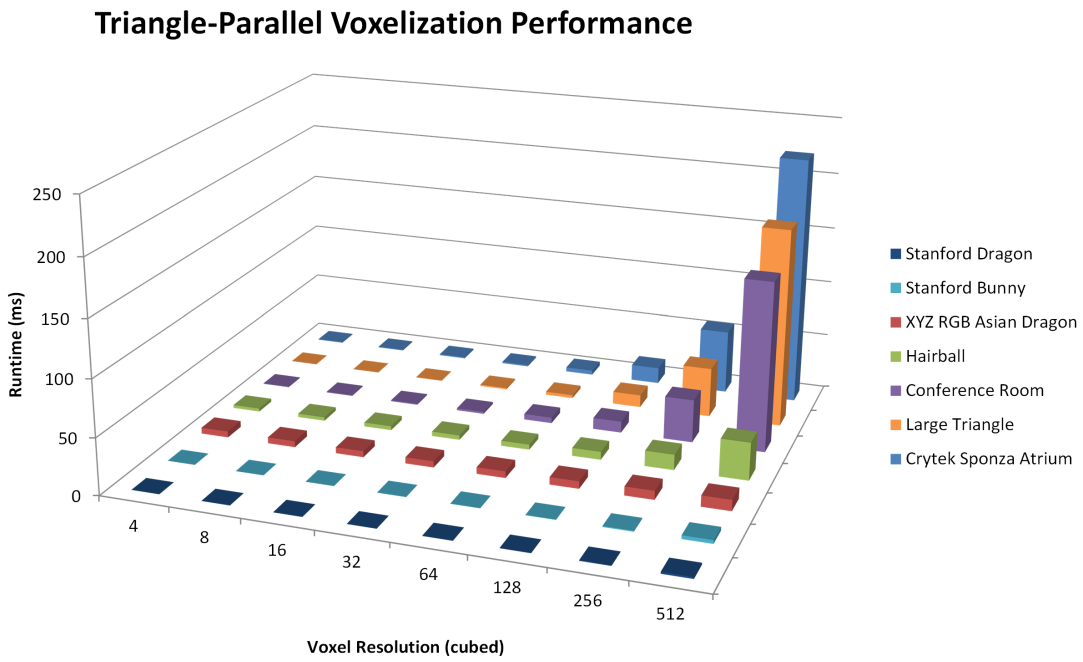


Figure 4.6: Performance of a naïve triangle-parallel voxelization. Exhibits poor performance on scenes containing large polygons. Performance decreases predictably with increase in voxel resolution.

4.1.2 Fragment-parallel voxelization

This observation of poor work-balance in unevenly tessellated scenes is what led Schwarz and Pantaleoni to introduce complex tile-assignment and sorting stages to their voxelization pipelines. Our fragment-parallel voxelization is based on the observation that much of our triangle-intersection routine can simply be moved to the fragment shader, providing the opportunity for vastly more parallelism. Thus, we exploit the fragment stage of the OpenGL pipeline as a sort of ad-hoc single-level of dynamic parallelism. There are several implementation particulars

required to ensure a gap-free voxelization, which will be discussed in a later section. The performance results of our single-pass fragment-parallel implementation can be observed in figure 4.7, and most noteworthy is the fact that it performs very well on the exact scenes that the triangle-parallel voxelization struggled with, and most poorly on scenes with large amounts of fine detailed geometry (XYZ RGB Dragon & Hairball).

The fragment-parallel implementation is far more unique and must be adapted to the pipeline in order to produce a correct voxelization. At present, only [CG12] describe a similar approach. Our utilization of the fragment stage allows us to benefit from the rasterization and interpolation acceleration provided by the graphics hardware. However, there are several issues we must concern ourselves with when endeavoring to produce a "gap-free" voxelization, (1) gaps within triangles caused by an overly oblique "camera" angle, and (2) gaps between triangles caused by OpenGL's rasterization rules.

As in the triangle-parallel approach values $\mathbf{n}$, $\mathbf{n}_{\mathbf{e}_i}^{\mathrm{XY}}$, $\mathbf{n}_{\mathbf{e}_i}^{\mathrm{YZ}}$, $\mathbf{n}_{\mathbf{e}_i}^{\mathrm{ZX}}$, $d_{\mathbf{e}_i}^{\mathrm{XY}}$, $d_{\mathbf{e}_i}^{\mathrm{YZ}}$, $d_{\mathbf{e}_i}^{\mathrm{ZX}}$ (for $i = 0, 1, 2$), $d_{\min}$, and $d_{\max}$ are precomputed. However, in this implementation they are calculated in the geometry shader, and passed as `flat` non-varying attributes to the fragment shader. Essentially, we allow the rasterizer to take over for iterating over the axes of the dominant planar projection, leaving the fragment shader to confirm overlap with the dominant plane, calculate the depth intersection range according to the desired separability rules, and confirm the remaining two planar projections. In the pseudocode in figures 4.4 and 4.5, the portion of code that would be moved into the fragment shaders goes from line 15 to line 20 in figure

4.4, and from line 14 to line 20 in figure 4.5.

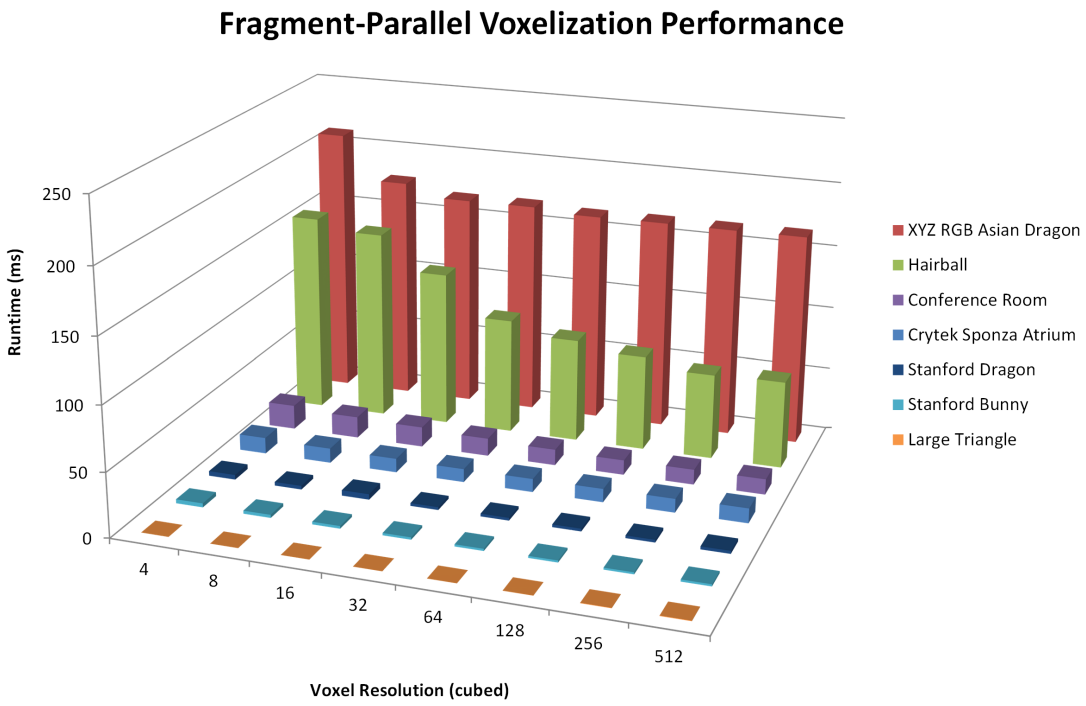


Figure 4.7: Performance of fragment-parallel voxelization. This exhibits poor-performance in scenes with large numbers of small triangles. Performance degradation is exacerbated as the ratio of voxel-size to triangle-size increases.

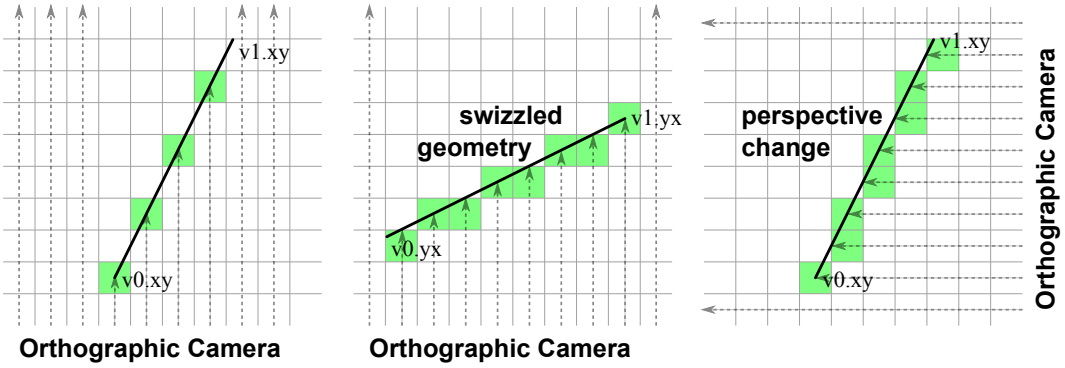


Figure 4.8: Naïve rasterization on input geometry can lead to gaps in the voxelization. This can be solved in two ways, the center image demonstrates swizzling the vertices of the input geometry, while the image on the right demonstrates changing the projection matrix.

Gap-Free Triangles We can solve the first problem, illustrated in figure 4.8, in one of two ways, both of which rely on determining the dominant normal direction of the triangle. The first approach relies on constructing an orthographic projection matrix per-triangle, which views the triangle against the axis of its maximum projection as determined by the dominant normal direction. Alternately, we can change the input geometry, again based on the dominant normal direction, such that the XY plane is always the axis of maximum projection. This can be accomplished by a simple hardware supported vector swizzle described below

$$\forall_{i=0}^{2}\mathbf{v}_{i,\mathrm{xyz}} = \begin{cases} \mathbf{v}_{i,\mathrm{yzx}} & \mathbf{n}_{\mathrm{x}} \text{ dominant} \\ \mathbf{v}_{i,\mathrm{zxy}} & \mathbf{n}_{\mathrm{y}} \text{ dominant} \\ \mathbf{v}_{i,\mathrm{xyz}} & \mathbf{n}_{\mathrm{z}} \text{ dominant} \end{cases} \tag{4.1}$$

However, we must be sure to "unswizzle" when storing in the destination texture. Additionally, a similar triangle swizzling approach can be used to reduce the number of cases taken in the [SS10] approach. With triangle swizzling, the number of cases drops from 9 to 3, one for each of the 1D, 2D, and 3D cases. Figure 4.9 depicts the selection of the largest triangle projection based on the dominant normal direction.

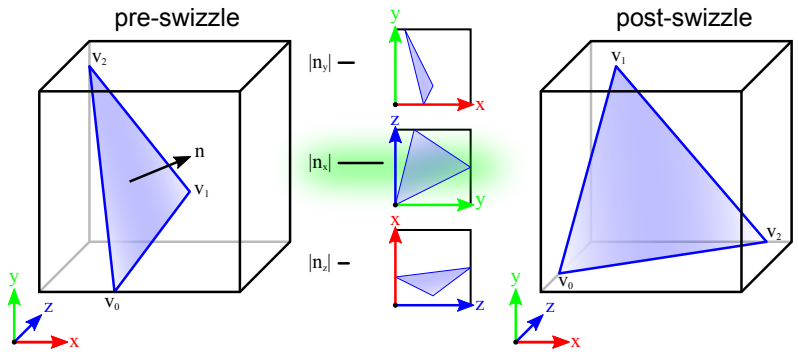


Figure 4.9: The largest component of the normal **n** of the original triangle determines the plane of maximal projection (XY, YZ, or ZX) and the corresponding swizzle operation to perform.

Conservative Rasterization The second problem can be solved with conservative rasterization. Conservative rasterization ensures that every pixel that touches a triangle is rasterized, which is counter to how the hardware rasterizer works. There are several approaches to overcome this, which generally involve "dilating" the input triangle. [HAMO05] dilated input triangles by expanding triangle vertices into pixel sized squares and computing the convex hull of the resultant geometry. Tessellation of this shape can be computed in the geometry shader.

Alternately, Hasselgren also proposed computing the bounding triangle of the dilated geometry from the previous approach and simply discarding in a fragment shader all fragments outside of the AABB. [HHW09] proposed a similar approach, computing the dilated triangle $\mathcal{T}'$ by constructing a triangle of intersecting lines parallel to the sides of the original triangle $\mathcal{T}$ at a distance of l, where l is half the length of the pixel diagonal, see figure 4.10 for examples of these techniques.

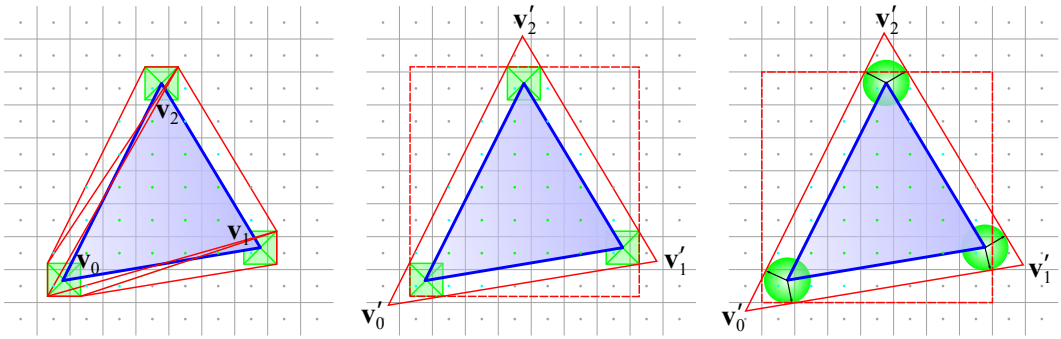


Figure 4.10: Various conservative rasterization techniques required in order to produce a "gap-free" voxelization. The first two images are from [HAMO05], the leftmost image shows the approach of expanding triangle vertices to size of pixel, and tessellating the resultant convex-hull. The middle image simply creates the minimal triangle to encompass the expanded vertices, and relies on clipping to occur later in the pipeline. The rightmost approach is from [HHW09], and simply expands the triangle by half the length of the pixel diagonal and also relies on clipping to remove unwanted pixels.

With the Hertel approach the dilated vertices $\mathbf{v}'_i$ of $\mathcal{T}'$ can be easily computed as

$$\mathbf{v}'_i = \mathbf{v}_i + l \quad \frac{\mathbf{e}_{i-1}}{\mathbf{e}_{i-1} \cdot \mathbf{n}_{\mathbf{e}_i}} + \frac{\mathbf{e}_i}{\mathbf{e}_i \cdot \mathbf{n}_{\mathbf{e}_{i-1}}} \quad . \tag{4.2}$$

In our case working on a 2D triangle projection in a premultiplied voxel space l will always be $\sqrt{2}/2$.

It should be noted that conservative rasterization has the potential to produce unnecessary overhead in the form of fragment threads that are ultimately rejected in the final voxelization intersection test. As triangles get smaller and l remains constant, the size of the dilated triangle $\mathcal{T}'$ to the size of the original triangle $\mathcal{T}$ causes the ratio $\frac{\text{area}(\mathcal{T})}{\text{area}(\mathcal{T}')}$ to become smaller. This ratio can be used to approximate an upper bound on the expected efficiency of per-triangle fragment thread utilization. This goes part of the way to explaining the fragment-parallel technique's poor performance in highly tessellated scenes with many small triangles, but is actually exacerbated further by poor quad utilization for small triangles. Since texture derivatives require neighbor information, even if only one pixel of a quad is covered, the entire quad is launched. This means that triangles smaller than a voxel will utilize only 25% of the threads allocated to them *before* triangle dilation is taken into account. After triangle dilation, thread utilization can be significantly worse, see figure 4.11, and in scenes with millions of sub-voxel sized triangles, can lead to massive oversubscription and poor performance.

Additionally, it was our observation that voxelization methods that relied purely on raster-based conservative voxelization methods tended to be overly conservative

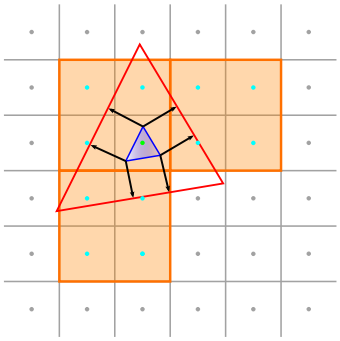


Figure 4.11: Sub-voxel sized triangle exhibiting thread utilization of only $8.\bar{3}\%$ after triangle dilation, note, that this can actually get much worse depending on the triangle configuration.

along their edges where clipping against the AABB couldn't help them, resulting in false positives, see figure 4.12. Since our approach maintains a computational intersection test inside the fragment shader, these voxels are still culled.

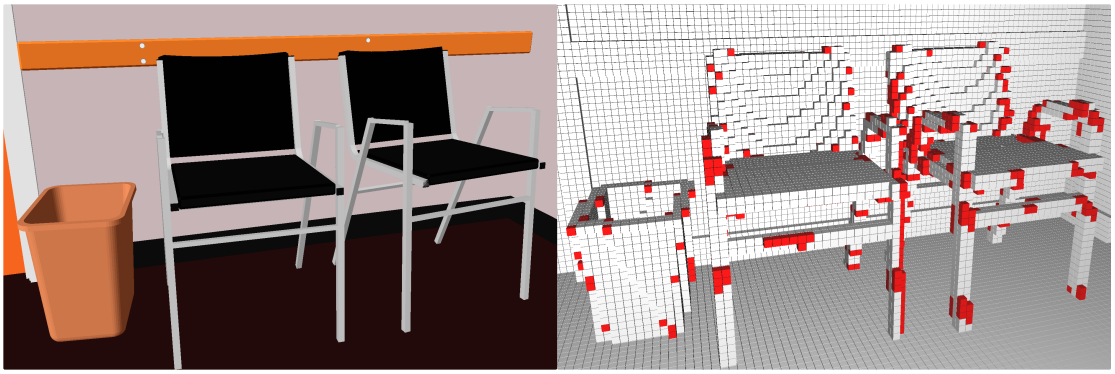


Figure 4.12: Thin (6-separable) voxelization of the Conference Room scene illustrating false positives (in red) resulting from a naïve conservative-rasterization based voxelization.

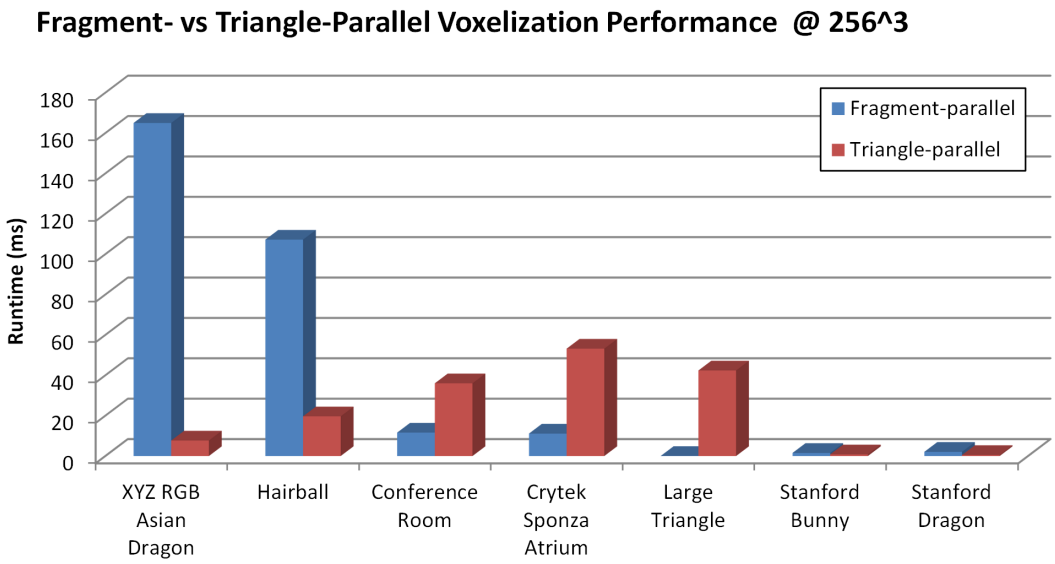


Figure 4.13: Comparison of the relative performance of Triangle-parallel and Fragment-parallel techniques. Note, where one technique performs poorly, the other performs well.

4.1.3 Hybrid Voxelization

Comparing the performance of both single-pass techniques side-by-side, as illustrated in figure 4.13, the inversion of strengths and weaknesses becomes even more apparent. By using the fragment shader to increase the available parallelism, the worst-case scenario for the triangle-parallel approach becomes the best case for the fragment-parallel case. Conversely, the best-case for the fragment-parallel approach is the worst case for the triangle-parallel approach. Thus, we logically arrive at a hybrid approach, one in which large triangles are divided into fragment-threads using the fragment-parallel technique, and small triangles are voxelized using the triangle-parallel technique, thus avoiding poor thread utilization and oversubscription.

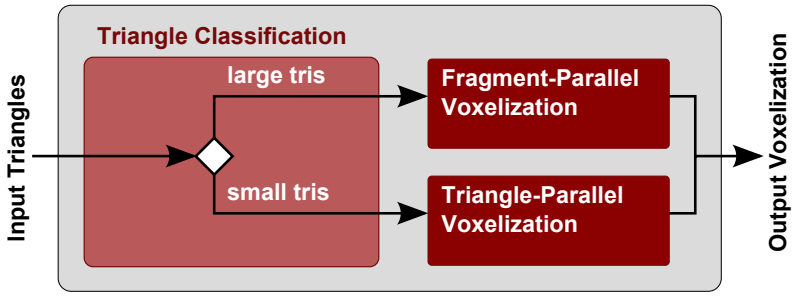


Figure 4.14: A simple classification routine run before the voxelization stage allows the creation of a hybrid voxelization pipeline and utilizes the optimal voxelization approach according to per-triangle characteristics.

We take care to preserve coherent execution among our shader threads with the introduction of a classification stage to our pipeline prior to voxelization, see figure 4.14, which outputs corresponding index buffers according to each triangle's classi-

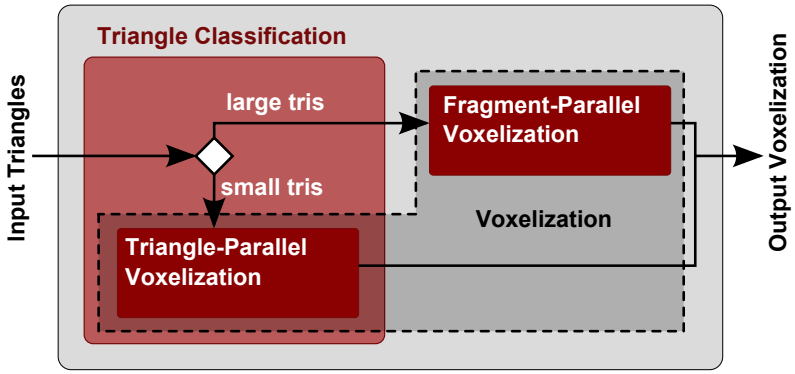


Figure 4.15: Our final hybrid voxelization implementation mitigates the cost processing the input geometry twice by immediately voxelizing input triangles classified as "small" and deferring only those triangles considered to be "large."

fication. These classified index buffers are then used to voxelize the corresponding geometry using the appropriate technique.

Triangle Selection Heuristic The crux of the hybrid-voxelization approach lies in the heuristic used for determining whether a triangle is most suitable for voxelization using a triangle-parallel approach or a fragment-parallel approach. The [SS10] approach is dependent on voxel extents of triangle bounding boxes, however, we have already determined that the fragment-parallel approach will handle all large triangles, and the triangle-parallel approach will handle all small triangles.

The heuristic for the selection of a cutoff value can be approached in many different ways, for instance, the size of the dilated triangle area ($\mathcal{T}'$) most accurately represent the number of potential voxel intersections to be evaluated in the fragment stage, but is not a fair representation of the amount of work required in the

triangle-parallel stage should the triangle be classified as small. Furthermore, the dilated triangle has a minimum size, which must be considered as undilated triangles approach zero area. The 3D voxel-extents provide a good indication of the amount of iteration required to voxelize a triangle in the geometry stage, however, since the depth-range is calculated, the 2D-projected voxel-extents provide a closer representation of the actual work performed. Additionally, we could consider the ratio of $\frac{\text{area}(\mathcal{T})}{\text{area}(\mathcal{T}')}$, which, as it varies from 0 to 1, indicates very small to very large triangles, respectively.

In our experiments, we found that simply considering the 2D projected area of the triangle $\mathcal{T}$ worked best, and for most scenes, a cutoff value of just a few voxel units squared provided a good starting cutoff value for triangle classification. In figure 4.16 we can see the full range of voxelization performance vary from that of the fragment-parallel approach at a cutoff of zero, to the performance of the triangle-parallel approach once the cutoff is large enough to encompass all triangles. Note that figure 4.16 represents an unreasonable range of cutoff values; this is meant to illustrate the performance characteristics as the cutoff value changes. Generally, there is a fairly large range of cutoff values corresponding to near-optimal performance.

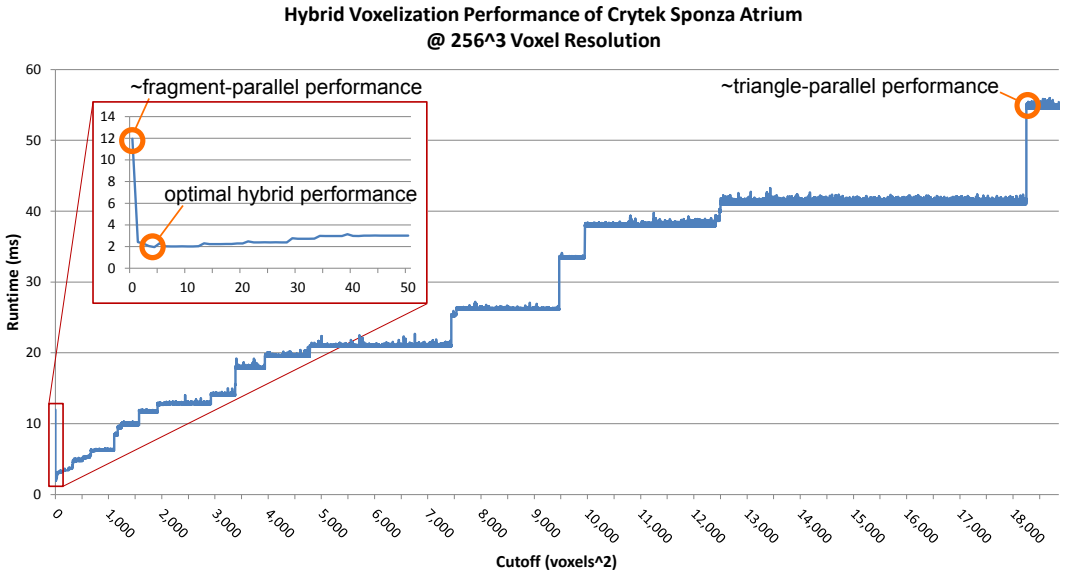


Figure 4.16: Initially at zero, all triangles are classified as "large" and therefore voxelized by the fragment-parallel shader. As the cutoff value (measured in voxel area) increases, triangles are classified and assigned to either the triangle-parallel or fragment-parallel approaches. As the cutoff continues to increase, performance exhibits a stair-step pattern as triangles are reclassified. Eventually all triangles are classified as "small" and performance reverts to that of the triangle-parallel approach.

We are, however, most interested in the cutoff value that will provide the minimal voxelization time, and these values tend to occur at much lower values. Figure 4.17 shows only the earlier range of cutoff values. Examination of the data confirms that for most inputs a cutoff value of just a few voxels squared provides for optimal voxelization timing. It is conceivable that a bracketing search could determine and adjust this value automatically [PTVF07].

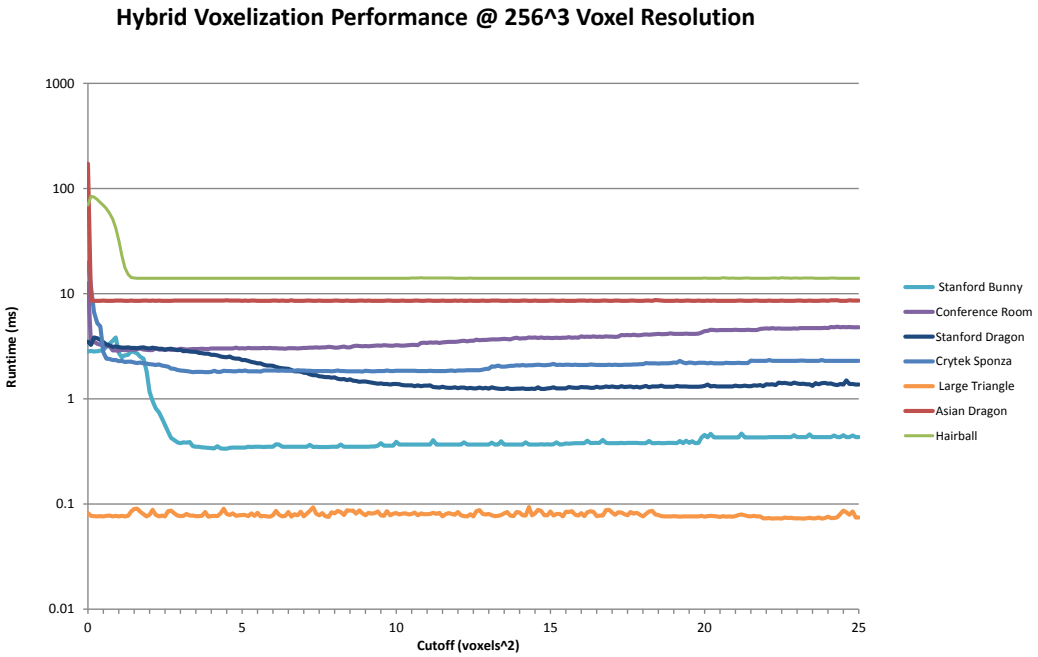


Figure 4.17: Logarithmic performance graph of the hybrid voxelization technique displaying a lower range of cutoff values such that the optimal cutoff can be clearly discerned.

Optimization In order to avoid requiring separate output buffers for all input attributes, we output only index buffers which are then used to render only the appropriate subset of the geometry with the voxelization method as determined by the classifier. On many scenes this allowed us to achieve improved performance over either the fragment-parallel or the triangle-parallel approach alone. However, when we examine the performance of a scene ideally suited to the triangle-parallel approach like the XYZ RGB Dragon, we observe that the best performance that can be achieved with our triangle-classifier is approximately twice that of the triangle-parallel approach alone. This can be explained by the amount of work it takes to

process the 7 million triangles in the scene. Each triangle is extremely small (generally less than the size of a voxel) and takes relatively little work to voxelize, and similarly little work to classify. In this case, run-time is dominated by the overhead of creating threads, rather than the work done in each thread, and with our current approach we have doubled the number of threads to be created. Fortunately, we can exploit the fact that in our classification, we employ the triangle-parallel approach only for small triangles. Combined with the fact that the number of small triangles in a scene almost always dominates the number of large triangles, we can dramatically decrease the overhead of our hybrid voxelization pipeline. As illustrated in figure 4.15, by moving the triangle-parallel voxelization into the classification shader and deferring only the larger triangles to be voxelized by the fragment shader, we effectively reduce a two-pass approach to a just slightly over one-pass approach, meaning, that while all triangles are processed at least once, only a few are processed twice. Furthermore, since the overhead of classification and voxelization of small triangles is so low, this makes our hybrid approach competitive on all scenes, even those tailored for a triangle-parallel approach. The full pipeline is shown in figure 4.18, illustrating the voxelization of the XYZ RGB Dragon scene.

4.1.4 Voxel-List Construction

Though we are primarily concerned with producing a voxelization stored in a dense 3D texture, it can also be useful to produce a sparse "voxel-list." Previously, it

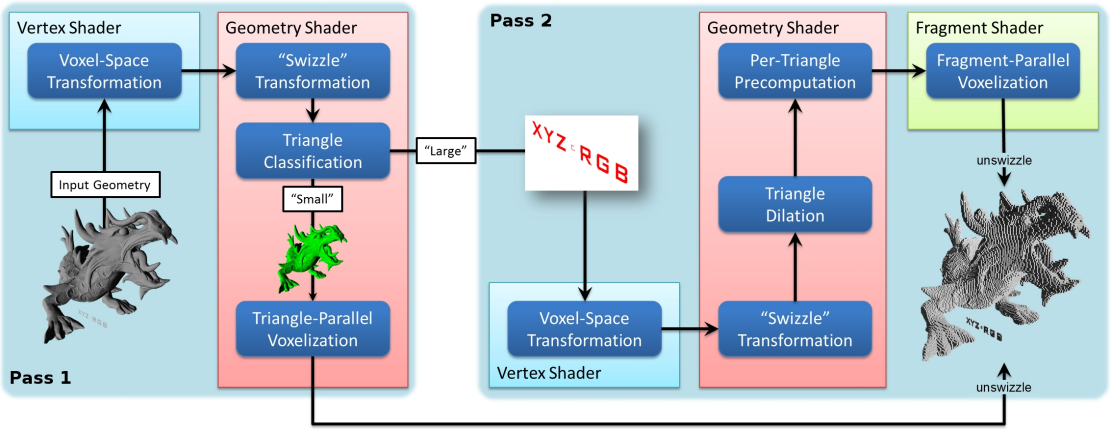


Figure 4.18: Full pipeline including shader stages. Note that while there are two "passes" only a very small subset of the geometry, that is classified as "large," is processed twice.

would be necessary to perform a dense voxelization and then perform a reduction, such as HistoPyramid compaction [ZTTS06], in order to produce such a list. However, with hardware support for atomic operations, this step can now be skipped. We can instead use an atomic counter to increment the index of an output buffer used to store the voxel's coordinates. [CG12] used such a technique to generate their "voxel-fragment-list," which they then used to construct a sparse hierarchical octree. With such an approach, multiple elements may refer to the same voxel location, which are later merged in hierarchy creation. To avoid duplicate voxel assignments, a dense 3D `r32ui` texture can be employed to provide mutexes at each voxel location. By employing an `imageAtomicCompSwap` operation at the voxel location, we can restrict incrementing the atomic counter to a single thread accessing the voxel location. This can be beneficial when your voxelization includes additional attribute outputs and there is not enough memory for a dense

3D texture for each attribute.

The reduced memory requirements of voxel-lists must be weighed against increased voxelization time. The use of atomic operations directly impacts voxelization performance, particularly in situations where many threads are attempting to access the same voxel. We observed that the additional voxel culling provided by a rigorous computational intersection test helped significantly in reducing the number of write conflicts for the atomics to resolve. It should be noted that when outputting attribute buffers, that on some architectures, correct averaging of attribute information (colors, normals, etc.) may require emulation of (as of yet) unsupported atomic operations [CG12].

4.1.5 Attribute Interpolation

Attribute interpolation must be handled manually in the triangle-parallel approach. But as a benefit of its usage of the graphics pipeline, the fragment-parallel approach can exploit the fixed-function interpolation hardware provided by the rasterizer. Since the fragment-parallel voxelization method relies on triangle dilation to ensure a conservative voxelization, care must be taken to correctly interpolate triangle attributes across the dilated triangle. To accomplish this, we calculate the barycentric coordinates of the dilated triangle vertices $\mathbf{v}'_i$ with respect to the undilated triangle vertices $\mathbf{v}_i$ using signed area functions.

$$\lambda_i\left(\mathbf{v}'_i\right) = \frac{\text{area}\left(\mathbf{v}'_i, \mathbf{v}_{i+1}, \mathbf{v}_{i+2}\right)}{\text{area}\left(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2\right)} \tag{4.3}$$

By applying the barycentric coordinates computed at the dilated triangle vertices $\mathbf{v}'_i$ to the vertex attributes, i.e. vertex colors, normals, or texture coordinates $\mathbf{t}_i$, we can calculate corresponding dilated attributes $\mathbf{t}'_i$ as follows

$$\mathbf{t}'_i = \lambda_0 (\mathbf{v}'_i) \mathbf{t}_0 + \lambda_1 (\mathbf{v}'_i) \mathbf{t}_1 + \lambda_2 (\mathbf{v}'_i) \mathbf{t}_2 \tag{4.4}$$

By passing dilated attributes in from the geometry shader to the vertex shader in this manner, we ensure that attributes interpolate across the undilated region of the dilated triangle in the same manner as they would on the undilated triangle. This holds regardless of the dilation factor l applied.

4.2 Voxelization Performance

We tested our hybrid voxelization approach against several different models at various voxel resolutions, and compared the results to purely triangle-parallel and purely fragment-parallel implementations, as well as the data available from [SS10], [Pan11], and [CG12]. We included the XYZ RGB Asian Dragon as an example of a pathological worst case-scenario for the fragment-parallel approach, and we included a single scene-spanning triangle as a pathological worst case for the triangle-parallel approach. All results were generated on an Intel Core i7 950 @ 3.07GHz with an NVIDIA GeForce GTX 480. Table 4.1 shows the performance comparison of the different techniques, and additionally the percentage of time spent in the first and second pass of the hybrid voxelization approach, see figure 4.18. Both Dragons, the Bunny, and the Hairball represent less than ideal conditions for our

Model	Grid size	6-separating (thin) binary voxelization						
		Triangle-parallel	Fragment-parallel	Hybrid @voxels^2	Pass 1/Pass 2	Schwarz & Seidel	VoxelPipe	Crassin & Greene (680)
Large Triangle (1 tri)	128^3	10.62	0.03	0.04 @na	36.1%/63.9%			
	256^3	42.4	0.06	0.07 @na	22.1%/77.9%			
	512^3	169.7	0.22	0.19 @na	12.0%/88.0%			
XYZ RGB Asian Dragon (7,219,045 tris)	128^3	6.37	165.2	8.51 @2.0	99.9%/0.1%	11.36	21.2	
	256^3	7.70	165.0	8.57 @1.7	99.7%/0.3%	14.73		
	512^3	9.80	164.6	10.3 @1.4	99.8%/0.2%	16.67	22.0	
Crytek Sponza Atrium (262,267 tris)	128^3	13.4	10.65	1.11 @2.8	87.7%/12.3%			
	256^3	53.2	11.13	1.80 @3.9	71.6%/28.3%			
	512^3	208.7	11.87	3.68 @3.1	52.8%/47.2%			
Conference (331,179 tris)	128^3	9.23	11.47	1.41 @0.5	68.5%/31.5%	3.9	3.3	
	256^3	36.04	11.62	1.82 @1.7	69.2%/30.8%			
	512^3	141.2	11.94	3.01 @0.9	52.2%/47.8%	59.3	4.3	
Stanford Bunny (69,666 tris)	128^3	0.28	1.58	0.19 @1.8	88.1%/11.9%	0.60		
	256^3	0.82	1.55	0.34 @4.5	91.6%/8.4%	0.89		
	512^3	3.12	1.82	1.08 @12.7	93.0%/7.0%	2.35		
Stanford Dragon (100,000 tris)	128^3	0.25	2.13	0.26 @13.3	97.8%/2.2%	3.44	4.8	1.19
	256^3	0.51	2.09	0.52 @5.9	93.4%/6.6%	3.96		
	512^3	1.61	2.25	1.25 @13.7	88.6%/11.4%	4.44	5.0	1.38
Hairball (2,880,000 tris)	128^3	7.09	74.8	7.37 @2.3	99.89%/0.11%	22.8	12.8	
	256^3	13.73	67.1	14.0 @2.4	99.94%/0.06%			
	512^3	33.47	68.4	33.9 @8.0	99.97%/0.03%	95.0	18.3	

Table 4.1: Running time (in ms) for different voxelization approaches, blue indicates the fastest voxelization method. Voxelizations are binary and performed into a single component dense 3D texture. The Large Triangle cutoff is listed as "na" since there are no suitable triangles to be reassigned.

approach as they do not have a large distribution of triangle sizes, yet are able to obtain better performance than the competing techniques in all but one instance. In several cases, the purely triangle-parallel approach beat the hybrid approach, which is understandable considering these scenes are ideally suited to the triangle-parallel approach. It should be noted that in all such cases besides the pathological worst case (the Asian Dragon), the hybrid approach was within 3% of the triangle-parallel approach, indicating the low overhead of our multi-pass approach. Despite its simple classification scheme, our approach provides a performance improvement for binary voxelization over its competitors, including [CG12] which used superior hardware (GTX 680). It should be noted that the cutoff values are likely to be

highly architecture dependent, we would expect them to change when executed on Nvidia's Kepler or AMD's Southern Islands architecture.

4.3 Discussion of Voxelization

We implemented a wide variety of voxelization and conservative rasterization techniques in our experiments. Our implementations targeted the capabilities described in the OpenGL 4.2 specification. Our approach relied on the ability to perform texture writes to arbitrary locations enabled by the image API. Our classification approach relied on indirect buffers to enable the asynchronous execution of the voxelization stage. A benefit of our OpenGL implementation is that it avoids the performance penalty of context switching and implicit synchronization points present in a CUDA or OpenCL implementation. With the introduction of OpenGL 4.3, the triangle-parallel approach could easily be implemented in a compute shader, but at present our experience has been that compute shaders incur an unknown overhead and are significantly less efficient than existing techniques.

Another application of our initial classification scheme, see figure 4.14, could be to "pre-classify" scenes. Then by maintaining two index-buffers, hybrid-voxelization could be employed absent the cost of classification. This would be most sensible when applying a non-voxel dependent triangle classifier, in scenarios where the orientation of the voxels may change relative to the scene geometry.

We found that several of our results agreed with [SEA08, HHW09], that geometry amplification of the first Hasselgren technique led to performance degradations.

We also found that atomic operations more greatly impacted the triangle-parallel approach, likely due to the fact that each triangle-parallel thread is responsible for more writes than each fragment-parallel thread.

This chapter has shown how a GPU-accelerated computational surface voxelization can be achieved without resorting to CUDA or OpenCL. Our hybrid approach to voxelization leverages the strengths of the graphics pipeline to improve parallelism where it is most needed without sacrificing the quality of the voxelization. It exhibits superior performance to existing techniques, especially on scenes with non-uniform triangle distributions.

Chapter 5: Voxel Storage, Sampling, & Mipmapping

Recasting the scene into a voxel format has many advantages, but also, some critical disadvantages, mainly high memory requirements. Dense 3D textures offer many advantages such as the availability of hardware for interpolation, but unfortunately allocate data for unused portions of the scene. Hence it is prudent to explore techniques for efficient sparse voxel storage. Another consideration is the format of the data being stored per voxel. A simple dense 512^3 texture storing a RGBA color value per voxel takes up over 500 megabytes when stored in a low precision RGBA8 texture, and over 2 gigabytes in a medium precision RGBA32F texture, already exhausting the 1.5 gigabytes of memory available on a typical GTX 480. These values become 3GB and 12GB respectively for an anisotropic voxel storage scheme which stores 6 directional color values per voxel. If it becomes necessary to store additional voxel attribute data (such as diffuse and emissive color values, and normals) then these already prohibitive storage requirements are multiplied several fold over again. Also, since texture filtering hardware on the GPU is fixed to function with specific texture formats, we are unable to define generic voxel attributes in a flexible manner as we may like. Instead we must make concessions to available functionality and hardware resources in order to "pack" our data into available texture formats.

5.1 Voxel Storage

Utilizing voxel storage that can be stored in hardware supported texture formats is critical for performance. Not only does this enable *trilinear* interpolation within a texture level, but assuming we use a consistent format across all texture hierarchy levels, we can enable *quadrilinear* interpolation. Adapting our storage mechanisms to enable the available texture filtering hardware allows us to produce smooth, visually pleasing results without adding computational overhead.

5.1.1 Isotropic Voxel Storage

Isotropic voxel storage is the simplest storage method and also has the lowest memory requirements. Essentially all we are storing is an approximation of the diffuse lighting at the voxel. Thus we are able to store it inside of a single `RGBA8` texture. We are able use this texture format with the `ARB_shader_image_load_store` functionality, even though it is not explicitly supported, by casting it to a supported `R32UI` format and manually converting and packing the appropriate bits. As our voxelization approach has the potential to create many threads attempting to write to the same location, we must find a way to resolve these conflicts. The simplest approach is to simply take the maximum value at the voxel using the `imageAtomicMax` function, this will ensure a consistent voxelization, albeit at the cost of sacrificing accuracy. Ideally we would want an `imageAtomicAverage` function, but one does not exist. However, in this case we may emulate the same functionality as described in [CG12] using the `imageAtomicCompSwap` function,

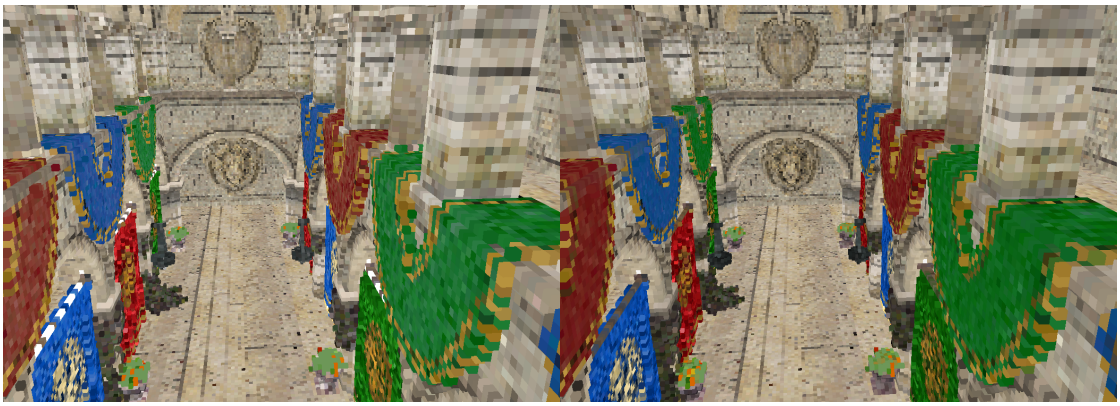


Figure 5.1: Images of the isotropic voxelization output using the builtin `imageAtomicMax` functionality on the left vs the emulated `imageAtomicAverage` on the right. Note that the `imageAtomicMax` version has a tendency to saturate the voxel color, but overall the result is quite acceptable. Both voxelization are performed using the fragment-parallel voxelization approach at a voxel resolution of 512^3.

see Figure B.1 located in Appendix B. Effectively what this code does is create a "spinlock," and updates the running average inside the loop until the lock stops "spinning."

We can observe the quality difference between emulated atomic average and the builtin atomic max functionality in Figure 5.1, and while there is a discernible quality difference between the two, it is surprisingly, not that noticeable. Considering that the initial voxelization is simply the input to a hierarchy construction from which filtering indirect illumination results are sampled, the quality difference becomes even more difficult to discern. Another consideration for the atomic max implementation is that the texture components should be swizzled such that the alpha component comes first, that is RGBA becomes ARGB. This can be accomplished either in the shaders directly, or with the `EXT_texture_swizzle` ex-

tension. The primary concern with performing the swizzle during voxel storage is to ensure that the appropriate unswizzle is performed during voxel access. In practice, however, while it makes sense to swizzle the components such that the alpha component dominates the atomic max comparison, it makes a virtually indistinguishable difference in quality in most cases. Ultimately, the decision for which method is selected to ensure a consistent voxelization output regresses to a familiar tradeoff between quality vs. computation time. The performance penalty incurred by the image atomic average emulation can be observed in Figure 5.4 in Section 5.1.4.

5.1.2 Anisotropic Voxel Storage

As described in [CNS*11], anisotropic voxel storage stores a color value for each voxel cube face. With this information, we can approximate a directional representation for the radiance emitted from the voxel. But this does increase our storage requirement by at least a factor of six. This can be accomplished in several ways. The most obvious is to have 6 `RGBA8` textures, one for each set of voxel faces. However, this comes with the caveat that it will take multiple passes to mipmap (at least 2) as each reduction requires that both the previous and current level be bound to an image unit, for a total of 12, exceeding the fixed limit of 8. Alternately, we could over-allocate a single texture, that is allocate a single texture that is 6 times larger than required and implement a custom indexing scheme such that each anisotropic voxel face was stored in its own section of the larger texture.

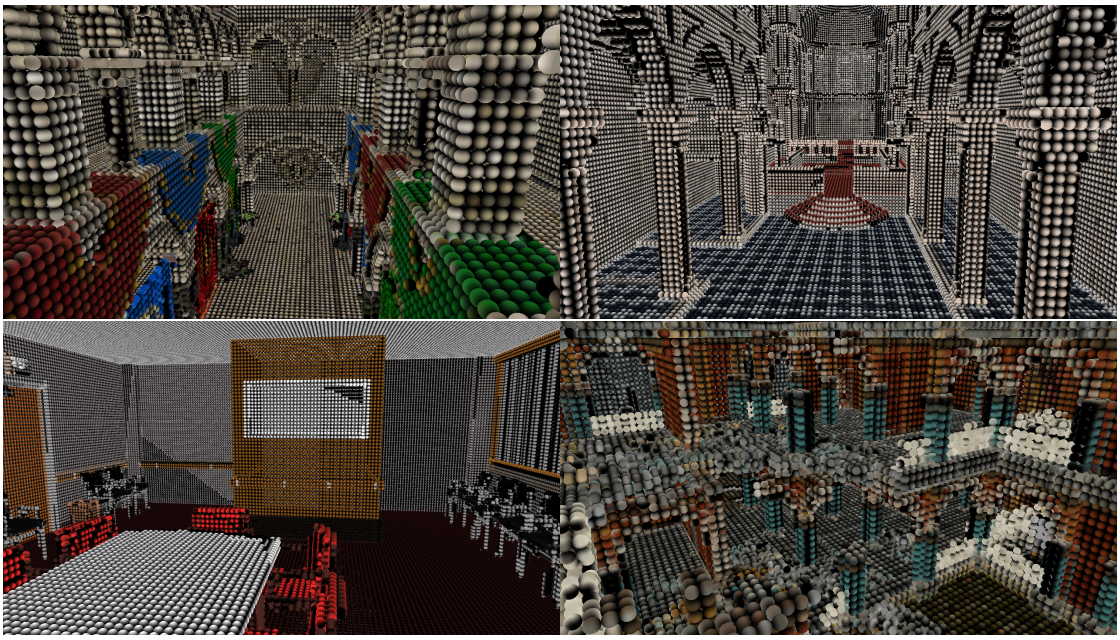


Figure 5.2: Anisotropic voxels initialized based on dominant normal direction and visualized as spheres using the method described in Section 5.2.2.

Instead of these approaches we adopt the new `ARB_bindless_textures` available on the latest generations of GPU hardware which allows us to exceed the previous image unit limitation.

Since we now have a directional storage format, we can employ a more sophisticated storage scheme than the "store color in voxel" isotropic approach. We can use the surface normal to select the dominant normal direction and store the computed color value in the appropriate face as seen in Figure 5.2. We can further elaborate on this approach and weight the color contribution by the normal components and store the results in the faces pointed at by the normal. We have experimented with both of these approaches. They both have the caveat that there is the potential for an active voxel to have uninitialized faces, which raises the question of how

to appropriately deal with these faces during mipmapping, (discussed in Section 5.3.2). For the sake of an objective comparison of the mipmapping process, we elected to treat the base level of the anisotropic textures as isotropic and duplicate the computed color value across the faces, allowing the anisotropic directionality of the voxels to be captured in the hierarchy construction process. Similar to the isotropic voxels, since the anisotropic voxels rely on the `RGBA8` texture format, we again have the option of using the builtin atomic max function or the emulated atomic average as shown in Figure B.1 in Appendix B.

5.1.3 Spherical Harmonic storage

As in [KD10], we limit our spherical harmonics to 2 bands. There are several reasons for this, the primary one being that the number of coefficients of spherical harmonics increases quadratically with the number of bands. With two bands we at least retain the possibility of fitting the coefficients of 2 band spherical harmonics in a 4 component texture. However, it quickly gets more complex than that. Unfortunately, an `RGBA8` texture does not provide sufficient accuracy to store spherical harmonic coefficients, and we are forced to resort to the use of a format that stores 32 bit floating point values. `RGBA32F` is the obvious candidate, unfortunately, like `RGBA8,` it is not natively supported by image atomic operations, and even worse it can neither be cast to another format, nor can it be "viewed" as another format (using `GL_ARB_texture_view`) supporting atomic operations. We can, however, use `R32F` with atomic operations thanks to the `NV_shader_atomic_float` exten-

sions, though with 4 coefficients required for each color channel (R,G, and B), this implies using at least 12 volumetric textures if we don't want to resort to clever indexing schemes. Ultimately, we implemented this twice, first relying on a fewer number of textures and a modified indexing scheme, but this was found to lead to extreme performance degradations due to incoherence in texture accesses as maintaining an ordering that allowed for hardware filtering required placing adjacent components (e.g. R and B) at a stride equal to the texture dimensions. This scheme was successful for prototyping but ultimately, bindless textures were adopted, via the extension `ARB_bindless_textures`, allowing us to circumvent the image unit limitations and increasing performance by an order of magnitude.

Since we are now using a medium precision floating point texture format (high precision being 'double') on Nvidia hardware, we exploit the availability of atomic floating point addition operations to sum spherical harmonic coefficients. Unfortunately, all attempts to create a clever scheme to either average or perform more complex spherical harmonic product operations over these values in the same shader invocation failed. Since we could no longer use the alpha bits of our texture as a "spinlock," we relied on creating a dedicated mutex texture, which allowed us to create a unique lock at each texture location using the `imageAtomicCompSwap` function, which should theoretically have allowed us to safely update the running average without interference from other threads. Tragically, it would seem that deficiencies in either the shader compiler or hardware rendered such approaches moot, and without access to a functioning debugger we were unable to accurately diagnose the problem. It turns out, however, that we are not alone in experiencing

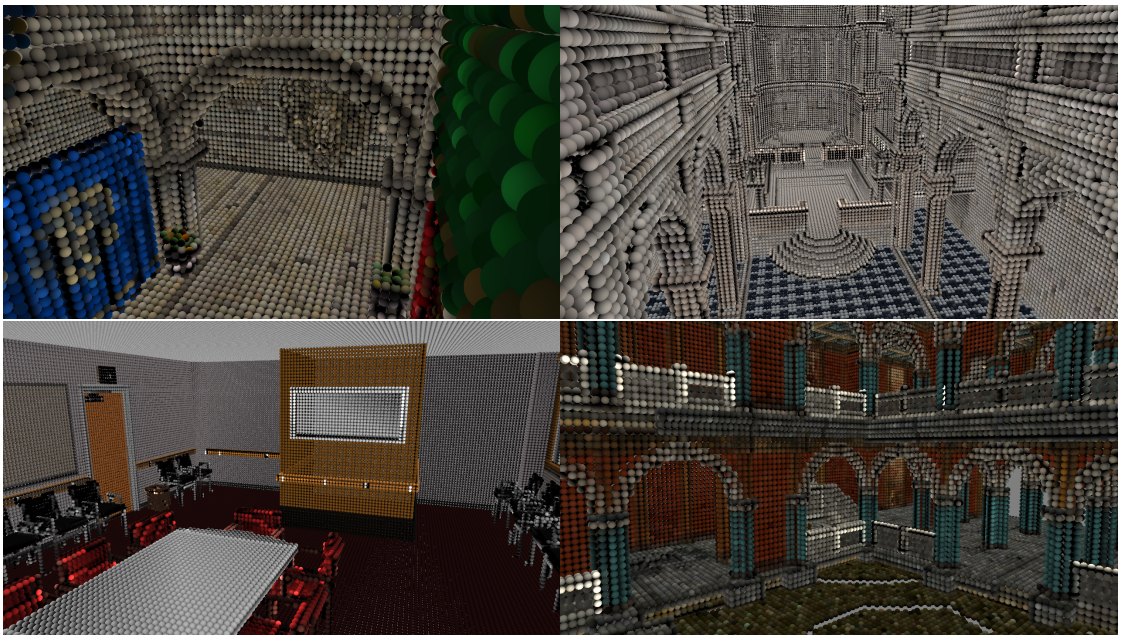


Figure 5.3: Visualization of the spherical harmonic functions stored at each voxel location. Each function is represented by a raytraced sphere and the color values are sampled from the spherical harmonic function at each location based on the normal. The Crytek-Sponza scene is shown in the upper left, the Sibenik Cathedral is shown in the upper right, while the Conference Room is shown in the lower left and the Ruins scene lower right. Note, spheres are unlit and unshaded.

difficulties trying to exploit such cutting edge features, for more information see *A Digression on Divergence* [Fol13]. Because of these issues, we were forced to take the extremely pedestrian route of simply normalizing the spherical harmonic coefficients in a second pass post-voxelization.

Much as with the anisotropic voxels, we can use the surface normal to encode directional information into spherical harmonics, the results of which can be seen in Figure 5.3. However, since these voxels are initialized from planar surfaces with only one normal direction, the result is often a voxel that is colored on one side

and simply black on the other side. This leads to problems during mipmapping as this lack of information is interpreted as lack of illumination and the dark portion of the spherical harmonics are projected onto their parent resulting in a significant darkening of the scene. So, much as with the anisotropic voxels, we effectively consider the base level of the spherical harmonic voxels to be isotropic as well and allow the hierarchy construction process (mipmapping) to capture the directionality information.

Considering the base level to be isotropic, and that we must already perform a second pass to normalize our values, we can optimize our spherical harmonic storage format. Instead of initializing the spherical harmonics directly, we perform the same isotropic voxelization as in Section 5.1.1, and then transfer the results to a spherical harmonic storage format. Initially, we replaced our 12 `R32F` textures with 3 `RGBA32F` textures, surprisingly however, this led to a moderate performance decrease, and thus we scrapped this approach. Ko et al. [KKZ08] describe quantization techniques for storing spherical harmonic coefficients efficiently, however, they targeted precomputed rendering techniques, and thus had the luxury of preprocessing their data. In our application we found the minimum, $SH_{\min}$, and maximum, $SH_{\max}$, spherical harmonic coefficients for several scenes, padded the results, and determined a conservative SH_{range} value. By dividing by SH_{range} on storage we reduce the potential range of value from $[-\texttt{FLT\_MAX}, \texttt{FLT\_MAX}]$ to $[SH_{\min}, SH_{\max}]$. This reduction in range allows us to store our spherical harmonic coefficients in a `RGBA8` texture without significant loss of accuracy. To recover our original values we simply multiply by SH_{range}. We have effectively reduced our

spherical harmonic storage costs by a factor of four, which makes it cheaper in total memory cost than anisotropic voxels.

5.1.4 Voxelization Performance & Costs

Since we have now described three radically different voxel storage formats, two of which have two mechanisms for computing voxels (`imageAtomicMax` and `imageAtomicAvg`), and a third which even has different storage targets; we must characterize the performance costs of these different approaches. These results can be seen in Figure 5.4, note that these voxelizations are not using the fully optimized voxelization technique described in Chapter 4 as this adds significant complexity, which is difficult to characterize, especially in the presence of so many atomic operations.

5.2 Voxel Sampling

Each of our three voxel storage methods requires its own sampling method. The isotropic voxels by definition have no dependence on sampling direction, while the anisotropic and spherical harmonic methods must both take into account directional sampling.

5.2.1 Isotropic Sampling

As there is no directionality to isotropic storage, the sampling scheme is trivial. The only thing worth noting here is that if the emulated atomic averaging scheme

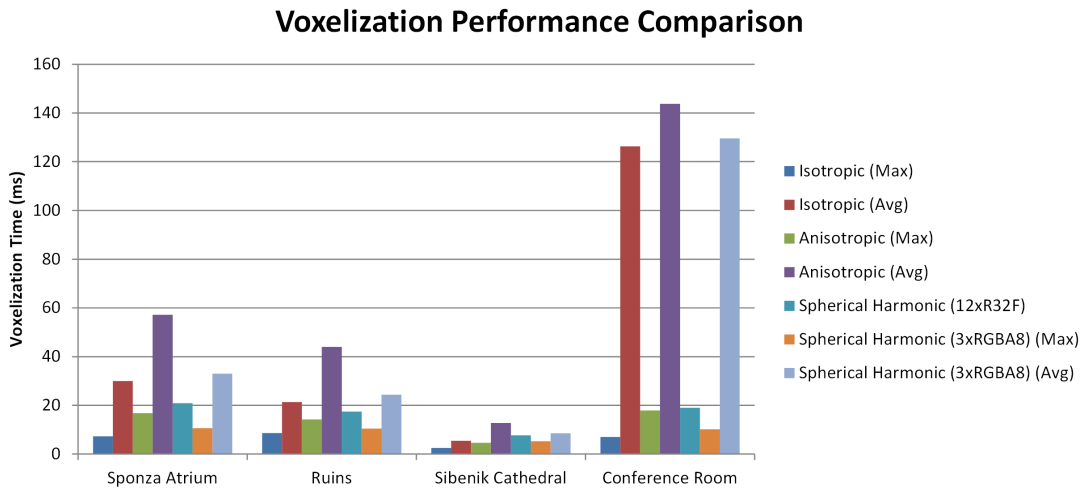


Figure 5.4: Performance of a fragment parallel voxelization for several computer graphics scenes. In general, the more complex storage formats have a higher voxelization cost. Also, the emulated image atomic average functionality can be severely detrimental to performance depending on degree of thread contention during voxelization. For example, the Conference Room scene relies on pure triangle density (as opposed to normal maps) to add additional detail to the scene, which causes severe busy-waiting in the atomic average's spin-lock. Note, that the `RGBA8` spherical harmonic voxelization outperforms isotropic voxelization, and is competitive with isotropic voxelization.

was used to initialize the isotropic voxels, the alpha component of the texture will not contain the expected value. Instead it will contain the count of the number of shader threads that attempted to write to the voxel location. In effect, the alpha component has become an indication of the amount of thread contention for that particular voxel during the voxel phase. In practice, we can generally assume base level voxels are opaque, ignore the alpha term, and sample the RGB components.

5.2.2 Anisotropic Sampling

It is not explicitly stated what sort of anisotropic sampling scheme is used by Crassin et al. in [CNS*11]. The technique we have selected is described by Mitchel et al. in [MMG06] and used for sampling "ambient cubes." It is essentially a simple weighted blending of the six directional face colors as a function of the world space direction, an example in 2D can be seen in figure 5.5. It is worth noting that spherical harmonics are directly mentioned in [MMG06] as a potential method for improving fidelity. GLSL code for the anisotropic sampling technique is listed in Figure B.2 in Appendix B.

5.2.3 Spherical Harmonic Sampling

Directional sampling for spherical harmonics is well defined. We shall describe it briefly here, for more detailed reference see Section 2.2, or *Stupid SH Tricks* [Slo08] and *Spherical Harmonic Lighting: The Gritty Details* [Gre03]. Since we are using

Figure 5.5: A 2D anisotropic voxel with per-face color values. Image courtesy of [Mit12].

only a 2 band spherical harmonic representation, we only need functions from the first two rows of Table 2.1 from Section 2.2. Essentially we take the dot product of the first four spherical harmonic functions $y_{0\ldots3}$ (i.e. the first two bands) with the components of the normalized negative direction vector substituted in with the functions recovered from the voxel storage, that is:

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} \left\langle \left(1, \frac{\sqrt{3}\vec{\mathbf{d}}_y}{2\sqrt{\pi}}, -\frac{\sqrt{3}\vec{\mathbf{d}}_z}{2\sqrt{\pi}}, \frac{\sqrt{3}\vec{\mathbf{d}}_x}{2\sqrt{\pi}}\right) \cdot (V_{r_0}, V_{r_1}, V_{r_2}, V_{r_3}) \right\rangle \\ \left\langle \left(1, \frac{\sqrt{3}\vec{\mathbf{d}}_y}{2\sqrt{\pi}}, -\frac{\sqrt{3}\vec{\mathbf{d}}_z}{2\sqrt{\pi}}, \frac{\sqrt{3}\vec{\mathbf{d}}_x}{2\sqrt{\pi}}\right) \cdot (V_{g_0}, V_{g_1}, V_{g_2}, V_{g_3}) \right\rangle \\ \left\langle \left(1, \frac{\sqrt{3}\vec{\mathbf{d}}_y}{2\sqrt{\pi}}, -\frac{\sqrt{3}\vec{\mathbf{d}}_z}{2\sqrt{\pi}}, \frac{\sqrt{3}\vec{\mathbf{d}}_x}{2\sqrt{\pi}}\right) \cdot (V_{b_0}, V_{b_1}, V_{b_2}, V_{b_3}) \right\rangle \end{pmatrix} \tag{5.1}$$

where $\vec{\mathbf{d}}$ is the normalized direction vector and $V_{r_{0\ldots3}}$, $V_{g_{0\ldots3}}$, and $V_{b_{0\ldots3}}$ are the spherical harmonic coefficients for each color component, (R, G, B), stored in the voxel location, V. Note, the RGB components must also be clamped from 0 to 1. Additionally, GLSL code for spherical harmonic sampling is listed in Figure B.3 in Appendix B.

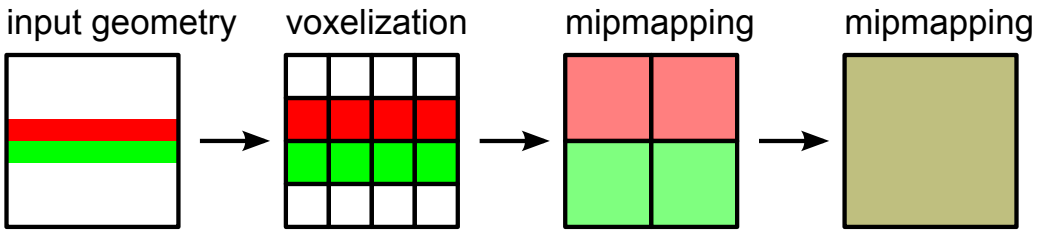


Figure 5.6: Illustration of isotropic voxel mipmapping, note the emergence of the red-green wall problem.

5.3 Voxel Mipmapping

After selecting what to store in the voxel and how to store it, the challenge then falls to the construction of a reasonable filtered hierarchy in which each element provides an approximation of the elements below it. There is necessarily a loss of accuracy in this process, but the challenge remains to preserve essence of the data contained therein.

5.3.1 Isotropic Mipmapping

Isotropic mipmapping is relatively straightforward, in fact there is not much more to be done other than to average voxel color values as weighted by their opacity. While simple, this approach does lead to several undesirable artifacts such as the red-green wall problem illustrated in Figure 5.6, which can result in two adjacent walls being represented as yellow voxels higher up the mipmap hierarchy. This problem applies to opacity values as well. If voxel opacity is averaged, it can turn

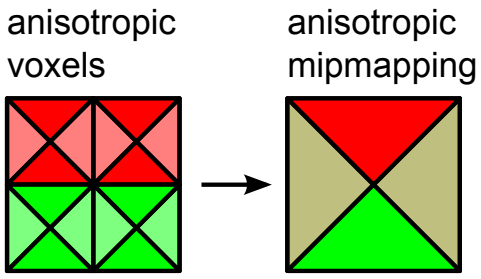


Figure 5.7: Illustration of the results of directionally dependent anisotropic mipmapping.

opaque walls transparent and lead to light leaking.

5.3.2 Anisotropic Mipmapping

Anisotropic mipmapping is slightly more involved. As we are not employing a brick-map storage scheme, there is no need for the transfer of illumination to adjacent bricks. The process is described in [CNS*11] and mentioned here in brief for the sake of completeness. As discussed in section 5.1.2, anisotropic voxels store 6 channels of directional values, one per major direction. The voxel contents are then filtered anisotropically along the major axial directions, as in figure 5.7, for more details see [Yeu13].

5.3.3 Spherical Harmonic Mipmapping

While it can be tempting to try and apply an approach such as a spherical harmonic product projection as described in section 2.2.2, this has the undesirable property that the result of a spherical harmonic product is non-commutative, so we will arrive at different results based on the ordering of our operations. Instead we take an approach more akin to the spherical harmonic propagation method outlined in [KD10]. Although instead of projecting the contribution of a spherical harmonic function onto the faces of the neighboring voxels, we are instead projecting the contribution of the (up to 8) spherical harmonic functions onto the faces of the parent voxel in which they are contained. This involves (assuming all interior voxels are active) 8 projections onto 6 faces for 3 sets of spherical harmonic coefficients. A 2D illustration of this concept can be seen in figure 5.8.

For all eight interior voxels, we must calculate the solid angle of each of the 6 faces of the larger voxel as seen from the unit sphere (i.e. the spherical harmonic function inside the smaller voxel). This solid angle is referred to as the subtended solid angle of the shape. We calculate this angle analytically using integration. First we set up the geometry by assuming that the center of the voxel we are projecting from lies at $(0, 0, 0)$ inside a voxel with an edge length of 2, meaning that the parent voxel has an edge length of 4. Figure 5.9 illustrates this construction for projections onto the six faces of the parent voxel for the top-left-front child voxel which defines the origin.

First we construct the integral for the back face projection. We integrate the

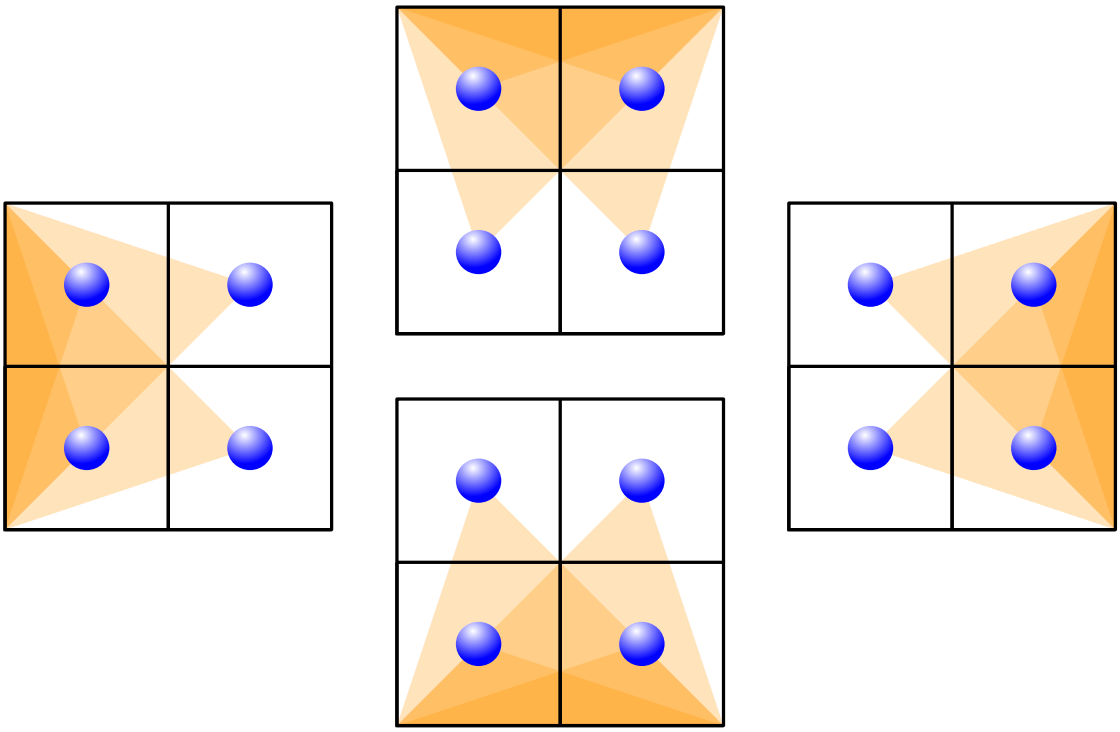


Figure 5.8: 2D projections onto each of the faces of a higher level (parent) voxel by the child voxels.

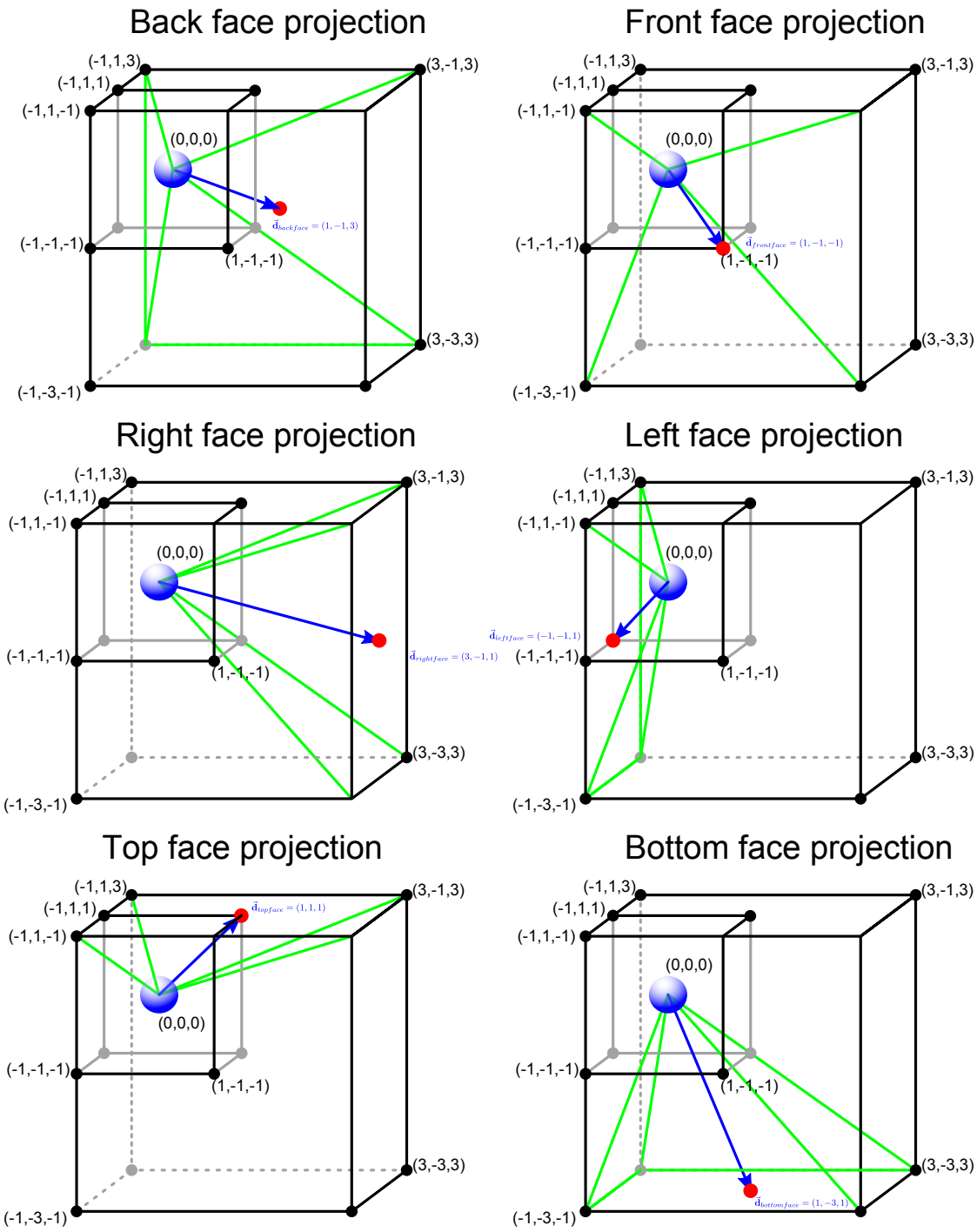


Figure 5.9: All 6 projections for the top-left-front voxel. Voxel centers are blue spheres, SH projections are in green, while $\vec{\mathbf{d}}_{face}$ vector are shown in blue traveling from the voxel center to the face center (in red). Note, the $\vec{\mathbf{d}}_{face}$ vectors in this diagram are not normalized.

radiance of its surface points $\mathbf{p} = (x, y, z)$, in this case z is a constant of value 3, while $x \in [-1 \ldots 3]$ and $y \in [-3 \ldots 1]$, so we construct our integral as follows:

$$\begin{aligned}
\Omega_{backface} &= \int_{-1}^{3}\int_{-3}^{1} L(\mathbf{p})\, dp \\
&= \int_{-1}^{3}\int_{-3}^{1} \frac{\langle \vec{\mathbf{n}}_{\text{backface}} \cdot \mathbf{p} \rangle}{|p|} \frac{1}{|p|^2} dx\, dy \\
&= \int_{-1}^{3}\int_{-3}^{1} \frac{\langle (0,0,1) \cdot \mathbf{p} \rangle}{|p|} \frac{1}{|p|^2} dx\, dy \\
&= \int_{-1}^{3}\int_{-3}^{1} \frac{z}{(x^2+y^2+z^2)^{\frac{3}{2}}} dx\, dy \\
&= \int_{-1}^{3}\int_{-3}^{1} \frac{3}{(x^2+y^2+9)^{\frac{3}{2}}} dx\, dy \\
&= 1.074793009
\end{aligned}$$

We follow a similar construction for the front face projection, except in this case $z = -1$ while x and y span the same values:

$$\begin{aligned}
\Omega_{frontface} &= \int_{-1}^{3}\int_{-3}^{1} L\left(\mathbf{p}\right) dp \\
&= \int_{-1}^{3}\int_{-3}^{1} \frac{\langle \vec{\mathbf{n}}_{\text{frontface}} \cdot \mathbf{p} \rangle}{|p|} \frac{1}{|p|^2} dx\, dy \\
&= \int_{-1}^{3}\int_{-3}^{1} \frac{\langle (0,0,-1) \cdot \mathbf{p} \rangle}{|p|} \frac{1}{|p|^2} dx\, dy \\
&= \int_{-1}^{3}\int_{-3}^{1} \frac{-z}{(x^2+y^2+z^2)^{\frac{3}{2}}} dx\, dy \\
&= \int_{-1}^{3}\int_{-3}^{1} \frac{1}{(x^2+y^2+1)^{\frac{3}{2}}} dx\, dy \\
&= 3.113997196
\end{aligned} \tag{5.2}$$

Now we can repeat this process 4 more times for the other four faces if we wish, and then another 42 times for the seven other voxels and their projections, or, we can observe that in all configurations there are actually only ever two different solid angles, which we shall term Ω_{narrow} and Ω_{wide}, which correspond to projections on far and near faces respectively, this holds for all voxels. To recap:

$$\begin{aligned}
\Omega_{narrow} &= \Omega_{farface} = 3.113997196 \\
\Omega_{wide} &= \Omega_{nearface} = 1.074793009
\end{aligned} \tag{5.3}$$

As a sanity check we can sum up the subtended solid angles of all the faces:

$$\begin{aligned}\forall_{faces}\Omega_{\text{face}} &= 3\Omega_{narrow} + 3\Omega_{wide} \\ &= 3 * 3.113997196 + 3 * 1.074793009 \\ &= 12.566370615 \\ &= 4\pi \end{aligned} \tag{5.4}$$

which is the total solid angle of a sphere in steradians, so we can be confident that our calculations are correct. Now, armed with the subtended solid angles (all two of them) for the 48 possible projections, we can begin the task of creating a single spherical harmonics function from this information. The only additional information we need for this is the normalized direction vectors from the centers of the child spherical harmonic functions to the faces of the parent voxel, $\vec{\mathbf{d}}_{face}$, and the coefficient's of the projection of a cosine lobe, $C(\theta)$, pointing in the direction of the z-axis. For the calculation of the cosine lobe coefficients we refer our reader to *Useful Results in Spherical Harmonics* [Ins10]. From [Ins10], we can compute the coefficients of the projection of a rotated lobe C' with coefficients $c'_{0\ldots3}$ such that the the peak of the lobe points along the direction vector $\vec{\mathbf{d}}$ as:

$$(c'_0, c'_1, c'_2, c'_3) = \frac{\sqrt{\pi}}{2}, \sqrt{\frac{\pi}{3}}\vec{\mathbf{d}}_y, \sqrt{\frac{\pi}{3}}\vec{\mathbf{d}}_z, \sqrt{\frac{\pi}{3}}\vec{\mathbf{d}}_x \tag{5.5}$$

Considering each spherical harmonic function as a VPL, in order to find the VPL for the parent voxel, we must first determine the VPLs representing the

contribution of every active child voxel onto every face of the parent voxel. After determining every directional VPL from every active child voxel to each face of the parent voxel, we can then sum the contributions of all these VPLs to arrive at a single VPL representing the parent voxel.

More formally, let us consider the parent voxel P with child voxels V_i, where $i \in [0 \dots 7]$, each with three sets of spherical harmonic coefficients for each color band, $V_{r_{0\dots3}}$, $V_{g_{0\dots3}}$, and $V_{b_{0\dots3}}$, and similarly for the SH coefficients of P. We define the vector $\mathbf{V}_{i,r}$, as the vector of red SH coefficient for voxel V, and use corresponding notation $\mathbf{V}_{i,g}$ and $\mathbf{V}_{i,b}$ for green and blue. That is

$$\begin{aligned} \mathbf{V}_{i,r} &= (V_{i,r_0}, V_{i,r_1}, V_{i,r_2}, V_{i,r_3}) \\ \mathbf{V}_{i,g} &= (V_{i,g_0}, V_{i,g_1}, V_{i,g_2}, V_{i,g_3}) \\ \mathbf{V}_{i,b} &= (V_{i,b_0}, V_{i,b_1}, V_{i,b_2}, V_{i,b_3}) \end{aligned} \tag{5.6}$$

and for any given child voxel, V_i, let us consider the normalized direction from its center to the center of the face we are projecting onto to be $\vec{\mathbf{d}}_{face}$. Then we define the function SH to evaluate the spherical harmonic coefficients of a face as:

$$SH \quad \vec{\mathbf{d}}_{face} = \left(1, -\frac{\sqrt{3}\vec{\mathbf{d}}_{face,y}}{2\sqrt{\pi}}, \frac{\sqrt{3}\vec{\mathbf{d}}_{face,z}}{2\sqrt{\pi}}, -\frac{\sqrt{3}\vec{\mathbf{d}}_{face,x}}{2\sqrt{\pi}}\right) \tag{5.7}$$

and the function $SH_{\cos}$ to evaluate the spherical harmonic coefficients of the cosine

lobe in the normalized direction $\vec{\mathbf{d}}_{face}$ as:

$$SH_{\cos}\ \ \vec{\mathbf{d}}_{face} \ = \ \frac{\sqrt{\pi}}{2}, -\sqrt{\frac{\pi}{3}}\vec{\mathbf{d}}_{face,y}, \sqrt{\frac{\pi}{3}}\vec{\mathbf{d}}_{face,z}, -\sqrt{\frac{\pi}{3}}\vec{\mathbf{d}}_{face,x} \tag{5.8}$$

Finally, this allows us to express the summed spherical harmonic coefficients for the parent voxel, P, as:

$$\begin{pmatrix} P_{r_{0\ldots3}} \\ P_{g_{0\ldots3}} \\ P_{b_{0\ldots3}} \end{pmatrix} = \begin{pmatrix} \sum_{i=0}^{7} \forall_{faces} \sum \Omega_{face} \left\langle SH\ \ \vec{\mathbf{d}}_{face}\ \cdot \mathbf{V}_{i,r} \right\rangle SH_{\cos}\ \ \vec{\mathbf{d}}_{face} \\ \sum_{i=0}^{7} \forall_{faces} \sum \Omega_{face} \left\langle SH\ \ \vec{\mathbf{d}}_{face}\ \cdot \mathbf{V}_{i,g} \right\rangle SH_{\cos}\ \ \vec{\mathbf{d}}_{face} \\ \sum_{i=0}^{7} \forall_{faces} \sum \Omega_{face} \left\langle SH\ \ \vec{\mathbf{d}}_{face}\ \cdot \mathbf{V}_{i,b} \right\rangle SH_{\cos}\ \ \vec{\mathbf{d}}_{face} \end{pmatrix} \tag{5.9}$$

Once we have the summed spherical harmonic coefficients for the parent voxel P, we must normalize them, which is accomplished simply by a division by the number of active voxels contributing to the sum (i.e. average the results).

5.4 Sparse Mipmapping Optimizations

Instead of naively processing an n^3 number of voxels when constructing our mipmap hierarchy, we can output an *active-voxel-list* at each mipmap level. The active-voxel-list consists of only those voxel locations which must be processed in the *next* mipmap level. This is accomplished using what we shall term "mutex textures" and hardware supported high performance atomic counters provided by the `ARB_shader_atomic_counters` extension.

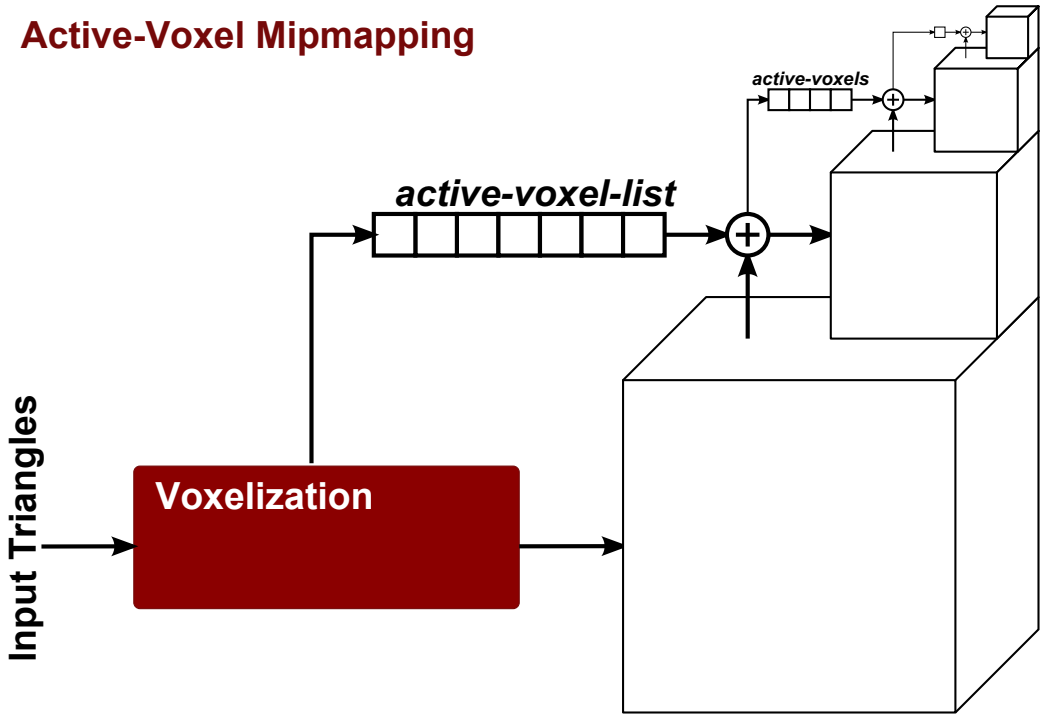


Figure 5.10: Implementation of an optimized voxel-mipmapping scheme which relies on the output of *active-voxel-lists* at each stage of the voxel-hierarchy.

The mutex texture is necessary to ensure that during the voxelization process the *active-voxel-list* is appended with the location of a voxel only once for each active voxel (in the next miplevel), as opposed to each time the voxel is accessed (as many threads will try to write to the same voxel location). The `imageAtomicCompSwap` operation is used on the mutex texture to uniquely "lock" a voxel location. The atomic counter is then incremented to provide a unique location to output to the "append buffer" (i.e. the *active-voxel-list*). The location of the voxel is then written to the *active-voxel-list*. What we are left with is a list of sparse *active-voxel* locations, i.e., only the voxels that actually need to be processed. By disabling the rasterizer and rendering a point list of only these active-voxel locations, we can sparsely process the relevant geometry stored inside our dense textures, see figure 5.3. While the addition of more atomic operations during voxelization does increase voxelization times, by employing this technique during hierarchy creation, we can drastically speed up the mipmapping process, resulting in an overall speedup, see Figure 5.11. In addition to outputting a sparse active-voxel-list for the next miplevel during voxelization, we additionally create a sparse voxel list for the current level. This is useful as it allows for sparse processing of the active voxels without processing the entire volume.

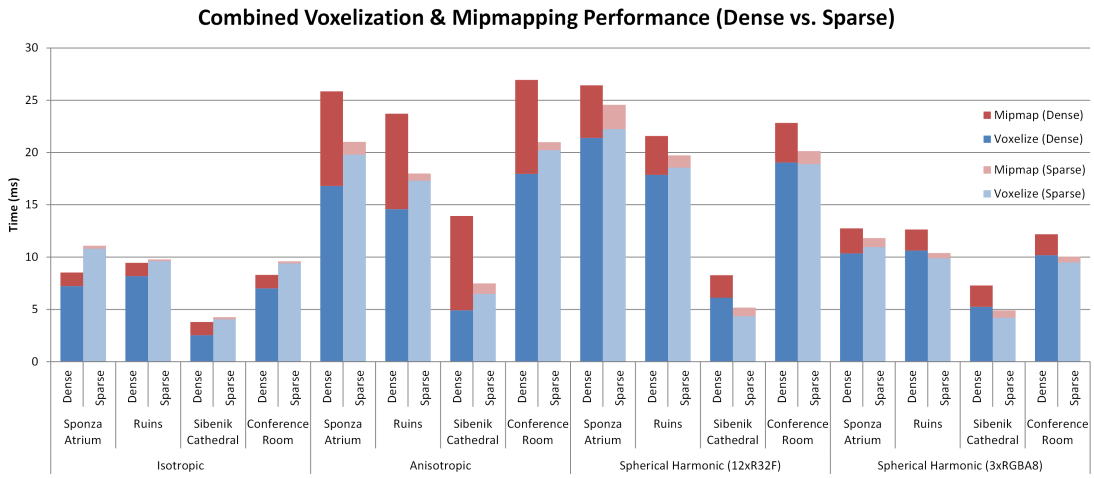


Figure 5.11: Combined performance of voxelization and mipmapping with active-voxel-lists disabled and enabled for several scenes and voxel storage formats. Note that for all voxel formats (besides isotropic) there is a net gain in performance for all scenes. Furthermore, for the spherical harmonic cases the active-voxel-list can be used in the post-voxelization step (normalization for 12x`R32F` and transfer for 3x`RGBA8`) resulting in an improvement in overall voxelization time as well.

Chapter 6: Voxel Based Illumination

Recently, voxel based methods have gained prominence among methods to compute global illumination solutions. This has been underscored by their success in commercially shipping game engines, notably the CryEngine in the case of Light Propagation Volumes [KD10], and tech demos in the case of the Unreal Engine and Voxel Cone Tracing [Mit12]. In this Chapter, we will review the underlying technique (Section 6.1), in addition to demonstrating its application to several illumination effects. Images depicting the incremental addition of voxel cone tracing based illumination effects can be seen in Figure A.3 and Figure A.4 in Appendix A.

6.1 Voxel Cone Tracing

Cone tracing provides an extremely high-performance alternative to ray tracing. As opposed to tracing many infinitesimally small rays, finding an approximate sampling over an area simply requires setting the appropriate cone aperture. The only caveat is that you must have an appropriate volumetric, hierarchical proxy of your scene to sample from. Fortunately, we have covered this in previous chapters (cf. Chapter 4).

Cone tracing is similar to volumetric ray-casting, save that we use the sample

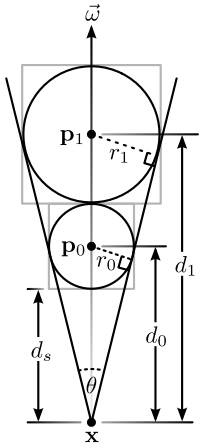


Figure 6.1: Geometric construction of samples, $\mathbf{p}_0$, and $\mathbf{p}_1$ along a cone of aperture θ degrees in direction $\vec{\omega}$. Note that the previous distance and radius is used to find the next sample location.

distance, d, from the originating point, $\mathbf{x}$, and the cone aperture, θ, to determine the radius, r, of the sample point, $\mathbf{p}$, which is, in turn, used to calculate the correct Level-of-Detail (LOD) to sample from (i.e. the correct mipmap level). The basic geometry of this construction is shown in Figure 6.1. We can easily compute the radius $r = d \cdot \sin\frac{\theta}{2}$; the radius of a sample point represents one half of the voxel extent, thus to compute the appropriate LOD level to sample from we take the base 2 logarithm of the sample's diameter or $\text{lod} = \log_2(2 \cdot r)$.

We are able to compute the appropriate radius for the second sample, $\mathbf{p}_1$, based on the aperture and value of the first sample, by exploiting the common ratios of the similar right triangles, $\frac{r_0}{d_0} = \frac{r_1}{d_1}$. That is $r_1 = (d_0 + r_0)\ \frac{\sin\frac{\theta}{2}}{1-\sin\frac{\theta}{2}}$, the second part of the expression remains constant for the evaluation of the cone and can be precomputed. We refer to this as the *cone ratio*, and through repeated application

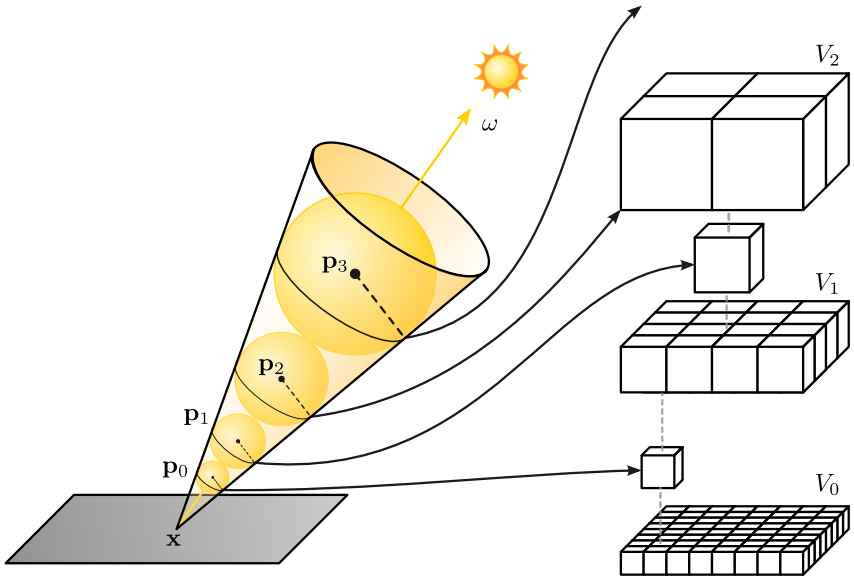


Figure 6.2: Illustration of the voxel cone tracing technique and the correspondence between the sampling radius of the cone and the quadrilinearly interpolated voxel value.

we can determine the points and radii along the cone, i.e. $r_{i+1} = (d_i + r_i) \quad \frac{\sin\frac{\theta}{2}}{1-\sin\frac{\theta}{2}}$. Thus in this manner we take successive samples $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n$ along the cone sampling from the voxels volume, V, along the cone. For the shadows we are only concerned with the alpha component $V_{\text{lod},\alpha}[\mathbf{p}_i]$, while for diffuse interreflection we would be interested in the color value $V_{\text{lod,xyz}}[\mathbf{p}_i]$. The sample is then quadrilinearly interpolated (trilinearly interpolated along spatial dimensions, then linearly interpolated between mipmap levels) by the texture filtering hardware. This sampling approach is illustrated in Figure 6.2. Additionally, GLSL code for a voxel cone tracing routine is listed in Figure B.4 in Appendix B.

6.1.1 Avoiding self intersection

Due to the volumetric nature of the voxelization, care must be taken to ensure that initial sampling points are not sampled within the voxels representing the geometry the ray is exiting from, see Figure 6.3. While we can simply stretch out the starting distance, d_s, shown in Figure 6.1, this doesn't necessarily guarantee that we will avoid sampling inside of the voxels we are trying to exit. Since a voxel occupies a fixed space, and the provoking geometry can intersect any part of the voxel, we must ensure that the first sampling point is at least the height of a voxel plus the minimal sampling radius (half a voxel) above the plane defined by the normal and the ray origin. However, when it comes to shadows, there's often a need for a fudge factor, so we set this value, f, to the distance of 1.5 voxels and allow it to be manipulated by the user.

Thus to compute the initial starting distance, we take one and a half, or f, times the size of the voxel times the dot product of the surface normal (not the bump-map normal) with the direction vector, assuming both are normalized:

$$d_s = \frac{f\,|v|}{(\vec{\mathbf{n}} \cdot \vec{\omega})} \tag{6.1}$$

An additional advisable practice is to clamp the initial distance between some minimum, i.e. $f_{\min}\,|v|$, and some selected maximum, $f_{\max}\,|v|$. This will prevent the cases where $\vec{\mathbf{n}} \cdot \vec{\omega}$ approaches zero, and when the d_s becomes too large and tries

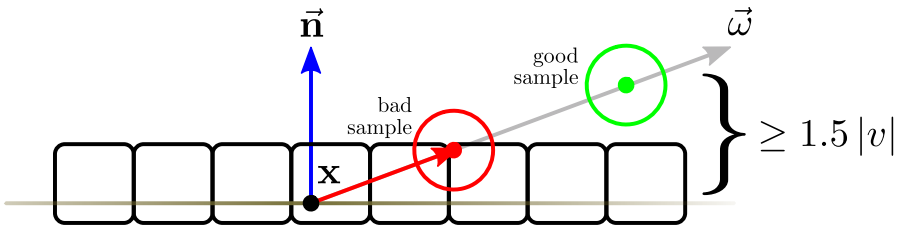


Figure 6.3: Illustration of technique

to sample outside of the volume, thus equation 6.1 becomes:

$$d_s = \begin{cases} f_{\min}\,|v| & \text{if } \vec{\mathbf{n}} \cdot \vec{\omega} > 1 \\ f_{\max}\,|v| & \text{if } \vec{\mathbf{n}} \cdot \vec{\omega} < \frac{f_{\max}}{f_{\min}} \\ \frac{f_{\min}|v|}{(\mathbf{n} \cdot \omega)} & \text{otherwise} \end{cases} \tag{6.2}$$

Note that we have specified $|v|$ as the size of the voxel, which is an indicator of the minimal length of travel needed to exit a voxel. This value differs depending on the selected voxelization method. If the voxelization is conservative, $|v|$ should be the voxel diagonal, that is $|v| = \sqrt{v_x^2 + v_y^2 + v_z^2}$. But if we are using a thin, 6-separating voxelization, $|v|$ should be the length of the longest side of the voxel, as this is the minimal length needed to exit the diamond inscribed inside the voxel.

This approach works extremely well when tracing specular cones (Section 6.5), works well on diffuse cones (Section 6.4), but does not work so well on ambient occlusion (Section 6.3), as the occlusion effect is too sensitive to the presence of self-occlusion. Since both diffuse interreflection and ambient occlusion can be calculated in the same pass, we must resort to a less elegant trick of simply shifting

the starting position, $\mathbf{x}$, some distance, f, along the normal for both. For shadowing, this technique can be helpful for light sources near the geometry, but causes artifacts for distant light sources.

6.1.2 Alternate Diffuse Cone Tracing

As will be discussed shortly in section 6.4, diffuse cones are generally traced at a wider cone aperture, commonly 60°, around a predetermined set of cone directions. At such large apertures, the challenge of avoiding self-illumination through self-intersection becomes even greater. Even if we avoid self-intersection with the first cone sample point using the method as described in section 6.1.1, we may still collide with higher mipmap levels as the sample radius expands. Another problem with such wide cones is the potential to skip through thin geometry. Thus we have implemented an alternative sampling technique for 60° cones, in which the sampling points fall precisely on each LOD level, and the radius and distance is precisely doubled with each iteration (this is less than the previous technique at 60°), this can be seen in Figure 6.4.

Another desirable property of this approach is that it removes the dependency on a mipmapped texture, since we are sampling precisely at each level of the hierarchy, we could potentially use different textures and formats for each level of the hierarchy. More on this in the Future Work section. This approach makes it easier to avoid self-intersection in higher mipmap levels. To avoid self-intersection at the base level we resort to simply offsetting the starting position, $\mathbf{x}$, some

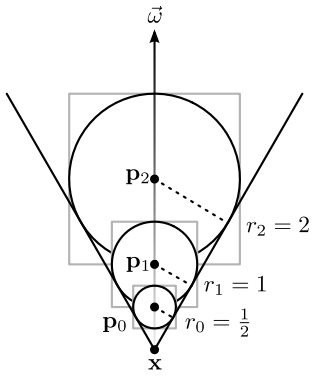


Figure 6.4: Geometric construction of the first three samples using the cone tracing technique specialized to diffuse cones at 60°, note the overlap in samples help to prevent skipping through thin geometry.

distance, f, along the normal. We have again provide GLSL code for the specialized 60° diffuse cone tracing routine in Figure B.5 in Appendix B.

6.2 Soft Shadows

Soft shadows are typically a challenging problem in computer graphics. Traditional techniques to evaluate shadows, e.g. shadow mapping, tend to create unpleasant aliasing artifacts. Over the years many techniques have attempted to address the shortcomings of shadow mapping, i.e. Variance Shadow Maps (VSM) [DL06], Percentage Closer Filtering (PCF) [RSC87], Cascaded Shadow Maps (CSM) [Dim07]. Invariably, these come with additional computational costs, memory requirements and code complexity. Ray-based approaches provide a straightforward approach to shadowing, but can potentially require tracing millions of extra rays to achieve a plausible soft shadow effect. We use soft shadows as an example of how cone-

tracing is performed, and describe in detail how to perform sampling using cone tracing.

Cone tracing completely inverts the relationship between computational complexity and shadow "softness." With cone tracing, the softer a shadow is, the easier it is to compute. This is a side effect of the cone tracing approach. In order to get "hard" shadows, we specify a cone aperture of zero degrees, which produces a result analogous to raycasting through a dense volumetric texture. All samples are taken from the lowest mipmap level, and the step size remains constant at the size of a single voxel. However, when we increase the cone's aperture, the sampling rate is reduced by a linear function of the *cone ratio*. As each sample is taken at a further distance from the previous, we start sampling from higher mipmap levels. Our shadow function thus takes on a form similar to that of the emission-absorption model from volume rendering, with the addition of hierarchical sampling, see equation 6.3.

$$S\left(\mathbf{x}, \vec{\omega}\right) = 1 - \prod_{i=0}^{n} \left(1 - V_{\text{lod},\alpha}\left[\mathbf{p}_i\right]\right) \tag{6.3}$$

We compute voxel-traced soft shadows in a deferred context, for every pixel on the screen we trace a cone towards the light source. If it arrives at the light, our view is unoccluded, otherwise we accumulate the degree of occlusion along the shadow ray. We can see the performance characteristics of tracing shadow cones of varying cone apertures in Figure 6.5. Examples of soft shadow tracing can be seen in Figure 6.6.

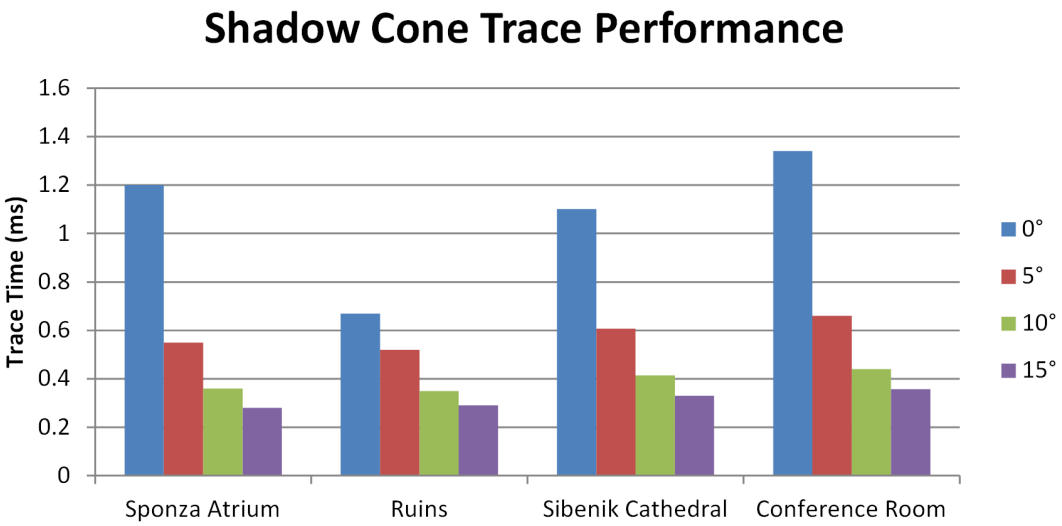


Figure 6.5: Shadow cone tracing performance of several scenes, varying the cone aperture by increments of 5°. For the Sponza and Ruins scenes, the light source is above the scene, while for the Sibenik and Conference Room scenes, the light source is inside the scene.

Figure 6.6: Voxel cone traced soft shadows of the column in the Sibenik Cathedral scene. Cone apertures vary from 0°, 2°, 4°, and 6°. Note that even the 0° cone aperture results in a slight soft shadowing effect due to the hardware based interpolation. This behavior could be modified by changing the hardware texture filtering parameters, but it is hard to imagine a scenario in which doing so would be desirable. Note, this scene exhibits no global illumination effects.

6.3 Ambient Occlusion

Ambient occlusion is often referred to as a simpler form of global illumination, when in reality it is a non-physically based heuristic used to determine an occlusion value based on the presence of local geometry. Commonly performed in screen space using the depth buffer, it provides a subtle (or sometimes not so subtle) shadowing effect at sharp concave creases in the scene geometry (i.e. interior room edges and corners). Lack of full geometry information in the depth buffer leads to artifacts at regions with depth discontinuities. Since we already have a fully volumetric filtered scene representation, we can implement a version that does not suffer from these artifacts. Much like we will see with diffuse interreflection (Section 6.4), ambient occlusion can be computed by tracing cones over the hemisphere Ω^+, and is computed as follows:

$$O(\mathbf{x}) = \frac{1}{\pi} \int_{\Omega^+} V(\mathbf{x}, \vec{\omega}) \langle \vec{\mathbf{n}} \cdot \vec{\omega} \rangle \, d\vec{\omega} \tag{6.4}$$

Applying the concept of voxel cone tracing to equation 6.4, we partition the integral into N conic regions:

$$O(\mathbf{x}) = \frac{1}{\pi} \sum_{k=0}^{N-1} \int_{\Omega_k^+} V(\mathbf{x}, \vec{\omega}_k) \langle \vec{\mathbf{n}} \cdot \vec{\omega}_k \rangle \, d\vec{\omega}_k \tag{6.5}$$

where Ω_i^+ is the portion of the upper hemisphere represented by cone i. We can move the visibility function, V, outside of the cone integral on the assumption that

visibility is the same within a cone:

$$O(\mathbf{x}) = \frac{1}{\pi} \sum_{k=0}^{N-1} V(\mathbf{x}, \vec{\omega}_k) \int_{\Omega_k^+} \langle \vec{\mathbf{n}} \cdot \vec{\omega}_k \rangle \, d\vec{\omega} \tag{6.6}$$

We can create a weight function, $W_i = \int_{\Omega_i^+} \langle \vec{\mathbf{n}} \cdot \vec{\omega} \rangle \, d\vec{\omega}$, and write the final ambient occlusion formulation as:

$$O(\mathbf{x}) = \frac{1}{\pi} \sum_{k=0}^{N-1} V(\mathbf{x}, \vec{\omega}_k) \, W_k \tag{6.7}$$

We are now left only with the task of determining the number and orientation of the cones and their associated weights. For instance, we can have 6 cones traced with 60 degrees over the hemisphere oriented as follows (courtesy of [Yeu13]):

$$\begin{aligned}
\vec{\omega}_0 &= (0.000000, 1.000000, 0.000000) \\
\vec{\omega}_1 &= (0.000000, 0.500000, 0.866025) \\
\vec{\omega}_2 &= (0.823639, 0.500000, 0.267617) \\
\vec{\omega}_3 &= (0.509037, 0.500000, -0.700629) \\
\vec{\omega}_4 &= (-0.509037, 0.500000, -0.700629) \\
\vec{\omega}_5 &= (-0.823639, 0.500000, 0.267617)
\end{aligned} \tag{6.8}$$

We can solve for the weights W_k analytically by integrating the Lambertian reflective surface for each region on the hemisphere:

$$
\begin{aligned}
W_0 &= \int_0^{2\pi}\int_0^{\frac{\pi}{6}} \cos\theta \sin\theta d\theta d\phi = \frac{\pi}{4} \\
W_1 &= \int_{-\frac{\pi}{5}}^{\frac{\pi}{5}}\int_{\frac{\pi}{6}}^{\frac{\pi}{2}} \cos\theta \sin\theta d\theta d\phi = \frac{3\pi}{20} \\
W_2 &= \int_{\frac{\pi}{5}}^{\frac{3\pi}{5}}\int_{\frac{\pi}{6}}^{\frac{\pi}{2}} \cos\theta \sin\theta d\theta d\phi = \frac{3\pi}{20} \\
W_3 &= \int_{\frac{3\pi}{5}}^{\pi}\int_{\frac{\pi}{6}}^{\frac{\pi}{2}} \cos\theta \sin\theta d\theta d\phi = \frac{3\pi}{20} \\
W_4 &= \int_{\pi}^{\frac{7\pi}{5}}\int_{\frac{\pi}{6}}^{\frac{\pi}{2}} \cos\theta \sin\theta d\theta d\phi = \frac{3\pi}{20} \\
W_5 &= \int_{\frac{7\pi}{5}}^{\frac{9\pi}{5}}\int_{\frac{\pi}{6}}^{\frac{\pi}{2}} \cos\theta \sin\theta d\theta d\phi = \frac{3\pi}{20}
\end{aligned} \tag{6.9}
$$

Which sums to π which is the expected result for a Lambertian surface:

$$\sum W_{0\ldots5} = \pi \tag{6.10}$$

Note, this should *not* sum to 2π, which is somewhat counter-intuitive, since 2π is the steradians for a hemisphere. The $\cos\theta$ term is responsible for this reduction. This is because each point on a Lambertian surface has a reflective *intensity* defined by the cosine function, however, the measured reflected radiance is still independent of the viewing direction.

Since it uses the same cones, ambient occlusion can be computed alongside diffuse interreflection with negligible extra cost. Generally, ambient occlusion cones will be traced at a much shorter distance, so we must be sure to stop updating the

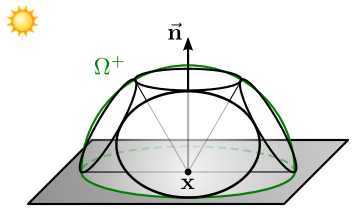


Figure 6.7: A set of cones emanating from the surface point $\mathbf{x}$, and oriented around the normal $\vec{\mathbf{n}}$ is used to compute ambient occlusion by calculating an "accessibility value" indicating the presence of nearby geometry.

occlusion value even as we continue to update the diffuse cones. Examples of the ambient occlusion calculation using cone tracing can be seen in Figure 6.8.

However, by performing ambient occlusion cone tracing in isolation, we can observe the performance characteristics of the ambient occlusion cone tracing method in Figure 6.9. Note that the ambient occlusion calculation remains the same for all the different voxel methods, thus it is not informative to differentiate between them.

6.4 Diffuse Interreflection

Diffuse interreflection works in much the same manner as ambient occlusion, except that instead of gathering occlusion values, we gather the reflected radiance from surrounding geometry, see Figure 6.10.

$$L\left(\mathbf{x} \rightarrow \vec{\omega}_o\right) = \int_{\Omega^+} f_s\left(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_o\right) \left\langle \vec{\mathbf{n}} \cdot \vec{\omega}_i \right\rangle L\left(\mathbf{x} \leftarrow \vec{\omega}_i\right) d\vec{\omega}_i \tag{6.11}$$

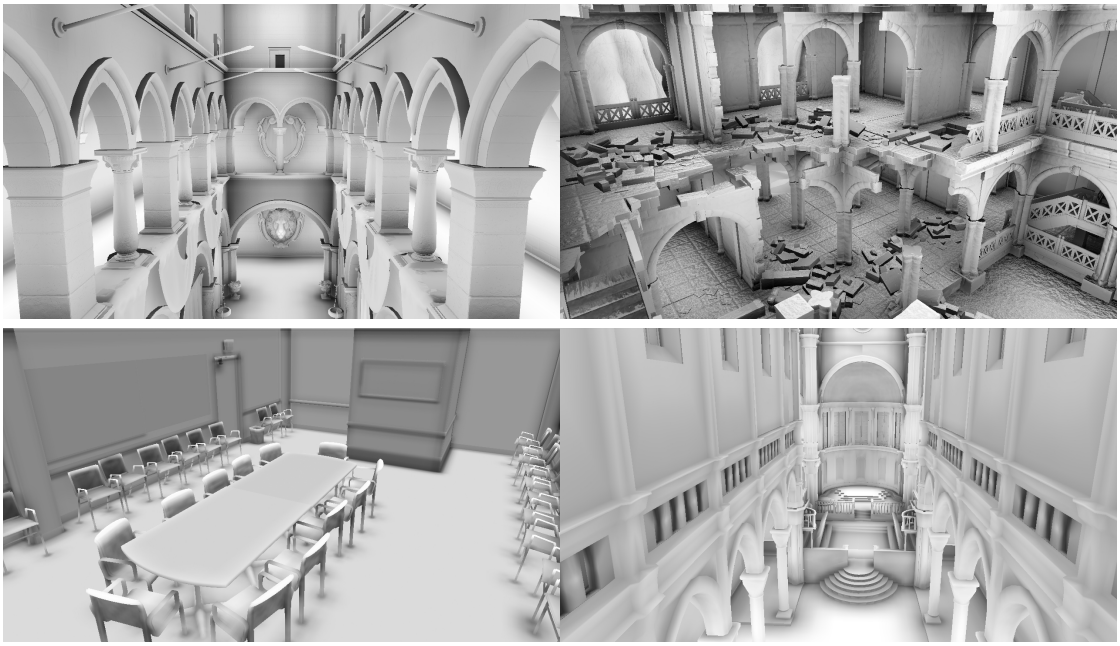


Figure 6.8: Examples of ambient occlusion computed for several classic computer graphics scenes.

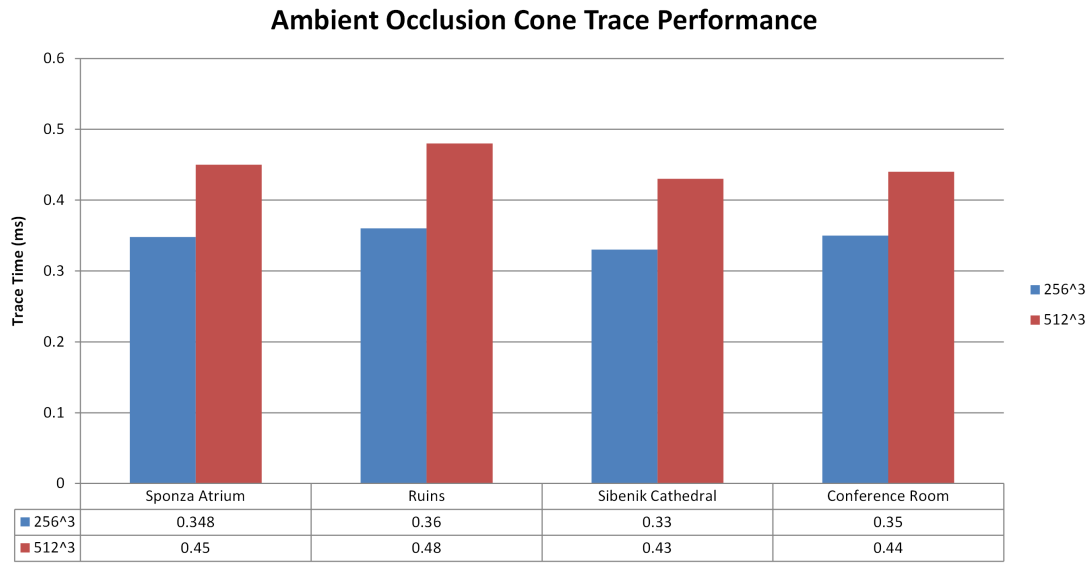


Figure 6.9: Ambient occlusion cone tracing performance for several classic computer graphics scenes at 256^3 and 512^3 voxel resolutions. Note, that the post voxelization cost of tracing different scenes is largely invariant with respect to scene geometry.

We can remove the BSDF function, f_s, as we are assuming diffuse reflectance (i.e. the same everywhere over the hemisphere):

$$L(\mathbf{x} \rightarrow \vec{\omega}_o) = \frac{\rho}{\pi} \int_{\Omega^+} \langle \vec{\mathbf{n}} \cdot \vec{\omega}_i \rangle \, L(\mathbf{x} \leftarrow \vec{\omega}_i) \, d\vec{\omega}_i \tag{6.12}$$

where $\rho \in [0, 1]$ is the diffuse reflection coefficient or *albedo*. Now it falls once again to partitioning the integral into N conic regions:

$$L(\mathbf{x} \rightarrow \vec{\omega}_o) = \frac{\rho}{\pi} \sum_{k=0}^{N-1} \int_{\Omega_k^+} \langle \vec{\mathbf{n}} \cdot \vec{\omega}_k \rangle \, L(\mathbf{x} \leftarrow \vec{\omega}_k) \, d\vec{\omega}_k \tag{6.13}$$

We can pull the incoming term out of the integral under the assumption that the incoming radiance is constant within a cone:

$$\begin{aligned} L(\mathbf{x} \rightarrow \vec{\omega}_o) &= \frac{\rho}{\pi} \sum_{k=0}^{N-1} L(\mathbf{x} \leftarrow \vec{\omega}_k) \int_{\Omega_k^+} \langle \vec{\mathbf{n}} \cdot \vec{\omega}_k \rangle \, d\vec{\omega}_k \\ &= \frac{\rho}{\pi} \sum_{k=0}^{N-1} L(\mathbf{x} \leftarrow \vec{\omega}_k) \, W_k \end{aligned} \tag{6.14}$$

where $W_k = \int_{\Omega_k^+} \langle \vec{\mathbf{n}} \cdot \vec{\omega}_k \rangle \, d\vec{\omega}_k$ is the same weight function as in Equation 6.10 which we can solve for or assign so long as the weights sum to 2π (the steradians over a hemisphere). The directions $\vec{\omega}_k$ (as show in equation 6.8) indicate the precomputed directions of the cones.

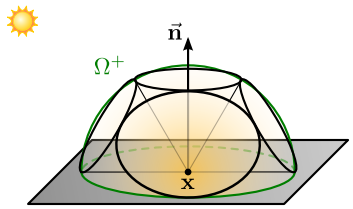


Figure 6.10: Much like ambient occlusion (cf. Figure 6.7), a set of cones emanating from the surface point $\mathbf{x}$, and oriented around the normal $\vec{\mathbf{n}}$ can be used to compute diffuse interreflection as well by accumulating the reflected illumination off of nearby geometry from the voxel based proxy.

We compare both the generic cone tracing technique described in section 6.1 and the specialized technique described in 6.1.2 and display the results in Figure 6.11.

Performance of the two techniques is far more differentiated than expected. The specialized cone tracing technique tends to outperform the generic cone tracing method, as seen in Figure 6.12, which is surprising, considering that it is actually sampling more frequently. It is possible that this is evidence that the final interpolation (between mipmap levels) of the quadrilinear interpolation is not actually performed in hardware, but instead emulated in software, or, the higher sampling rate is simply causing the specialized cone tracing to saturate and terminate earlier.

6.5 Specular Reflection

Specular reflections are achieved by tracing a single cone along a ray that is mirrored about the surface normal, see Figure 6.13. In general, the specular cone has

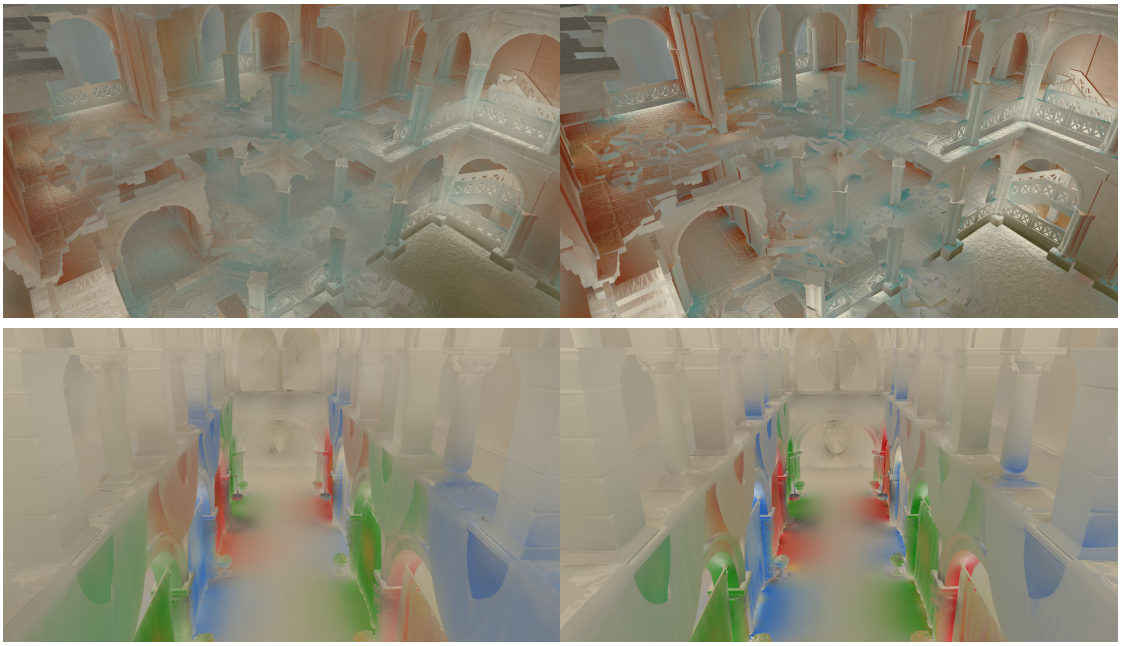


Figure 6.11: Comparison of diffuse interreflection techniques, the generic cone tracing technique at 60° is on the left, while the specialized technique for diffuse cones of 60° is on the right. We observe that the specialized cones seem to do slightly better at avoiding self-illumination, yet exhibit somewhat brighter highlights.

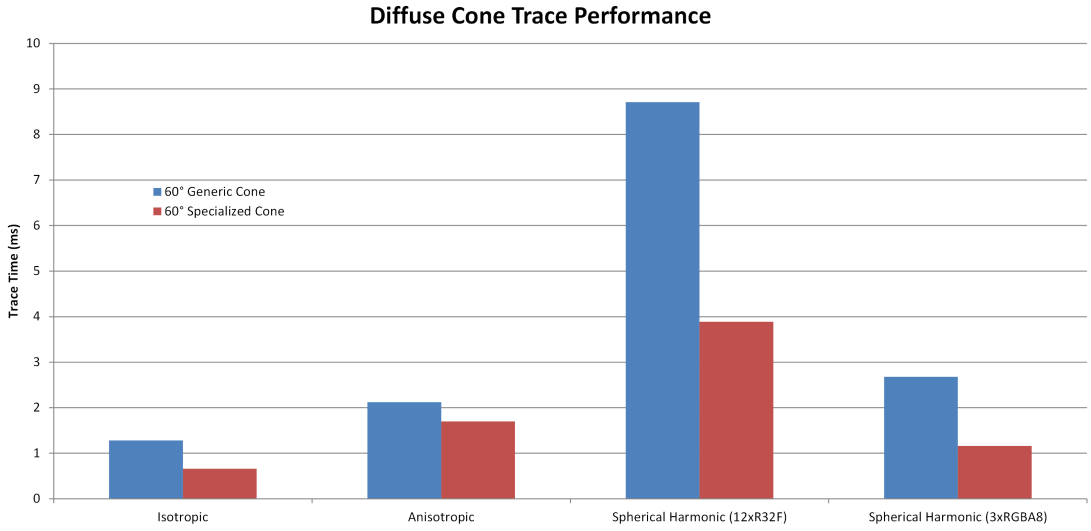


Figure 6.12: Timing data of the diffuse interreflection methods for the Sponza scene at a voxel resolution of 256^3 comparing diffuse cone tracing performance for the three implemented voxel methods and the generic vs. specialized cone tracing methods. The specialized cone tracing method provides a performance increase across all implementation, and in the case of the spherical harmonic method a speedup of over 50%. Note, the compact spherical harmonic storage (`RGBA8`) is quite competitive with isotropic and anisotropic trace times.

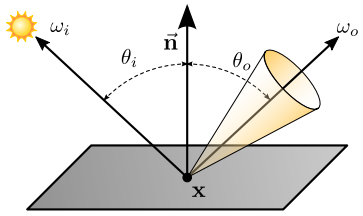


Figure 6.13: A specular cone is reflected around the normal. The cone aperture can be determined by the glossiness of the material.

a much tighter aperture than the diffuse cones, which leads to a greater sampling rate and higher traversal costs.

In the case of cones with zero degree apertures, cone tracing degenerates into ray casting within the base levels of volume textures and the specular surface reverts into a mirror, albeit one that mirrors the voxel based proxy geometry rather than the actual geometry of the scene. As the cone aperture is increased in size, the sampling rate decreases, performance increases and the result is an extremely plausible specular effect, see Figure 6.14 for examples.

The performance of this method increases dramatically as the cone aperture increases see Figure 6.15; while it may not be suitable for perfect mirror reflections, its performance for glossy type reflections is extremely competitive, and more than suitable for real-time applications.

Specifying the cone aperture is a good way to determine performance metrics, but in general, the specular reflection is based on the glossiness of the material, g, and the cone aperture calculation would be based upon this and the selected illumination model.

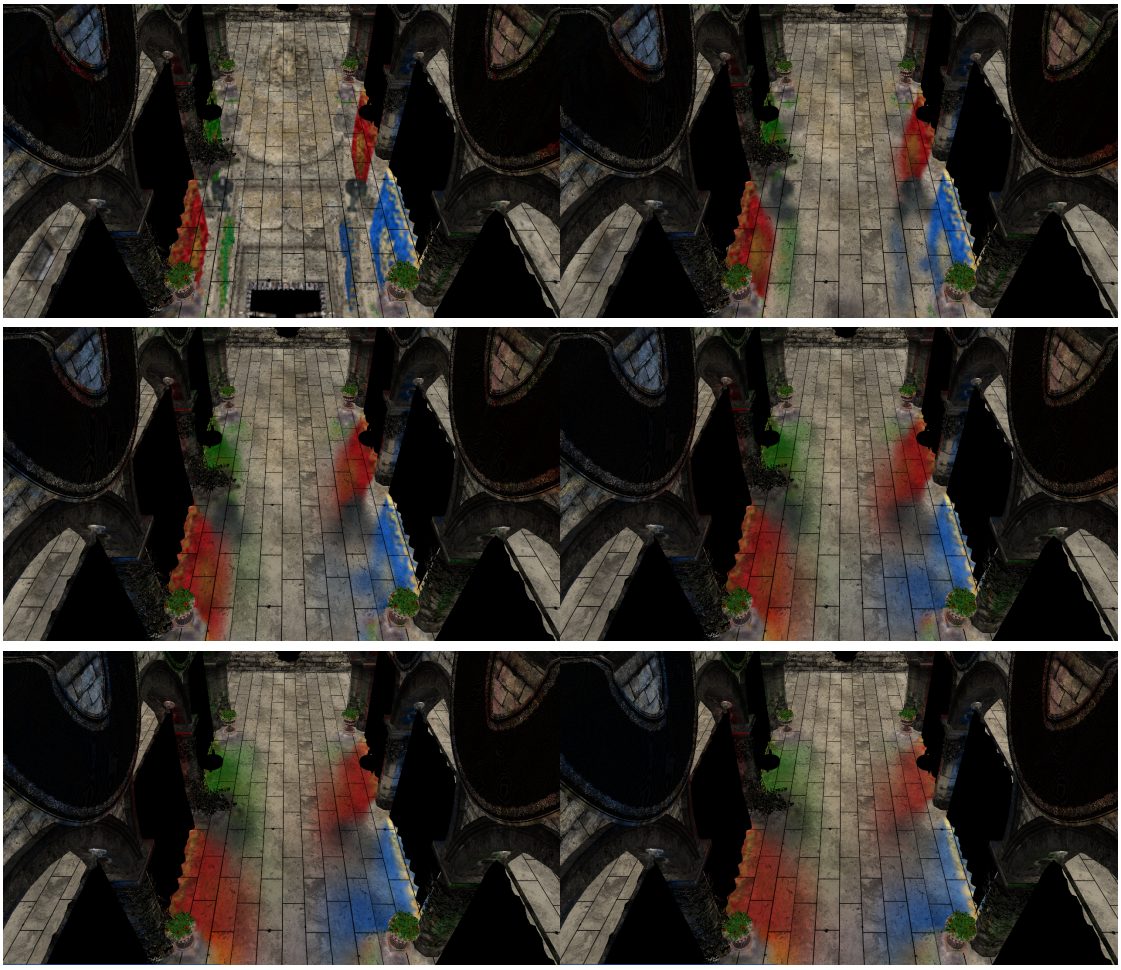


Figure 6.14: Images of specular tracing in the Crytek Sponza scene. From left to right and top to bottom cone apertures are 0, 5, 10, 15, 20, and 25 degrees respectively. Dark surfaces are not specularly reflective.

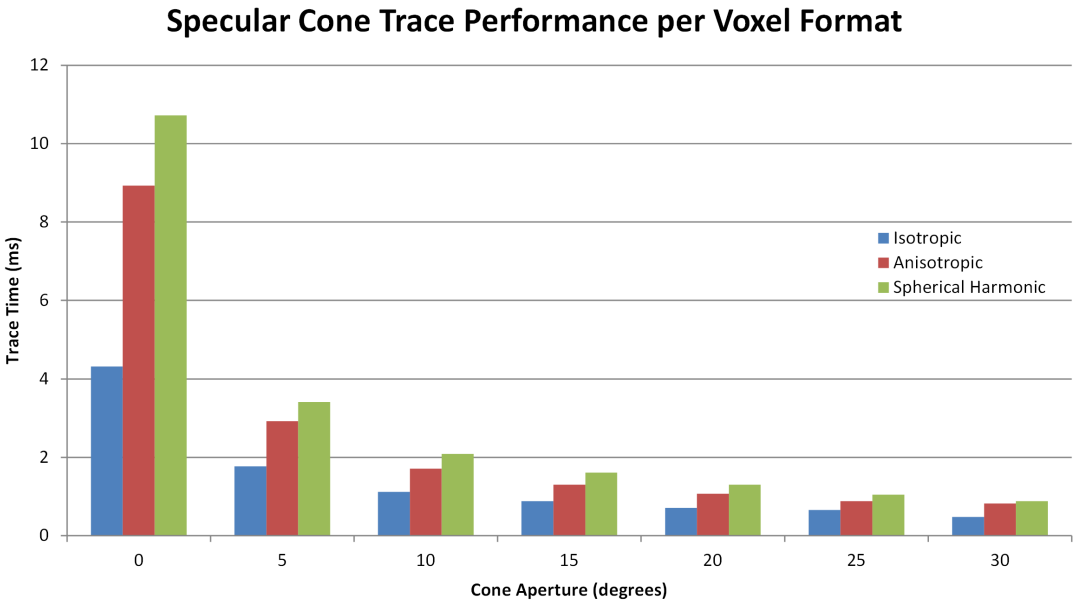


Figure 6.15: Comparison of tracing times for specular cones for the three implemented voxel formats. Tracing time for the specular cones decreases rapidly as the cone aperture increases.

Chapter 7: Voxel Based Pipeline

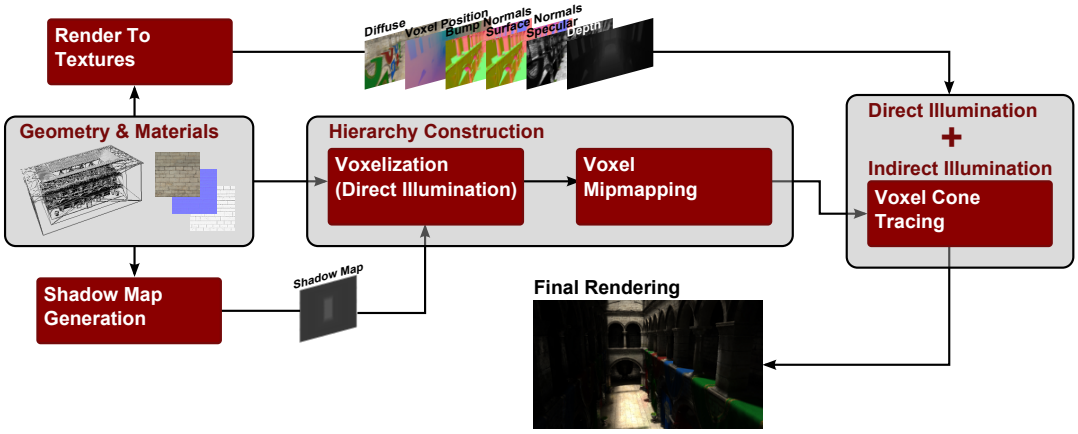


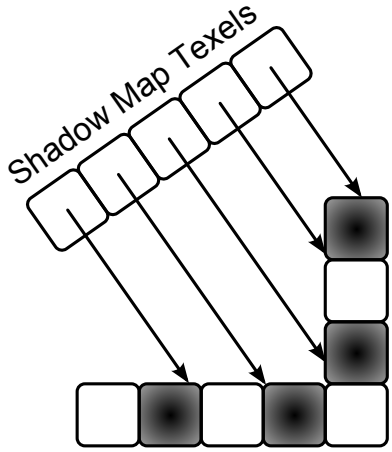
Figure 7.1: Illustration of the full voxel based lighting pipeline. We construct a filtered mipmap hierarchy of direct illumination values, which is then used to calculate the per-pixel indirect illumination component to accumulate with the direct illumination computed in a deferred context.

In Chapter 6 we have described and shown how to achieve many effects required for a global illumination solution, but not how to assemble them into a complete solution. Additionally in Chapter 5 we have discussed a variety of voxel storage approaches. Ultimately, to compute the final global illumination solution, we must incorporate both the direct and indirect illumination.

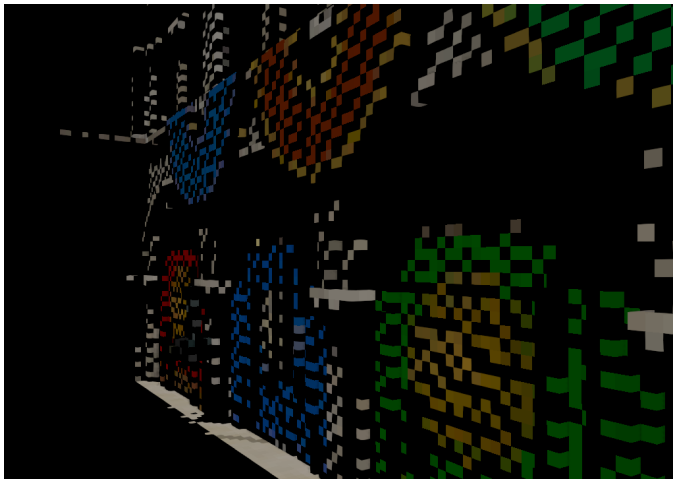
7.1 Direct Illumination

In order to compute the indirect illumination we must compute the direct illumination component for the voxels first. Effectively, we perform the direct lighting computation in two places in our pipeline (cf. Figure 7.1). We must initialize the voxel hierarchy with direct illumination values in order for it to be a viable source of gathered indirect illumination values for the final rendering pass. The initialized voxel color must also take into account the presence of shadowing information. As the voxel hierarchy is not yet constructed, we cannot employ voxel based shadows (as described in Section 6.2). Thus, we resort to using traditional shadow mapping techniques. We render shadow maps from the light sources, and pass these along to the voxelization stage along with all lighting information. This has the benefit of allowing us to avoid the light injection stage present in [CNS*11, KD10, Yeu13], but comes at the cost of requiring re-voxelization in the event of lighting changes. There are, however, several ways in which the re-voxelization costs can be mitigated and/or eliminated. For now, let us focus on the benefits of sampling from the shadow maps, rather than attempting to "inject" light into the volume. By sampling from the shadow map we avoid the problem illustrated in Figure 7.2a where a low resolution shadow map is unable to provide full coverage for the scene voxels, an effect that is displayed in Figure 7.2b. By sampling the shadow map *from* the voxels, as in Figure 7.2c, we avoid gaps in our shadow coverage.

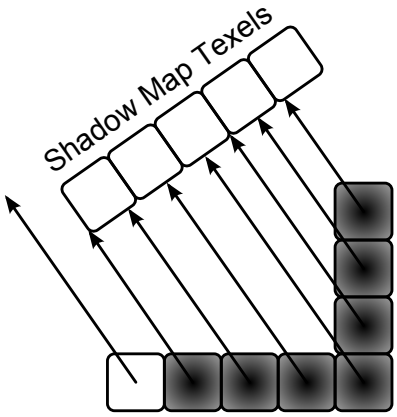
The re-voxelization cost can be mitigated somewhat by only re-voxelizing scene geometry at a different frequency than the render frame rate (i.e. re-voxelize only



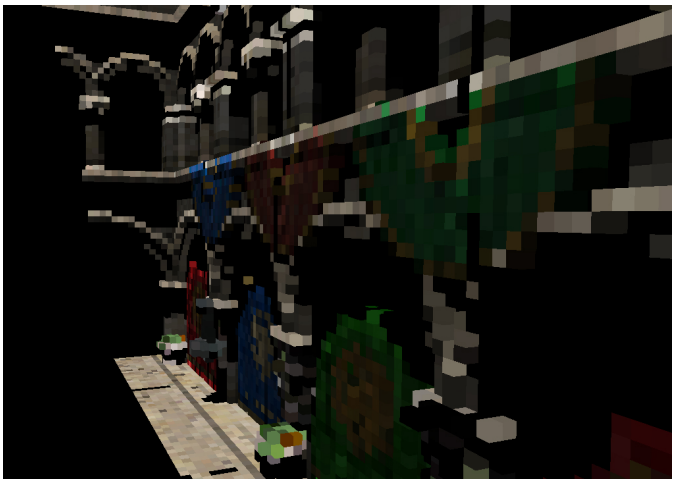
(a)



(b)



(c)



(d)

Figure 7.2: (a) Demonstrates the improper coverage of a shadow map with insufficient resolution, the result can be seen in (b) credit: [Yeu13]. By sampling the shadow map from the voxel, as in (c),we avoid shadow coverage gaps, as evidenced in (d) (which also has direct lighting information).

every 5th frame). Re-voxelization required by changing lighting conditions can be avoided entirely if during voxelization a sparse *active-voxel-list* was constructed. In this case, the active voxels can be processed and updated directly.

The computation for the color of the voxel, $\mathbf{C}_{\text{voxel}}$, is straightforward. Essentially, its color is derived from the standard direct illumination lighting equation, omitting the specular and ambient terms, as they are, quite literally, hacks to approximate indirect illumination. So, $\mathbf{C}_{\text{voxel}}$, is really just the lit and shadowed diffuse coloring:

$$\mathbf{C}_{\text{voxel}} = \mathbf{C}_{\text{light}}\mathbf{C}_{\text{diffuse}} \quad \vec{\mathbf{n}} \cdot \vec{\mathbf{l}} \quad S \tag{7.1}$$

where $\mathbf{C}_{\text{diffuse}}$ is the color sampled from either the object's diffuse texture, or it's diffuse material, $\mathbf{C}_{\text{light}}$ is the light color, $\vec{\mathbf{n}}$ is the surface normal, or the bump map normal in the presence of a bump map, and $\vec{\mathbf{l}}$ is the normalized light vector, $\vec{\mathbf{n}} \cdot \vec{\mathbf{l}}$ can also be referred to as I_{diffuse}, the intensity of the diffuse lighting contribution, and S is the shadowing contribution sampled from a traditional shadow map.

Direct illumination for the pixel, $\mathbf{C}_{\text{pixel}}$, is calculated in the exact same manner as for the voxel, except that in this case we have the option to use traditional shadow mapping as with $\mathbf{C}_{\text{voxel}}$, or, use the voxel cone tracing based shadow function described in Eq. 6.3, from Section 6.2. Since we are demonstrating the viability of voxel based rendering approaches, we used the voxel based shadowing approach for $\mathbf{C}_{\text{pixel}}$.

7.2 Indirect Illumination

Indirect illumination is computed using the technique described in Chapter 6. Since we are rendering in a deferred context, for every pixel in the scene we have access to its diffuse color, voxel position, bump map normal, surface normal, and depth. We use the voxel position and normal information to initiate the indirect diffuse color computation, $\mathbf{C}_{\text{indirectDiffuse}}$, as per the methodology described in 6.4 as a side effect of tracing diffuse cones. We also compute (using an additional guard on sample distance) the ambient occlusion, O, in the scene at no additional computational cost. To compute the indirect specular component, $\mathbf{C}_{\text{indirectSpecular}}$, we use the method described in Section 6.5.

7.3 Final Rendering & Results

We perform our final rendering in a deferred context, accomplished by rendering a single screen covering triangle, and sampling from deferred textures holding diffuse, normal, specular, voxel position, and depth information. We have described how to find the direct and indirect lighting contributions, and now have all the components needed to compute our final global illumination solution. But before we proceed, we discuss how best to incorporate the ambient occlusion contribution, O, since it is not a physically based quantity, we can elect to omit it, but ambient occlusion can be useful to hide artifacts such as light leakage. Let us define the ambient occlusion contribution, c_O, which can be varied from 0 to 1. Since, ambient occlusion is a shadow term, we modulate the shadow term, S, by the ambient

occlusion contribution, O, with a weight determined by c_O. That is:

$$S = O\,(1 - c_O) + S \cdot O \cdot c_O \tag{7.2}$$

allowing the user to control the ambient occlusion contribution as desired. Finally, now that we have defined all the terms of our final lighting formula, we can express the final color value, $\mathbf{C}_{\text{final}}$, as:

$$\mathbf{C}_{\text{final}} = \mathbf{C}_{\text{light}} \left(\vec{\mathbf{n}} \cdot \vec{\mathbf{l}}\right) S + \mathbf{C}_{\text{indirectDiffuse}} \mathbf{C}_{\text{diffuse}} + \mathbf{C}_{\text{indirectSpecular}} \mathbf{C}_{\text{specular}} \tag{7.3}$$

where $\mathbf{C}_{\text{specular}}$ is the specular sample stored in the deferred specular texture.

Final renderings can be seen in Figure 7.3, note the presence of soft shadowing, diffuse interreflection, and glossy surfaces. Additionally, images depicting the incremental addition of voxel cone tracing based illumination effects can be seen in Figure A.3 and Figure A.4 in Appendix A.

Unless otherwise mentioned (as in Chapter 4), all results were generated on an Intel Core i7 950 @ 3.07 GHz with an NVIDIA GeForce Titan GPU. The selection of the Titan GPU was made because its 6 gigabytes of on-board RAM allowed us to implement our approach using dense 3D textures while waiting for inevitable release sparse texture support. Otherwise it would have been necessary to expend significant time and effort towards the creation of complex data-structures which have nothing to do with our research. Additionally, until the late addition of bindless textures (a feature only available on Kepler class hardware) our approach

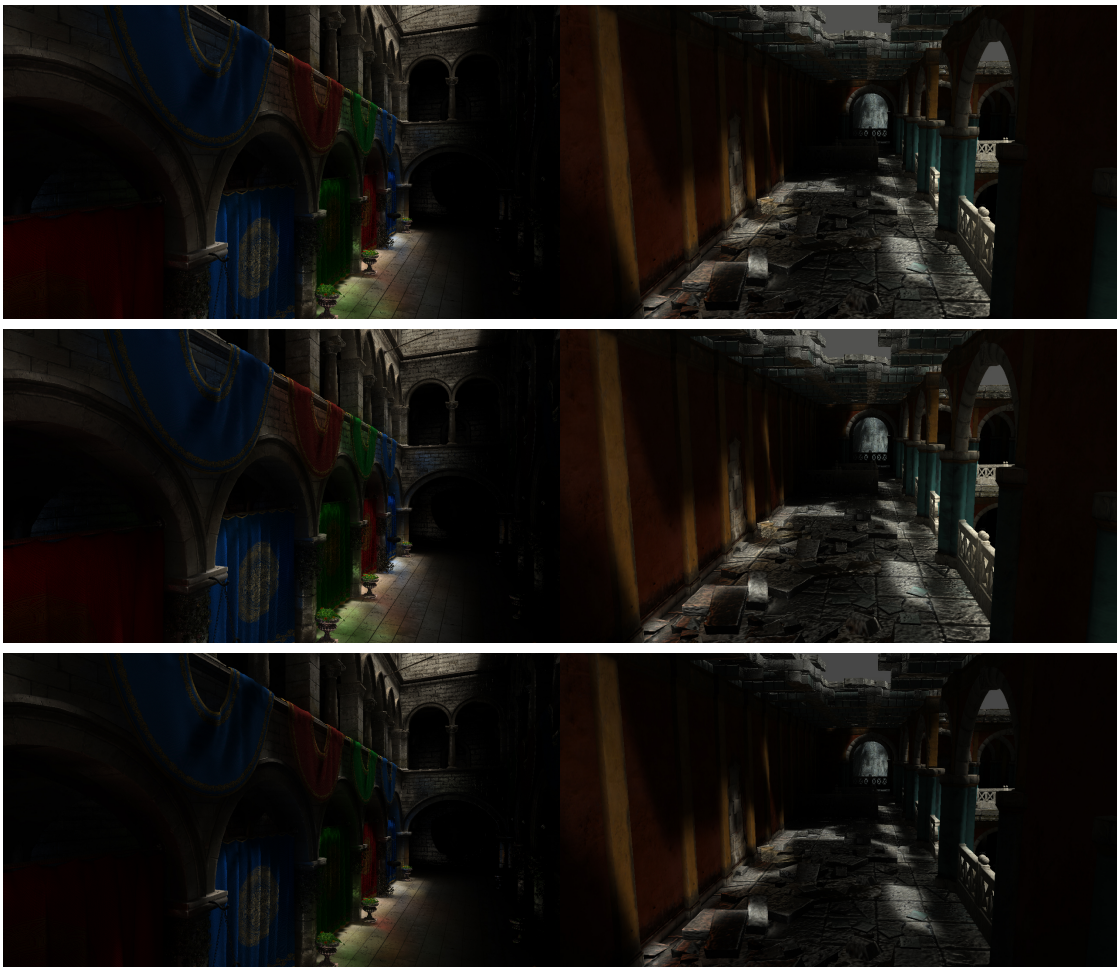


Figure 7.3: Full global illumination final renderings for the Sponza Atrium and the Ruins. The images are rendering using isotropic, anisotropic, and spherical harmonic voxels from the top row to the bottom, respectively.

worked just fine on a much older GTX 480 (Fermi class hardware) with 1.5GB of on-board ram, albeit at a lower voxel resolution. We have largely displayed timings and results for each component of the rendering (or pipeline component) as we have introduced them. All scenes have been rendered at a resolution of 1280×720 (720p). We now display the final render timings for Sponza Atrium and Ruins scenes in 7.4 as a sum of their component parts. These scenes were selected as they provided the best approximation of the geometric and material complexity likely to be found in modern games. In many ways the timings illustrated in Figure 7.4 are a representation of the worst case performance results. We have not enabled sparse-voxelization, nor have we fully specialized the voxelization pipeline to each target storage format using the methods in Chapter 4. The entire rendering engine is solely the product of a single graduate student, working without the benefit of guidance from an industry rendering expert. Furthermore, as will be discussed, there are many further optimizations that could be implemented that do not directly relate to the subject matter of this dissertation.

Nevertheless, the results are sufficient for comparing the strengths and weakness of the implemented storage formats. As expected, the simplest format, isotropic, is the fastest, while the complexity of the anisotropic and spherical harmonic formats predictably increase rendering times. This trend holds true when looking at voxelization performance in isolation. While there is a performance cost going from the isotropic to anisotropic formats, there is perhaps not so great a cost as expected when moving from the anisotropic to spherical harmonic formats. Surprisingly, the spherical harmonic mipmapping is over twice as fast as the anisotropic mipmap-

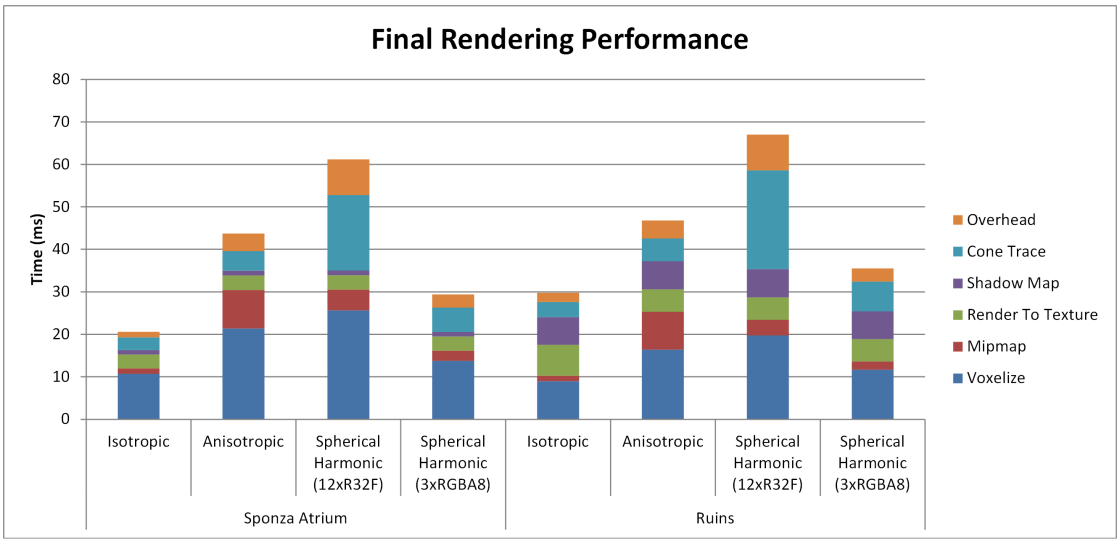


Figure 7.4: Complete profiles of the final rendering times for the Sponza Atrium scene and the Ruins scene, for the isotropic, anisotropic, and spherical harmonic storage formats.

ping. Unfortunately, the sampling cost is by far the highest for the spherical harmonic approach, as it is clearly superior in capturing directionality information, as best seen in Figure A.2 in Appendix A. In the end, these timings serve less as an indicator of their suitability as real-time rendering approaches, but rather an indicator of their time until more widespread adoption.

Chapter 8: Conclusions and Future Work

In this dissertation, we have presented three voxel storage models for use in voxel cone tracing. These storage models allow for the comparison between directionally independent isotropic voxels vs. directionally dependent anisotropic voxels implemented as a discrete set of per face color values (cf. Section 5.1.2), and directionally dependent anisotropic voxels implemented as spherical harmonics (cf. Section 5.1.3), a novel contribution, unique to this work.

In support of this work, we have contributed an innovative voxelization approach, detailed in Chapter 4, that is currently the fastest known method for generating a regular discretized geometry representation from an input triangular surface geometry on consumer grade graphics hardware.

To facilitate voxel cone tracing, we have detailed efficient real-time filtered hierarchy creation within the mipmap levels of the supported texture formats, enabling hardware accelerated *quadrilinear* filtering of our voxelized volumetric scene proxy (cf. Section 5.3). Once again, the method for the mipmapping of spherical harmonics (cf. Section 5.3.3) represents a new contribution found only in this work. Additionally, we have contributed a method for performing sparse computations in a dense voxel storage medium by employing *active-voxel-lists*, vastly accelerating mipmapping time at a small cost to voxelization time (cf. Section 5.4).

We have described in unprecedented detail the geometric construction of cone

tracing based sampling methods (cf. Section 6.1), and described methods for avoiding self-intersection when tracing within a voxelized environment (cf. Section 6.1.1). We have further expanded upon the concept of voxel cone tracing by noting the potential for optimizing and specializing cone based sampling methods based upon cone aperture. In Section 6.1.2, we described a cone tracing method optimized for tracing cones with an aperture of 60°, an angle ideally suited for tracing diffuse interreflection. Furthermore, this alternate cone tracing method has the side effect of sampling precisely at each mipmap level, obviating the need for the previously desirable quadrilinear hardware filtering, and enabling instead the possibility of using different voxel storage methods (and texture formats) for different levels of the voxel hierarchy, i.e. higher order spherical harmonics at higher levels of the hierarchy.

In Chapter 6 we described the computation of many global illumination effects using voxel cone tracing, which many previously considered too expensive for real-time rendering. These include Soft Shadows (Section 6.2), Ambient Occlusion (Section 6.3), Diffuse Interreflection (Section 6.4), and glossy Specular Reflections (Section 6.5). Our care in constructing a generic voxel based cone tracing pipeline allowed every one of these effects to be executed for any voxel storage method on the condition that appropriate methods were provided for the following: directional voxel sampling, voxel storage, and voxel mipmapping.

There are many techniques that can enhance final rendering performance that are orthogonal to the approaches discussed in this thesis. For instance, performing cone tracing at half resolution and up-sampling the indirect lighting results back

to full resolution, or exploiting temporal coherence to avoid a full indirect lighting computation for each frame as in [SHR*09]. In the same vein, assuming a naive scheme of re-voxelizing all scene geometry, we can lower the update frequency of the voxelization and mipmapping without introducing too many artifacts. This will generally be dependent on the speed of moving geometry in the scene. Considering that voxelization is often the most expensive part of the pipeline (especially for simpler voxel storage formats), we can further amortize the cost of voxelization by trivially implementing a system that flags and re-voxelizes only the scene geometry that has actually been changed, providing a dramatic cost savings in practical real-world scenarios.

Furthermore, considering the presence of dedicated rasterization hardware on modern GPUs and the many similarities between voxelization and rasterization, it would seem (at least to this author) that the extension of the rasterization hardware to more natively support voxelization would be a fruitful avenue of exploration yielding yet faster and more easily implemented dynamic voxelizations. Outside the realm of theoretical hardware improvements, the recent introduction of broad API support for sparse textures should lead to dramatically reduced memory requirements and/or higher voxel resolutions.

Shadow cone tracing is interesting, but falls in the awkward position of requiring that it execute after the stage in which it would be most useful; that is, it must occur after hierarchy construction in which shadowing information is useful for constructing the initial radiance distribution. That being said, it does still have potential application as a means of computing soft penumbra regions when coupled

with traditional shadow mapping.

Another interested avenue of exploration would be to further experiment with combinations of the light propagation volumes and voxel cone tracing techniques. For instance, we could select a level of the spherical harmonic mipmap hierarchy and perform light propagation based radiance diffusion, and sample from this for the indirect diffuse component of our lighting equation, and combine this with specular cone tracing for the indirect specular component of our lighting equation.

Future work in improving voxelization performance could exploit *true* dynamic parallelism facilities currently only available in CUDA 5 to spawn exactly one thread for each triangle/voxel pair. While this would still obviate the need for complex tiling and sorting strategies, it would unfortunately remove the ability to exploit the remaining fixed-function hardware present on the GPU exposed to the graphics pipeline. Additionally, we could explore more robust cutoff prediction strategies, techniques for automatic minimum detection, and more sophisticated classification approaches.

The voxel cone tracing approach is, at the same time, deceptively simple, and extremely robust. It excels at reproducing effects that would otherwise be extremely computationally expensive. Ultimately, its adoption in major game engines seems inevitable [Mit12], and the evolution of voxel cone tracing will invariably be tied to accuracy of the directional radiance sampling in the upper levels of its voxel hierarchy. In order to fully exploit the available filtering hardware of modern GPUs, this directionally dependent radiance storage *must* be based on orthogonal functions, thus naturally leading to spherical harmonics. In this dissertation

we have not only endeavored to, but succeeded in demonstrating that spherical harmonics represent a viable voxel storage format for voxel cone tracing.

Bibliography

[AKDS04] Annen T., Kautz J., Durand F., Seidel H.-P.: Spherical harmonic gradients for mid-range illumination. *Proceedings of Eurographics Symposium on Rendering* (Jan 2004).

[AM05] Akenine-Möller T.: Fast 3d triangle-box overlap testing. In *ACM SIGGRAPH 2005 Courses* (New York, NY, USA, 2005), SIGGRAPH '05, ACM.

[ATS94] Arvo J., Torrance K., Smits B.: A framework for the analysis of error in global illumination algorithms. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1994), SIGGRAPH '94, ACM, pp. 75–84.

[AUW07] Akerlund O., Unger M., Wang R.: Precomputed Visibility Cuts for Interactive Relighting with Dynamic BRDFs. *Proceedings Pacific Graphics* (Jan 2007).

[AWB08] Arbree A., Walter B., Bala K.: Single-pass scalable subsurface rendering with lightcuts. *Eurographics* (Jan 2008).

[BB11] Bolz J., Brown P.: `ARB_shader_image_load_store`, June 2011.

[BD08] Bodt T. D., Dutre P.: *Coherent Lightcuts*. Tech. rep., Katholieke Universiteit Leuven, August 2008.

[BF89] Buchalew C., Fussell D.: Illumination networks: fast realistic rendering with general reflectance functions. *ACM SIGGRAPH Computer Graphics* (Jan 1989).

[BT99] Bala K., Teller S.: Radiance interpolants for accelerated bounded-error ray tracing. *ACM Transactions on Graphics (TOG)* (Jan 1999).

[Bun05] Bunnell M.: Dynamic ambient occlusion and indirect lighting. *GPU Gems* (Jan 2005).

[BWG03] Bala K., Walter B., Greenberg D. P.: Combining edges and points for interactive high-quality rendering. In *ACM SIGGRAPH 2003 Papers* (New York, NY, USA, 2003), SIGGRAPH '03, ACM, pp. 631–640.

[CB04] Christensen P., Batali D.: An irradiance atlas for global illumination in complex production scenes. *Eurographics Symposium on Rendering 2004* (Dec 2004), 10.

[CG85] Cohen M., Greenberg D.: The hemi-cube: a radiosity solution for complex environments. *ACM SIGGRAPH Computer Graphics* (Jan 1985).

[CG12] Crassin C., Green S.: *Octree-Based Sparse Voxelization Using The GPU Hardware Rasterizer*. CRC Press, Patrick Cozzi and Christophe Riccio, 2012.

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

[CHH03] Carr N. A., Hall J. D., Hart J. C.: GPU algorithms for radiosity and subsurface scattering. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (Aire-la-Ville, Switzerland, Switzerland, 2003), HWWS '03, Eurographics Association, pp. 51–59.

[CHL04] Coombe G., Harris M., Lastra A.: Radiosity on graphics hardware. *GI '04: Proceedings of Graphics Interface 2004* (May 2004).

[CLS97] Christensen P., Lischinski D., Stollnitz E.: Clustering for glossy global illumination. *ACM Transactions on Graphics (TOG)* (Jan 1997).

[CMS87] Cabral B., Max N., Springmeyer R.: Bidirectional reflection functions from surface bump maps. *ACM SIGGRAPH Computer Graphics* (Jan 1987).

[CNS*11] Crassin C., Neyret F., Sainz M., Green S., Eisemann E.: Interactive indirect illumination using voxel cone tracing. *Computer*

Graphics Forum (Proceedings of Pacific Graphics 2011) 30, 7 (sep 2011).

[Cra11] Crassin C.: *GigaVoxels: A Voxel-Based Rendering Pipeline For Efficient Exploration Of Large And Detailed Scenes*. PhD thesis, UNIVERSITE DE GRENOBLE, July 2011. English and web-optimized version.

[DCB*04] Dong Z., Chen W., Bao H., Zhang H., Peng Q.: Real-time voxelization for complex polygonal models. In *12th Pacific Conference on Computer Graphics and Applications, 2004. PG 2004. Proceedings* (2004), pp. 43–50.

[Dim07] Dimitrov R.: *Cascaded shadow maps*. Tech. rep., NVIDIA Corporation, August 2007.

[DKTS07] Dong Z., Kautz J., Theobalt C., Seidel H.-P.: Interactive global illumination using implicit visibility. In *Proceedings of the 15th Pacific Conference on Computer Graphics and Applications* (Washington, DC, USA, 2007), PG '07, IEEE Computer Society, pp. 77–86.

[DL06] Donnelly W., Lauritzen A.: Variance shadow maps. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2006), I3D '06, ACM, pp. 161–165.

[DS97] Drettakis G., Sillion F. X.: Interactive update of global illumination using a line-space hierarchy. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1997), SIGGRAPH '97, ACM Press/Addison-Wesley Publishing Co., pp. 57–64.

[DS05] Dachsbacher C., Stamminger M.: Reflective shadow maps. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2005), I3D '05, ACM, pp. 203–231.

[DS06] Dachsbacher C., Stamminger M.: Splatting indirect illumination. *Proceedings of the 2006 symposium on Interactive 3D graphics* (Jan 2006).

[DSDD07] Dachsbacher C., Stamminger M., Drettakis G., Durand F.: Implicit visibility and antiradiance for interactive global illumination. vol. 26, ACM.

[DZTS07] Dyken C., Ziegler G., Theobalt C., Seidel H.-P.: *HistoPyramids in Iso-Surface Extraction*. Tech. rep., Max-Planck-Institut fÃŒr Informatik, 2007.

[ED06] Eisemann E., Décoret X.: Fast Scene Voxelization and Applications. *ACM SIGGRAPH* (2006), 71–78.

[ED08] Eisemann E., Décoret X.: Single-pass GPU solid voxelization for real-time applications. In *Proceedings of graphics interface 2008* (2008), Canadian Information Processing Society, pp. 73–80.

[FC00] Fang S., Chen H.: Hardware accelerated voxelization. *Computers and Graphics 24* (2000).

[FCL05] Fan S., Chenney S., Lai Y.: Metropolis photon sampling with optional user guidance. *Eurographics Symposium on Rendering* (Jan 2005).

[Fol13] Foley T.: A digression on divergence. http://tangentvector.wordpress.com/2013/04/12/a-digression-on-divergence/, April 2013.

[Gai12] Gaitatzes A.: *Interactive Diffuse Global Illumination Discretization Methods for Dynamic Environments*. PhD thesis, University of Cyprus, 2012.

[GBP08] Gautron P., Bouatouch K., Pattanaik S.: Temporal radiance caching. In *ACM SIGGRAPH 2008 classes* (New York, NY, USA, 2008), SIGGRAPH '08, ACM, pp. 79:1–79:29.

[GKBP05] Gautron P., Krivanek J., Bouatouch K., Pattanaik S.: Radiance cache splatting: a gpu-friendly global illumination algorithm (sketch). *International Conference on Computer Graphics and ...* (Jan 2005).

[GKPB04] Gautron P., Krivanek J., Pattanaik S., Bouatouch K.: A novel hemispherical basis for accurate and efficient rendering. *Proc. Eurographics Symp. Rendering* (Jan 2004).

[Gre03] Green R.: Spherical harmonic lighting: The gritty details. *Archives of the Game Developers Conference* (2003).

[GTGB84] Goral C., Torrance K., Greenberg D., Battaile B.: Modeling the interaction of light between diffuse surfaces. *ACM SIGGRAPH Computer Graphics* (Jan 1984).

[Guo98] Guo B.: Progressive radiance evaluation using directional coherence maps. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1998), SIGGRAPH '98, ACM, pp. 255–266.

[HAMO05] Hasselgren J., Akenine-Möller T., Ohlsson L.: Conservative Rasterization. In *GPU Gems 2*. 2005, pp. 677–690.

[Hec91] Heckbert P.: *Simulating global illumination using adaptive meshing.* PhD thesis, UC Berkeley, June 1991.

[HHS05] Havran V., Herzog R., Seidel H.: Fast final gathering via reverse photon mapping. *Computer Graphics Forum* (Jan 2005).

[HHW09] Hertel S., Hormann K., Westermann R.: A hybrid gpu rendering pipline for alias-free hard shadows. In *Proceedings of Eurographics 2009* (2009).

[HPB06] Hašan M., Pellacini F., Bala K.: Direct-to-indirect transfer for cinematic relighting. In *ACM SIGGRAPH 2006 Papers* (New York, NY, USA, 2006), SIGGRAPH '06, ACM, pp. 1089–1097.

[HPB07] Hašan M., Pellacini F., Bala K.: Matrix row-column sampling for the many-light problem. *SIGGRAPH '07: SIGGRAPH 2007 papers* (Aug 2007).

[HSA91] Hanrahan P., Salzman D., Aupperle L.: A rapid hierarchical radiosity algorithm. *ACM SIGGRAPH Computer Graphics* (Jan 1991).

[HSM10] Howell J. R., Siegel R., Menguc M. P.: *Thermal Radiation Heat Transfer, 5th Edition*, 5 ed. CRC Press, 9 2010.

[IDYN07] Iwasaki K., Dobashi Y., Yoshimoto F., Nishita T.: Precomputed radiance transfer for dynamic scenes taking into account light interreflection. *Proceedings of Eurographics Symposium on Rendering 2007* (2007).

[Ins10] Insomniac Games: Useful results in spherical harmonics (mainly 2-band). October 2010.

[IZT*07] Ihrke I., Ziegler G., Tevs A., Theobalt C., Magnor M., Seidel H.-P.: Eikonal rendering: efficient light transport in refractive objects. *SIGGRAPH '07: SIGGRAPH 2007 papers* (Aug 2007).

[Jar08] Jarosz W.: *Efficient Monte Carlo Methods for Light Transport in Scattering Media*. PhD thesis, UC San Diego, Sept. 2008.

[JB02] Jensen H. W., Buhler J.: A rapid hierarchical rendering technique for translucent materials. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2002), SIGGRAPH '02, ACM, pp. 576–581.

[JC95] Jensen H., Christensen N. J.: Efficiently rendering shadows using the photon map. *Compugraphics'95* (1995).

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

[JMLH01] Jensen H. W., Marschner S. R., Levoy M., Hanrahan P.: A practical model for subsurface light transport. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2001), SIGGRAPH '01, ACM, pp. 511–518.

[Kaj86] Kajiya J. T.: The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1986), ACM, pp. 143–150.

[KAMJ05] Kristensen A., Akenine-Möller T., Jensen H.: Precomputed local radiance transfer for real-time lighting design. *Proceedings of ACM SIGGRAPH 2005* (Jan 2005).

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

[KDS96] Koenderink J. J., Doorn A. J. v., Stavridi M.: Bidirectional reflection distribution function expressed in terms of surface scattering modes. In *Proceedings of the 4th European Conference on Computer Vision-Volume II - Volume II* (London, UK, UK, 1996), ECCV '96, Springer-Verlag, pp. 28–39.

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

[KGBP05] Křivánek J., Gautron P., Bouatouch K., Pattanaik S.: Improved radiance gradient computation. In *Proceedings of the 21st spring conference on Computer graphics* (New York, NY, USA, 2005), SCCG '05, ACM, pp. 155–159.

[KGPB08] Křivánek J., Gautron P., Pattanaik S., Bouatouch K.: Radiance caching for efficient global illumination computation. In *ACM SIGGRAPH 2008 classes* (New York, NY, USA, 2008), SIGGRAPH '08, ACM, pp. 75:1–75:19.

[KGW*07] Křivánek J., Gautron P., Ward G., Arikan O., Jensen H. W.: Practical global illumination with irradiance caching. In *ACM SIGGRAPH 2007 courses* (New York, NY, USA, 2007), SIGGRAPH '07, ACM.

[KKZ08] Ko J., Ko M., Zwicker M.: Practical methods for a prt-based shader using spherical harmonics. In *ShaderX6 - Advanced Rendering Techniques*, Engel W., (Ed.). Charles River Media, 2008.

[KL05] Kontkanen J., Laine S.: Ambient occlusion fields. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2005), I3D '05, ACM, pp. 41–48.

[KLA04] Kautz J., Lehtinen J., Aila T.: Hemispherical rasterization for self-shadowing of dynamic objects. *Proceedings of the Eurographics Symposium on Rendering* (Jan 2004).

[KLS*05] Kniss J., Lefohn A., Strzodka R., Sengupta S., Owens J. D.: Octree textures on graphics hardware. In *ACM SIGGRAPH 2005 Sketches* (New York, NY, USA, 2005), SIGGRAPH '05, ACM.

[KSS02] Kautz J., Sloan P.-P., Snyder J.: Fast, arbitrary BRDF shading for low-frequency lighting using spherical harmonics. *Proceedings of the 13th Eurographics workshop on Rendering* (Jan 2002).

[KTHS06] Kontkanen J., Turquin E., Holzschuch N., Sillion F. X.: Wavelet radiance transport for interactive indirect lighting. In *Proceedings of the 17th Eurographics conference on Rendering Techniques* (Aire-la-Ville, Switzerland, Switzerland, 2006), EGSR'06, Eurographics Association, pp. 161–171.

[KW00] Keller A., Wald I.: *Efficient Importance Sampling Techniques for the Photon Map*. Tech. rep., Jan 2000.

[LC04] Larsen B., Christensen N. J.: Simulating photon mapping for real-time applications. *Rendering Techniques* (Jan 2004).

[LD08] Lagae A., Dutré P.: Compact, fast and robust grids for ray tracing. In *ACM SIGGRAPH 2008 talks* (New York, NY, USA, 2008), SIGGRAPH '08, ACM, pp. 20:1–20:1.

[Leh04] Lehtinen J.: *Foundations of Precomputed Radiance Transfer*. Master's thesis, Helsinki University of Technology, Jan 2004.

[LF96] Lewis R. R., Fournier A.: Light-driven global illumination with a wavelet representation of light transport. In *Proceedings of the eurographics workshop on Rendering techniques '96* (London, UK, UK, 1996), Springer-Verlag, pp. 11–ff.

[LFWK05] Li W., Fan Z., Wei X., Kaufman A.: GPU-based flow simulation with complex boundaries. *GPU Gems 2* (2005), 747764.

[LP03] Lavignotte F., Paulin M.: Scalable photon splatting for global illumination. In *Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia* (New York, NY, USA, 2003), GRAPHITE '03, ACM, pp. 203–ff.

[LSKL07] Laine S., Saransaari H., Kontkanen J., Lehtinen J.: Incremental instant radiosity for real-time indirect illumination. *Proceedings of Eurographics Symposium on Rendering 2007* (Jan 2007).

[LSSS04] Liu X., Sloan P.-P., Shum H.-Y., Snyder J.: All-frequency precomputed radiance transfer for glossy objects. *Eurographics Symposium on Rendering* (Jan 2004).

[LTG92] Lischinski D., Tampieri F., Greenberg D.: Discontinuity meshing for accurate radiosity. *Computer Graphics and Applications* (Jan 1992).

[MAH07] Malmer M., Assarsson U., Holzschuch N.: Fast precomputed ambient occlusion for proximity shadows. *Journal of Graphics Tools* (Jan 2007).

[Mak96] Makhotkin O.: Analysis of radiative transfer between surfaces by hemispherical harmonics. *J. Quant. Spectrosc. Radiat. Transfer* (Jan 1996).

[McG12] McGuire M.: *The Graphics Codex*. iTunes, 2012.

[Men12] Menzel R.: Shader model and glsl versions. http://renderingpipeline.com/2012/03/shader-model-and-glsl-versions/, 3 2012.

[Mit12] Mittring M.: The Technology Behind the "Unreal Engine 4 Elemental demo", 2012.

[MM02] Ma V. C. H., McCool M. D.: Low latency photon mapping using block hashing. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (Aire-la-Ville, Switzerland, Switzerland, 2002), HWWS '02, Eurographics Association, pp. 89–99.

[MMG06] Mitchell J., McTaggart G., Green C.: Shading in valve's source engine. *ACM SIGGRAPH 2006* (2006), 129–142.

[MPT98] Martín I., Pueyo X., Tost D.: A two-pass hardware-based method for hierarchical radiosity. In *Computer Graphics Forum* (1998), vol. 17, Wiley Online Library, pp. 159–164.

[MSW04] Mei C., Shi J., Wu F.: Rendering with spherical radiance transport maps. *Computer Graphics Forum* (Jan 2004).

[Nij03] Nijasure M.: *Interactive Global Illumination on the Graphics Processing Unit*. Master's thesis, University of Central Florida, Jan 2003.

[NN77] Nicodemus F., NBS.: *Geometrical Considerations and Nomenclature for Reflectance*. NBS Monograph. U.S. Government Printing Office, 1977.

[NPG05] Nijasure M., Pattanaik S., Goel V.: Real-Time Global Illumination on GPUs. *Journal of Graphics Tools* (Jan 2005).

[NRH03] Ng R., Ramamoorthi R., Hanrahan P.: All-frequency shadows using non-linear wavelet lighting approximation. *ACM Transactions on Graphics (TOG)* (Jan 2003).

[NRH04] Ng R., Ramamoorthi R., Hanrahan P.: Triple product wavelet integrals for all-frequency relighting. In *ACM SIGGRAPH 2004 Papers* (New York, NY, USA, 2004), SIGGRAPH '04, ACM, pp. 477–487.

[Pan11] Pantaleoni J.: VoxelPipe : A Programmable Pipeline for 3D Voxelization Blending-Based Rasterization. *HPG* (2011).

[PB96] Pattanaik S., Bouatouch K.: Haar wavelet: A solution to global illumination with general surface properties. *Photorealistic Rendering Techniques* (1996).

[PBMH02] Purcell T. J., Buck I., Mark W. R., Hanrahan P.: Ray tracing on programmable graphics hardware. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2002), SIGGRAPH '02, ACM, pp. 703–712.

[PDC*03] Purcell T. J., Donner C., Cammarano M., Jensen H. W., Hanrahan P.: Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (Aire-la-Ville, Switzerland, Switzerland, 2003), HWWS '03, Eurographics Association, pp. 41–50.

[Pel99] Pellegrini M.: Rendering equation revisited: how to avoid explicit visibility computations. In *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms* (Philadelphia, PA, USA, 1999), SODA '99, Society for Industrial and Applied Mathematics, pp. 725–733.

[PLPB07] Pan M., Liu X., Peng Q., Bao H.: Precomputed radiance transfer field for rendering interreflections in dynamic scenes. *Computer Graphics Forum* (Jan 2007).

[PP98] Peter I., Pietrek G.: Importance driven construction of photon maps. In *Rendering Techniques '98*, Drettakis G., Max N., (Eds.), Eurographics. Springer Vienna, 1998, pp. 269–280.

[PSV90] Puech C., Sillion F., Vedel C.: Improving interaction with radiosity-based lighting simulation programs. *Proceedings of the 1990 symposium on Interactive 3D graphics* (Jan 1990).

[PTVF07] Press W. H., Teukolsky S. A., Vetterling W. T., Flannery B. P.: *Numerical Recipes 3rd Edition: The Art of Scientific Computing*, 3 ed. Cambridge University Press, New York, NY, USA, 2007.

[Ram02] Ramamoorthi R.: *A Signal-Processing Framework for Forward and Inverse Rendering*. PhD thesis, Stanford, CA, USA, Jan 2002.

[Rau12] Rauwendaal R.: *Hybrid Computational Voxelization Using the Graphics Pipeline*. Master's thesis, Oregon State University, Nov 2012.

[RB13] Rauwendaal R., Bailey M.: Hybrid computational voxelization using the graphics pipeline. *Journal of Computer Graphics Techniques (JCGT) 2*, 1 (March 2013), 15–37.

[RDGK12] Ritschel T., Dachsbacher C., Grosch T., Kautz J.: The state of the art in interactive global illumination. *Computer Graphics Forum 31*, 1 (Feb 2012), 160–188.

[RGKM07] Ritschel T., Grosch T., Kautz J., Müller S.: Interactive illumination with coherent shadow maps. *Proceedings of Eurographics Symposium on Rendering 2007* (2007).

[RGKS08] Ritschel T., Grosch T., Kautz J., Seidel H.-P.: Interactive global illumination based on coherent surface shadow maps. *Proceedings of graphics interface 2008* (Jan 2008).

[RH01] Ramamoorthi R., Hanrahan P.: An efficient representation for irradiance environment maps. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2001), SIGGRAPH '01, ACM, pp. 497–500.

[RH02] Ramamoorthi R., Hanrahan P.: Frequency space environment map rendering. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2002), SIGGRAPH '02, ACM, pp. 517–526.

[RSC87] Reeves W. T., Salesin D. H., Cook R. L.: Rendering antialiased shadows with depth maps. In *ACM SIGGRAPH Computer Graphics* (1987), vol. 21, ACM, pp. 283–291.

[RWS*06] Ren Z., Wang R., Snyder J., Zhou K., Liu X., Sun B.: Real-time soft shadows in dynamic scenes using spherical harmonic exponentiation. *Proceedings of ACM SIGGRAPH 2006* (Jan 2006).

[SA13] Segal M., Akeley K.: Opengl 4.3 core profile specification, 2013.

[SAG94] Smits B., Arvo J., Greenberg D.: A clustering algorithm for radiosity in complex environments. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1994), SIGGRAPH '94, ACM, pp. 435–442.

[SAS92] Smits B. E., Arvo J. R., Salesin D. H.: An importance-driven radiosity algorithm. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1992), SIGGRAPH '92, ACM, pp. 273–282.

[SAWG91] Sillion F. X., Arvo J. R., Westin S. H., Greenberg D. P.: A global illumination solution for general reflectance distributions. In *Proceedings of the 18th annual conference on Computer graphics and*

interactive techniques (New York, NY, USA, 1991), SIGGRAPH '91, ACM, pp. 187–196.

[SB97] Stürzlinger W., Bastos R.: Interactive rendering of globally illuminated glossy scenes. *Eurographics Rendering Workshop 1997* (Jan 1997).

[Sch12] Schwarz M.: Practical binary surface and solid voxelization with Direct3D 11. In *GPU Pro 3: Advanced Rendering Techniques*, Engel W., (Ed.). A K Peters/CRC Press, Boca Raton, FL, USA, 2012, pp. 337–352.

[SDS95] Sillion F., Drettakis G., Soler C.: A clustering algorithm for radiance calculation in general environments. In *Rendering Techniques '95*, Hanrahan P., Purgathofer W., (Eds.), Eurographics. Springer Vienna, 1995, pp. 196–205.

[SEA08] Sintorn E., Eisemann E., Assarsson U.: Sample based visibility for soft shadows using alias-free shadow maps. *Computer Graphics Forum (Proceedings of the Eurographics Symposium on Rendering 2008) 27*, 4 (2008), 1285–1292.

[SGCH94] Schroder C., Gortler S., Cohen M., Hanrahan P.: Wavelet methods for radiance calculations. *Proceedings of Fifth Eurographics Workshop on Rendering* (Jan 1994).

[Sha97] Shaw E.: Hierarchical radiosity for dynamic environments. *Computer Graphics Forum 16*, 2 (1997), 107–118.

[SHHS03] Sloan P.-P., Hall J., Hart J., Snyder J.: Clustered principal components for precomputed radiance transfer. In *ACM SIGGRAPH 2003 Papers* (New York, NY, USA, 2003), SIGGRAPH '03, ACM, pp. 382–391.

[SHR*09] Soler C., Hoel O., Rochet F., Jay F., Holzschuch N.: *Hierarchical Screen Space Indirect Illumination For Video Games*. Rapport de recherche RR-7162, INRIA, Dec. 2009.

[SHS98] Slusallek P., Heidrich W., Seidel H.-P.: Radiance maps: an image-based approach to global illumination. In *ACM SIGGRAPH 98*

Conference abstracts and applications (New York, NY, USA, 1998), SIGGRAPH '98, ACM, pp. 264–.

[Sil95] Sillion F. X.: A Unified Hierarchical Algorithm for Global Illumination with Scattering Volumes and Object Clusters. *IEEE Transactions on Visualization and Computer Graphics 1*, 3 (Sept. 1995), 240–254.

[SKS02] Sloan P.-P., Kautz J., Snyder J.: Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2002), SIGGRAPH '02, ACM, pp. 527–536.

[Slo08] Sloan P.-P.: Stupid Spherical Harmonics (SH) Tricks. Game Developers Conference, 2008.

[SLS05] Sloan P.-P., Luna B., Snyder J.: Local, deformable precomputed radiance transfer. In *ACM SIGGRAPH 2005 Papers* (New York, NY, USA, 2005), SIGGRAPH '05, ACM, pp. 1216–1224.

[SS95] Schröder P., Sweldens W.: Spherical wavelets: efficiently representing functions on the sphere. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1995), SIGGRAPH '95, ACM, pp. 161–172.

[SS00] Simmons M., Séquin C.: Tapestry: A dynamic mesh-based display representation for interactive rendering. *Proceedings of the Eurographics Workshop on Rendering* (Jan 2000).

[SS10] Schwarz M., Seidel H.-P.: Fast parallel surface and solid voxelization on GPUs. *ACM Transactions on Graphics 29*, 6 (Proceedings of SIGGRAPH Asia 2010) (Dec. 2010), 179:1–179:9.

[SSG*00] Stamminger M., Scheel A., Granier X., Perez-Cazorla F., Drettakis G., Sillion F. X.: Efficient glossy global illumination with interactive viewing. *Computer Graphics Forum 19*, 1 (2000), 13–25.

[ST07] Saleh B., Teich M.: *Fundamentals of Photonics*. Wiley Series in Pure and Applied Optics. Wiley, 2007.

[SW00] Suykens F., Willems Y. D.: Density control for photon maps. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000* (London, UK, UK, 2000), Springer-Verlag, pp. 23–34.

[SZS*08] Sun X., Zhou K., Stollnitz E., Shi J., Guo B.: Interactive relighting of dynamic refractive objects. *SIGGRAPH '08: SIGGRAPH 2008 papers* (Aug 2008).

[Tin03] Tinkham M.: *Group Theory and Quantum Mechanics*. Dover Publications, 12 2003.

[TMS04a] Tawara T., Myszkowski K., Seidel H.-P.: Efficient rendering of strong secondary lighting in photon mapping algorithm. In *Theory and Practice of Computer Graphics, 2004. Proceedings* (2004), pp. 174–178.

[TMS04b] Tawara T., Myszkowski K., Seidel H.-P.: Exploiting temporal coherence in final gathering for dynamic scenes. *Proceedings of the Computer Graphics International* (Jan 2004).

[TPWG02] Tole P., Pellacini F., Walter B., Greenberg D.: Interactive global illumination in dynamic scenes. *ACM Transactions on Graphics* (Jan 2002).

[Vea98] Veach E.: *Robust monte carlo methods for light transport simulation*. PhD thesis, Stanford, CA, USA, 1998.

[WABG06] Walter B., Arbree A., Bala K., Greenberg D.: Multidimensional lightcuts. *SIGGRAPH '06: SIGGRAPH 2006 Papers* (Jul 2006).

[War94] Ward G. J.: The radiance lighting simulation and rendering system. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1994), SIGGRAPH '94, ACM, pp. 459–472.

[WAT92] Westin S. H., Arvo J. R., Torrance K. E.: Predicting reflectance functions from complex surfaces. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1992), SIGGRAPH '92, ACM, pp. 255–264.

[WC92] Wyant J., Creath K.: Basic wavefront aberration theory for optical metrology. *Applied Optics and Optical Engineering* (Jan 1992).

[WDG02] Walter B., Drettakis G., Greenberg D.: Enhancing and optimizing the render cache. *Proceedings of the 13th Eurographics workshop on Rendering* (Jan 2002).

[WDP99] Walter B., Drettakis G., Parker S.: Interactive rendering using the render cache. *Rendering Techniques* (Jan 1999).

[WFA*05] Walter B., Fernandez S., Arbree A., Bala K., Donikian M., Greenberg D.: Lightcuts: a scalable approach to illumination. *SIGGRAPH '05: SIGGRAPH 2005 Papers* (Jul 2005).

[WH80] Whitted T., Holmdel N.: An improved illumination model for shaded display. *Communications 23*, 6 (June 1980), 343–349.

[WH92] Ward G., Heckbert P.: Irradiance gradients. *Third Eurographics Workshop on Rendering* (Jan 1992).

[Wik13] Wikipedia: Directx, 2013. [Online; accessed 28-June-2013].

[WRC88] Ward G. J., Rubinstein F. M., Clear R. D.: A ray tracing solution for diffuse interreflection. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1988), SIGGRAPH '88, ACM, pp. 85–92.

[Yeu13] Yeung S.: Implementing voxel cone tracing. http://simonstechblog.blogspot.com/2013/01/implementing-voxel-cone-tracing.html, 02 2013.

[YIDN07] Yue Y., Iwasaki K., Dobashi Y., Nishita T.: Global illumination for interactive lighting design using light path pre-computation and hierarchical histogram estimation. *Computer Graphics and Applications* (Jan 2007).

[ZCEP07] Zhang L., Chen W., Ebert D. S., Peng Q.: Conservative voxelization. *Visual Computer 23*, 9 (Aug. 2007), 783–792.

[ZHL*05] Zhou K., Hu Y., Lin S., Guo B., Shum H.-Y.: Precomputed shadow fields for dynamic scenes. In *ACM SIGGRAPH 2005 Papers* (New York, NY, USA, 2005), SIGGRAPH '05, ACM, pp. 1196–1201.

[ZTTS06] Ziegler G., Tevs A., Theobalt C., Seidel H.-P.: On-the-fly point clouds through histogram pyramids. In *11th International Fall Workshop on Vision, Modeling and Visualization 2006 (VMV2006)* (Aachen, Germany, 2006), Kobbelt L., Kuhlen T., Aach T., Westermann R., (Eds.), European Association for Computer Graphics (Eurographics), Aka, pp. 137–144.

APPENDICES

Appendix A: Additional Images

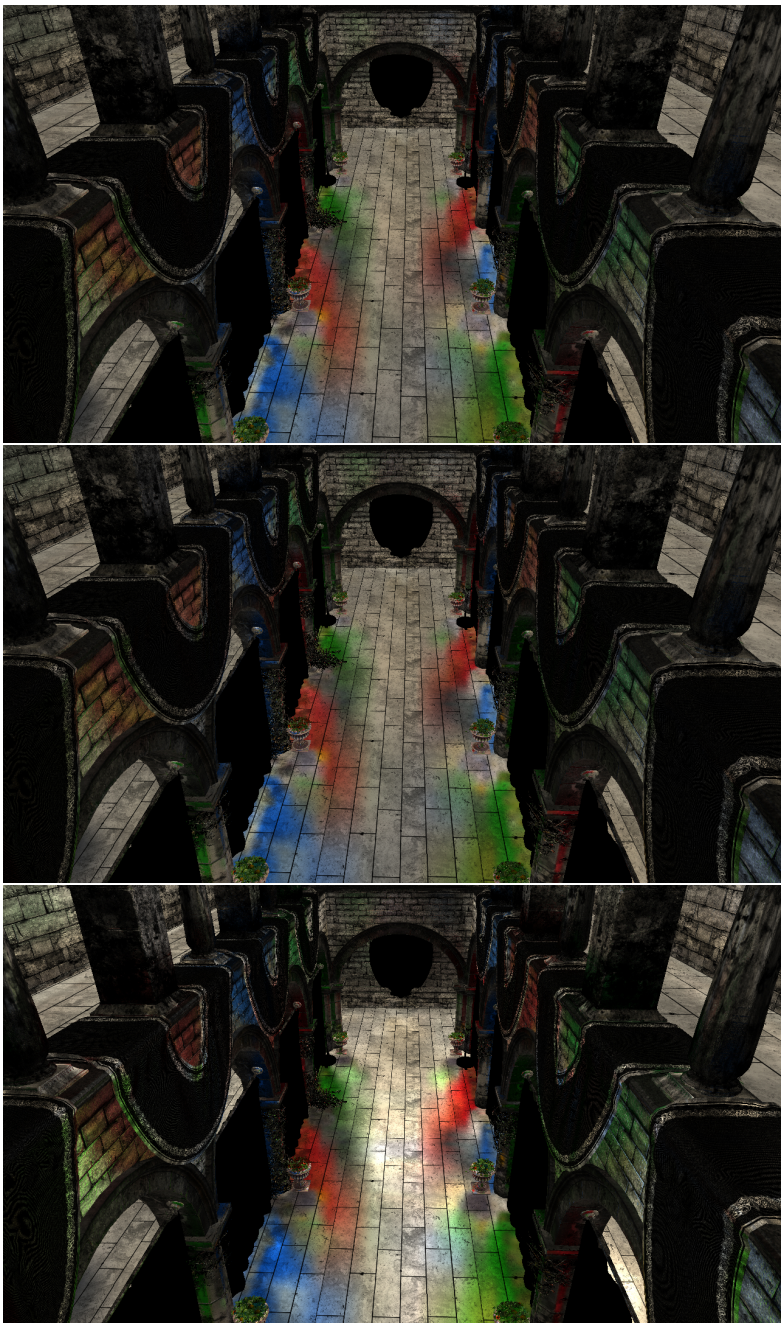


Figure A.1: Comparison of the quality of specular cone tracing for the three voxel formats for the Sponza scene at 256^3, from top to bottom istropic, anisotropic, and spherical harmonic voxel formats respectively.

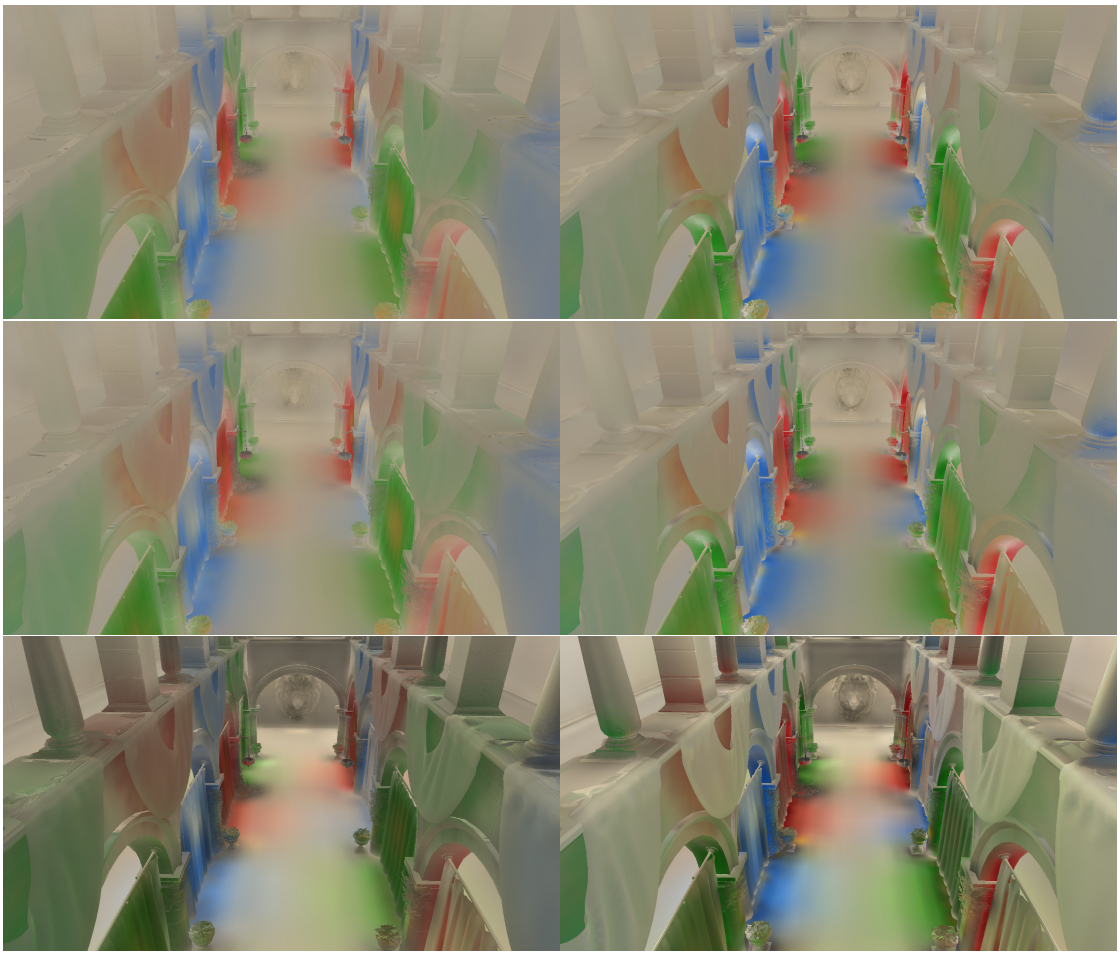


Figure A.2: Comparison of the quality of diffuse cone tracing for the three voxel formats for the Sponza scene at 256^3, from top to bottom istropic, anisotropic, and spherical harmonic voxel formats respectively. Images with diffuse cones traced using the generic method are on the left, while images with diffuse cones traced using the specialized method are on the right.

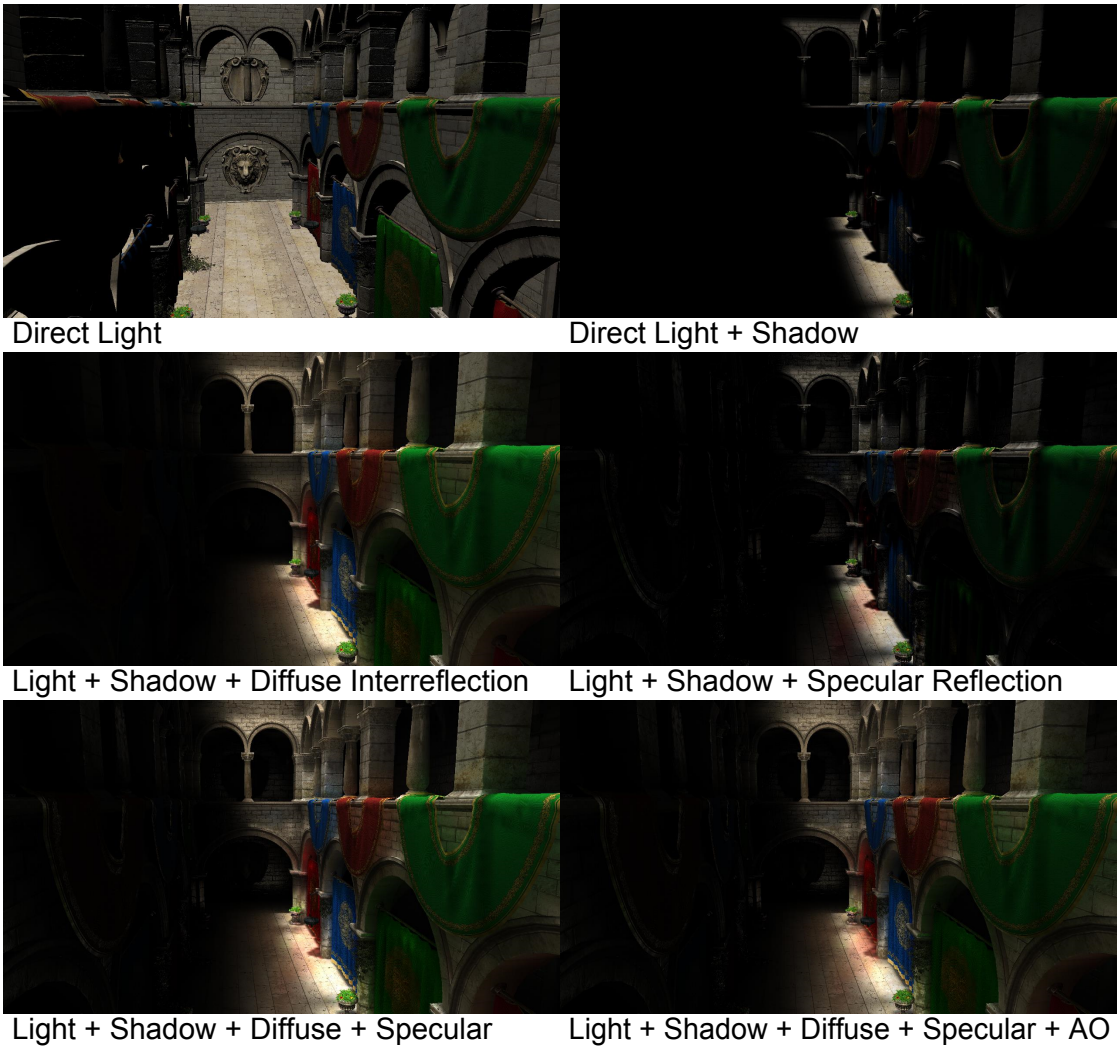


Figure A.3: Collage of Sponza Atrium images illustrating the incremental additional of direct and indirect illumination effects, and the improved realism of the scene.

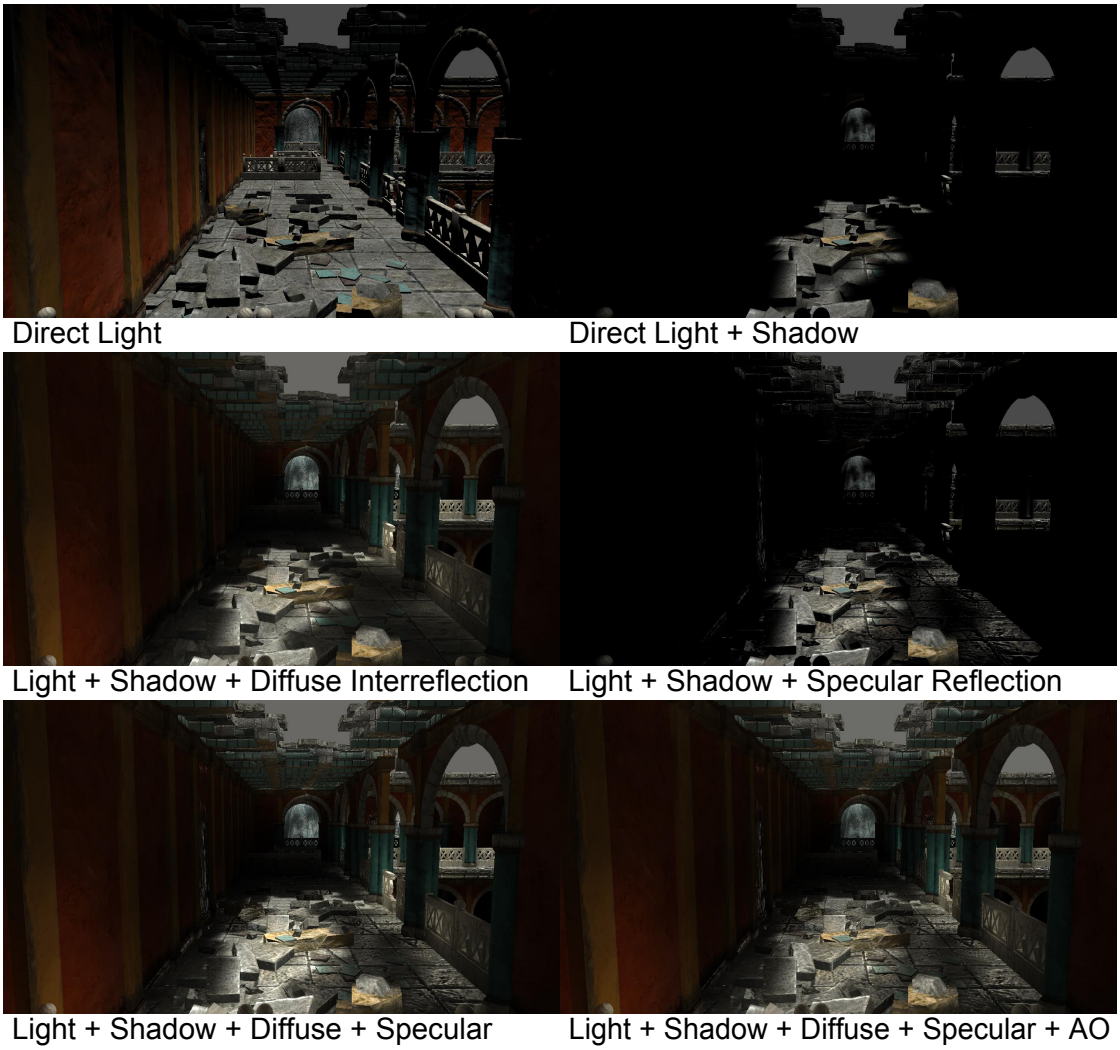


Figure A.4: Collage of images of the Ruins scene illustrating the incremental additional of direct and indirect illumination effects, and the improved realism of the scene.

Appendix B: Shader Code

```
vec4 unpackRGBCount(uint val)
{
  vec4 rgba;
  //mask of the approach quadrant of the uint then shift it to the end
  rgba.r = float((val & 0x000000FF)       );  //red
  rgba.g = float((val & 0x0000FF00) >>  8u);  //green
  rgba.b = float((val & 0x00FF0000) >> 16u);  //blue
  rgba.a = float((val & 0xFF000000) >> 24u);  //count
  return rgba;
}
uint packRGBCount(vec4 val)
{
  //Mask of the last 8 bits then shift them to the appropriate quadrant
  uint r = (uint(val.r) & 0x000000FF);
  uint g = (uint(val.g) & 0x000000FF) <<  8u;
  uint b = (uint(val.b) & 0x000000FF) << 16u;
  uint a = (uint(val.a) & 0x000000FF) << 24u;

  //OR the values together
  return (r | g | b | a);
}
void imageAtomicAverageRGBA8Custom(layout(r32ui) coherent volatile uimage3D ↩
    voxels, ivec3 coord, vec3 nextVec3)
{
  uint nextUint = packRGBCount(vec4(nextVec3*255.0f,1));
  uint prevUint;
  uint currUint = 0;//packRGBCount(vec4(0,0,0,999));
  vec4 currVec4;

  vec3  average;
  uint  count;

  //Loop as long as destination value gets changed by other threads
  //compares currUint to nextUint
  while( ( prevUint = imageAtomicCompSwap(voxels, coord, currUint, nextUint) ) ↩
      != currUint )
  {
    currUint = prevUint;          //store packed rgb average and count
    currVec4 = unpackRGBCount(currUint);  //unpack stored rgb average and count

    average =      currVec4.rgb / 255.0f; //extract rgb average
    count   = uint(currVec4.a);       //extract count

    //Compute the running average
    average = (average*count + nextVec3) / float(count+1);

    //Pack new average and incremented count back into a uint
    nextUint = packRGBCount(vec4(average*255.0f, (count+1)));
  }
}
```

Figure B.1: Implementation of a moving average using `imageAtomicCompSwap`.

```
vec4 anisoVoxelFetch(vec3 pos, float lod, vec4 n)
{
  vec4 sampleX = (n.x < 0.0f) ? textureLod(VoxelsNegX, pos, lod) : textureLod(↩
      VoxelsPosX, pos, lod);
  vec4 sampleY = (n.y < 0.0f) ? textureLod(VoxelsNegY, pos, lod) : textureLod(↩
      VoxelsPosY, pos, lod);
  vec4 sampleZ = (n.z < 0.0f) ? textureLod(VoxelsNegZ, pos, lod) : textureLod(↩
      VoxelsPosZ, pos, lod);
  vec3 nSquared = n.xyz * n.xyz;
  vec4 filtered = nSquared.x * sampleX + nSquared.y * sampleY + nSquared.z * ↩
      sampleZ;
  return filtered;
}
```

Figure B.2: Anisotropic voxel sampling using the "ambient cube" method described in [MMG06].

```
#define shCoeff vec4(0.2820947918, -0.488602512, 0.488602512, -0.488602512)

vec4 shEvaluate(vec3 dir)
{
  return shCoeff * vec4(1.0f, dir.yzx);
}

vec4 shVoxelFetch(in vec3 pos, in float lod, in vec4 dir)
{
  vec4 rSH, gSH, bSH;
  rSH[0] = textureLod(r0Tex, pos, lod).x;
  rSH[1] = textureLod(r1Tex, pos, lod).x;
  rSH[2] = textureLod(r2Tex, pos, lod).x;
  rSH[3] = textureLod(r3Tex, pos, lod).x;
  gSH[0] = textureLod(g0Tex, pos, lod).x;
  gSH[1] = textureLod(g1Tex, pos, lod).x;
  gSH[2] = textureLod(g2Tex, pos, lod).x;
  gSH[3] = textureLod(g3Tex, pos, lod).x;
  bSH[0] = textureLod(b0Tex, pos, lod).x;
  bSH[1] = textureLod(b1Tex, pos, lod).x;
  bSH[2] = textureLod(b2Tex, pos, lod).x;
  bSH[3] = textureLod(b3Tex, pos, lod).x;

  vec4 shColor;
  vec4 shCoeff = shEvaluate(-normalize(dir.xyz));
  shColor.r = clamp(dot(rSH, shCoeff), 0.0, 1.0);
  shColor.g = clamp(dot(gSH, shCoeff), 0.0, 1.0);
  shColor.b = clamp(dot(bSH, shCoeff), 0.0, 1.0);
  shColor.a = textureLod(aTex, pos, lod).a;

  return shColor;
}
```

Figure B.3: Spherical harmonic voxel sampling method described in Section 5.2.3.

```
vec4 coneTrace(vec3 origin, vec3 nS, vec3 dir, float coneRatio, float maxDist, ↩
    float aoDist, float voxSize, float volDim, float fMin, float sinHalfAngle)
{
  vec4  accum   = vec4(0);
  float opacity = 0.0f;

  float    NdotR = dot(nS,dir);
  float invNdotR = 1 / NdotR;

  float h  = fMin*voxSize;
  float d0 = h * invNdotR;
  float r0 = d0*sinHalfAngle;

  float startDist = d0-r0;
  for(float dist = startDist; dist <= maxDist && accum.w < 1.0;)
  {
    float sampleRadius   = coneRatio * dist;
    float sampleDiameter = max(2.0f*sampleRadius, voxSize);
    float sampleLOD      = log2(sampleDiameter * volDim);
    vec3  samplePos      = origin + dir * (dist + sampleRadius);
    vec4  sampleValue    = voxelFetch(samplePos, sampleLOD, vec4(dir,1));
    float sampleWeight   = 1.0f - accum.w;
    accum += sampleValue * sampleWeight;
    dist  += sampleDiameter;
    opacity = (dist < aoDist) ? accum.w : opacity;
  }
  return vec4(accum.xyz,1-opacity);
}
```

Figure B.4: Shader code for a generic voxel cone tracing routine. Note, the `voxelFetch` function must be implemented appropriately for the selected voxel storage format.

```
vec4 diffuseConeTrace60(vec3 origin, vec3 nS, vec3 dir, float coneRatio, float ↩
    maxDist, float aoDist, float voxSize)
{
  vec4  accum   = vec4(0);
  float opacity = 0.0f;
  vec3  samplePos;
  vec4  sampleValue;
  float sampleWeight;
  float sampleRadius = 0.5*voxSize;
  float sampleLOD = 0;
  for(float dist = voxSize; dist <= maxDist && accum.w < 1.0;)
  {
    samplePos    = origin + dir * dist;
    sampleValue  = voxelFetch(samplePos, sampleLOD, vec4(dir,1));
    sampleWeight = 1.0f - accum.w;
    accum.xyz += sampleValue.xyz * sampleWeight;
    accum.w   += sampleValue.w   * sampleWeight;
    sampleLOD    += 1.0f;
    sampleRadius *= 2.0f;
    dist *= 2.0f;
    opacity = (dist < aoDist) ? accum.w : opacity;
  }
  return vec4(accum.xyz,1-opacity);
}
```

Figure B.5: Shader code for a specialized 60° diffuse cone tracing routine. Note, the `voxelFetch` function must be implemented appropriately for the selected voxel storage format.