Vector Occluders: An Empirical Approximation for Rendering Global Illumination Effects in Real-Time

by

William Sherif

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Ontario Institute of Technology

Abstract

Vector Occluders: An Empirical Approximation for Rendering Global Illumination Effects in Real-Time

William Sherif

Master of Science

Graduate Department of Computer Science

University of Ontario Institute of Technology

2013

Precomputation has been previously used as a means to get global illumination effects in real-time on consumer hardware of the day. Our work uses Sloan's 2002 PRT method as a starting point, and builds on it with two new ideas.

We first explore an alternative representation for PRT data. "Cpherical harmonics" (CH) are introduced as an alternative to spherical harmonics, by substituting the Chebyshev polynomial in the place of the Legendre polynomial as the orthogonal polynomial in the spherical harmonics definition. We show that CH can be used instead of SH for PRT with near-equivalent performance.

"Vector occluders" (VO) are introduced as a novel, precomputed, real-time, empirical technique for adding global illumination effects including shadows, caustics and interreflections to a locally illuminated scene on static geometry. VO encodes PRT data as simple vectors instead of using SH. VO can handle point lights, whereas a standard SH implementation cannot.

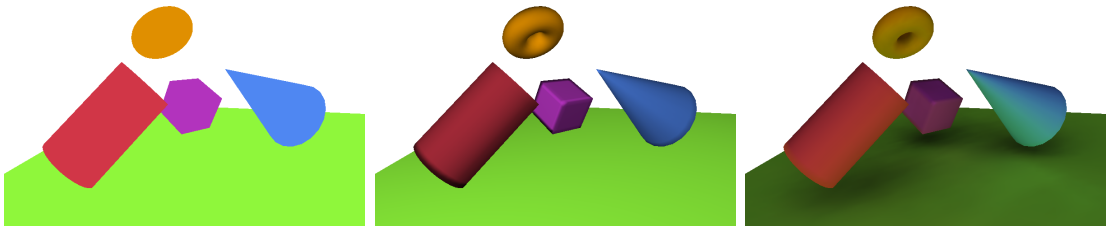
Contents

Chapter 1

Introduction

1.1 Problem: real-time global illumination

Global illumination (GI) adds spatial and depth cues to computer graphics renders that help the viewer make sense of the scene. Consider the scene shown in figure 1.1.



(a) Unshaded shapes (b) Local illumination only (c) Global illumination, with shadows and interreflections

Figure 1.1: More spatial relationship "information" is added to the renders as we go from left to right

The leftmost image in figure 1.1a does not give much information as to the shape of the objects. They all appear completely flat. We are not sure *what* the orange ellipse-shaped object actually is. Adding basic *local illumination* shading in figure 1.1b gives us additional information about the shape of the objects. In figure 1.1b, we learn

that the ambiguous orange ellipse from 1.1a is actually a torus. Finally, adding *global illumination* in figure 1.1c gives us additional cues about the proximity of the objects by adding shadows and interreflections. For example, we see how high off the ground plane the shapes are by looking at the shadows in figure 1.1c (you cannot tell this in figure 1.1b).

The main difference between local illumination and global illumination is that GI *considers other scene geometry as each scene point is rendered.* Computing GI (as compared to computing local illumination only) is very expensive, because of all of the extra geometry considerations. Computing GI in *real-time* is still an area of active research.

In this work, we assume a target frame rate of 60 frames per second (fps) for a real-time application. The work by [36] says that for a computer application to remain "interactive," response times must be under 100 ms. For reasons described in [59], game developers often target a frame rate of 60 fps.

1.2 Significance

The primary applications for real-time global illumination are computer games, realistic simulation software, and real-time lighting design software packages.

Games

Games benefit from real-time GI lighting because GI effects are aesthetically pleasing, and these effects can help make the game more interesting or immersive to the player. Military simulation titles such as the *America's Army* try to place the player in as life-like a scenario as possible. It is reasonable to use realistic lighting effects as an integrated part of the gameplay, such as to add challenge by using realistic camoflague and shaded areas.

For games especially, we need our GI algorithm to be very fast and lightweight. Because we assume a target frame rate of 60 fps, we only have *a maximum* of 16 ms (ideally we should use much less) to complete all of our lighting computations before the next presentation interval. Some current techniques "cheat" on this and allow global illumination effects to lag by a couple of frames to allow time for a GI solution to complete. This is argued to be imperceptible [33] except under "flashes" of light (then the lag of the GI effects being displayed is quite obvious).

Lighting design

One of the tasks of an architect is to use both natural and artificial light effectively in his building design. Architects like to see how the interior of their buildings will look due to natural light at different times of the day, and under different weather conditions [27].

With fast simulations of global illumination and light transport, we can do this interactively for the architect, so he doesn't have to wait minutes or hours to view a render every time he moves the light source. Game level designers and movie set producers also benefit from seeing how lighting will affect their designs in real-time.

1.3 Contributions

We present two contributions in this thesis. The first contribution is the introduction of the ***cpherical harmonics*** (CH), where we substitute the Chebyshev polynomial in the place of the Legendre polynomial in the spherical harmonics definition. The motivation for this was to see whether cpherical harmonics would give significantly different results than spherical harmonics in a global illumination render. Our results showed that cpherical harmonics were roughly equivalent in performance and image quality. Cpherical harmonics are fully described in section 6.5.

The second (main) contribution of this thesis is ***vector occluders*** (VO): an empirical technique for real-time shadows, caustics and interreflections, fully described in chapter 7.

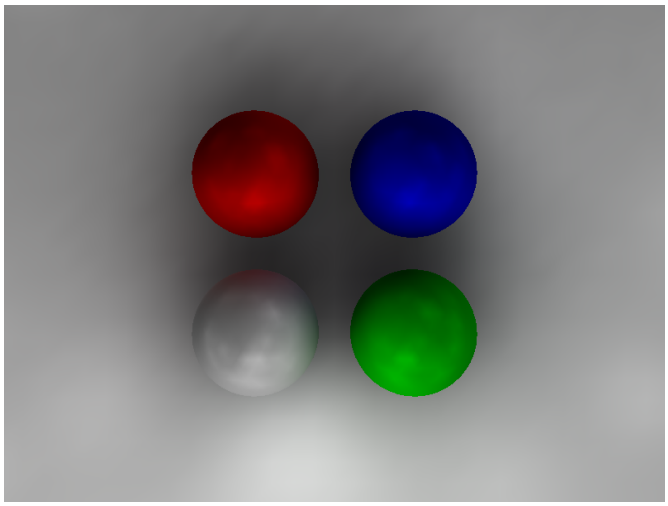


Figure 1.2: A vector occluders sample render

VO is a type of precomputed radiance transfer (PRT). VO simply uses vectors (instead of spherical harmonics or wavelets as existing PRT techniques do) to store *information about remote surface occluders* as vector data local at each vertex.

VO's render quality (pictured in figure 1.2) is comparable to low band spherical harmonics, with soft shadows and diffuse color bleeding. VO also offers approximated glossy interreflections as well. Memory consumption for VO can be comparable to SH, but VO's memory consumption depends on the number of occluders in the scene, while SH's consumption depends on the number of bands used.

VO differs from SH in that it can easily handle point lights, where SH usually uses infinitely distant lights. In addition, because the components are vector based, special effects can be added to VO renders, simulating effects such as smoke passing overhead or underwater caustics on the ocean floor.

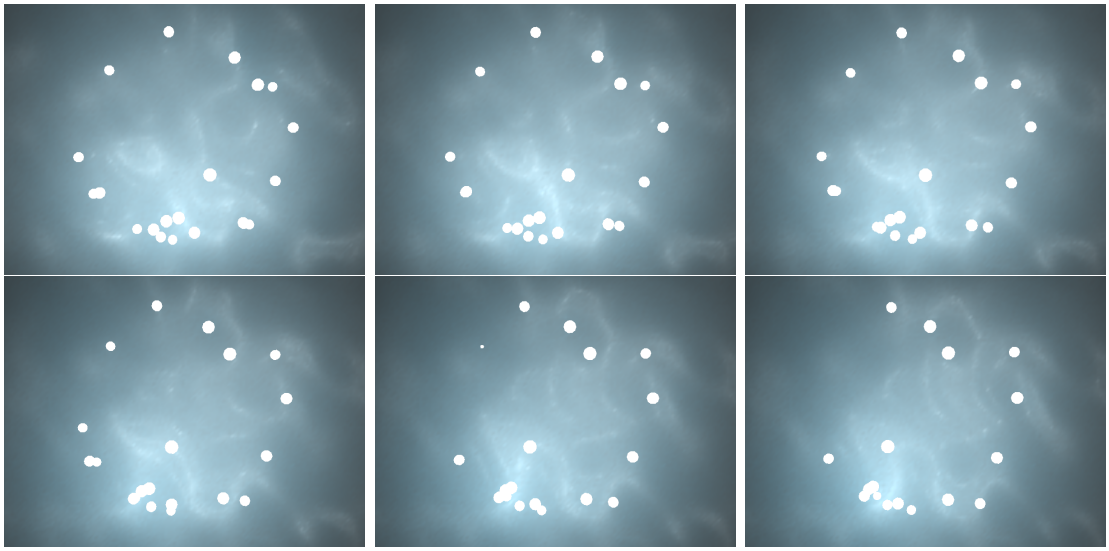


Figure 1.3: VO rendering a real-time simulated ocean floor caustic (by normal bending) with fully dynamic, positioned light sources. This 30k polygon scene ran at 560 fps with 20 light sources (some are under the ocean floor) and a 250 second precompute time.

The magnitude of these effects are easily tuneable, from a "realistic" default setting, to exaggerated effects.

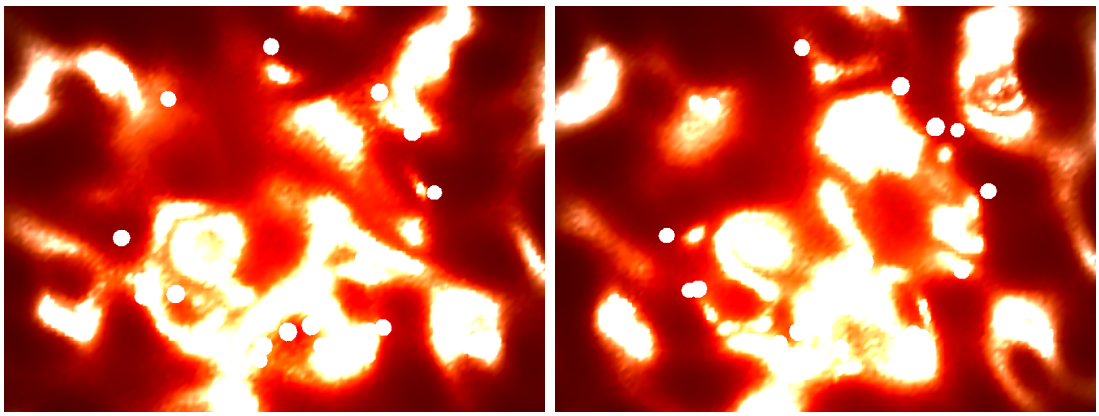


Figure 1.4: "Fire lake" simulation by bending the normals with a sinusoidal wavetrain and applying the VO caustic formula.

1.4 Roadmap

The problem space we are exploring in this thesis is real-time global illumination. We begin chapter 2 by describing shading and reflection and identifying what global illumination is exactly. In section 2.4 we give a reverse-chronological survey of some of the approaches used to display ***real-time approximations*** to global illumination.

After discussing the mathematical background of the global illumination problem in chapter 3, we discuss the details of two classic full solutions to global illumination in chapter 4: ray tracing and radiosity. In chapter 5 we discuss some of the more recent real-time "screen-space" GI methods. Following this we move onto precomputed radiance transfer in chapter 6, and discuss the spherical harmonic, cpherical harmonic, and wavelet approaches to real-time global illumination. We then present our main contribution, an empirical technique for rendering global illumination effects in real-time that we have called ***vector occluders*** in chapter 7.

A source package accompanies this thesis with an implementation of each technique described at `http://github.com/superwills/` [48].

1.5 Coordinate system conventions

1.5.1 Axes

The coordinate system used in mathematical descriptions for this thesis are consistent throughout this document. A right-handed coordinate system with directions as labelled in figure 1.5 is used.

(a) Right-handed Cartesian system (b) Spherical coordinates

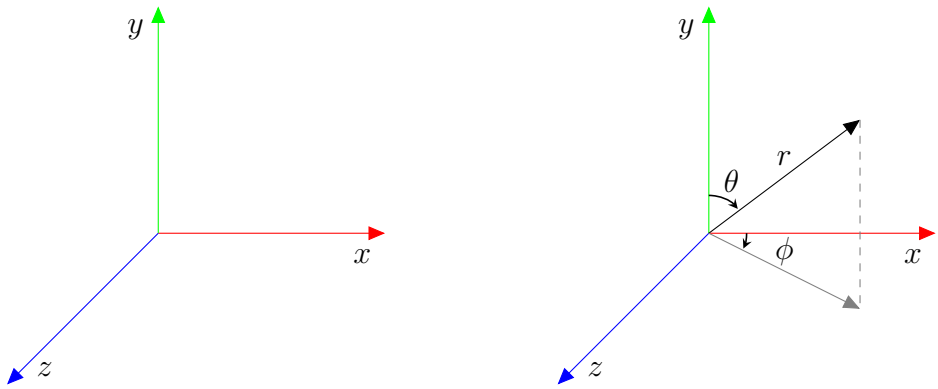


Figure 1.5: The coordinate system conventions used consistently throughout this thesis, with the exception of cubemap orientation.

When spherical coordinates are used, the *inclination angle* is called $\theta \in [0, \pi]$ and is measured from the zenith, or **y**-axis. The *azimuthal angle* is called $\phi \in [0, 2\pi]$ and is measured from the **x**-axis. Spherical coordinates are specified in the order (r, θ, ϕ).

Be aware that different literature uses different coordinate system conventions, especially with regard to spherical harmonics. The most common differences are to switch the roles of θ and ϕ in spherical coordinates, or to use the **z**-axis as the zenith instead of the y-axis in spherical coordinates, or to use a left-handed coordinate system.

1.5.2 Cube maps

Next we describe the default cubemap orientation. The convention described here is the orientation that Direct3D 11 expects, so unfortunately for our convention it is left-handed. We use left handed cubemaps in this text since [48] uses Direct3D 11. Color coordination as axes in 1.5. (eg PX is red, NX is cyan (inverse of red))

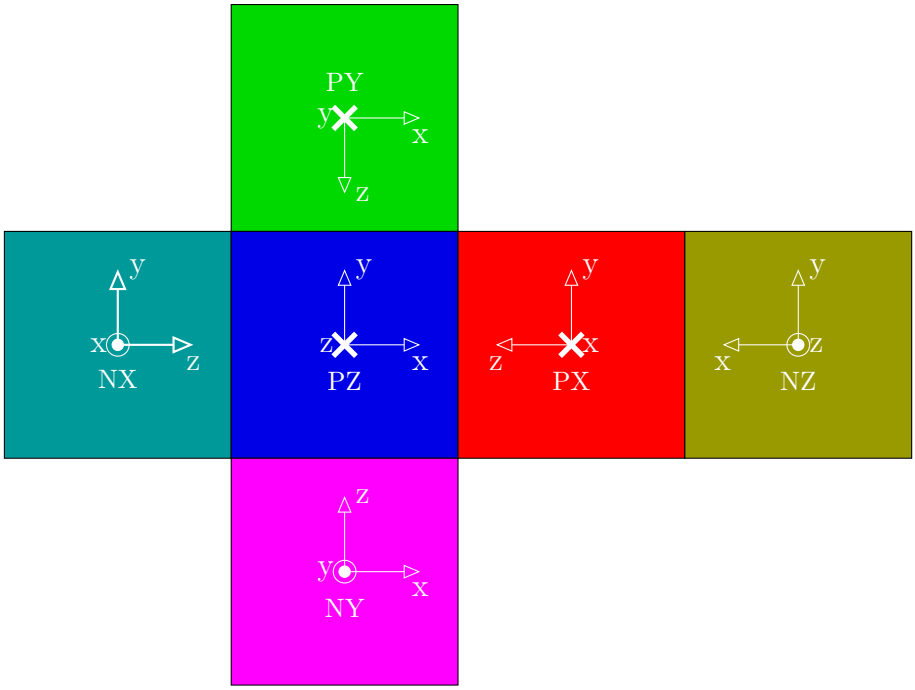


Figure 1.6: Direct3D 11 convention for cubemap orientation. This is the only left-handed system used in this thesis, every other axis is going to be right-handed.

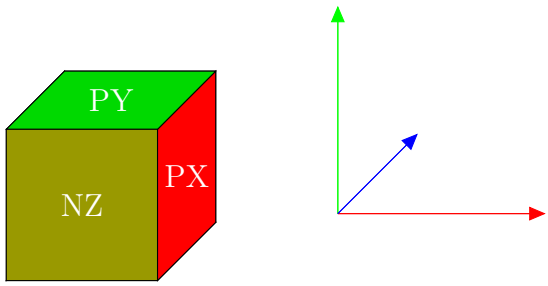


Figure 1.7: External view of the cube map for additional perspective

Chapter 2

Background

2.1 Shading

Methods of generating realistic-looking lighting have always been central to computer graphics. Generating a color at a point in the scene is called *shading*[1]. From physics we know how a surface appears depends primarily on incident light and material properties as well as the viewing angle.

2.1.1 Specularity and diffusivity

To find a surface's response to the light in a scene, we usually break it into two basic terms: *specular* reflection and *diffuse* reflection. Specular reflection is mirror-like reflection. The shiny side of aluminum foil is mostly specular. Diffuse means "to spread". Rough paper is mostly diffuse. Specular reflection occurs everywhere you can see the image of a light source reflected on a shiny surface.

Every material exhibits some specularity and some diffusivity. For example, the typical *eggshell*, *satin*, *high gloss* categories of wall paints adds more specularity to the paint as

[1] Hence the terms *vertex shader* and *pixel shader*, computer programs that determine the shaded color at a vertex and pixel respectively.

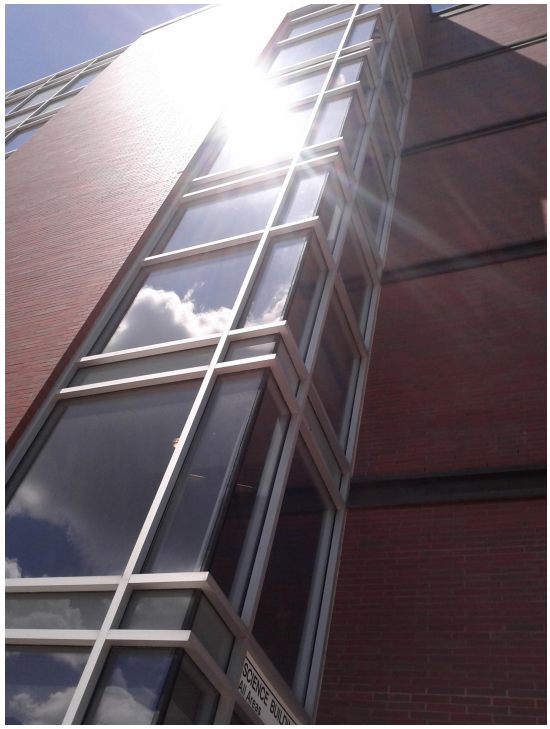
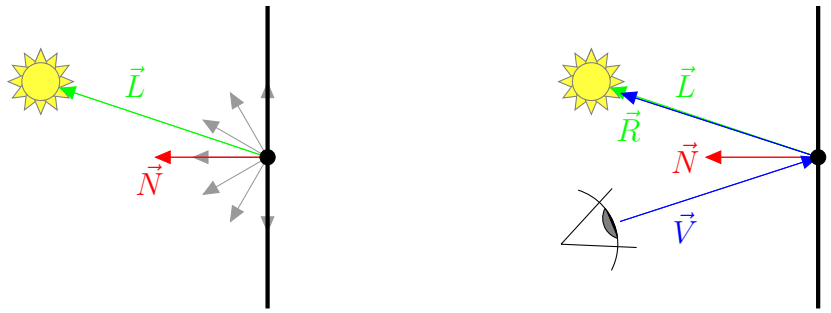


Figure 2.1: Diffuse and specular reflection from a real life scene. The bricks on the right side of the image are exhibiting purely diffuse reflection. The bricks in the center-left are showing a combination of diffuse and specular reflection. The windows are showing only specular reflection. You should note here that sky and clouds, although they are diffuse reflectors themselves, are actually "light sources".

we go from *eggshell* to *high gloss*.

A common model for diffuse reflection in computer graphics is *Lambertian reflectance*. A Lambertian surface, like eggshell paint, *looks the same color from every viewing angle*. The Lambertian model stipulates that the surface scatters light equally in all directions. The amount of light that a Lambertian surface scatters in every direction is proportional to the cosine of the surface normal's angle with a vector from the surface to the light source. This is shown in figure 2.2a.



(a) Diffuse, scatters all directions (b) Specular, mirror-like

Figure 2.2: Diffuse and specular and reflection. Diffuse reflection requires only the surface normal and the vector to the light source. Specular reflection also requires our view direction.

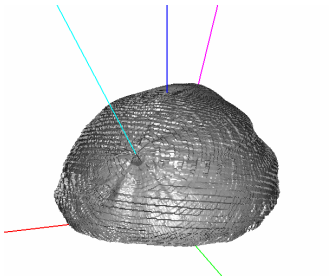
Specular reflection (figure 2.2b) is *view dependent*, because it requires another vector in addition to the surface normal and the vector to the light source: the vector from the *eye* to the surface point. We need to know how much the reflected view vector "lines up" with the vector that goes to the light source. In figure 2.2b, we simply reflect $\vec{V}$ about $\vec{N}$, call that $\vec{R}$, and dot $\vec{R}$ with $\vec{L}$. We raise $\vec{R} \cdot \vec{L}$ to some exponent depending on the material properties (shininess factor). This is Phong's model [40]. Phong's model for specular reflection is an empirical model and does not claim to be physically accurate [40]. The final "shading color" at a point in the scene is simply the sum of the specular and diffuse responses at that point.

2.1.2 Accuracy and other models

Bidirectional Reflectance Distribution Function

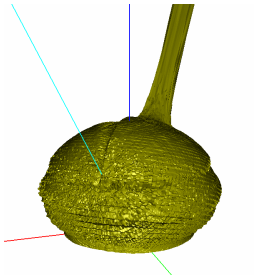
The "line" between specularity and diffusivity is actually blurred for real world materials. Materials are rarely ever purely specular or purely diffuse, but often a combination of both. This "semispecularity" is also called "gloss". In addition, *just how specular* a physical surface behaves is frequently dependent on the angle of incidence of

the light. If you look at a surface from a "grazing angle", the surface usually appears "more specular" than it does from an oblique angle. For example, a smooth table might start to show stronger mirror reflections when your viewing angle is nearly parallel to the surface of the table. This is all encoded for in something called a "bidirectional reflectance distribution function", or BRDF.



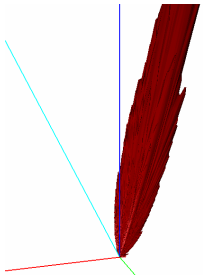
(a) BRDF of pure rubber. Highly diffuse.

(b) Observe lack of specularity



(c) Aluminum oxide: Semispecular. Note the specular lobe on the reflection angle.

(d) Specular highlights *in addition to* the diffuse response.



(e) Steel BRDF, almost purely specular

(f) No diffuse response, only specular.

Figure 2.3: BRDFs (left) and rendering of a statue (right), created in Disney's BRDF Explorer [6]. Incident angle of light for the BRDF visualizations is shown in cyan.

For each incident angle on the hemisphere, the BRDF has an entire hemispherical

response. This hemispherical response tells us the directions and power that we reflect light from the surface for a particular entry angle. Some BRDFs are shown in figure 2.3.

In the BRDF visualizations down the left side of figure 2.3, the incident angle of the light is shown in cyan. The solid meshes show the magnitude and directions that the incident light would be reflected in. Specular materials only send light along the mirror reflection angle. Diffuse materials send light *evenly* along every exit angle, regardless of the entry angle. As you can see, figure 2.3a is highly diffuse. Figure 2.3c is an example of a "semispecular" surface, with both a strong diffuse and strong specular component. Figure 2.3e is almost purely specular in its response.

2.2 Local illumination models

From the earliest computer generated images, academics have tried to find ways to illuminate objects both realistically and efficiently. Early renders by Gouraud [16] and Phong [40] used simple shading algorithms that lit each scene point by considering only the color, position and orientation that each scene point had with respect to the light source. These early models were called *local illumination models*, because the algorithm for lighting only considers data *local* to the point it is lighting. An example local illumination render is in figure 2.4.

However, there is more to consider. There should be ***shadows***: a light source *should not* illuminate a point if that light source is occluded. There should be ***interreflections***: to find the lighting at a point, we should have to find *how all other objects in the room illuminate that point.* Light does not just leave a light source and land on every surface, it reflects and refracts many times before being absorbed into a surface. When we light each scene element considering each of the light sources *and* how light bounces from *all other scene geometry*, then this is called "global illumination."

Figure 2.4: A tree lit by direct illumination only using the Phong local illumination model. Shadows and interreflections are not present here.

2.3 Shadows, caustics, and interreflections

If a scene is lighted with a local illumination model, using only vertex normals, the direction to the light source, and reflection vectors as shown in figure 2.4, we see that some very obvious phenomena are missing:

- Shadows
- Caustics
- Interreflections

These are the "global illumination effects" the techniques studied seek to bring to our renders. **Direct illumination** is the response of a surface to light directly incident on the surface. Figure 2.4 is *direct illumination* only. **Shadows** are where the light source should not be able to reach a surface point because of occluding scene geometry. Direct illumination should be *attenuated* there. **Interreflections**, also called "color bleeding", are where scene surfaces actually act as *secondary light sources*. That is, every surface in the scene that reflects light *also* must behave *as a light source*. **Caustics** are where one piece of geometry re-focusses light onto another piece of geometry, like a

magnifying glass can focus the sun onto a sidewalk (*caustic from refraction*), or a curved sheet of metal (*caustic from reflection*).

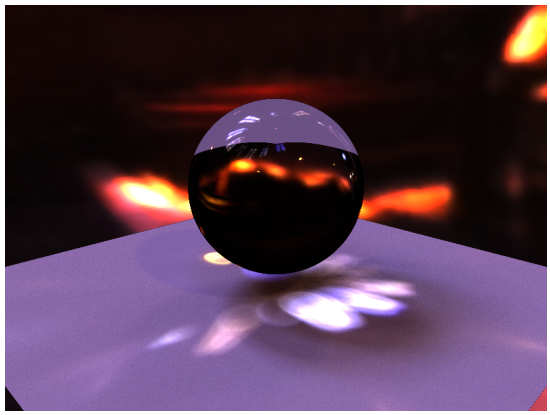


Figure 2.5: Ray traced scene showing some global illumination effects. The transparent ball has focussed the light sources around it, to form the bright *caustics* on the ground.

To re-iterate, shadows, caustics and interreflections (anything beyond *direct illumination*) have come to be termed "global illumination effects" in the literature. A "global illumination effect" is one that considers light's interactions with *other scene geometry* when lighting a point in the scene, not just the relationship of a point and the light source.

We will need to identify the different types of caustics and interreflections, so that we can model them in our lighting algorithms later. We consider three different types of interreflection:

1. **Diffuse from diffuse**: The diffuse response of a surface to *the brightness of another diffuse reflector around it.* Commonly called "color bleeding."

2. **Diffuse from specular** (or *caustic from reflection*): When a diffuse surface is illuminated by light that was specularly reflected from another surface. An example of this type of reflection is shown in figure 2.6.

3. **Specular from specular**: Light bouncing off two mirror surfaces before reaching the eye is specular from specular reflection. An example of this is if you see the

sun in your rearview mirror reflecting off a glass building.

Figure 2.6: Diffuse from specular (*caustic from reflection*) in real life. The bright patches on the sidewalk are a diffuse response to the specular reflection of the sun from the building's glass windows

How can we write an algorithm that expresses lighting at a point including interaction with all other scene geometry? Is there a mathematical equation for global illumination? Can global illumination be solved for every scene point in real-time? What has been done so far towards that end? These questions will be examined in this thesis. We begin in section 2.4 with an overview of what is current in real-time global illumination.

2.4 What's current in real-time global illumination

Real-time Ray tracing

Recently, corporations such as NVIDIA [9] and special interest groups such as [43] have put out real-time ray tracing demos that utilize nearly 100% of the GPU and can handle ray-traced global illumination effects with limited amounts of scene geometry.

Figure 2.7: Rigid Gems 2.0 [43]

Real-time Radiosity

Through 2008-2010, Geomerics **Enlighten** software emerged and achieved "real-time" radiosity in shipped, AAA game titles such as Battlefield 3. In fact, the radiosity computation is imperceptibly lagged by about 60 ms depending on machine power [33].

Figure 2.8: Geomerics' Enlighten real-time radiosity official sample scene

Deferred shading and screen-space methods

Around 2008 there was a surge in the development of *screen-space deferred shading* techniques. Deferred shading was first introduced in 1988 by Deering [12]. Deferred shading gathers *information* from the first rendering pass, and "defers" actual shading computation until a second pass pixel shader.

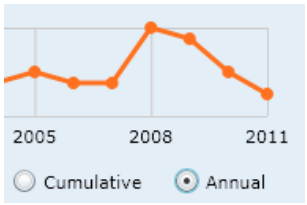


Figure 2.9: Spike in citations (8 total) in 2008 for [12]

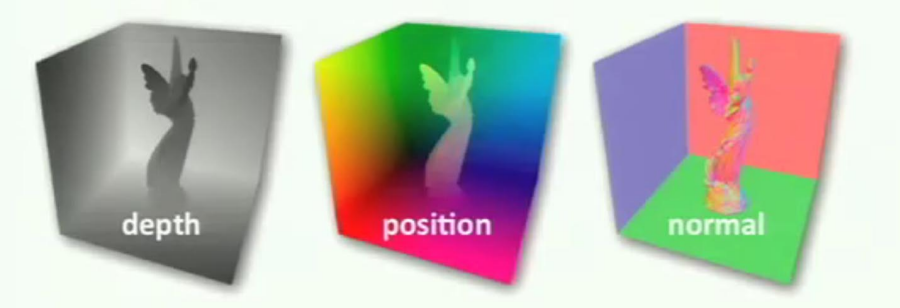


Figure 2.10: Deferred shading data gathered on first pass in "geometry buffer" (G-buffer). Here, depth, position, and normals are gathered into the G-buffer on the first pass. Image from [10].

As a refresher, lighting is usually computed in either *world space* or *eye space*. The idea of computing lighting in screen space was the new idea presented by Deering [12]. Using information from neighbouring pixels in a deferred shading technique, we can compute *screen-space ambient occlusion* (SSAO [3], shadows only) and *screen space directional occlusion* (SSDO [44], shadows and interreflections) in real-time on modern graphics cards.

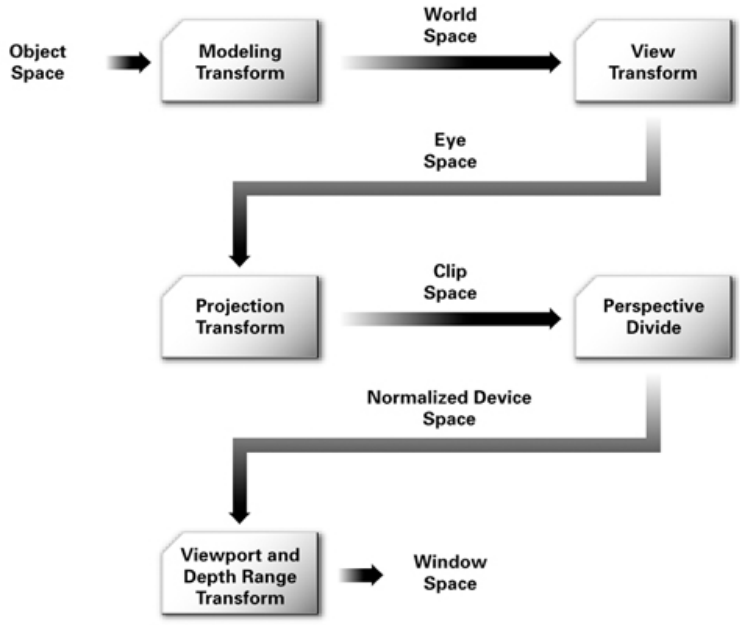


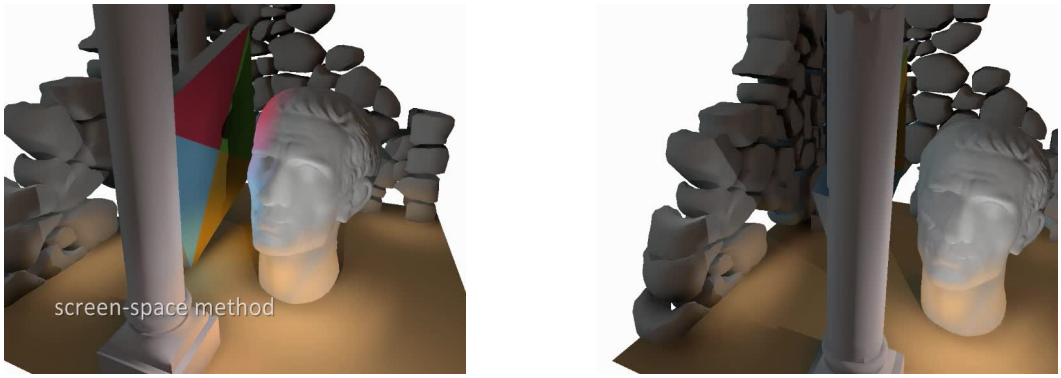
Figure 2.11: Different coordinate spaces, as illustrated in [14]. Window space is synonymous with screen-space.

One of the main takeaways of deferred rendering is we are able to render global illumination effects independent of scene complexity – in screen-space, there are no

polygons, only pixels. Compute time for any effect computed with deferred shading depends only on screen resolution.

So, in a sense, screen-space methods are a type of *ultimate culling* as far as global illumination is concerned. We perform global illumination computations on *precisely* the visible surfaces, that will be shown to the end user in that frame, and *precisely* nothing more.

But that very advantage is actually screen-space's major flaw. If the object is behind something then it doesn't send an interreflection.



(a) Visible pinwheel sends interreflection (b) When hidden, no interreflection

Figure 2.12: A flaw with screen space methods, as illustrated by [54]. The bust does not receive an interreflection from the pinwheel when the pinwheel is hidden.

Accounting for this by depth peeling, ray marching [10] or using multiple views [44] is possible, but this increases cost significantly.

Precomputed Radiance Transfer

In 2002 Sloan published a technique called "precomputed radiance transfer" [50].

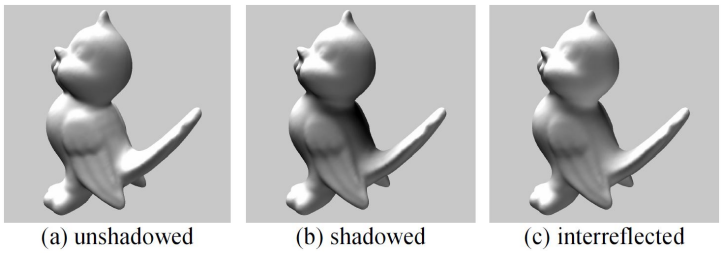


Figure 2.13: Precomputed radiance transfer [50]

This technique received a lot of attention because Sloan had shown that soft shadows and interreflections could be computed in real-time for static geometry and a dynamic, infinitely distant light source.

PRT parameterizes the global illumination response of each vertex based on surrounding geometry. Each vertex contains a function that can output the global illumination given only the incident lighting. There is no need to "feel around" for surfaces that shadow or interreflect at runtime.

The chief drawback to PRT is that scene geometry cannot deform at runtime, otherwise the precomputed data is rendered incorrect.

Chapter 3

Mathematical foundations for rendering

To solve a problem using a computer, we should be able to model it accurately. The mathematical model for rendering is outlined below.

3.1 The rendering equation

There are two common presentations for the rendering equation. We present both in order to use them in our discussions.

3.1.1 Kajiya's form

Kajiya [24] was the first to express global illumination for computer graphics as a mathematical equation

$$I(x, x') = g(x, x') \left[\epsilon(x, x') + \int_S \rho(x, x', x'') I(x', x'') dx'' \right] \tag{3.1}$$

Kajiya adapted the rendering equation from radiative heat transfer, of which the formula is actually a generalization. In equation (3.1), we are illuminating a point x

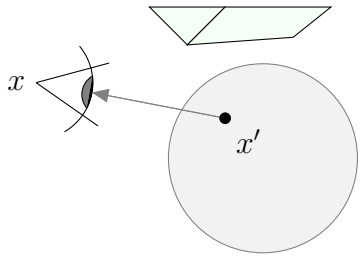
with light sent from point x', where

- $I(x, x')$ is the intensity of light from x' to x
- $g(x, x')$ is called a "geometry" term - but it really has to do with *what ratio of exiting energy at x' will reach x* due to surface orientation, position, material properties, etc. This number will always be between 0 and 1.
- $\epsilon(x, x')$ is the intensity of light that x' emits towards x
- $\rho(x, x', x'')$ is the intensity of light *another* remote patch x'' will send to x *via* a bounce first from x'
- $I(x', x'')$ is exactly what it looks like: the intensity of light from x'' to x'

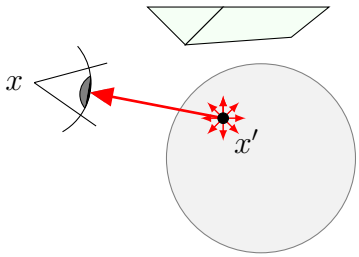
Pay special attention to the integral term on the right hand side of the equation. It involves evaluation of $I(x', x'')$, while $I(x, x')$ is what we are trying to solve for! To evaluate $I(x, x')$ we must evaluate $I(x', x'')$ on a hemisphere. Figure 3.1 illustrates what this means.

A key point to note is that in figure 3.1d, to find the light at x sent from x', $I(x, x')$, we also have to find the light sent from *every other* visible point x'' to x'. Once we have this quantity, we can find the total amount of light that all the x'' are sending to x *via* x'.

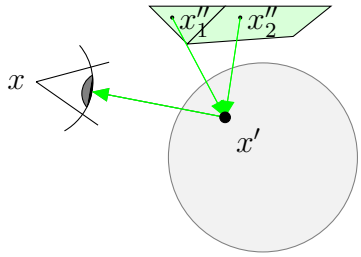
Intuitively you should see that if we delete the integral term on the right hand side of equation (3.1) we would just get the emission from light sources only. If we evaluate equation (3.1) in a computer and cut off evaluation after finding one $I(x', x'')$ integral, then that would correspond to finding just one bounce of light.



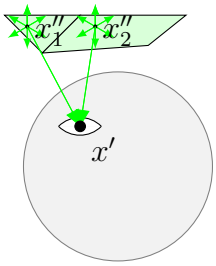
(a) $I(x, x')$: Light transfer to "eye" at x from x' is ..



(b) .. $g(x, x')\epsilon(x, x')$: illumination sent directly from x' reaching x + ..



(c) .. $g(x, x') \int_S \rho(x, x', x'')I(x', x'')$: The light x' reflects towards x due to light gathering at x' from all other points x''_j in the scene ..



(d) .. which requires us to put the "eye" at x' to see the light gathered at x', $I(x', x'')$

Figure 3.1: Illustrating the rendering equation, written as:
$I(x, x') = g(x, x')\epsilon(x, x') + g(x, x') \int_S \rho(x, x', x'')I(x', x'')dx''$

3.1.2 Differential angle formulation of the rendering equation

We will use an alternative formulation of the rendering equation called the "differential angle form" from Cohen [7] and also in Green [17] in some of our discussions. This alternative formulation is simply a refactoring of Kajiya's original formula.

$$L(x, \vec{\omega_o}) = L_e(x, \vec{\omega_o}) + \int_S V(x, x')G(x, x')L(x', \vec{\omega_i})f_r(x, \vec{\omega_i} \rightarrow \vec{\omega_o})d\omega_i \tag{3.2}$$

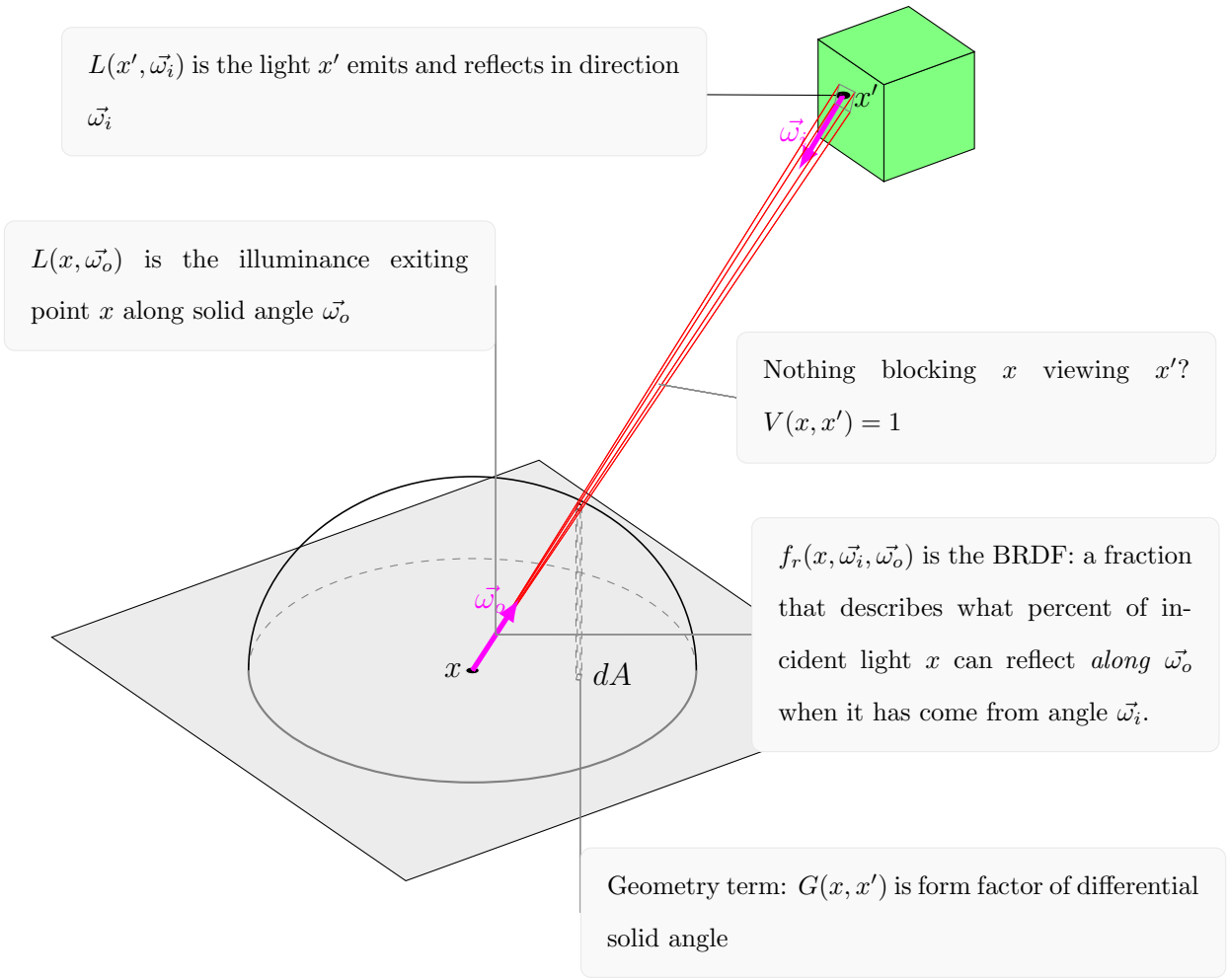
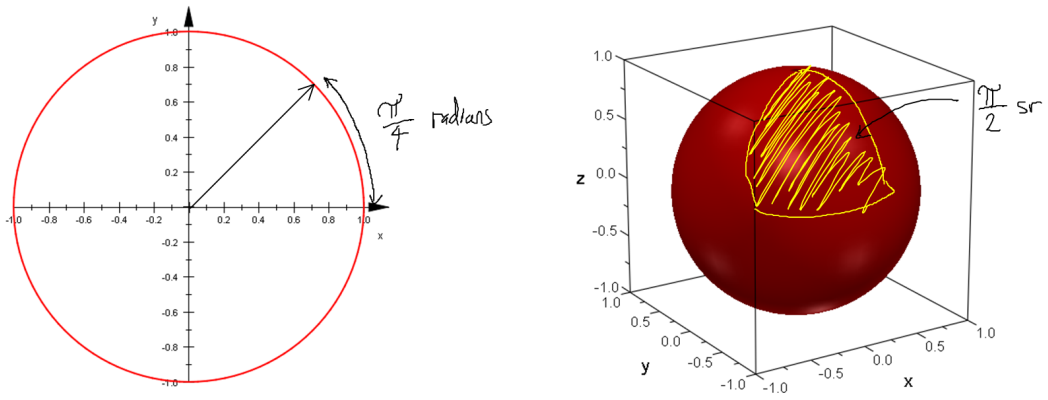


Figure 3.2: Illustration of the parameters of the solid angle version of the rendering equation (3.2)

Equation (3.2) says the same thing that Kajiya's formulation does in (3.1), but it is expressed using two additional concepts explicitly that Kajiya implicitly "hid" in the $g(x, x')$ "geometry term" in his formulation. Those two concepts are the *solid angle* $\vec{\omega_o}$, and the *visibility function* $V(x, x')$.

A *solid angle* is actually an *area* on the unit sphere, units *steradians* (sr). The unit sphere has a total solid angle of 4π steradians available over the 2 dimensions of its surface. This is analogous to how an *angle* is actually an arc length on the unit circle.

A "steradian" is like a "square radian"[2].



(a) An "angle" is an arc length on the unit circle. There are 2π radians to go around.

(b) A solid angle is an area on the unit sphere. There are 4π *sr* to go around.

Figure 3.3: Angles and solid angles

Now the left hand side of the equation (3.2) is talking about an incoming quantity of light $L(x, \vec{\omega_o})$ to point x *though the solid angle* $\vec{\omega_o}$. Note the solid angle measure $\vec{\omega_o}$ is a vector quantity in this equation, so it talks about both a magnitude *and* a direction for the incoming light. $L(x, \vec{\omega_o})$ will be the sum of the light emitted by x through $\vec{\omega_o}$ *plus* the sum of light from *all* surfaces x' *that we can see* ($V(x, x')$) over the hemisphere of directions at x ($\int_S$). Each "differential ray" of light that is sent from a remote surface x' comes in on solid angle $\vec{\omega_i}$. The integral visits *all* remote surfaces x' that are visible over the entire hemisphere of directions surrounding point x as illustrated in figure 3.2. The second thing in equation (3.2) that is not explicitly shown in equation (3.1) is the *visibility function* $V(x, x')$. $V(x, x')$ has the value 1 where the remote surface patch at x' is *visible* from the vantage point x, and the value 0 where the remote surface patch is *not visible*. It basically has the effect of killing the integral that gathers light from the remote surfaces x' if the surface x' is not visible from the vantage point x.

[2] There is also a non-SI unit called the "square degree", used in astronomy.

3.1.3 Solving the rendering equation

The rendering equation is actually a Fredholm equation of the second kind [42]. Fredholm equations of the second kind have the form:

$$\phi(x) = f(x) + \lambda \int_a^b K(x,y)\phi(y)dy \tag{3.3}$$

Fredholm equations of the second kind can be solved analytically by the Neumann series [58], or numerically by Gauss-Legendre quadrature [41]. In the specific case of the rendering equation, it ends up being intractably difficult to solve analytically [42]. Not only is solving the rendering equation difficult, even forming the equation to begin with is also hard. Specifically, the geometry term $g(x, x')$ (which ends up inside the kernel $K(x, y)$ in equation (3.3)) is difficult to find analytically. The geometry term $g(x, x')$ speaks of what fraction of exiting light at x' should reach x. In a usual computer graphics scene, such as the one in figure 3.4, there may be thousands or millions of polygons. Yet to solve the rendering equation we would need to know $g(x, x')$ for every pair of points in the scene.

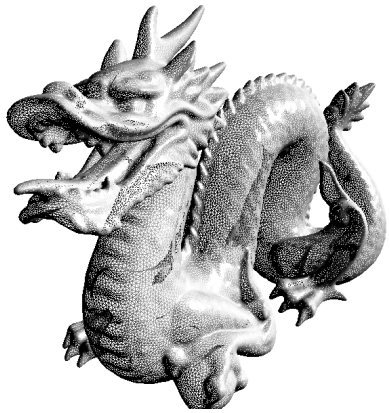


Figure 3.4: A dragon with about 600k polygons

In our solutions, we will call the geometry term $g(x, x')$ the *form factor*, *view factor*, or *shape factor*[3] [15]. The analytic formula for the form factor between two arbitrarily

[3]These different terms can be used interchangeably.

oriented polygons in three space was found quite recently in 1993 by Schröder and Hanrahan [46]. The expression for the form factor is extremely complicated, for further details the reader is referred to [46].

Because form factors are so difficult to express analytically, the task of finding form factors is also usually left to so-called "Monte Carlo" methods - or random sampling. So the two most common routes to solve the rendering equation will end up using Monte Carlo methods at some point:

1. Radiosity: Find form factors by using Monte Carlo methods, and solve the resultant radiosity equation using finite element methods

2. Ray tracing: Solve the rendering equation directly from scene geometry representation with Monte Carlo methods

3.2 Monte Carlo methods

Random sampling can be used to estimate integrals. The laws of probability will say, as we shall see, that we will come out with the correct solution.

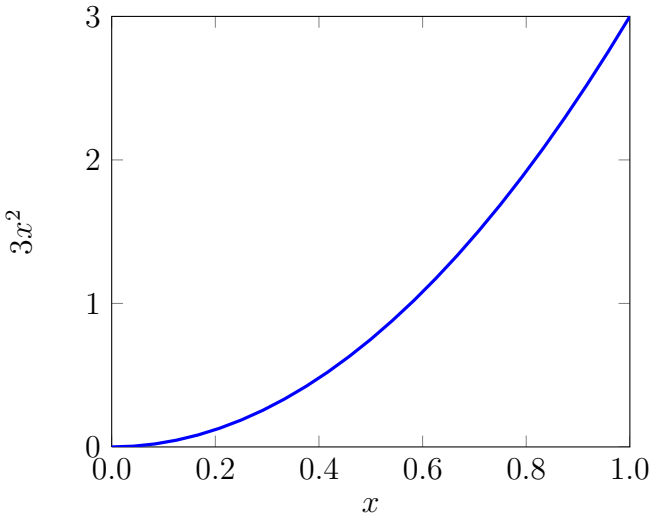
3.2.1 Why probability?

We use probabilty theory to verify that casting rays ("sampling") can indeed estimate an integral and so, help us find a correct solution to the rendering equation. The main technique we will use is called Monte Carlo integration. We begin by introducing a couple of terms.

3.2.2 Definitions

Probability density function

A *probability density function* $p(x)$, or pdf, is a function whose value p is how likely a random variable x has that value. For example, looking at the plot of $p(x) = 3x^2, x \in [0, 1]$:



What this pdf means is that there is a very small chance the random variable x takes on smaller values, for example between 0 and 0.5, and a very high chance that x will take on larger values, for example between 0.5 and 1.

To find the probability that the random variable x falls in a *range of values*, we have to integrate. For example, the probability that x takes a value on the interval $[0.5, 1.0]$ for the pdf above is $\int_{0.5}^{1} 3x^2 dx = 0.875$. This means that the random variable x has an 87.5% chance of having a value between 0.5 and 1, and only a 12.5% chance of being between 0 and 0.5. The integral over a pdf's domain must be equal to 1 (otherwise the function may not be called a pdf).

Expected value

The *expected value* of a function $f(x)$, denoted $E[f(x)]$, is an average value that we *expect* $f(x)$ to be when we consider *both* the values that $f(x)$ takes on over its domain

as well as the *likelihood* that the function $f(x)$ will *get to take on* that value. The likelihood that $f(x)$ will take on a certain value is of course dictated by the probability distribution of the underlying independent variable x.

For a function $f(x)$ that we have an explicit formulation for, and a probability density function $p(x)$, the expected value is simply

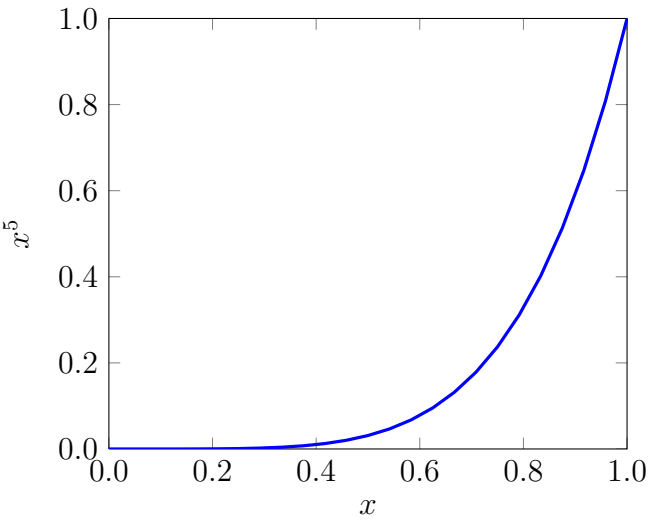
$$E[f(x)] = \int f(x)p(x)dx \tag{3.4}$$

If x has uniform distribution, then the *expected value* degenerates into the mean value of $f(x)$.[4]

If we have a simple function

$$f(x) = x^5$$

Whose graph is



Then, the average value of $f(x)$ assuming all values of x on $[0, 1]$ are equiprobable is the integral:

[4] The mean value of $f(x)$ on $[a, b]$ is given by $\int_a^b \frac{f(x)}{b-a}dx$. Because the integral of a pdf must be 1 over its domain, a uniform pdf has a constant value $\frac{1}{b-a}$, where a and b are the limits of its domain. Thus $\int f(x)p(x)dx = \int \frac{f(x)}{b-a}dx$, which equals the mean value of $f(x)$ as long as you integrate over the entire domain $x \in [a, b]$.

$$\int_0^1 x^5 dx = \frac{1}{6}$$

But if the distribution of x values is as according to the pdf in 3.2.2, then low values of x are less common and high values of x are more common. Intuitively, then, we *expect* that the *expected*[5] value of $f(x)$, therefore, should be quite a bit higher than $\frac{1}{6}$.
From the definition of expected value in (3.4) we have $E[f(x)] = \int f(x)p(x)dx$. That is, the expected value of $f(x)$ is equal to the sum of infinitesimals $f(x)dx$ *weighted by* the probability that f *should take on* those values $f(x)$ at each x.
So actually the expected value will be:

$$\int_0^1 f(x)p(x)dx = \int_0^1 (x^5)(3x^2)dx = \frac{3}{8}$$

Where the function $f(x)$ is unknown, for example while estimating an integral that we cannot solve analytically, the expected value can be used to find an estimate for *the correct value* of the integral. We show this below in section 3.2.2.
For the discrete case, the integral becomes a sum.

Variance

Variance is a measure of the "swing" of the results of the random experiment around the expected value. To compute the variance analytically you need to have the expected value:

$$\text{Var}[f(x)] = E[(f(x) - \mu)^2]$$

where $\mu = E[f(x)]$.
We are really looking for magnitude fluctuation of $f(x)$ about the expected value μ, so variance is computed as the expected value of the square of the difference between the

[5]Intentionally redundant

value of the function at each x, and μ (which is the expected value of the function *over all x*).

Variance manifests in a render as a speckly-looking noise.

Application of probability to the rendering equation

How does this apply to graphics? Well, we can estimate the value of an integral using expectation.

The rule we will use here is called the Law of Large Numbers:

$$E[f(x)] \approx \frac{1}{N}\sum_{i=1}^{N} f(X_i) \tag{3.5}$$

Where X is a random variable. In English this says "the average value of a function is approximately equal to the average of N random samples taken on the domain of that function." Of course, the more samples we take, the better the estimate.

Since we can write

$$\int f(x)dx = \int \frac{f(x)p(x)}{p(x)}dx$$

and

$$E[f(x)] = \int f(x)p(x)dx$$

therefore

$$\int f(x)dx = \int \frac{f(x)p(x)}{p(x)}dx = E\left[\frac{f(x)}{p(x)}\right] \tag{3.6}$$

Using equation (3.5) together with equation (3.6), we have the Monte Carlo Estimator [17]:

$$\int f(x)dx = E\left[\frac{f(x)}{p(x)}\right] \approx \frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)} \tag{3.7}$$

Where X is a random variable. All the Monte Carlo estimator says is that the integral of $f(x)$ is approximately equal to the weighted average of N different values $f(X_i)$.

Each $f(X_i)$ is weighted (divided) by the probability $p(X_i)$ that X_i should occur.

What Monte Carlo integration means for us is we can estimate an integral to arbitrary precision simply by taking a lot of random samples and averaging them.

This fits perfectly with our problem in solving the rendering equation, where we don't always have an explicit formulation for the integrand (recall: analytic formulae for $g(x, x')$ are complicated!), but finding numerical values for that integrand is easy.

If we use a uniform distribution for $p(x) = P$ such that each X_i is equiprobable, then the formula for Monte Carlo integration becomes even simpler:

$$\int f(x)dx \approx \frac{1}{NP}\sum_{i=1}^{N} f(X_i)$$

We will use Monte Carlo integration later to project an occlusion function into spherical harmonics as well as to calculate pixel color values for Monte Carlo ray traces.

Realizing that Monte Carlo ray tracing is a probabilistic method, we can start to identify noise and artifacts in our rendered image as what they are in probability: variance. More advanced methods for laying out where we take our samples can help us reduce the variance in our image by changing the way we sample the integral.

Importance sampling

The general idea behind importance sampling is to focus your solution efforts on areas that help you get the correct solution faster.

Various techniques exist that attempt to minimize the variance by judiciously choosing where to take additional samples in the Monte Carlo integration estimate. For example, Metropolis light transport chooses to favor the tracing paths that have previously let light through. This enables the algorithm to better light narrow corridors or small holes with fewer samples.

3.3 Evaluating Monte Carlo integrals at run time and precomputing part of the solution

Evaluating a Monte Carlo integral of a scene with thousands of polygons at run time is very expensive. We are just on the brink of being able to run a Monte Carlo ray trace on state of the art hardware in real-time [35].

So we must reduce the amount of computation that must be done in each frame at runtime to get interactive performance on less powerful hardware. In order to do this, we *precompute* some part of calculation. In doing so, we consume some memory, and sacrifice some flexibility.

3.3.1 Totally precomputed: solving the rendering equation once

It is possible to solve the rendering equation on the scene geometry using a method like radiosity, and *save the resultant color values out to a texture file.*

At runtime, we simply render the scene with these "baked" color values. This is the technique that has been used for simulating real-time global illumination since Quake in 1996.

While the results look very good, this solution is the most inflexible, as neither the geometry nor the light sources can move, otherwise the solution is made incorrect.

3.3.2 Precomputed radiance transfer: parameterizing a solution to the rendering equation

Another thing we can do is ***precompute the light-sharing and light-blocking relationships between scene geometry*** (ie precompute the form factors). Here we fix the scene geometry, and let only the light sources vary, in both position and

Figure 3.5: Quake by ID Software

intensity.

This is where most of the algorithms studied here get their speed, because it removes the *recursion from the runtime evaluation of the rendering equation*, instead working out a function that will give you the result of that recursion immediately[6].

We must keep in mind that any precomputed data that is based on a specific arrangement of scene geometry will get "dirty" if any of the geometry in the scene moves at all. Precomputed data is typically not easily updated.

In sum, the speed up from precomputed techniques comes from the fact that we don't have to "feel out" what geometry is around us at runtime. We then use that precomputed information for global illumination at runtime and get very good frame rates.

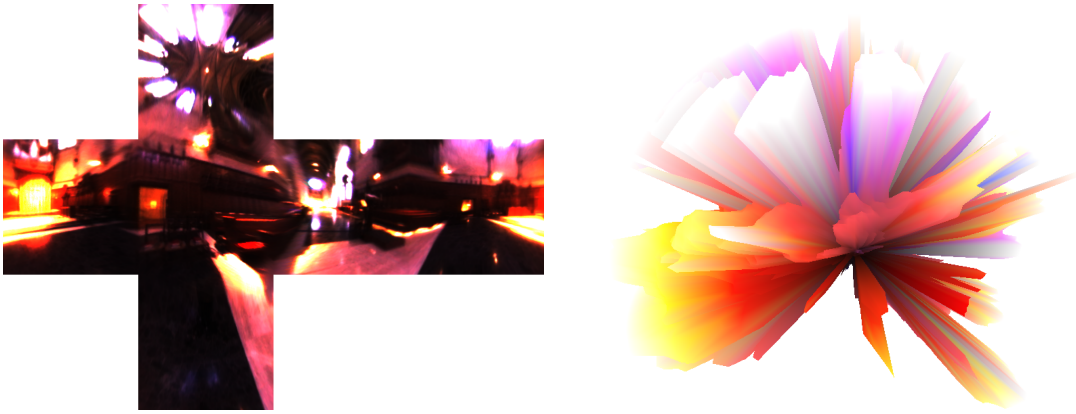
[6]Recall (3.1): to find $I(x, x')$ we have to find $I(x', x'')$ for all the x'' that x' can see. What we are going to do is precompute part of $I(x', x'')$ so that run time evaluation is very fast.

3.4 Light, shadows, and directions

We begin to describe how we can represent *direct illumination light functions*, shadow-producing *occlusion functions*, and color bleeding *interreflection functions*. Combining these three functions, we will get global illumination on static geometry, where the *direct light function* is a *parameter*.

3.4.1 Light functions

Unlike the traditional Phong model, where lights are simply point sources, we will think of ambient light as a *function* of values on a sphere. For each (θ, ϕ) direction on the sphere, we have an RGB color value for the light intensity coming from that direction. Ambient light can be most conveniently specified by the color values on a *cube map*. (The convention for the orientation of cube maps is in figure 1.6.)



(a) Cube map, standard orientation

(b) Visualization of directions light comes from on cube map

Figure 3.6: Visualizing direct lighting. The left plot is the Grace Cathedral cubemap and it specifies light power as a texel color for each of $6 \times 256 \times 256$ distinct directions. The right plot is a 3d visualization of the cubemap. You should be able to clearly see that the purple-white blocks of light in 3.6b are from the lights on the ceiling in 3.6a

Now, how can we light a scene point by a function like this? If we have the *occlusion* function for a point (ie the set of directions for which light is blocked and allowed to pass through at a point), then we can determine how much light that point will receive *as a function* of the ambient light.

Similarly, if we have the *interreflection* function for a point, (ie the set of directions for which interreflected light is received, on what color bands, and in what intensities), then we can determine how much *interreflected* light will reach that point *as a function* of the ambient light, *without having to feel out what geometry is around the point at runtime.*

To create the occlusion and interreflection functions, at each scene point, we need to find what directions:

1. Will block light (creating shadows)

2. Will send additional light (interreflections and caustics)

3.4.2 Shadows

To create shadows, we need to know the directions for which direct lighting is blocked.

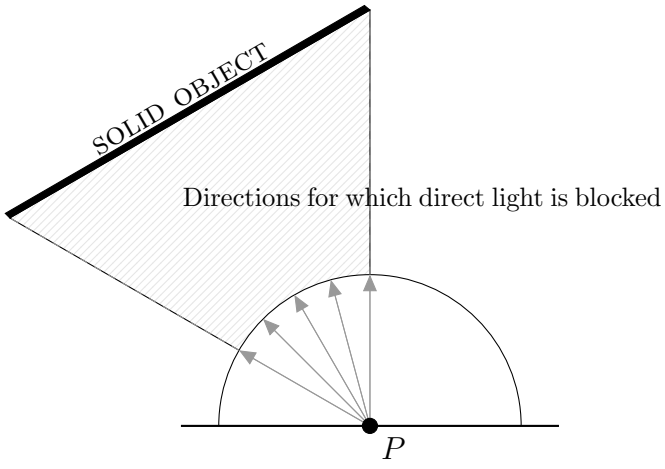


Figure 3.7: Occlusion function

You can see now how we can easily shade the point P in figure 3.7 by simply zeroing light coming from the occluded directions and allow lighting by "open" directions only. A "light blocking" function like this can be created using ray casting.

At this point you should observe that if any of the scene geometry moves (e.g. the "solid object" in figure 3.7), then our occlusion function becomes incorrect and we need to recompute it. However, the scene objects can rotate and move rigidly, as long as the geometry does not move in such a way that the occlusion functions change.

3.4.3 Interreflections and *transferred radiance*

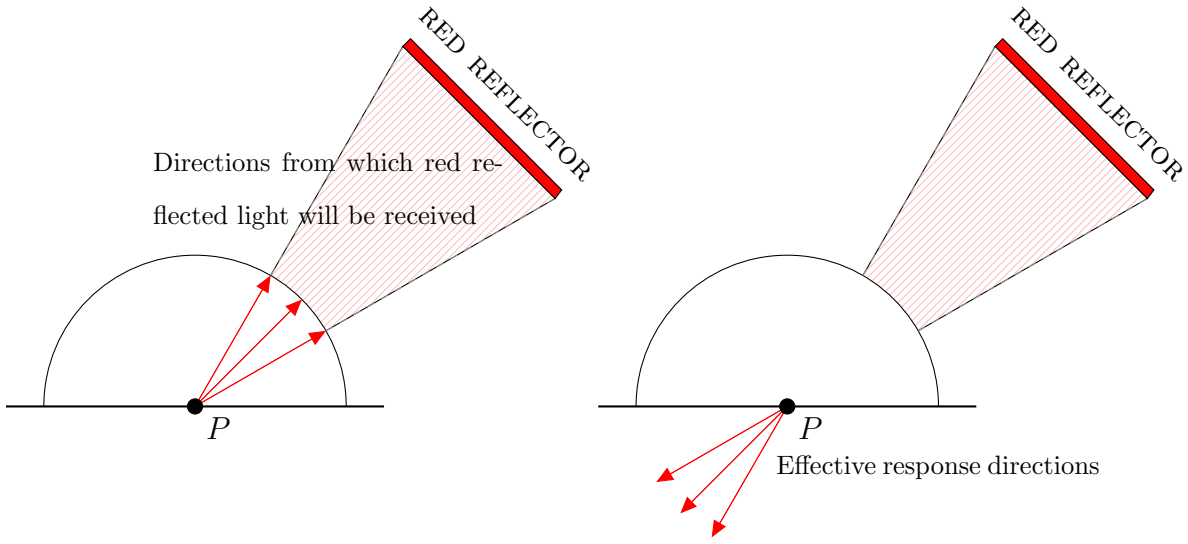
To handle interreflections, we need to know *what directions send reflected light*, and in what ratio *to the incident light* from the direct lighting distribution. Assume we have two surfaces that share light, A and B. **If we already know *how* A will respond to the brightness of B, as a function of the incident light, then we don't need to compute the brightness of B at runtime to get B's effect on A.**

This is called *transferred radiance* in the literature [50]. A simple example with a red diffuse reflector is given in figure 3.8.

Even though this seems counter-intuitive, we will light the point P in figure 3.8 by direct lighting *that comes from behind the surface that P sits on* (from the directions labelled *"Effective response directions"* in figure 3.8). This models the interreflection of RED REFLECTOR onto P. ***This is a key difference between vector occluders and virtual point lights.*** While VPLs are virtual light sources placed in the scene, VO doesn't place any "extra light sources" in the scene. Instead, it uses the *transferred radiance* at each vertex to account for the interreflections.

Combining the shadow and interreflection functions gives a single function whose output is the global illumination at a scene vertex, and can be evaluated very fast, without having to integrate at runtime.

The challenge ahead of us is *representing the shadowing and interreflection functions in*



(a) Red reflector will send red light to P *from* directions shown

(b) P now *effectively* responds to ambient light that comes from behind it

Figure 3.8: Parameterizing interreflection

such a way that runtime lighting can be evaluated very fast (much less than 16 ms!), and in such a way that the precomputed data does not consume too much memory. We will need to "approximate" these functions because the raw data takes up too much memory, and with so many data components the number of multiplies to evaluate at run time becomes too much. We need compact data that at the same time represents the whole lighting function.

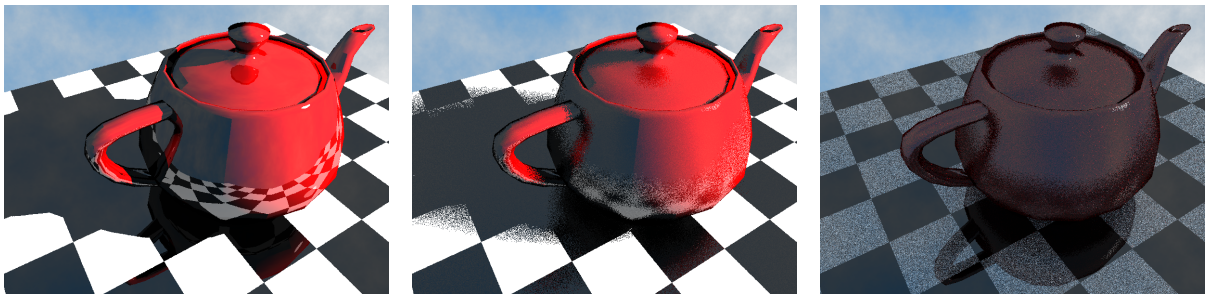
This is the main motivation for using spherical harmonics and wavelets – a space-efficient representation of spherical data becomes possible. Also, by working in an orthonormal basis such as spherical harmonics or wavelets, integration become a cheap dot product.

Chapter 4

Full solutions

4.1 Ray tracing

Ray tracing was first introduced in 1980 by Whitted [60]. Since then several variations of ray tracing emerged, including *distributed ray tracing* by Cook in 1984 [8], and *path tracing*, which was introduced by Kajiya in the same paper that he introduced the rendering equation in 1986 [24].



(a) **Whitted**: sharp shadows, specular reflection only

(b) **Distributed**: Adds to Whitted: soft shadows, glossy interreflections

(c) **Path**: Adds to distributed: color bleeding. The color balance is very different because no shadow ray is created.

Figure 4.1: Types of ray tracing. Scene is solid red teapot on a checkerboard platform in a Perlin sky.

Mathematically ray tracing is a Monte Carlo estimation of the rendering equation. The different flavours of ray tracing simply have different sampling domains and sampling orders.

Practically, ray tracing shoots a number of rays from the eye, and follows it through the scene backwards, until a light source is reached. If no light source is reachable then that point is in darkness.

Every time a ray hits a surface, that ray can be reflected or refracted[7]. There are two types of reflection: specular and diffuse reflection as mentioned previously in 2.1.1 (transmission is commonly classified as a type of specular reflection).

The mechanics of ray tracing are quite simple. We define a few types of ray:

- **Shadow ray**: A ray shot towards the light source. If the ray gets blocked by another object then the point that shot the shadow ray is in shadow (hence the name of this type of ray).
- **Reflection ray**: Shot in the specular (mirror) direction $\vec{R} = \vec{V} - 2\vec{N}(\vec{V} \cdot \vec{N})$[8]
- **Transparency rays**: Ray shot through an object, and bent by Snell's law

Publications [18] and [1] have a nice notation using regular expressions that describes the set of possible paths that different tracer algorithms will trace that will result in a *lit* pixel (hence why all paths end in reaching a light source L).

4.1.1 Whitted tracing

$E[S^*]D?L$[9]

Whitted tracing can follow paths from the eye, take any number of specular bounces, and then compute a single diffuse color to the light. A path such as $EDDDL$ is never

[7] The ray can also be absorbed, but because the basic ray tracing algorithm does not model absorption explicitly we do not mention it. Russian roulette does model absorption in a sense explicitly, however.

[8] $\vec{R}$ is the reflected vector, $\vec{V}$ is the original vector, $\vec{N}$ is the normal at the point of intersection

[9] Eye, zero or more Specular reflections, possible Diffuse bounce, reach Light

going to be explored in a Whitted tracer: bouncing is specular only, so Whitted ray traces don't have diffuse interreflection (color bleeding).

4.1.2 Distributed tracing

$E(D|S)^*R\{n\}L$[10]

Whitted tracing was limited to sharp shadows, sharp reflections, and sharp refractions. This was because ray directions on reflection or refraction were completely determined by the simple reflection formula, or Snell's law. These formulae are deterministic: two different incident rays travelling in the exact same direction will *reflect and refract in exactly the same direction*, every time. Because of this level of determinism massive sub-pixel supersampling was required to get realistic blurring of these phenomena. However, Cook [8] found that if we introduce some randomness and jitter into reflection and refraction directions, we can get soft shadows and blurred ("glossy") reflections without oversampling.

To do this, at each intersection, we shoot n rays *in a distribution of directions* in accordance with material properties (instead of all in the same direction). Shinier materials have a narrower spread, more glossy materials have a wider spread. There is now a continuum between specularity and diffusivity that better reflects[11] reality.

If the rays are distributed over *time* (instead of space) then motion blur becomes possible [8].

4.1.3 Path tracing

$E(D|S)^*L$[12]

The regular expression for walkable paths looks similar to distributed tracing, but here

[10] Eye, zero or more Specular *or* Diffuse reflections, n shadows Rays to light
[11] Pun intended
[12] Eye, zero or more Specular *or* Diffuse reflections, Light

$n = 1$: the number of rays for diffuse color gathering is always 1. This is called a *random walk* or a *Markov chain* [1]. It does a good job of modelling of the real world process of following a photon through the scene from the eye (back to) the light source from which it came.

4.1.4 Variance reduction techniques

There are a few techniques we can use to reduce the variance in the image for the same number of samples by carefully choosing the *shape* of the distribution of sampling rays on the hemisphere.

Importance sampling

We can use importance sampling anywhere we are computing an integral by Monte Carlo methods. The main idea behind importance sampling *is to shoot more rays in the directions we know they will count.* So in estimating the lighting at a vertex in distributed ray tracing, we can shoot rays with a cosine lobe distribution instead of a hemispherical one. We shoot a large number of rays with small angles with the normal (which is where they really count), and fewer rays with large angles with the normal (a ray shot at 89° to the normal will be attenuated by a factor of $\cos 89° = 2\%$, so we don't want a lot of those).

Metropolis Light Transport

MLT [56] encourages fast convergence of the integral by favoring tracing through and near paths that are known to be unblocked. Narrow unprobable paths are thus explored more thoroughly with fewer total samples.

Photon mapping

The idea behind Photon mapping [23] was to pre-process light transport by "parking" photons in the scene prior to ray tracing the image. This makes getting a stable estimate for effects such as caustics in a ray traced image much faster.

4.1.5 Real-time ray tracing

Some impressive demonstrations of real-time ray tracing already exist [43].

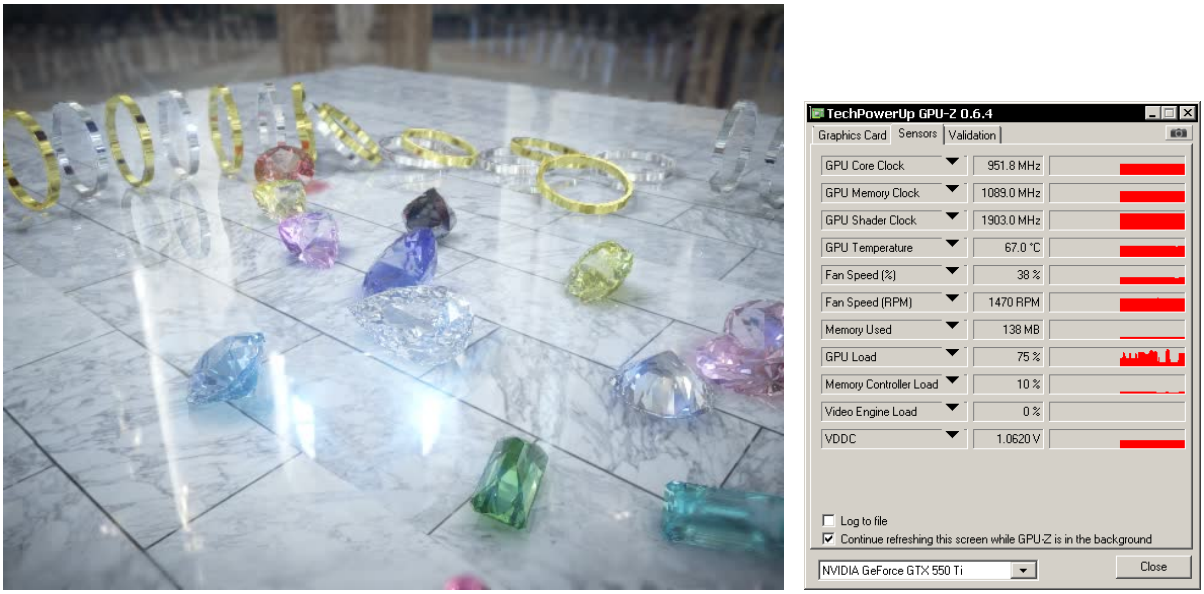


Figure 4.2: A screen capture from Rigid Gems 2.0 [43], running at 60 FPS in DirectX11, consuming 3% CPU and 75% GPU on the experiment machine described in B.

We expect these demos to get better year by year, but the fact remains that real-time ray tracing requires bleeding edge hardware and uses most of it, (RigidGems requires DX11 and Shader Model 5.0, and has 75% GPU usage for the scene shown in figure 4.2). We will not see real-time ray tracing on portable hardware for some time.

4.2 Radiosity

As early as 1984 Goral et al. [15] discussed using techniques from radiative heat transfer to find the total flux of radiant energy from surfaces. This technique has been called "radiosity". Radiosity is a finite element method approach to solving the rendering equation.

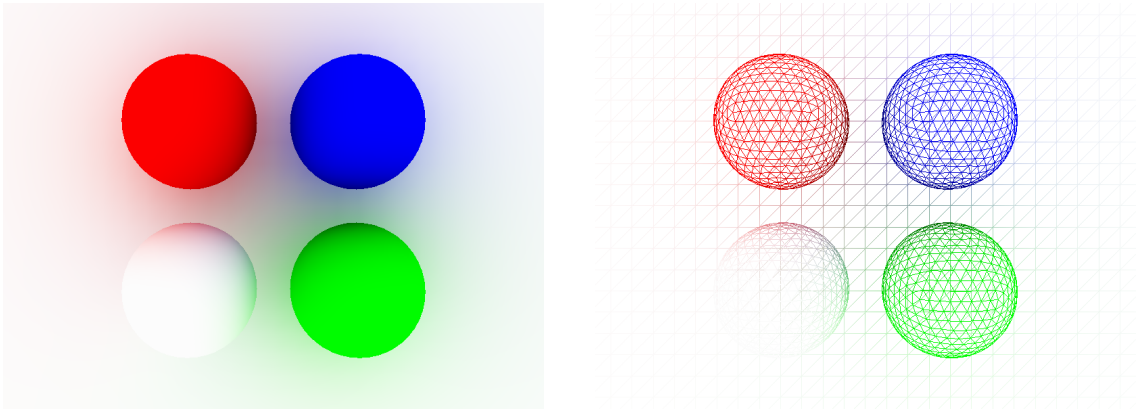


Figure 4.3: An example of a radiosity render (left) with wireframe (the "finite elements", right). Note the diffuse interreflections and soft shadows.

The "radiosity" of something is defined as its *radiant exitance*[13][2]. Radiant exitance is simply the electromagnetic radiative ***power*** emitted from a surface per square metre. The SI unit for radiant exitance is $[W/m^2]$. Keep in mind that the *Watt* is actually energy in time ($[J/s]$), so radiant exitance is actually measured in $[(J/s)/m^2]$. In other words, radiant exitance is the energy you get from the surface per second (if you're standing in front of it, capturing all of it) *per* metre squared of surface that there is.

4.2.1 Radiosity equation

To create a radiosity solution for a scene, we first must break the scene into patches (the "finite elements" in "finite element method"), then we create a matrix that

[13] "Radiosity" is both the *name* of the technique, as well as being the measurable quantity you want to find. We will prefer the term "radiant exitance" to avoid confusion.

describes each patch's light-sharing relationship with every other patch in the scene. This will be an $n \times n$ matrix called *the matrix of form factors*, where n is the number of patches in the scene.

The radiosity equation that we need to solve as formulated in [2] is:

$$B_i = E_i + \rho_i \sum_{j=1}^{n} F_{ij} B_j \tag{4.1}$$

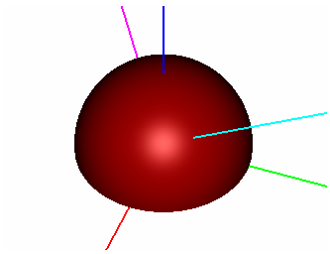
Where:

- B_i is the *final* radiant exitance of the i^{th} surface element, after the light has bounced around the room. Note that B appears on both the left and right hand sides of the equation. This is because the lighting at patch i depends on not only the radiation incident from light sources, but also due on the light interreflected from all other patches in the scene.
- F_{ij} is the form factor relationship between patches i and j. F_{ij} is the fraction of exiting radiance *sent* from j that is received by i. Form factors are thoroughly described in section 4.2.3
- E_i is the initial radiant exitance of patch i. A patch i that is *not* a light source will have 0 for its E_i entry.
- ρ_i is the *surface reflectivity constant* for patch i. It represents the ability of i to reflect light, and will usually have three (R, G, and B) values. If the value is 1.0, then the surface reflects 100% of the light incident on it for that color band.

4.2.2 Relating the radiosity equation to the rendering equation

The radiosity equation (4.1) is a discretization of the rendering equation (3.1). To write the radiosity equation, however, we made one important assumption about the scene's

surface properties. This assumption is that *all surfaces are perfect diffuse emitters*. A perfect diffuse emitter sends exactly the *same* amount of light in every direction, regardless of the angle from which the light came.



(a) Lambertian BRDF visualization

(b) Lambertian armadillo

Figure 4.4: BRDF of a perfect diffuse (Lambertian) emitter and a Lambertian armadillo rendered using [6]. Note the incident ray (cyan line) spreads in a perfect hemisphere. This means that an equal magnitude of light is sent in every direction on the hemisphere. No extra light is sent along the mirror reflection direction (magenta line), so the surface is not at all specular. Also note that if we change the angle of incidence (cyan line) there will be absolutely no change in the BRDF response (it will still be a perfect hemisphere).

As shown in figure 4.4, there is no specularity to a perfect diffuse emitter.
With this assumption in place, we can completely drop the view dependence from the rendering equation (repeated here for convenience):

$$I(x,x') = g(x,x')\left[\epsilon(x,x') + \int_S \rho(x,x',x'')I(x',x'')dx''\right] \tag{3.1}$$

To get the rendering equation *for perfect diffuse emitters*:

$$I(x) = \epsilon(x) + \rho_x \int_S g(x,x')I(x')dx' \tag{4.2}$$

Equation (4.2) says:

> The light intensity from x, called $I(x)$, is the light directly emitted from x, $\epsilon(x)$, plus the light reflected by x that came to x from all other scene surfaces x'.

Note that because x is a perfect diffuse emitter now, ρ_x has lost its directional dependence. Compare the left-hand side quantity I in (4.2) and (3.1). In (4.2) it is just $I(x)$, not $I(x, x')$: in other words there is no longer a *destination* surface x' to send the light *to*; radiant intensity in (4.2) is described as just light intensity *from* x. We don't have to say where that light is going *to* simply because we already know the radiant intensity is actually the same in *all directions*, under the *perfect diffuse emitter* assumption. Each surface *sends the same* ***amount*** *of radiance out in every direction that it sends any radiance at all.*

Comparing (4.2) directly with (4.1)

$$B_i = E_i + \rho_i \sum_{j=1}^{n} F_{ij} B_j$$

- B_i is $I(x)$, the final radiant exitance from patch i
- E_i is $\epsilon(x)$, the initial radiant exitance from patch i
- ρ_i is a *reflectivity constant*. It has been pulled out of the summation because a perfect diffuse reflector can be more or less reflective, but it is just scaling, perfect diffuse reflectivity is always a completely even distribution, ie there is no dependence on which patch light is being sent to j.
- Inside the summation loop for $j = 1..n$:
  - F_{ij} is the form factor or light sharing relationship of patch i with patch j and corresponds with $g(x, x')$
  - B_j is the final radiant exitance of patch j

4.2.3 Form factors

We introduced the form factor matrix in equation (4.1). Now we intuitively describe what a *form factor* is. We mentioned previously that physically, the form factor

between two patches i and j, written F_{ij} (or sometimes written $F_{i \to j}$), is the fraction of exiting radiance *sent* from i that is received by j. It is the ratio $\frac{\text{energy into } j \text{ from } i}{\text{total energy out } i}$. In other words, the form factor F_{ij} is the fraction of radiant energy that j will capture *of the radiant energy that i sends out.* Form factors are also called view factors, and we will also say that the view factor F_{ij} is the fraction of total radiant energy that j ***sees*** of the total radiant energy that i sends out.

There is an analogy here of form factors with vision that is very intuitive. What you *see* at any moment in time may be considered as is a rough estimate of the form factors of the surfaces in front of you with your eye's retina.

View factors example: Fictional room

Let us take the first row of a form factor matrix from a fictional room with only 4 patches.

	Patch 1	Patch 2	Patch 3	Patch 4
Patch 1	[0.0	0.5	0.25	0.25]

From this we conclude:

- Patch 1 sees 0% of its own exiting radiance
- Patch 2 sees 50% of patch 1's exiting radiance
- Patches 3 and 4 sees 25% of patch 1's exiting radiance (each)

The row sums to 1, meaning 100% of patch 1's exiting radiance is accounted for (by being "seen" by other patches in the room).

The diagonal of the form factor matrix should always be all zeroes, because $f_{11}, f_{22}, f_{33}..f_{nn}$ represents a patch's form factor with itself. This should always be zero unless curved patches such as a parabolic dish reflector are allowed. Curved patches aren't commonly allowed in computer graphics applications, because they cannot be easily rasterized.

The sum of the entries of each row of the form factor matrix will be exactly 1 if the room doesn't have any holes in it. If the room does have holes (and radiance will escape into space) then some of the rows may not sum to 1. If a row sums to 0, then it means the patch that row represents actually faces out of the scene and cannot see any other patch at all.

4.2.4 Computing form factors

Ashdown [2] gives the mathematical formula for the form factor between patches i and j as:

$$F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{\cos\theta_i \cos\theta_j}{\pi r^2} dA_j dA_i \tag{4.3}$$

There is a closed expression to compute the form factor between two arbitrarily oriented polygons in [46], but the formulae are very complicated. We use Nusselt's analog to compute form factors in our work.

Nusselt's analog

Nusselt's analog states that the form factor of a piece of geometry j with a differential element dA_i can be found by projecting j onto a unit hemisphere surrounding dA_i,

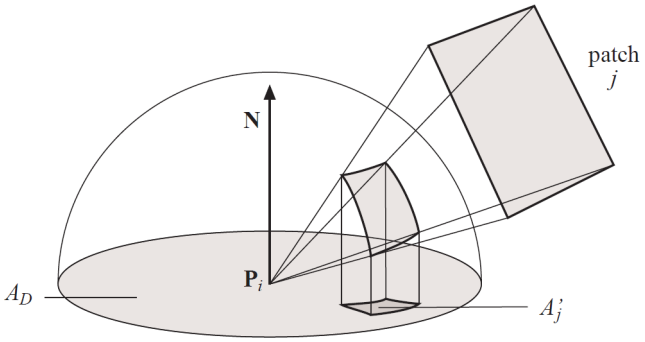


Figure 4.5: Illustration of Nusselt's analog, from [55]

followed by a projection of that result onto the unit circle at the base of that hemisphere. This is illustrated in figure 4.5, which was reprinted from [55].
Finding the projection onto the hemisphere is very easy to do in a computer graphics application using raycasting or hemicubes. The details of how to do so are in [2].

4.2.5 Matrix form of the radiosity equation

The B (final exitance) term (which is the quantity we are solving for) being on both the left and right sides of the equation is problematic.
Let us refactor the radiosity equation so we can solve it:

$$E_i = B_i - \rho_i \sum_{j=1}^{n} F_{ij} B_j$$

So now, *the first row* can be expanded as:

$$E_1 = B_1 - (\rho_1 F_{11} B_1 + \rho_1 F_{12} B_2 + \cdots + \rho_1 F_{1n} B_n)$$

So,

$$\begin{aligned}
E_1 = &\quad (1 - \rho_1 F_{11}) B_1 &+ &\quad (-\rho_1 F_{12}) B_2 &+ \cdots + &\quad (-\rho_1 F_{1n}) B_n \\
E_2 = &\quad (-\rho_2 F_{21}) B_1 &+ &\quad (1 - \rho_2 F_{22}) B_2 &+ \cdots + &\quad (-\rho_2 F_{2n}) B_n \\
&\vdots \\
E_n = &\quad (-\rho_n F_{n1}) B_1 &+ &\quad (-\rho_n F_{n2}) B_2 &+ \cdots + &\quad (1 - \rho_n F_{nn}) B_n
\end{aligned}$$

This can be rewritten as a matrix

$$\begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{bmatrix} = \begin{bmatrix} 1 - \rho_1 F_{11} & -\rho_1 F_{12} & \cdots & -\rho_1 F_{1n} \\ -\rho_2 F_{21} & 1 - \rho_2 F_{22} & \cdots & -\rho_2 F_{2n} \\ \vdots & \ddots & & \vdots \\ -\rho_n F_{n1} & -\rho_n F_{n2} & \cdots & 1 - \rho_n F_{nn} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{bmatrix} \tag{4.4}$$

It should be clear that if we premultiply E by the inverse of A, then we will have B, which is the final exitances of each patch.

4.2.6 Iterative solution methods

To solve the system $Ax = b$, an iterative technique starts with an initial guess solution vector for x, call it x_G, and progressively updates x_G until it is close enough to the actual solution to be acceptable.

The way we know how close x_G is to being the correct solution is by computing the *residual* R:

$$R = Ax_G - b \tag{4.5}$$

When x_G is close to being correct, R will be close to the zero vector.

Next we will discuss the Jacobi method and two *relaxation* methods, Gauss-Seidel and Southwell[14]. A *relaxation* method "fixes" one of the x_{Gi} at a time, by choosing a new value for it such that the corresponding residual R_i is zero. Note that relaxation methods *don't always work* on every matrix A. It is possible for a relaxation method to *diverge*, with x_G values that get further and further away from a solution on each iteration instead of closer. For radiosity matrices, however, these methods will always converge because radiosity matrices are diagonally dominant[15].

Jacobi method

Starting from the radiosity equation

$$Ax_G = b$$

[14] Technically the Jacobi method isn't a relaxation method.

[15] The condition for diagonal dominance is $A_{ii} \geq \sum_{j \neq i} |A_{ij}|$. We have this condition because in (4.4), for any given row i, the diagonal entry is $1 - \rho_i F_{ii}$, $\rho_i \leq 1$, $F_{ii} = 0$, and the sum of the rest of the form factors on any row i is $\sum_{j \neq i} F_{ij} \leq 1$.

We write this matrix multiplication as an explicit summation

$$\sum_{j=1}^{n} A_{ij} x_{Gj} = b_i$$

If we want to "fix" x_{Gi} to it's "correct" value, so that the residual (4.5) is 0, we simply have to isolate x_{Gi}. We give the "fixed" x value subscript N (for "next guess") instead of G.

$$x_{Ni} = \frac{b_i - \sum_{j \neq i} A_{ij} x_{Gj}}{A_{ii}} \tag{4.6}$$

Note while getting a 0 residual for x_{Gi}, we may increase the residual for other values in x_G. This is why the Jacobi method is iterative, and we might have to find each x_{Gi} many times before we get close to correct values.

The Jacobi method can be easily parallelized when implemented because we find all of the x_N using the same x_G.

Gauss-Seidel method

The Gauss-Seidel method is similar to the Jacobi method, but with one change. The x_{Ni} are found using a residual computed from the "up to date" x_N instead of only using x_G.

$$x_{Ni} = \frac{b_i - \sum_{j=1}^{i-1} A_{ij} x_{Nj} - \sum_{j=i+1}^{n} A_{ij} x_{Gj}}{A_{ii}}$$

Here we've used as many x_N values as are available in updating x_{Ni} (the first $i-1$ elements of x_N). The rest of the update is constructed from the current best guess x_G. This one simple change can reduce the total number of iterations required significantly.

Southwell method

The Southwell method is slightly different from both of the above iterative methods. Instead of walking the rows of A, and updating each row of x_G in-order, the Southwell

method searches through the residual vector for the row k with the largest error. It updates this row using

$$x_{Gk} = \frac{b_k - \sum_{j \neq k} A_{kj} x_{Gj}}{A_{kk}}$$

"Fixing" the estimate with the largest error has the effect of greatly reducing the number of iterations, but the cost of recomputing and searching through the residual that often has a larger performance impact that offsets the gains.

4.2.7 Radiosity in real-time

While radiosity produces very good looking results, it remains an offline method. Instant radiosity (1997) [25] and incremental instant radiosity (2007) [29] simulate radiosity by depositing "virtual point lights" in the scene by casting rays. Some earlier work from 1999 [11] added time-dependence to a radiosity solution, with the restriction that all motion must be known ahead of time. More recently in 2011, a technology company named Geomerics shipped a real-time radiosity solver called Enlighten [31] [32]. Enlighten does not provide a true solution to real-time radiosity because it has restrictions such as running the radiosity solution on the *static meshes* in the scene only. The bottleneck to truly real-time radiosity solution remains in the gathering of form factors, which must be done *every frame* if scene geometry is allowed to move freely.

4.2.8 Instant radiosity

Introduced in 1997 was the idea of using many virtual point lights (VPLs) to approximate the radiosity solution [25]. This is similar to PRT [50] but not identical. VPLs place actual extra lights in the scene, while PRT does not.

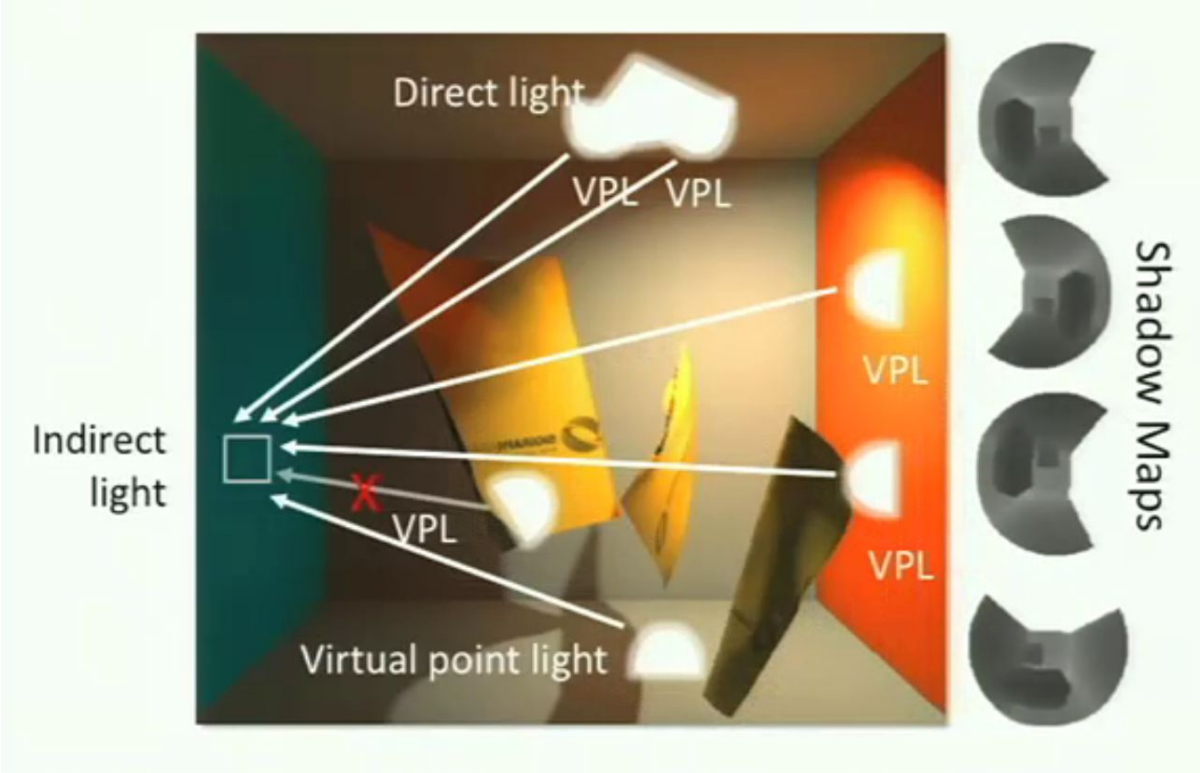


Figure 4.6: Instant radiosity places virtual point lights everywhere there would be a diffuse interreflection. It creates shadows using shadow maps to determine which surfaces are blocked from which point lights. Image from [10].

4.2.9 Architecture: Geomerics Enlighten

Geomerics' Enlighten middleware has convergence for the radiosity matrix in 60 ms [33]. With a 16 ms frame, this means lighting results are about 4 frames delayed. This is argued to be imperceptible except under flash lighting such as muzzle flash or grenades. Geomerics achieved this by clever architecture, not by newer or faster algorithms. In Enlighten, there is a clear cut separation between objects that are static and objects that are dynamic. Static geometry has radiosity solved for it in each frame, but form factors *are never recomputed* while the game is running. Dynamic objects (such as players or moving objects) *do not* have radiosity solved for them – but they are however allowed to sample the radiosity solution from nearby static geometry.

Precompute step

Enlighten's pipeline has a significant precomputation.

- Cut the large scene into smaller, independent "systems" that can be solved quickly (recall iterative methods have complexity $O(n^2)$, so smaller systems really do pay off)
- Create two sets of geometries: a coarse set, and a detailed set. Both sets of geometries are *made by the artist*.

Figure 4.7: Coarse and fine meshes. Note rounded edges in doorway arches become completely angular in coarse geometry.

> Radiosity is computed for the *target* geometry in figure 4.7 and mapped to the detailed geometry. The mapping of the target geometry to the detailed geometry is done as mesh-to-mesh projection offline

In addition to the precomputed steps, there is a bit of paralleism in the architecture. Specifically, Enlighten separates the computation of direct lighting (done on the GPU) and indirect lighting (done on the CPU). The direct lighting is done each frame and is always correct. The indirect lighting is done using the direct lighting *as its input*, and its accuracy depends on how fast radiosity can be solved (which depends on CPU speed).

The "Standard lighting" in 4.8 is computed on the GPU using a cascaded shadow map. The pieces of the target mesh that receive direct lighting act as light sources for the

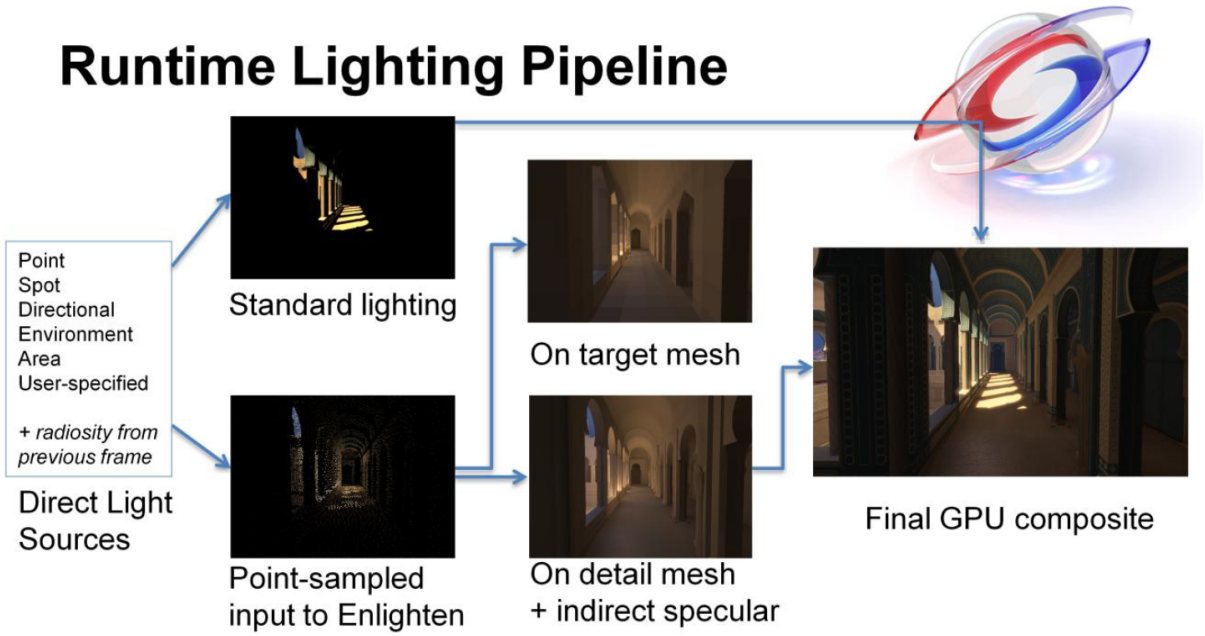


Figure 4.8: Geomerics Enlighting pipeline as in [33]

radiosity computation – which will be handled on a CPU thread *completely asynchoronously* to the next frame's direct lighting computation.

The radiosity computation uses first bounce lighting as it's input – the radiosity computation *does not* use the *light sources* directly as an input. The stroke of genius here is that if the light source moves, ***absolutely nothing has changed about the form factors as far as the radiosity computation is concerned***, because the light source geometry is *not* part of the radiosity computation. Thus we can have moving light sources without re-computing form factors. The radiosity engine will only "hear about" the changes to what patches are acting as light sources after GPU direct lighting has been recalculated.

This is the architecture Geomerics used to achieve real-time radiosity on commodity hardware. It is actually *not* completely dynamic. The main trick they appear to have used is if the form factors of the scene geometry don't change, then you only need to re-solve the existing system with a different initial exitance vector.

Chapter 5

Screen-space methods

Global illumination effects can be approximated in real-time by working in screen-space. The notion of working in screen-space was introduced in 2.4. Some points are repeated here. Screen-space methods are a type of *ultimate culling* as far as global illumination is concerned. We perform global illumination computations on *precisely* the visible surfaces, that will be shown to the end user in that frame, and *precisely* nothing more. Screen-space methods typically operate on the output data of the first pass of a deferred shader, ie the depth and normal G-buffers. This results in a number of advantages [10], namely:

- No precomputation is required
- Compute time for the *GI effects* depends on only the screen resolution, not on the polygon count of the underlying scene (we can only sample surrounding *pixels* for GI data in screen space)
- Scene can be fully dynamic
- Screen-space methods can be computed in real-time on modern GPUs

5.1 Ambient occlusion

Static mesh ambient occlusion (AO) was first introduced in 1998 by Zhukov [62]. Ambient occlusion estimates the blockage at each vertex with a scalar number [62] and calls this scalar quantity the "obscurance". Some works [10] speak in terms of the "accessibility", which is the opposite of obscurance. Generally $accessibility = 1 - obscurance$. The accessibility of a point tells you "how much that point is open to receiving ambient light."

The ambient occlusion of each vertex can be computed by integrating over the hemisphere S oriented with it's apex being pierced by the vertex normal as

$$\text{accessibility} = \frac{1}{2\pi}\int_S V(\vec{\omega})d\omega$$

Where $V(\theta, \phi)$ is 1 when the direction is *open*[16] and $V(\theta, \phi)$ is 0 when the direction is *closed*.

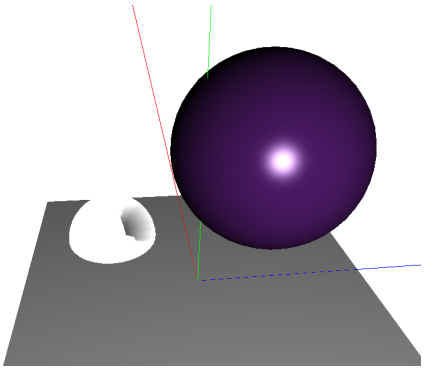


Figure 5.1: The purple sphere "punches a hole" in the directions from which a vertex on the ground plane will receive ambient light.

The accessibility can be precomputed by using ray casting.

To use the AO values in a render, direct illumination is scaled down by the obscurance. This gives recessed pieces of geometry a darker appearance.

[16] In this chapter it is important to remember that an *open* direction is an unblocked one

Figure 5.2: Man rendered without (left) and with (right) ambient occlusion, from [39]

Ambient occlusion generally adds a sense of depth to a render. For light sources that are *extremely* spread out, like a cloudy sky, AO looks fairly reasonable, somewhat resembling radiosity with the soft shadows that it adds. There are two main problems with AO:

i) AO shadows are uncolored (gray). Shadows should actually be colored by the color of the occluder, assuming the occluder is lit.

ii) When there a lot of directionality to the light source, AO fails to indicate the direction of the light source very well.

Both of these problems with AO are corrected by both VO (covered in chapter 7) and SSDO (in section 5.2). Note that the directions for which light is blocked are dissolved under the integral where AO is precomputed. This is where VO makes its biggest improvement upon AO - simply by remembering the *average* directions for which light is blocked, interreflected, and re-focussed to form a caustic.

5.1.1 Real-time ambient occlusion

A programmer on the Crytek engine found he could compute the accessibility of a fragment using only the depth buffer [37]. Crytek's method, as summarized in [10], is to

randomly sample depth values in a sphere around the pixel you are shading (in screen space).

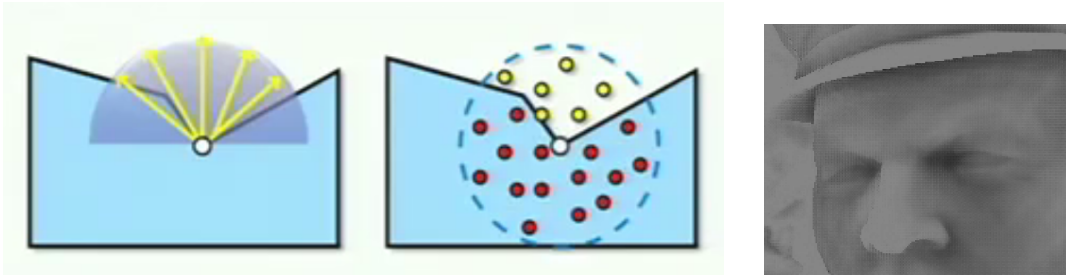


Figure 5.3: Crytek's method for SSAO centers a "sphere of dots" around the point whose AO we are estimating. On the right, we have a sample render of AO from [37].

Each (red, yellow) sample in 5.3 has a depth value. The red dots are "buried" by the surrounding geometry, while the yellow points are above it.

The accessibility of the white dot in 5.3 is then going to be equal to the ratio $\frac{Y}{T}$, where Y is the number of samples that are *closer to the camera* (ie not buried) than the actual fragment at that pixel (colored yellow in 5.3), and T is the total number of samples.

Other methods to compute AO in a deferred shader in real-time have also been developed including horizon-based SSAO [4], which uses per pixel normals and ray marching to determine the horizon angle of the nearest surrounding edges.

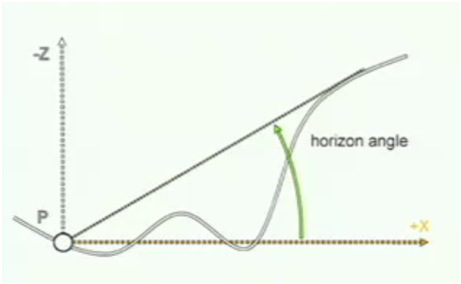


Figure 5.4: Determining the horizon angle at point P by ray casting. Multiple rays be cast to accurately determine the correct horizon angle. Image from [10].

5.2 Directional occlusion

Screen-space directional occlusion [44] is a real-time, dynamic method for calculating interreflections.

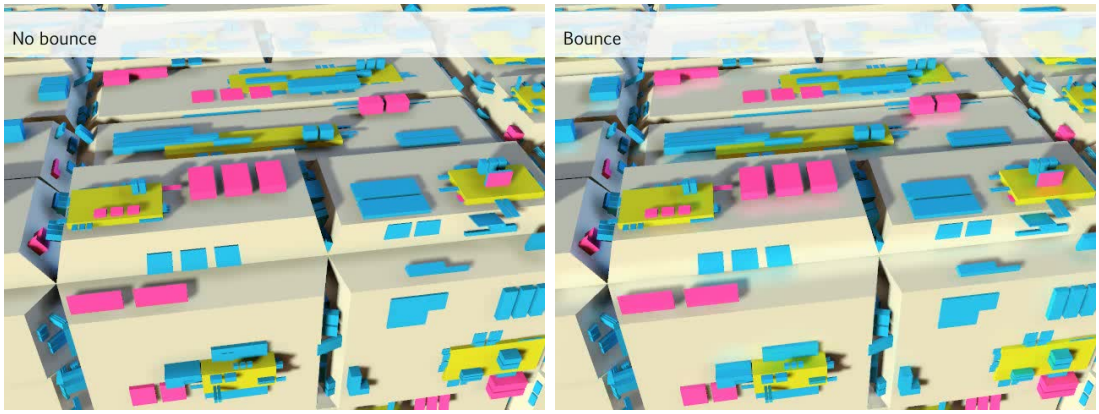


Figure 5.5: On the left we have an SSAO render, note the gray shadows. On the right, is Ritschel's SSDO render from [44] with bounced lighting. Note the "colored shadows" SSDO provides on the cream colored block as interreflections from the front side of the pink and blue cubes. These interreflections are strong in the front and weak in the back because of the location of the light source.

SSDO is very similar to SSAO. Basically SSDO "colors the shadows" with the color of the sender surface that is causing the shadow (assuming the sender surface is illuminated).

SSDO uses the clever observation that when the fragment at a sample location is occluded, as the red dots are in figure 5.3, *then the fragment at that sample location should not only block light, but* ***it should also send a diffuse interreflection***.

The diffuse interreflection is gathered by computing the blocking pixel's response to the environment map or light sources. This can be done in real-time with only about 10% additional processing overhead on top of SSAO [10].

Temporal coherence for SSDO as the viewpoint changes is good as well, there is little flickering in the demonstration video that accompanies [44].

Chapter 6

Precomputed radiance transfer

The basic ideas behind precomputed radiance transfer (PRT) were introduced in 3.3.2 - 3.4.3. PRT relies on a parameterization of a vertex's response to light, computed ***ahead of render time***, so that the shadowed, indirect illumination response can be found using only the input lighting and information already precomputed and stored at that vertex.

We will find what *directions* receive shadows, and what *directions* receive interreflections. Then we will write a *function* (called the transport operator) that can compute the final lighting at a vertex given the incident lighting *in one step* using the precomputed transport operator data.

PRT as defined by Sloan [50] does 3 important things to make real-time evaluation of global illumination possible:

1. Assume that scene lighting is ***infinitely distant***, so that a per-vertex transformation of the light source is not necessary

2. Define the ***transport operator***

3. Write the light source and transport operator coefficients in an orthonormal basis that requires ***few basis coefficients*** to represent the data accurately

6.1 Infinitely distant light sources

Say we place a light source inside of a room.

Figure 6.1: Lighting at every point in this room is dependent on *position*

Every point in the room in 6.1 sees the light source as coming from a different direction. To make the problem of global illumination simpler, we want L to be *the same* for every vertex in the scene. So, we take L infinitely far away, and make L very bright.

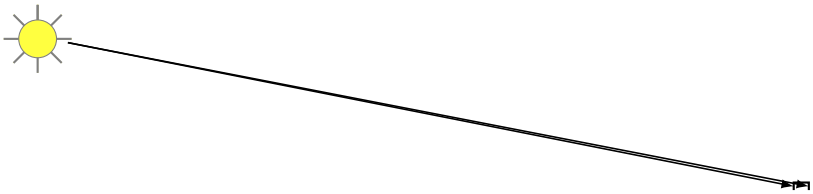


Figure 6.2: A very far light source appears to come from the same direction for every point in the scene

Now the light source loses its dependence on scene position x, and we simplify the rendering equation to

$$L(x, \vec{\omega_o}) = \int_S L_i(\vec{\omega_i}) f_r(x, \vec{\omega_i} \rightarrow \vec{\omega_o}) V(x, \vec{\omega_i}) G(x, \vec{\omega_i}) d\vec{\omega_i} \tag{6.1}$$

as shown in [27].

6.2 Transport operator

The transport operator is a function that accepts incoming light as a parameter and outputs the lighting at a vertex that would reach our eye.

From the rendering equation (3.2):

$$L(x, \vec{\omega_o}) = L_e(x, \vec{\omega_o}) + \int_S L(x', \vec{\omega_i}) V(x, x') G(x, x') f_r(x, \vec{\omega_i} \rightarrow \vec{\omega_o}) d\omega_i$$

Define a ***transport operator*** [27]:

$$T(\vec{\omega_i}, \vec{\omega_o}) = V(x, \vec{\omega_i}) G(x, x') f_r(x, \vec{\omega_i} \rightarrow \vec{\omega_o}) \tag{6.2}$$

$T(\vec{\omega_i}, \vec{\omega_o})$ groups together the BRDF response, the visibility function, and the geometry factor. We can now write the rendering equation in terms of L and T

$$L(x, \vec{\omega_o}) = \int_S L(x', \vec{\omega_i}) T(\vec{\omega_i}, \vec{\omega_o}) d\vec{\omega_i} \tag{6.3}$$

Equation (6.3) says that the light output at a point x over outgoing solid angle $\vec{\omega_o}$ is the integral over the sphere of the light function with the transport operator. If L is infinitely distant, then using the infinitely distant light assumption (6.1), we can further simplify (6.3) so that the light has no dependence on x

$$L(x, \vec{\omega_o}) = \int_S L_i(\vec{\omega_i}) T(\vec{\omega_i}, \vec{\omega_o}) d\vec{\omega_i} \tag{6.4}$$

If L and T are represented in an orthonormal basis, then the integral in (6.4) becomes a dot product.

$$L_o(x, \vec{\omega_o}) = \sum_k L_k T_k \tag{6.5}$$

If we ***precompute*** T, and L *stays the same* for each vertex under our infinitely distant light source assumption, then global illumination can be evaluated quite simply in a vertex shader. T will be stored as per-vertex data (possibly as TEXCOORDs), and L will be stored a uniform parameter (the same for every vertex).

This dot product in (6.5) will clearly be fastest ***when*** k ***is small***. Our choice of basis to represent L and T therefore is quite important. Good quality lighting results are produced with small k *if* the orthonormal basis chosen can represent a spherical

function reasonably accurately using only a few basis function coefficients. We will see spherical harmonics is able to approximate spherical functions quite well with relatively few basis coefficients. We will also see wavelets can also represent spherical functions quite accurately with few coefficients. With wavelets, we will consider the spherical functions as 2D functions on the 6 faces of a cube, or as a contiguous lattice over the 6 faces of the cube.

6.3 Orthonormal basis functions

A family of orthonormal basis functions are a group of functions ψ such that the inner product of any two of these basis functions is always either 1 or 0. Specifically[17],

$$\langle \psi_x, \psi_y \rangle = \int \psi_x \psi_y = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{if } x \neq y \end{cases} \tag{6.6}$$

There are many sets of orthonormal basis functions available, including the Fourier series, and different families of wavelets such as Haar and Daubechies.

6.3.1 Integrating orthonormal basis functions

A special thing about orthonormal basis functions is integration becomes very simple. When f and g have been projected into the orthonormal basis ψ, they will be represented as a linear combination of ψ's basis functions.

$$f = 8\psi_0 + 5\psi_1 + 7\psi_2$$

$$g = 2\psi_0 + 4\psi_1 + 6\psi_2$$

Now, the integral $\int fg$, when computed in the ψ basis, will work out to be

$$\int fg = \int (8\psi_0 + 5\psi_1 + 7\psi_2)(2\psi_0 + 4\psi_1 + 6\psi_2)$$

[17] The integral must be taken over some domain of ψ_x and ψ_y and the ψ should be a function of some independent variable, say t, but we chose a simplified presentation here.

If we expanded this, we would have an explosion of terms

$$\int fg = \int (8\psi_0 2\psi_0 + 8\psi_0 4\psi_1 + 8\psi_0 6\psi_2) + \cdots$$

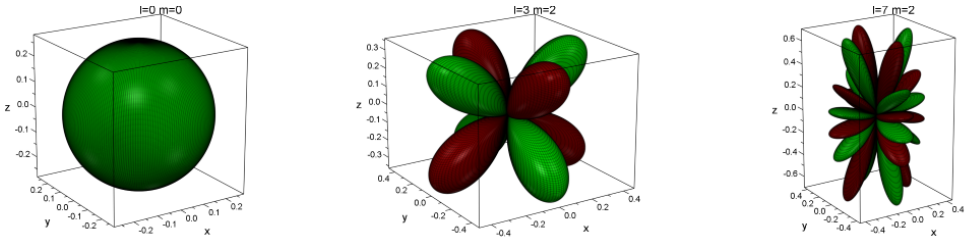
However, the integral $\int \psi_0\psi_0 = 1$, and $\int \psi_0\psi_1 = 0$, $\int \psi_0\psi_2 = 0$, by the definition of orthonormal basis function in (6.6). Thus the integral $\int fg$ actually equals the dot product of the ψ basis coefficients, or

$$\int fg = (8)(2) + (5)(4) + (7)(6)$$

This fact greatly simplifies runtime evaluation when representing L and T in an orthonormal basis.

6.4 Spherical harmonics

We need a set of orthonormal basis functions that can accurately represent a function defined on the sphere with only a few basis coefficients. Spherical harmonics (SH) are one such set of orthonormal basis functions.



(a) $\ell = 0$, $m = 0$ (b) $\ell = 3$, $m = 2$ (c) $\ell = 7$, $m = 2$

Figure 6.3: Some examples of spherical harmonic basis functions. Green values are positive and red are negative. Spherical harmonic approximations of functions are weighted sums of these basis functions. "Spikier" functions with higher ℓ values can retain higher frequency data.

6.4.1 Definition of spherical harmonics

The real spherical harmonics are defined as follows, with $\theta \in [0, \pi]$ as the inclination angle measured from the zenith and $\phi \in [0, 2\pi]$ as the azimuthal angle:

$$Y_\ell^m = \begin{cases} \sqrt{2} K_\ell^m \, P_\ell^m(\cos\theta) \cos(m\phi) & \text{if } m > 0 \\ K_\ell^0 \, P_\ell(\cos\theta) & \text{if } m = 0 \\ \sqrt{2} K_\ell^{-m} \, P_\ell^{-m}(\cos\theta) \sin(-m\phi) & \text{if } m < 0 \end{cases} \tag{6.7}$$

ℓ with $\ell \geq 0$ is called the *band* index, and m must have values satisfying $-\ell \leq m \leq \ell$. $P_\ell(x)$ is the Legendre polynomial

$$P_\ell(x) = \frac{1}{2^\ell \, \ell!} \frac{\partial^\ell}{\partial x^\ell} (x^2 - 1)^\ell \tag{6.8}$$

with $\ell \geq 0$, and $P_\ell^m(x)$ are the *Associated* Legendre polynomials

$$P_\ell^m(x) = (1 - x^2)^{\frac{m}{2}} \frac{\partial^m}{\partial x^m} P_\ell(x) \tag{6.9}$$

with $\ell \geq 0$ and $0 \leq m \leq \ell$ (note in (6.7) the m's sign is reversed when $m < 0$). K_ℓ^m is the normalization factor

$$K_\ell^m = \sqrt{\frac{(2\ell + 1)}{4\pi} \frac{(\ell - |m|)!}{(\ell + |m|)!}} \tag{6.10}$$

Multiplication by the normalization factor in (6.7) is required to make spherical harmonics an orthonormal basis.

The Legendre polynomials

The Legendre polynomials (6.8) are orthogonal on the interval $-1 \leq x \leq 1$. The degree of the ℓ^{th} Legendre polynomial is equal to ℓ. A few example polynomials are $P_0(x) = 1$, $P_1(x) = x$, and $P_2(x) = \frac{1}{2}(3x^2 - 1)$.

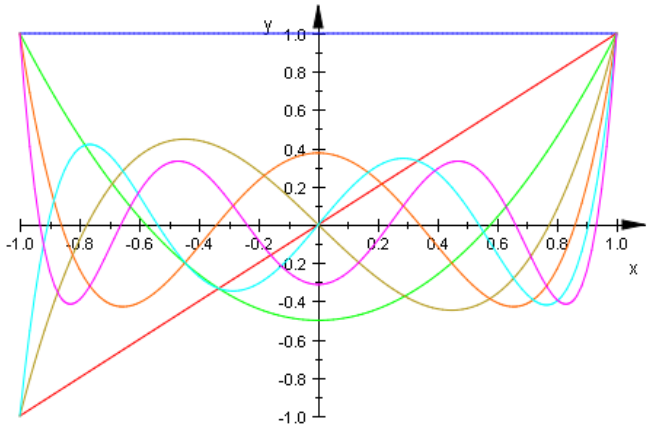


Figure 6.4: Legendre polynomials $P_\ell(x)$, $\ell = 0$ to $\ell = 6$ and $-1 \leq x \leq 1$. Higher ℓ means a higher degree polynomial (curvier)

The Legendre polynomials can be used as orthogonal basis functions. Any 1D function can be approximated as a weighted sum of Legendre polynomials.
The *associated* Legendre polynomials $P_\ell^m(x)$ in (6.9) are defined in terms of the "regular" Legendre polynomials $P_\ell(x)$. The *associated* Legendre polynomials create 2ℓ additional functions per band level ℓ, for a total of $2\ell + 1$ functions per band level.

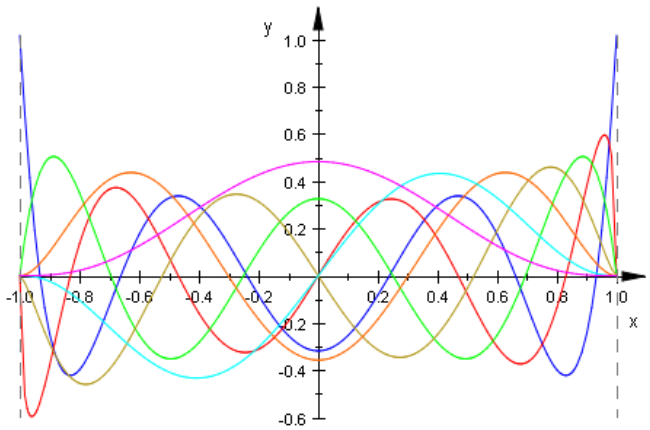


Figure 6.5: Graph of $K_6^m P_6^m(x)$, all of the Associated Legendre polynomials multiplied by their normalizing factors for $\ell = 6$ ($m = 0$ to $m = 6$). Higher magnitudes of m on a specific ℓ band produce the functions with *fewer* oscillations in that ℓ band.

In (6.9), we should note that if $m > \ell$, then $P_\ell^m(x) = 0$ due to the derivative. Thus we

must always have $|m| \leq \ell$.

Cordon-Shortley phase factor in associated Legendre polynomial

It is important to note that there are *two* popular flavors of associated Legendre polynomial in the literature. The definition we have provided in (6.9) is the definition that we will use in all of our work. The other definition, however, has an *additional* factor of $(-1)^m$, called the Cordon-Shortley phase factor:

$$P_\ell^m(x) = (-1)^m (1-x^2)^{\frac{m}{2}} \frac{\partial^m}{\partial x^m} P_\ell(x)$$

This distinction is easy to overlook and a common source of error[18]. For instance, the Ivanic-Ruedenberg SH rotation matrices [20] uses *no Cordon-Shortley phase factor* in its definition of spherical harmonic rotation matrices, while other SH graphics papers such as [17] do use it.

Use of associated Legendre polynomials in spherical harmonics definition

The associated Legendre polynomials, as used in the spherical harmonic functions (6.7), take as their *input* variable x the cosine of the elevation angle θ.

[18]The Cordon-Shortley phase factor comes from quantum mechanics and it has meaning for quantum physicists, but it does not have meaning for our application in computer graphics, since we are not using the SH functions as a quantum physics model.

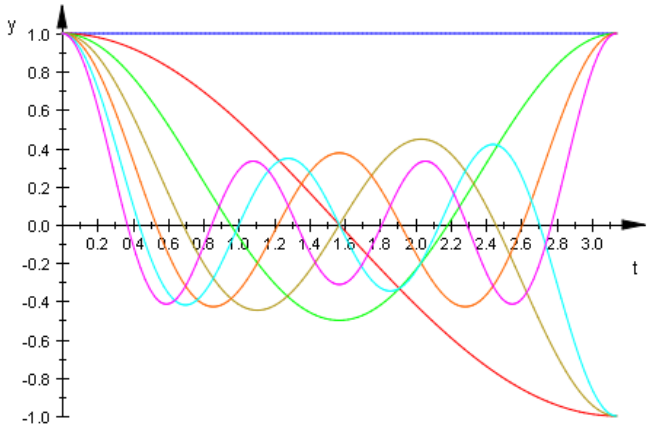


Figure 6.6: Legendre polynomials $P_\ell(\cos(\theta))$, $\ell = 0$ to $\ell = 6$ and $0 \leq \theta \leq \pi$. The output of a cosine has been substituted for x.

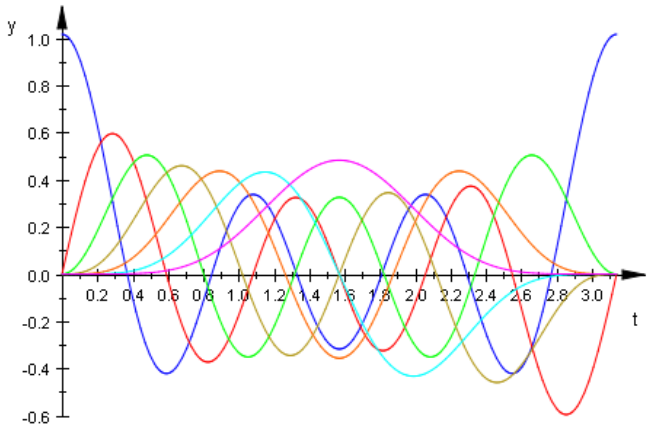


Figure 6.7: Normalized Associated Legendre polynomials $K_6^m P_6^m(\cos(\theta))$, $m = 0$ to $m = 6$. The output of a cosine has been substituted for x.

Visualizing the spherical harmonic functions

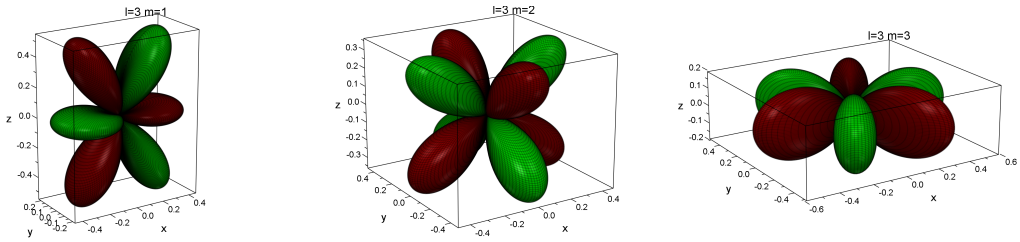
Dropping the constant terms from the spherical harmonic equations in (6.7), we have for $m > 0$

$$P_\ell^m(\cos\theta)\cos(m\phi)$$

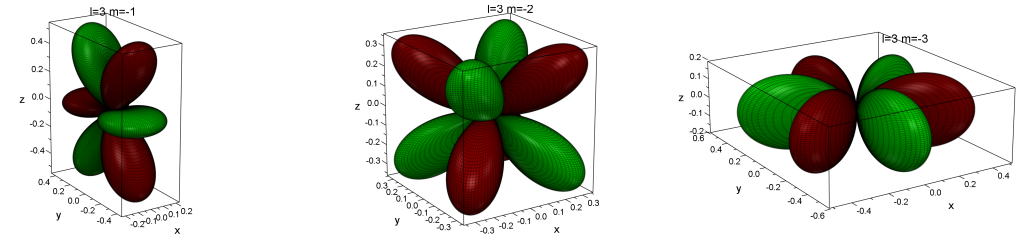
That is, these spherical harmonic functions are just the product of an associated Legendre polynomial, whose independent variable is the output of the cosine of the

inclination angle θ, and a sinusoid varying in azimuthal angle ϕ around the equator of the sphere. The frequency of the "equatorial sinusoid" increases with m. The equatorial sinusoid is simply phase shifted for negative values of m. For $m < 0$,

$$P_\ell^{-m}(\cos\theta)\sin(-m\phi)$$



(a) $\ell = 3$, $m = 1$ (b) $\ell = 3$, $m = 2$ (c) $\ell = 3$, $m = 3$



(d) $\ell = 3$, $m = -1$ (e) $\ell = 3$, $m = -2$ (f) $\ell = 3$, $m = -3$

Figure 6.8: Graphs of spherical harmonic functions for $\ell = 3$, various values of m. As we increase m, SH functions will have more lobes around the equator but fewer lobes in the vertical direction

The associated Legendre polynomials "flatten" as we increase m, due to the derivative (6.9) and reduction in the degree of the polynomial, as shown in figure 6.5.

The SH functions have no ϕ dependence for any of the *zonal harmonics*, the SH functions with $m = 0$:

$$P_\ell(\cos\theta)$$

Note that these *zonal harmonics* are circularly symmetric about the polar axis.

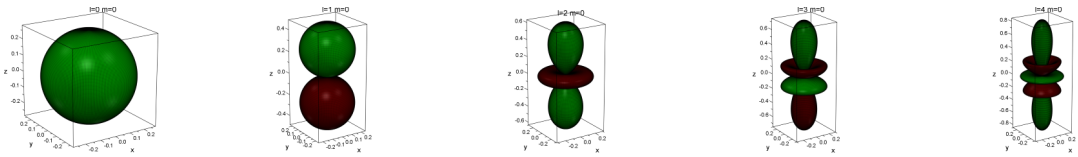


Figure 6.9: Zonal harmonics for increasing ℓ, from $\ell = 0$ at left to $\ell = 4$ at the right. The zonal harmonics are circularly symmetric about the polar axis.

Sloan noted the circular symmetry makes rotations for the zonal harmonics simpler and faster than for the other harmonic functions, and exploited this fact in a paper written on locally deformable PRT in 2005 [51].

6.4.2 Projecting into SH

Any function that can be represented on the 2-sphere can be projected into spherical harmonics. Like a Fourier decomposition, the more spherical harmonic basis functions we use, the better our approximation of the original function.

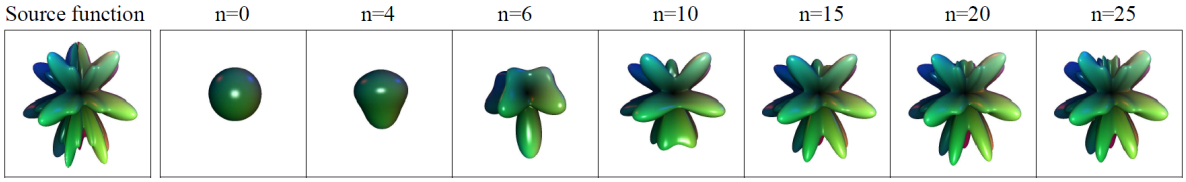


Figure 6.10: A source function and its representation with n spherical harmonic basis coefficients, from [45]

Here, "high frequency" refers to rapid oscillation on the surface of the sphere, or "sharp detail." Spherical harmonics does a good job of representing sharp detail, but as with the Fourier series for 1D functions, at increasing cost.

To see the value of using a spherical harmonic basis for representing spherical functions, consider how many Cartesian coordinate space points or (r, θ, ϕ) values you would need to represent any of the shapes in 6.10. The meshes in figure 6.10 are fairly high resolution, so if we use 128 slices (orange slices, or longitude) and 128 stacks (latitude)

on a sphere and store *only* the r values for each (θ, ϕ) direction on the sphere, we would need to store $126 \times 128 = 16,128$ distinct r values (two of the stacks are subtracted – the poles only have 1 value each).

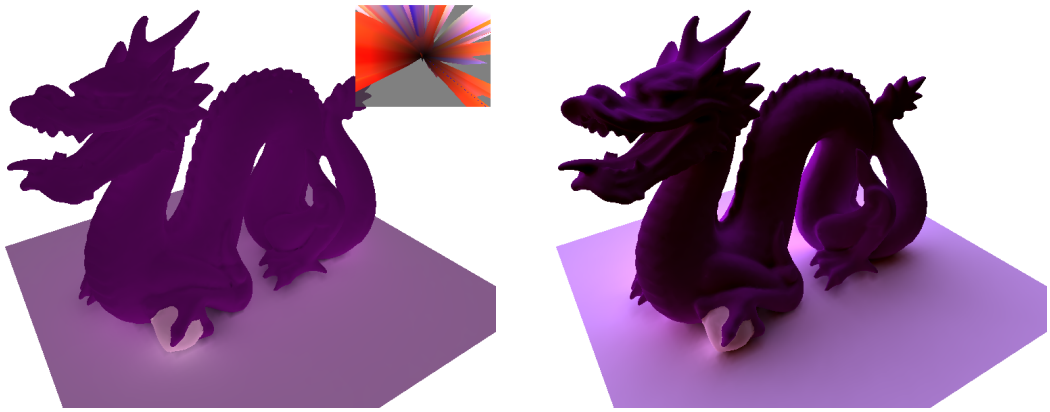
Consider also that these shapes are exactly represented in the spherical harmonic basis using only $(n+1)^2$ SH basis coefficients (where n is the number atop each image in 6.10). The reasonable $n = 10$ approximation to the original function only takes 121 SH basis coefficients to represent that shape exactly in the spherical harmonics basis. From the SH basis coefficients we can construct the visualization mesh at any time. We usually work entirely in spherical harmonics however, without ever needing to project back into the spatial domain.

In our uses in computer graphics, projecting into spherical harmonics is an extremely effective form of ***data compression*** for all types of spherical functions, including visibility and BRDF functions.

6.4.3 Spherical harmonics and the rendering equation

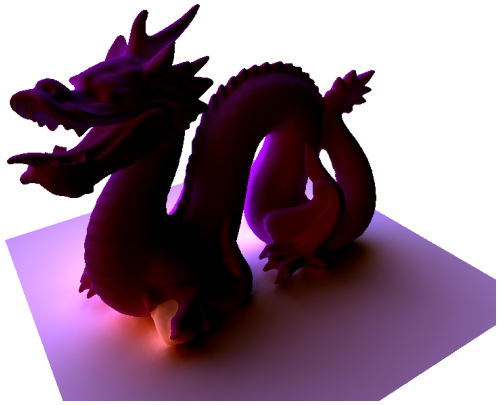
Once we project our light function L and our visibility function T into spherical harmonics, we can evaluate the lighting at each vertex by evaluating (6.5). We shall see spherical harmonics produces good lighting results even when we only use the first few SH bands.

Examples of a simple shapes scene in figure 6.11 for SH bands 1 through 8. Note the sharpness of the shadows and clarity of the interreflections as we increase the number of SH bands.



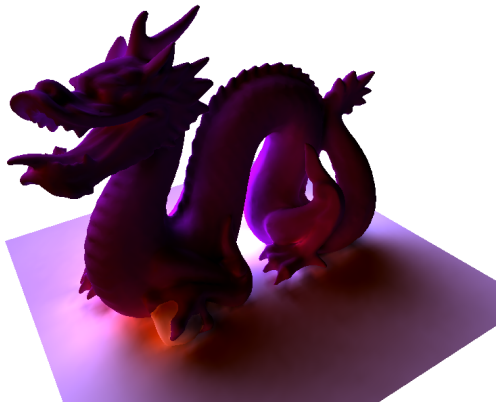
(a) SH 1 band, 1650 fps

(b) SH 2 bands, 1170 fps

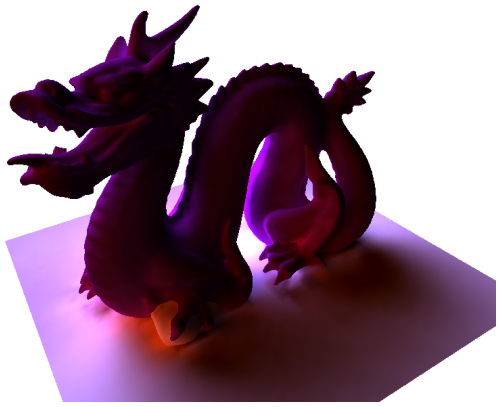


(c) SH 3 bands, 700 fps

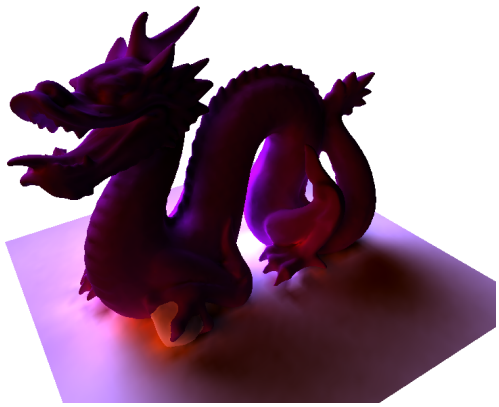
(d) SH 4 bands, 430 fps



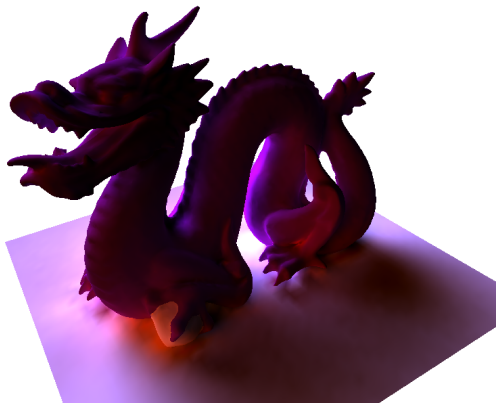
(e) SH 5 bands, 300 fps



(f) SH 6 bands, 200 fps



(g) SH 7 bands, 150 fps



(h) SH 8 bands, 114 fps

Figure 6.11: Diffuse SH. More SH bands generally means sharper shadows

1-band SH is very much like ambient occlusion, only it adds colored interreflections to the render. In 1-band SH, both the light source and visibility functions are scaled versions of the Y_0^0 function, which is simply a sphere (refer to 6.3a for the visualization of Y_0^0). As such, a 1-band SH light source is "stuck" being omnidirectional, radiating light equally in all directions. The only thing you can change about a 1-band SH light source is its color and magnitude. The occlusion functions are also omnidirectional, and they have different magnitudes depending on how "open" the vertex is to receiving ambient light. Again we note the similarity to AO 5.1. Rotating a 1-band SH light source can have no effect because you are simply rotating scaled spheres.

Using higher band SH functions allows us to keep some directionality for our light source and occlusion functions. Intuitively you should see that the sharper lobes in increasing band level ℓ in 6.3 mean preservation of sharper detail for both the light source and occlusion functions.

As we use more and increasing higher ℓ band SH functions, the occlusion and light functions are represented more exactly. However, we pay an increasing runtime and storage price for using more SH bands. Is the 600 fps cost worth the quality gain for going from 3 band SH to 8 band SH in figure 6.11?

After this study, one can see why Sloan and others present spherical harmonics as a method for "low frequency dynamic lighting." Higher frequency SH becomes intractable for real-time due to the number of coefficients we are required to store, and the number of multiplies we are required to execute per-vertex at runtime.

6.4.4 Spherical harmonics rotation

[20] neatly describes a method to implement spherical harmonics rotation given any regular 3×3 rotation matrix. Implementation code is provided in our code [48]. It should be noted that a few typographical errors have been spotted by Ivanic and others in the formulae in [20], and a correction note was issued [21], but there are still errors in

that correction paper. Email discussion with Ivanic allowed us to verify we have the correct formulation in our code, but a final errata notice for [20] and [21] have not been publicly posted yet to our knowledge.

The complexity of spherical harmonics rotation is quite high ($O(n^3)$ [28]), and so again, rotation is a place where we pay a nonlinear (cubic) increasing cost for using a higher number of spherical harmonics bands. As one would expect spherical harmonics rotation is very smooth, and there are no visible artifacts or discontinuities that arise due to rotating the SH basis functions, no matter the number of bands used.

6.5 Cpherical harmonics

We substituted the Chebyshev polynomial in the place of the Legendre polynomial in the definition of spherical harmonics. The result is a new orthonormal basis that we called the *cpherical harmonics*.

We define the cpherical harmonics as:

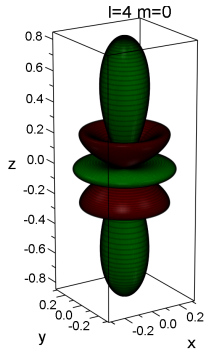
$$C_\ell^m = \begin{cases} \sqrt{2} A_\ell^m \, T_\ell^m(\cos\theta) \cos(m\phi) & \text{if } m > 0 \\ A_\ell^0 \, T_\ell(\cos\theta) & \text{if } m = 0 \\ \sqrt{2} A_\ell^{-m} \, T_\ell^{-m}(\cos\theta) \sin(-m\phi) & \text{if } m < 0 \end{cases} \tag{6.11}$$

with $\ell \geq 0$, $-\ell \leq m \leq \ell$.

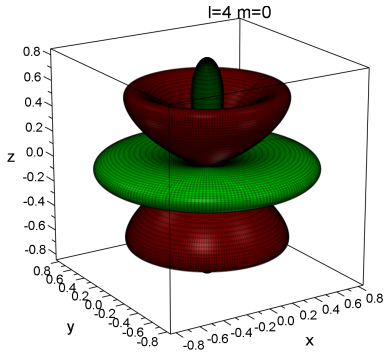
Here $T_\ell(x)$ is the Chebyshev polynomial, and we define the associated Chebyshev polynomial $T_\ell^m(x)$ analogously to the associated Legendre polynomial as

$$T_\ell^m = (1 - x^2)^{\frac{m}{2}} \frac{\partial^m}{\partial x^m} T(\ell) \tag{6.12}$$

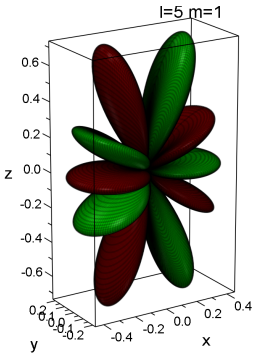
The formulae for the A_ℓ^m normalization constants differ significantly from the K_ℓ^m normalization constants of spherical harmonics. The analytical formula for A_ℓ^0 was



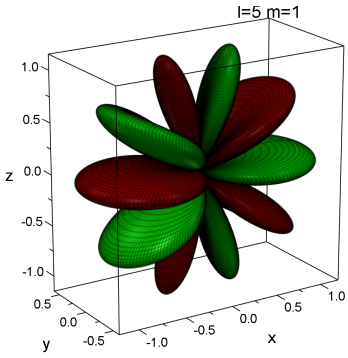
(a) SH $\ell = 4, m = 0$



(b) CH $\ell = 4, m = 0$



(c) SH $\ell = 5, m = 1$



(d) CH $\ell = 5, m = 1$

Figure 6.12: A couple of spherical harmonic (SH) and cpherical harmonic (CH) function graphs.

derived as

$$A_\ell^0 = \sqrt{\frac{2\ell+1}{4\pi}\frac{2\ell-1}{2\ell^2-1}} \tag{6.13}$$

The A_ℓ^m normalization constants used in A.1 however were determined numerically by integrating each C_ℓ^m function against itself over the sphere using a symbolic math package. We leave determination of A_ℓ^m normalization formula for arbitrary m for future work[19].

[19]The interested reader may refer to quantum mechanics texts, such as Cohen-Tannoudji for hints on how the spherical harmonics normalization factors are derived. In the interests of time we found it necessary to curtail the work, since stored constants for the A_ℓ^m were all we needed for our purpose.

We should note that while the associated Legendre polynomials solve a well known differential equation called the *general* or *associated Legendre equation*, the associated Chebyshev polynomials do not solve any similar well known system to our knowledge. Finding further meaning for the Associated Chebyshev polynomials may be considered for future work as well.

6.5.1 Computer implementation of cpherical harmonics

We have a couple of choices when it comes to the explicit expression we use for evaluating Chebyshev polynomials. We could use recurrence relations, the explicit expression involving roots, or a trigonometric expression. In our testing, it turned out that the cosine expression is the fastest to evaluate on a computer.
We define:

$$T_\ell(x) = \cos\left(\ell \cos^{-1}(x)\right) x \in [-1, 1]$$

To calculate the mth derivative of $T_\ell(x)$, we use the explicit formulation for the derivatives of the Chebyshev polynomials provided by [13]:

$$T_\ell^m(x) = \sum_{\substack{n=0 \\ (n+\ell-m) \text{ even}}}^{\ell-m} b_{n,\ell}^m T_n(x), \ell \geq m \tag{6.14}$$

with

$$b_{n,\ell}^m = \frac{2^m \ell}{(m-1)! c_n} \frac{(s-n+m-1)!(s+m-1)!}{(s)!(s-n)!} \tag{6.15}$$

$c_0 = 2$, $c_n = 1, n \geq 1$, and

$$s = \frac{\ell + n - m}{2}$$

See A.1 for reference implementation code.

(a) 6-band diffuse CH

(b) 6-band diffuse SH

(c) 6-band specular CH

(d) 6-band specular SH

Figure 6.13: 6-band cpherical harmonics (left) vs 6-band spherical harmonics (right) for different orientations of the light source. The renders very similar, as they should.

6.5.2 Spherical harmonics vs cpherical harmonics

We wanted to know if cpherical harmonics would offer sharper shadows than spherical harmonics. The graphs of the cpherical harmonic functions look similar to the spherical harmonic functions. The zonal harmonics for CH tend to be much wider than for their SH counterparts. Some graphs of the basis functions are shown in figure 6.12.

Runtime performance

Cpherical harmonics performs similarly to spherical harmonics in our runtime tests. As such, we didn't find any significant advantage to using the cpherical harmonics in our

work here. The resultant image quality is close (figure 6.13), the precompute time (projection into CH) is very similar, and the rotation matrices that work for spherical harmonics appear to work just as well for cpherical harmonics[20]. A 3-way numerical comparison of runtime performance of VO vs SH vs CH is in later section 7.6. The point of interest about cpherical harmonics that we believe warrants further exploration is the runtime cpherical harmonic evaluation code.

Future work could attempt to optimize the code in A.1, attempt to put the cpherical harmonic evaluation code on the GPU, and compare that code with an optimized spherical harmonics implementation (one such SH implementation can be found in [41]).

6.6 Wavelets

We explore alternative representations for spherical functions. We want to represent the light and visibility functions *more accurately* ***and*** with *less data*. In 2003, Ng [38] suggested we use a *non-linear wavelet approximation* to do so. Accuracy in the light and visibility function signals will preserve sharp shadows and solve the rendering equation more correctly. The less data we use for the lighting and visibility functions, the fewer multiplies we need to evaluate lighting at runtime (which is critical for interactive framerates).

As we saw in section 6.4.3, spherical harmonics has excellent performance for low frequency dynamic lighting. But, the large number of coefficients (and so, multiplies per vertex) required for higher frequency SH can make the frame rate drop below an acceptable level. In addition SH rotation becomes much more expensive at higher band levels. Ng showed that a non-linear wavelet approximation has excellent ability to retain all-frequency data with few data components and still maintain interactive frame rates.

[20] We should note explicitly here that confirming the derivation of the rotation matrices in CH is beyond the scope of this work.

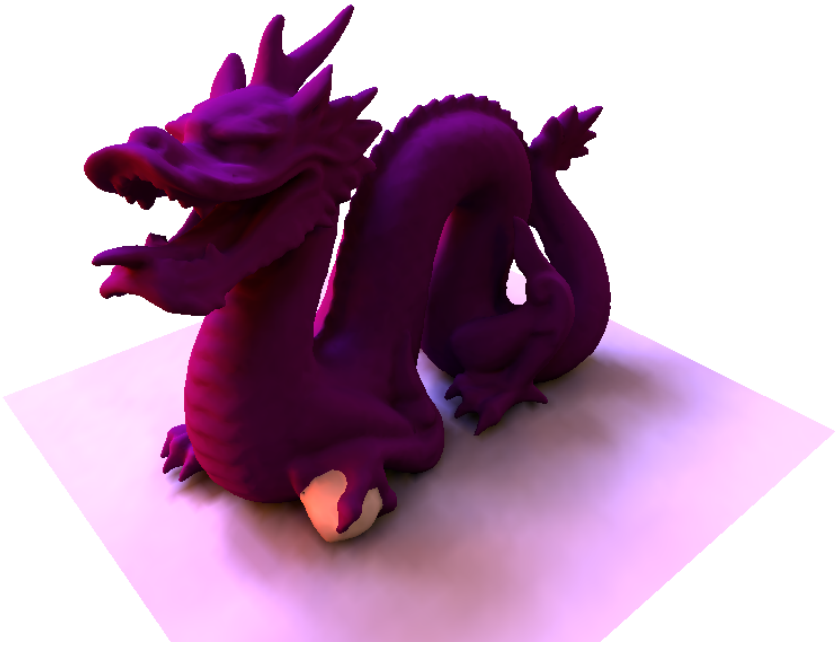


Figure 6.14: Sample wavelet render with shadows running at 8 fps, keeping 4.5% of original lighting coefficients (1100 components) and 0.6% of visibility function coefficients (150 components) at each vertex.

Non-linear wavelet approximation with lossy data compression

We will use a *non-linear* wavelet approximation for data compression. What this means is we project the T and L functions into the wavelet basis first, and then we simply drop all the near-zero coefficients. What we are left with are only the strongest components in the signal. This is highly lossy compression, but due to the wavelet transform's ability to decorrelate data [53], these dominant components are going to be enough to reproduce the original signal with acceptable quality, and to evaluate global illumination with acceptable quality as well. It turns out that for our purposes in rendering, we will only need about 4.5% of the lighting vector coefficients, and an average of 0.6% of each visibility function's coefficients.

6.6.1 Using wavelets to solve the rendering equation

The wavelets we use are going to be orthonormal bases, so when we project the light and visibility functions into the wavelet basis, we can evaluate global illumination with a simple dot product as described in 6.3.1, just as we did for spherical harmonics lighting. Wavelets produce excellent looking images, with sharp shadows and interreflections. An example render is in 6.14.

Wavelet transforms on the surface of the sphere are available [47] [30], but we did not use them here. Instead, we represented our spherical functions as 2D Haar wavelet transforms on the six faces of a cubemap, as did Ng [38]. A number of texts such as [52] cover the wavelet basis in detail. A spherical/cubemap function visualization is shown in figure 3.6.

6.6.2 Lifting scheme

There are two general approaches to performing a wavelet transform. The "traditional" approach uses the Fourier transform, and high and low pass filters. The more "modern" lifting scheme introduced by Sweldens [53] performs a series of additions and subtractions on the signal to transform it *in place*, without requiring the large amount of memory needed to make a transformed signal copy. The lifting scheme is used to keep precompute time low.

6.6.3 Wavelet rotation

[57] showed that it is possible to rotate a function in the wavelet domain. It is also possible to rotate the cubemap in the pixel basis and fast wavelet transform the result. For each texel in the rotated cubemap, we "reach back" for a source texel using an inverted rotation matrix. We can then perform a fast wavelet transform using lifting on the rotated cubemap.

However rotating in the pixel basis and transforming the result introduces its own set of problems, as we shall show.

Figure 6.15: Example of a rotated cubemap by resampling. On the left is the original source cubemap, and on the right is a rotated version.

Pixel-basis rotation followed by wavelet transformation works well, as long as the light vector and visibility functions are not compressed too heavily. If there is too much compression, then two undesirable artifacts emerge: blocky shadows, and "flickering". These two artifacts are shown in figure 6.16.

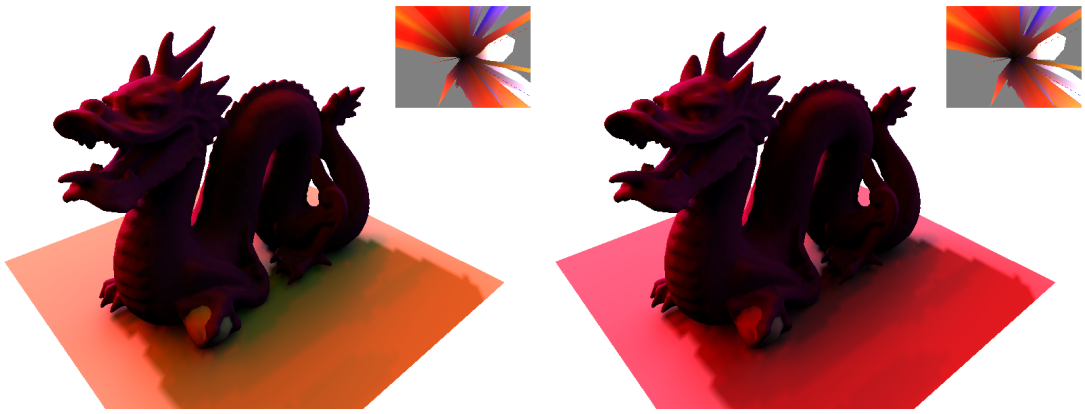


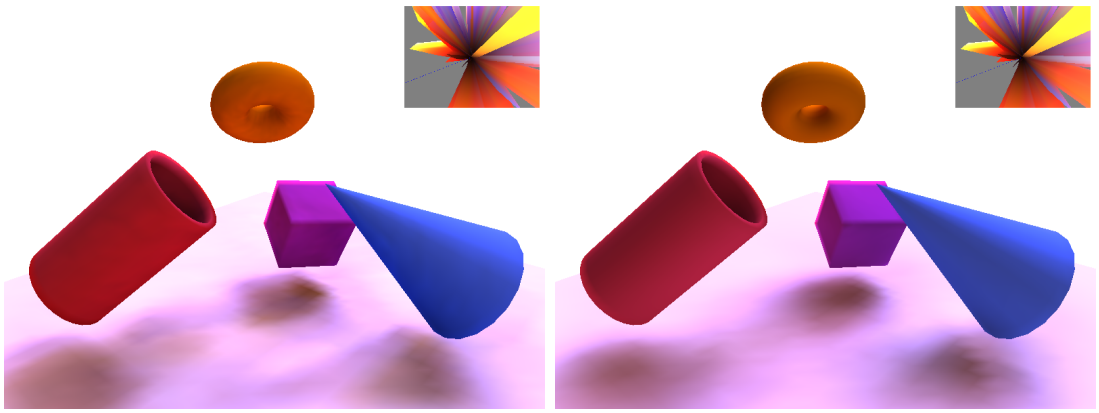
Figure 6.16: Pixel-basis cubemap rotation followed by heavy Haar wavelet compression gives good frame rates (40 fps), but introduces two problems: blocky shadows and flickering. This image was rendered using 0.7% of the light cubemap's coefficients (179 components), and 0.03% (10 components) of visibility function coefficients.

In figure 6.16, the cubemap has been rotated by only 1° between the left and right images, but there is a strong, very obvious jump in the color balance. This flickering problem does not appear as intensely if compression is not too heavy.

The reason for this flickering is because we used pixel-based cubemap rotations, under heavy compression, small changes to the source cubemap images can have large effects on the transformed and filtered result. We see a large change in the RGB color balance for small changes in rotation of the light source cubemap in figure 6.16 due to this effect. The wavelet-basis rotation matrices described in [57] may remedy this problem.

6.6.4 Wavelet interreflection

Although [38] did not mention this, it is possible to compute interreflections using wavelets using the same method that Sloan did with spherical harmonics in [50]. After the initial visibility functions are determined, rays are cast at each vertex, and the wavelet compressed visibility functions of surrounding geometry is simply added to the vertex's own visibility function. This produces the result in 6.17, which is highly comparable to the spherical harmonics result.



(a) Wavelet interreflection, 8 fps (b) 6 band SH interreflection, 1000 fps

Figure 6.17: Wavelet interreflection is possible, but it hurts the frame rate for wavelets. Adding in interreflections for SH has no effect the frame rate however.

Unlike spherical harmonics, however, *there is an additional runtime cost* to factoring interreflections into the visibility functions for wavelets, because it reduces the sparsity of the visibility functions at each vertex. SH does ***not*** experience a runtime hit for adding interreflections because the dimensionality of the SH vector is not increased by adding together two SH functions. For example, adding two 9-band SH functions, each with 81 components, is always just another 9-band SH function (with 81 components).

6.6.5 Spherical harmonics vs Wavelets

We identified two main problems with wavelets as compared to spherical harmonics that would make us prefer spherical harmonics for a real-time application over wavelets.

1. Wavelets become noisy when too few coefficients are used, whereas SH degrades gracefully into progressively softer shadows (a much more acceptable artifact)

2. Nonlinear wavelet approximation uses sparse matrices, but sparse matrix multiplies are not as fast as SH's dense vector multiplies for the same number of coefficients

Wavelet noise

Recall that to get real-time performance, we must keep only a very small number of basis coefficients, so that T and L are both small in (6.5). Wavelets' fast runtime hinges on compressing the data enough that it becomes extremely sparse. If you do not compress the data enough, the code will run very slow (because of the large number of multiplies that need to be executed in lighting).
The main point of difference between spherical harmonics and wavelets is ***the effect of using a courser approximation with fewer basis function coefficients***. In particular, wavelets have a very visible blocky, JPEG-like noise introduced for using too few coefficients, while SH gracefully degrades into smoother shadows.

We showed in figure 6.11 that using fewer SH bands resulted in softer shadows, but there are no obviously unacceptable "noisy artifacts" introduced into the renders with fewer SH bands.

On the other hand, as shown in figure 6.16, with wavelets the shadows tend to become blocky when too few coefficients were used. Blockiness is the signature of a highly compressed Haar wavelet. Using pixel-based rotations does not work very well under high compression because of the introduction of the "flickering" problem shown in figure 6.16. Wavelet-basis rotation [57] should be used instead.

Sparse matrix multiplies

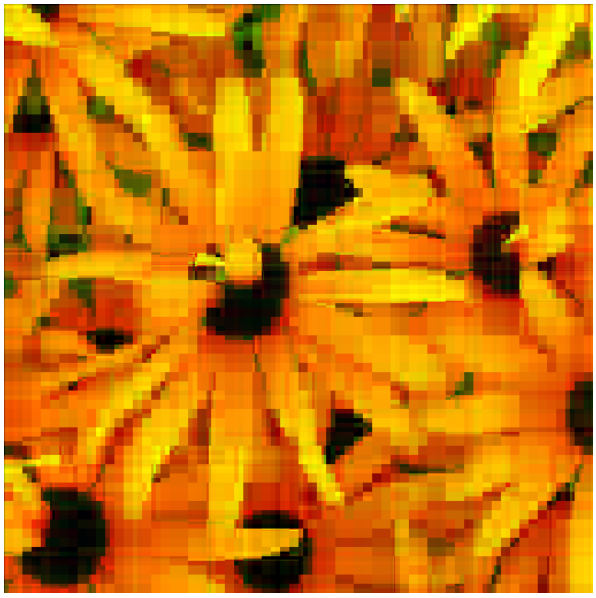
Dot products of sparse vectors, while faster than dense ones of the same size, are still more expensive than the dense, short vectors that spherical harmonics uses, due to the need to match components. Even with only a few coefficients to evaluate, index lookup adds to runtime cost because it essentially adds a *search* step to the execution of each multiply. Some work has been done on efficiently executing sparse matrix multiplies on the GPU [5], which would help improve the frame rate for wavelet lighting.

6.6.6 Quincunx lattice wavelet transform

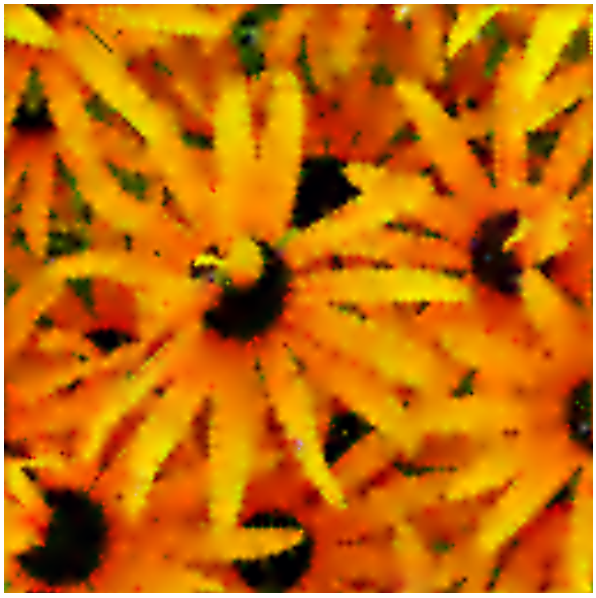
We performed some experiments involving the quincunx lattice transform [22] in an attempt to remedy the flickering issues cited above. Quincunx lattice transforms have interesting properties as compared to Haar, and it was thought they would be a better match for the case of a wrapping object such as a cubemap.

In order to perform the quincunx lattice transform on a cubemap, the cubemap must be made into a contiguous lattice by making each side have $N - 1$ pixels, and sharing an additional "rail" of pixels between neighbouring faces (colored red in 6.18).

This looks as follows:



(a) Haar wavelet, restored from 99.5% compression



(b) Quincunx lattice wavelet, restored from 99.5% compression

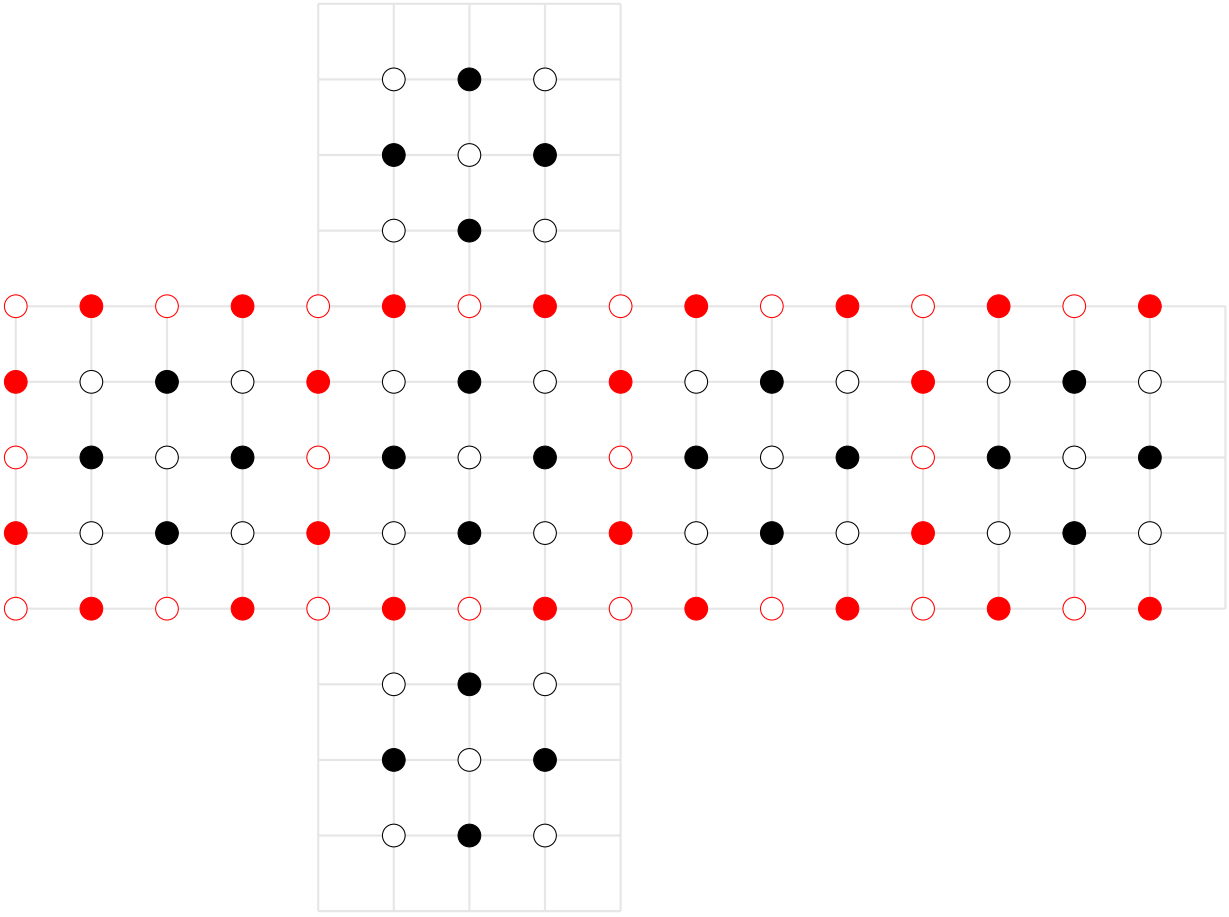


Figure 6.18: Layout for a cubemap with shared edges for quincunx lattice transform. The hollow points are "even" and the solid points are "odd"

The results of compressing with the quincunx lattice were much the same as using Haar wavelets, and they did not satisfactorily resolve the flickering issue under rotation in the pixel basis.

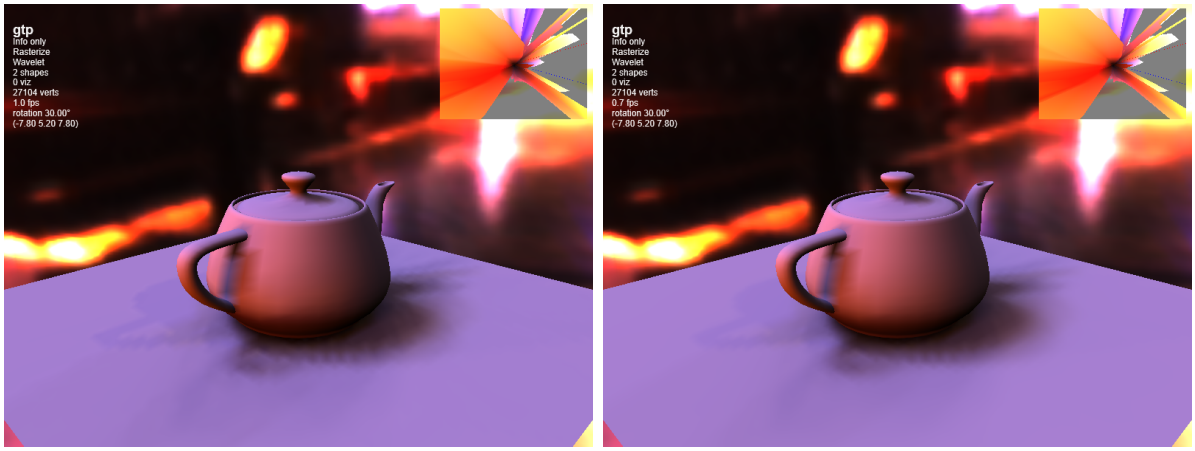


Figure 6.19: Quincunx lattice result (left) and Haar wavelet result (right) look very similar. The shadows in the quincunx lattice result are a bit darker, but both are very sharp. Both experienced flickering issues when the light source cube map was rotated.

6.7 Conclusion

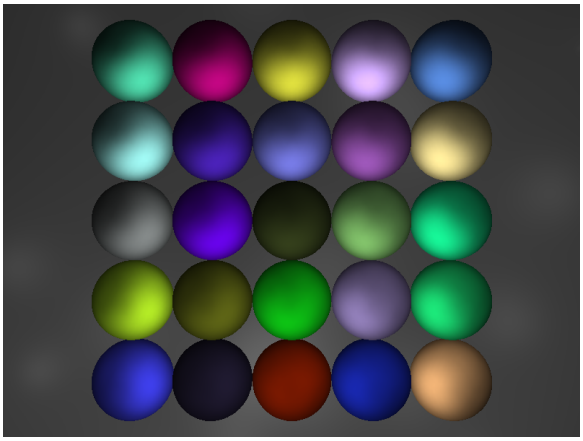
Precomputed radiance transfer is a great way to get real-time global illumination effects. Spherical harmonics seems the more viable alternative to wavelets due to the superior performance of dense vectors versus the sparse matrices of wavelets, and the *graceful* degradation of SH image quality when the number of coefficients used at each vertex is reduced, where wavelets tends to introduce an unpredictable noise.
PRT using spherical harmonics is already a real-time method, but it only works on static geometry. Attempts to make PRT work on deformable geometry have been made [51], and there was even some work done on using spherical harmonics in screen-space [19].

Chapter 7

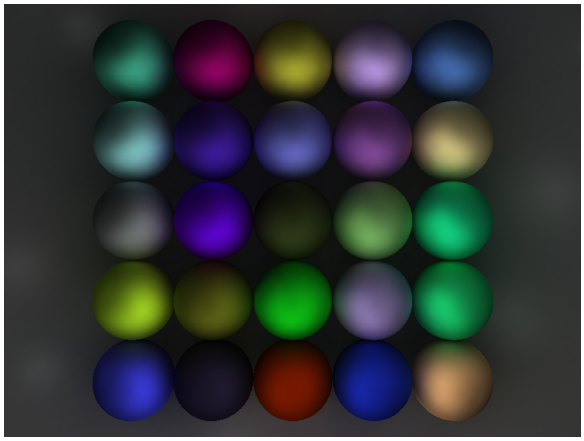
Vector occluders

Global illumination by vector occluders (VO) is a novel, ***empirical*** method for calculating real-time shadows, caustics, and interreflections on a static mesh with any number of dynamic light sources. VO uses ideas from radiosity (form factors), ambient occlusion, and ideas from precomputed radiance transfer [50] to empirically simulate global illumination effects in real-time. VO does this by storing not only a measure of "how blocked" each vertex is as ambient occlusion (AO) does, but it also stores *information about the blockers* at the vertex, including *the directions from which* blockage is occuring, the color of the blocker, and the average surface normal of the blocker. Others [61] [26] [34] have added directionality to precomputed AO through various means, but VO is the only work to our knowledge to include the caustic *diffuse from specular* and *specular from specular* interreflection terms.

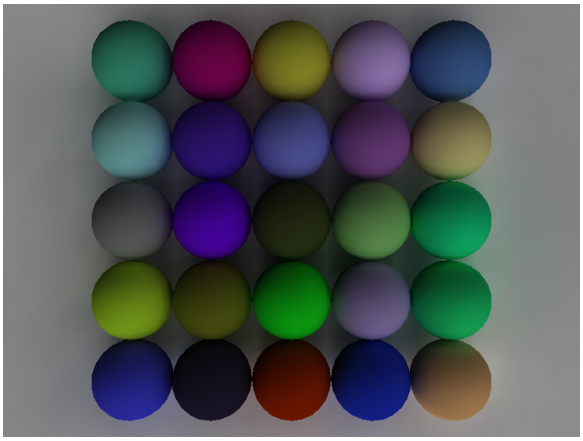
We must emphasize that VO is an *empirical* method. *Empirical*, for the purposes of computer graphics means the method doesn't necessarily generate a correct solution to the rendering equation, but nonetheless can generate visually pleasing, consistently *correct-looking* results. The Phong shading model itself is a canonical example of an *empirical* method.



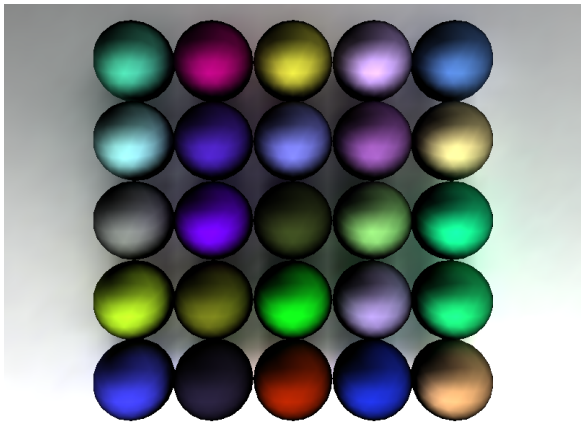
(a) Phong per-vertex shader @ 2150 fps



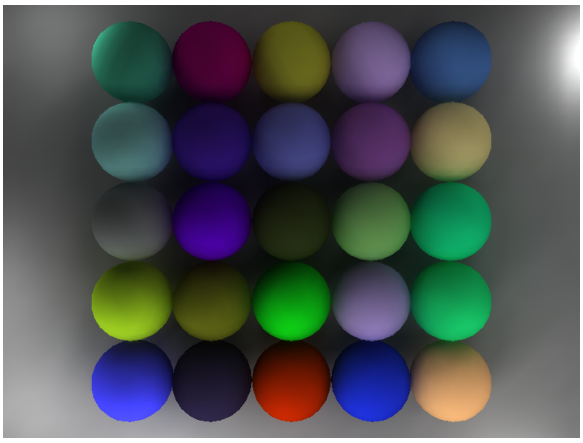
(b) VO @ 66 fps



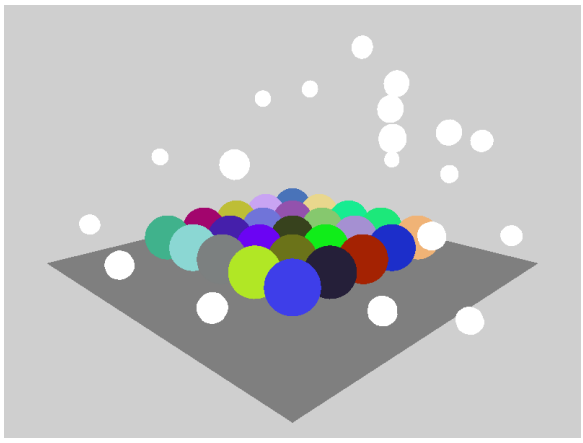
(c) SH diffuse GPU @ 610 fps



(d) SH glossy CPU @ 14 fps



(e) Baked-in radiosity @ 2400 fps



(f) Unshaded view of scene arrangement Lights are shown as spheres of the same size.

Figure 7.1: 25k vertices and 52k polygons under 20 point lights.

7.1 VO: intuition

If the form factors stay the same, then changing the lighting is trivial and fast. We exploit the same principle that orther PRT methods do and require static meshes. We make extensive use of form factors in our estimation of shadow darkness, caustic strength, and interreflection strength. Recall from 4.2.3 that the form factor of a surface A (e.g. your finger) with another surface B (e.g. your eye) is proportional to the solid angle that A takes up of B's view. A will *appear large* to B if either:

i) A is very close to B (e.g. your finger is very close to your pupil) *or*

ii) A is actually extremely large but further away from B (e.g. a giant's finger further away from your pupil)

It should be intuitive that the shadowing and interreflections "sent" by A to B are proportional to the form factor of A with B. All of VO's shadow, caustic, and interreflection computations are based on form factors.

7.1.1 Directions to occluders

The main idea of VO is to store *average directions, colors and normals of all the visible occluders*[21] as data local ***at each vertex*** (hence the name *vector occluders*). An OCCLUDER can be any mesh of arbitrary complexity. Using only these vectors that store information about the OCCLUDERs, we will be able to compute empirical estimations of global illumination effects in a regular vertex shader. The way we obtain the vectors to the OCCLUDERs surrounding each vertex is in a raycasting precompute stage very similar to Sloan's method in [50]. We cast tens of thousands of "feeler rays" from each vertex to identify what geometry surrounds the vertex. We aggregate data *by*

[21] Or, *blockers*. "Blocker" is synonymous with "occluder"

OCCLUDER. This data is then fed through the GPU pipeline either as vertex texture coordinates, or as texels on an auxiliary texture.

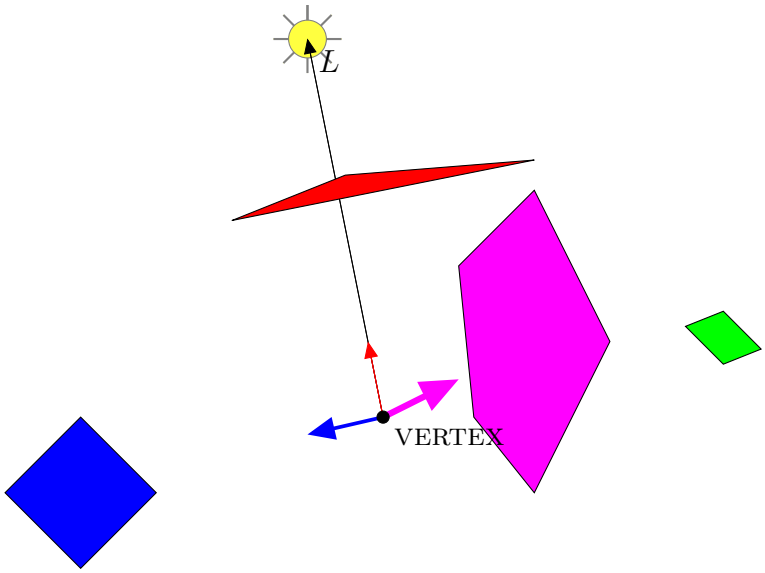


Figure 7.2: The vectors shown here are ***occlusion vectors***, the weight of these vectors is roughly proportional to the form factor that each occluder would have with VERTEX.

Consider the example scene in figure 7.2. There are 4 OCCLUDERS, the blue square, the purple pentagon, the green square, and the red triangle[22]. Say we are lighting the point VERTEX (normally VERTEX would be part of a surface, but for simplicity we are considering VERTEX an isolated point in space here).

It should be intuitive, based on the arrangement of the geometry and the position of the single light source L, that the red triangle in 7.2 casts a mild shadow on VERTEX and sends no interreflection to VERTEX[23]. The purple pentagon casts no shadow on VERTEX, but sends a considerable purple interreflection. The blue surface also casts no shadow on VERTEX, but sends a bit of a blue interreflection.

Depending on each occluder's *material properties*, and the current position of the light source, each occluder can cause up to three distinct global illumination effects to be

[22]Here we have defined an occluder as a single primitive, but this definition is flexible. You can break a single object into multiple occluders.

[23]No *first bounce* interreflection, that is.

displayed at VERTEX:

1. A ***shadow*** (attenuation of light) if the material of the OCCLUDER is opaque, and the OCCLUDER stands between VERTEX and the light source

2. A ***caustic*** (magnification of light) if the material of the OCCLUDER is translucent to any degree, and the OCCLUDER stands between VERTEX and the light source

3. An ***interreflection***, of which there are 3 types (see 2.3):

   (a) *Diffuse from diffuse*: OCCLUDER color bleeds onto VERTEX. This is the type of interreflection modelled by radiosity (see figure 4.3)

   (b) *Diffuse from specular*: OCCLUDER sends a specular reflection to VERTEX, and VERTEX has a diffuse response to the reflected light (see figure 2.6)

   (c) *Specular from specular*: OCCLUDER sends a specular reflection to VERTEX, and VERTEX sends a specular reflection to the eye (see figure 7.3).

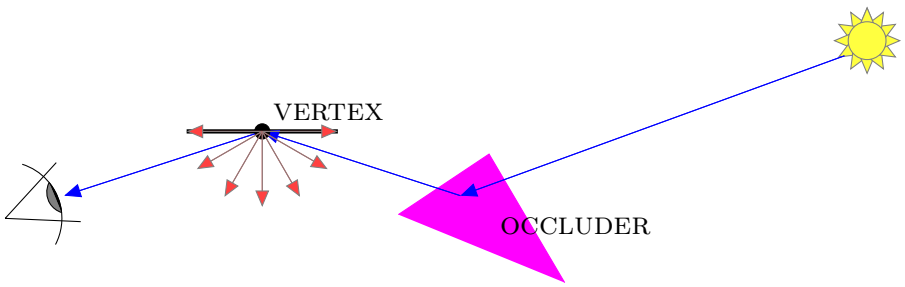


Figure 7.3: **Secondary specular effects**. The magenta OCCLUDER sends a specular reflection to VERTEX. VERTEX has ***two responses***: a diffuse response (indicated by red radiating arrows) and a specular response (blue arrow that leads towards eye).

7.2 The components of VO

Now we delve into a more complete description of the components that VO uses. For each VERTEX, we store the following eight[24] vectors for each OCCLUDER (or *reflector group*)[25] ***that*** VERTEX ***can see***:

1. **Position vector**: Average position *of visible portion of* OCCLUDER

2. **Occlusion vector**: Average directions for rays cast that reached OCCLUDER. This will be the average direction for which light is blocked due to OCCLUDER, and it is used in the shadow strength calculation

3. **Caustic vector**: Average direction from which a caustic is received (after diffraction through OCCLUDER *and* bouncing or passing through all other surface geometry until reaching free space). If a light source comes from this direction, you get a caustic at VERTEX.

4. **Transmissive color**: Average color OCCLUDER can transmit. Affects both *shadow* color and *caustic* color

5. **Diffuse normal**: Average normal of diffuse reflecting faces on OCCLUDER ***that*** VERTEX ***can see***. Used for diffuse-from-diffuse interreflection.

6. **Specular normal**: Average normal of specularly reflecting faces on OCCLUDER. In order for a feeler ray to count, it must bounce off the face and reach free space (ie reflection direction *should not be blocked).* This is mainly why the specular normal is different from the diffuse normal (diffuse normal does not check if it is open on the reflection direction).

[24] This is a "full" representation. Simplified representations to conserve memory are possible. For example, caustics can be eliminated if all surfaces will be solid.

[25] Reflector groups which can be defined to be any size you like. We found good results with using one reflector group *per general shape or model.* However, an individual shape can be broken into any number of reflector groups that the model designer wishes.

7. **Diffuse color**: Average diffuse color of visible portion of OCCLUDER. Affects diffuse interreflected color

8. **Specular color**: Average specular reflective color of visible portion of OCCLUDER. Affects color due to specular interreflection.

We should make clear that we are not proposing to store only 8 vectors per-vertex and get GI effects with it. We are storing a copy of the 8 vectors listed above ***per*** OCCLUDER ***that*** VERTEX ***can see***. For the example in figure 7.2, $8 \times 3 = 24$ vectors would actually be stored at VERTEX, because VERTEX can see 3 of the 4 occluders in the scene.

So we see here that VO will have the same data-size vs quality trade off that spherical harmonics and wavelets had previously. The more occluders we use, the better quality the results will be. However, the more occluders we use, the more data we need to store, and the more data we store, the more computations we will end up running at lighting time.

7.2.1 The position vector of an occluder ($P_{\mathbf{avg}}$)

The position vector of an OCCLUDER is denoted P_{avg}. P_{avg} is the average position *of visible portion of* OCCLUDER. P_{avg} is found just as the average intersection point of the rays cast from VERTEX that hit OCCLUDER during the VO raycasting step.

7.2.2 Shadows

Occlusion vectors are found in the VO preprocess by casting thousands of rays from VERTEX, and summing together ***form-factor weighted directions*** for all the rays that hit the same occluder.

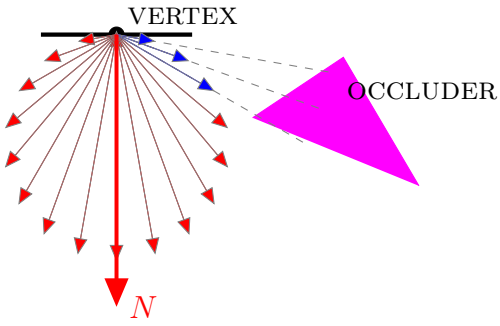


Figure 7.4: Raycasting form-factor weighted rays to find shadow vector at VERTEX due to OCCLUDER. The vectors with the blue heads are the ones that hit OCCLUDER

Weighting by form factors

We use the idea of form factor from radiosity in creating VERTEX's occlusion vectors for each OCCLUDER to get accurate-looking results. By Nusselt's analog (discussed in section 4.2.4), we can find the form factor of a piece of geometry by projection onto the hemisphere surrounding VERTEX, followed by projection down onto the base of the unit sphere surrounding VERTEX.

[55] shows that projection onto the cosine lobe is equivalent to Nusselt's analog, described in the paragraph above. The cosine lobe is described in figure 7.5.

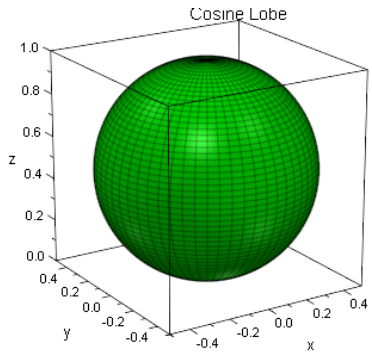


Figure 7.5: A graph of spherical function $r = \cos\theta$, $0 \leq \theta \leq \frac{\pi}{2}$, $0 \leq \phi \leq 2\pi$. We call this "the cosine lobe" in surrounding text. The integral of the cosine lobe is $\int_0^{2\pi} \int_0^{\frac{\pi}{2}} \cos\theta \sin\theta d\theta d\phi = \pi$.

What we essentially want to do with our raycasting is numerically evaluate the portion of the cosine lobe 7.5 that each occluder *blocks*. We just sum the form factors of the individual rays cast from VERTEX that hit OCCLUDER. That will give us the form factor of OCCLUDER from the vantage point of VERTEX. The ***occlusion vector*** is going to be the sum of the form-factor weighted ray directions that hit OCCLUDER. The form factor of each ray cast is equal to the dot product of each ray's direction with VERTEX's normal, multiplied by the solid angle each ray represents, divided by π.

$$F_{\text{ray}} = max\left((\vec{N}\cdot\vec{d}), 0\right)\ \frac{4\pi}{\text{Num rays cast}}\ \frac{1}{\pi}$$

$$F_{\text{ray}} = max\left((\vec{N}\cdot\vec{d}), 0\right)\ \frac{4}{\text{Num rays cast}} \tag{7.1}$$

Where $\vec{N}$ is the direction of the normal at VERTEX, and $\vec{d}$ is the direction vector of the ray cast. $\frac{4\pi}{\text{Num rays cast}}$ is the solid angle that each ray cast represents[26]. We discard rays that form an obtuse angle with the normal using the *max* function and dot product, because these obtuse rays don't match the cosine lobe. Physically, they shoot into the surface that VERTEX sits upon. We divide the result by π because the integral of a cosine lobe is π, and we want to normalize that value to 1. If an occluder has a form factor of 1 with VERTEX, then that occluder *is all* VERTEX *can see*.

Computing shadows

To compute shadowed lighting at VERTEX using the ***occlusion vector*** computed in the previous section, a second stage is added to the regular local illumination model. First, each light sources' direct illumination is computed at VERTEX, just as it would be for any local illumination model. After that, each OCCLUDER at VERTEX will ***"try" to attenuate*** the direct lighting received at VERTEX. Maximal attenuation occurs when

[26]It should be intuitive that integrating the unit sphere with these rays would give 4π, or the surface area of the unit sphere

the ***occlusion vector*** direction lines up with the vector from VERTEX to the light source. Physically that means the occluder sits directly between VERTEX and the light source. No attenuation occurs when the shadow vector is at a right or obtuse angle with the vector from VERTEX and the light source.

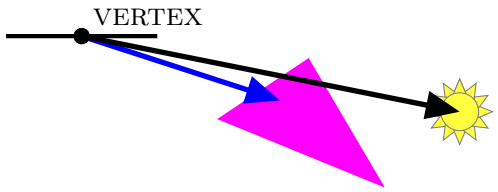


Figure 7.6: The purple polygon will attenuate an amount of light proportional to *both a)* how much the (blue) *occlusion vector* lines up with the black vector from VERTEX to the light, and *b)* the form factor of the purple polygon with VERTEX.

The important thing to note is that at each vertex, the unshadowed direct illumination due to *each* light is computed *first* (exactly as it would be for any local illumination model), and following that, the direct illumination is *attenuated* by each of the blockers that VERTEX can see. It is possible that the direct illumination get completely zeroed out by one of the blockers.
The scalar value of "how much" to attenuate the light at a VERTEX x[27] due to the OCCLUDER S is given by:

$$\text{blockageFactor} = \sin^{2N_h F_{xS}}(b) \tag{7.2}$$

Where:

- b is the angle between the ***occlusion vector*** of S and the *vector from x to the light source* $\overrightarrow{xL}$
- F_{xS} is the *form factor* of the OCCLUDER with the VERTEX point being lit x

[27] x will be taken to abbreviate VERTEX in the formulae, and S to abbreviate OCCLUDER

- N_{sh} is the *shadow strength adjustable parameter*. This parameter requires hand tuning, much like the shininess N_s term in the Phong lighting model. See figure 7.8.

Our use of the sine function here is an empirical choice that we found to produce good results in our experiments.

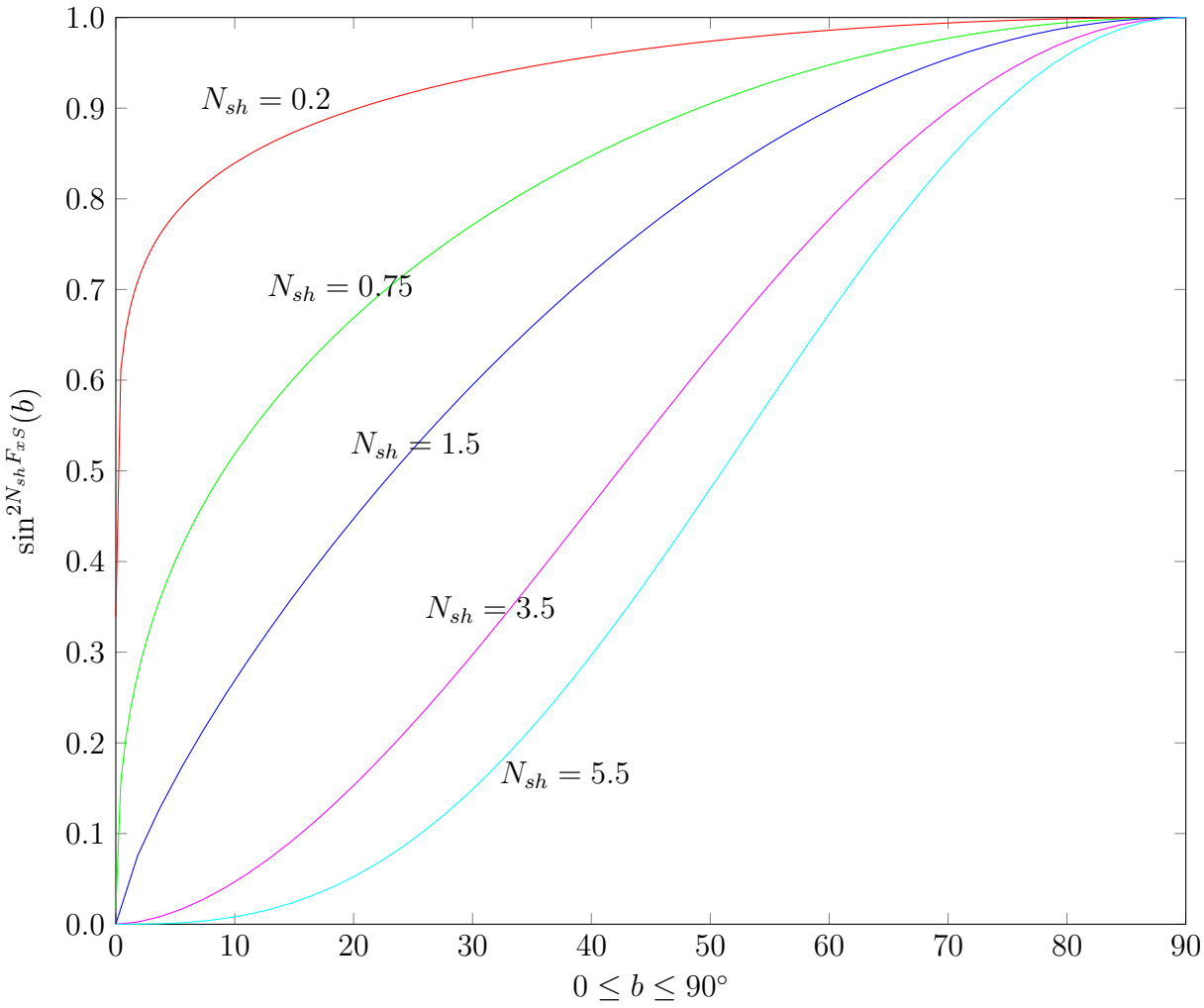
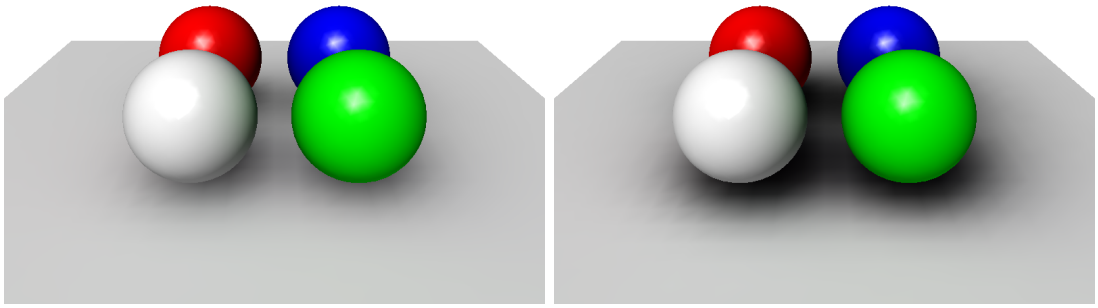


Figure 7.7: Shadow attentuation curve plots for $\sin^{2N_{sh}F_{xS}}(b)$, $F_{xS} = \frac{1}{4}$, N_{sh} values ranging from 0.2 to 5.5. When N_{sh} has lower values, shadows are only cast when the vector to the light source lines up almost exactly with the vector to the blocker. Higher values of N_{sh} mean a "fatter shadow" for an OCCLUDER of the same form factor.

We needed a function that produced values near 0 for small angles and values near 1 for angles near 90°[28]. The sine function fit here, but other functions for obtaining shadows may be possible.



(a) Shadow strength=1.5 (b) Shadow strength=5.5

Figure 7.8: For VO shadow darkness and caustic strength are adjustable parameters, analogous to the exponent used for "shininess" used for specular reflection in the Phong shading model.

The *blockage factor* is a value between 0 and 1 that describes how much local illumination a particular blocker will intercept. A blockage factor of 0 means ***full blockage***, and a blockage factor of 1 means *no blockage*.

Figure 7.7 shows the shadow attenuation curves for different values of N_{sh}. When $b \approx 0°$, regardless of N_{sh}, the vector from VERTEX to the light source ($\overrightarrow{xL}$) and the vector from VERTEX to the OCCLUDER $\overrightarrow{xS}$ line up and ***point in almost exactly the same direction***. In other words, we are in the umbra of the blocker: the blocker is coming directly between the vertex and the light source and completely occluding it. An example of this situation is the red blocker in figure 7.2. (Note this sharp zeroing of light may be responsible for the pointy shadows problem described in section 7.7.2.)

[28] You should note that occluders that are behind VERTEX will never have a shadow cast on VERTEX because they never will have been *seen* by the raycasting step

For any other value of $0 < b < 90°$, the amount of shadowing that OCCLUDER exerts on VERTEX will be determined by N_{sh}. If N_{sh} is very large (eg $N_{sh} = 5.5$ in figure 7.7), then OCCLUDER will tend to obliterate the light even for a large angle between $\overrightarrow{xL}$ and $\overrightarrow{xS}$ (large b). For example, when $N_{sh} = 5.5$, OCCLUDER still attenuates half the light for an angle of $b = 45°$. When N_{sh} is small, (eg $N_{sh} = 0.2$), almost 90% of the light is let through for $b > 20°$. When $b = 90°$, the vector to the light source is orthogonal from the blockage direction, so no blockage occurs.

If translucent surfaces are allowed, then the **blockageFactor** should be component wise multiplied by 1 – ***transmissive color*** of the OCCLUDER.

7.2.3 Caustics

VO computes caustics as the opposite of shadows. Instead of geometry attenuating light at VERTEX when the light comes from certain directions, the geometry magnifies it. The caustic vectors are found by raycasting, just as the shadow vectors are. The difference is rays are refracted through translucent surfaces until reaching free space. When light comes into VERTEX from the ***caustic direction***, OCCLUDER causes a magnification of light at VERTEX.

The formula for computing caustic strength, once we have the *caustic direction* and form factor of the caustic-sending surface is:

$$\text{causticStrength} = \sin^{-2N_c F_{xS}}(c) \tag{7.3}$$

Where[29]:

- c is the angle between the *caustic direction* and the light source
- F_{xS} is the *form factor* of the caustic-sending surface with the point being lit x

[29]Note this is not the inverse trigonometric function arcsin, rather it is just the regular sin function with a negative exponent.

- N_c is the *caustic strength adjustable parameter*. Requires hand tuning.

The **causticStrength** should be component-wise multiplied by the RGB *transmissive color* of the OCCLUDER. Note that the factor of 2 in the exponent for both the shadow and caustic formulae are there because they allow us to use the identity

$$\sin^2 t = 1 - \cos^2 t$$

To compute the sine of the angle between two vectors, we can use the dot product directly and avoid using arccos, and avoid using square roots.

$$\sin^2 t = 1 - (V \cdot L)^2$$

Where V is can be either the *normalized occlusion direction* or *normalized caustic direction* and, L is the normalized direction towards the light source.
Since the exponent is multiplied by adjustable parameters N_{sh} or N_c anyways, we can easily reduce the exponent to 1 by setting N_c or N_{sh} to $\frac{1}{2}$.

7.2.4 Interreflections

In figure 7.2, the purple pentagon will send strong interreflections to x. The three different types of interreflection described in section 2.3 each work slightly differently. Here we shall describe each, using the purple pentagon as our example OCCLUDER. Again, x is synonymous with VERTEX.

Diffuse-from-diffuse (DD) interreflection

Finding the *diffuse normal*

To find the diffuse-from-diffuse (DD) interreflection (or color bleeding) at x due to an OCCLUDER such as the purple pentagon, we have to find the brightness of that purple pentagon due to direct light, and then "bleed" some of its purple color onto x. Recall

that we do all this in a vertex shader *while* we are shading vertex x. If the purple pentagon is in a recessed area and generally obscured from light, then it won't bleed much color. We will therefore use AO to dampen an OCCLUDER's DD contribution. The amount of color bleeding at x sent from the purple pentagon will depend on two main things:

1. The angle of the *diffuse normal* (N_D) of the purple pentagon with the vector $\overrightarrow{P_{\text{avg}}L}$ in figure 7.9. Recall that P_{avg} is the average position of OCCLUDER from the vantage point x.

2. The *diffuse color* of the purple pentagon (determines color bled)

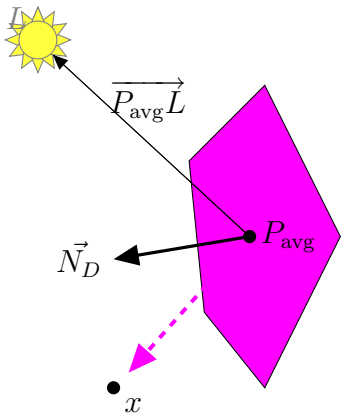


Figure 7.9: ***Sending a DD interreflection using the diffuse normal.*** The *diffuse normal* of the purple pentagon occluder is the average surface normal that the casted feeler rays saw when they hit it. The purple pentagon is planar so every ray cast will see the same normal, but OCCLUDER does ***not*** have to be a flat shape. The strength of the DD interreflection will depend on the purple polygon's form factor with x.

The way we compute the N_D of the purple polygon OCCLUDER for VERTEX x in figure 7.9 requires two passes. On the first pass, we find the ambient occlusion of every VERTEX in the scene and store the AO at each VERTEX. The reason we do this is we can sample the AO computation on the second pass.

On the second pass, we cast thousands of rays from each VERTEX and find what OCCLUDERs VERTEX can see. Previously we computed the shadow vector in figure 7.6 by a form-factor weighted sum of the direction vectors that hit each occluder. Now we compute the ***diffuse normal*** vector as the *ambient occlusion-damped, form factor weighted sum of the* ***surface normals*** *of the* OCCLUDER*s that we hit.*

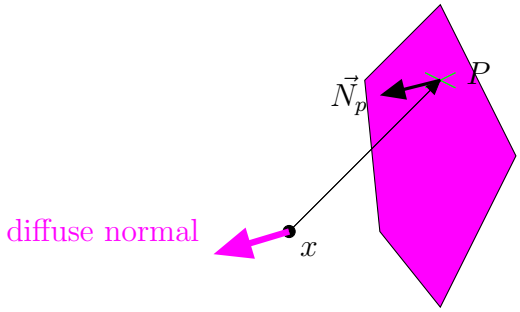


Figure 7.10: A ray cast from x hits the point P on the purple pentagon. AO

This is illustrated in figure 7.10. Using the ray cast from x to the point P as an example, we would add $\text{AO}_P \times F_{\text{ray}} \times \vec{N_p}$ to the *diffuse normal* at x. Note that we dampen each normal summed in by not only the form factor of the ray cast F_{ray} but also by the average ambient occlusion of the point hit AO_P. If we do not weigh by the ambient occlusion at the point hit interreflections will appear too strong in the final render.

It is important to note that x will experience the strongest interreflection from the purple pentagon ***when*** L ***comes from the diffuse normal direction***. Or in other words, when L comes from a direction that gives the purple pentagon the strongest response. This is the same idea that PRT uses to compute interreflections, and was previously described in figure 3.8.

Computing DD lighting in the VO vertex shader

At runtime, the way the DD interreflection sent by each OCCLUDER is found is quite simple. In figure 7.9, for example, the vertex x would have **two** diffuse responses to the light source L, not just one. x would have a direct response to L computed with x's

own surface normal (not shown), and x would also add in the diffuse response ***due to the purple pentagon***, using the ***diffuse normal*** vector for the purple pentagon occluder that it stored earlier during the raycasting process.

The only thing VO does to add DD interreflections, then, is to have each VERTEX effectively have ***many*** normals. Each normal has to be properly weighted of course. The strength of the $N \cdot L$ lighting of remote surfaces must be toned down by the form factor of the remote surface with VERTEX, and the ambient occlusion of the remote surface, otherwise the interreflections will appear too bright and unrealistic.

Diffuse-from-specular (DS) and specular-from-specular (SS) interreflection

Finding the *specular normal*

The diffuse-from-specular (DS) and specular-from-specular (SS) terms are going to be calculated using the *specular normal* and *specular color* vectors of each occluder listed in section 7.2.

To gather the *specular normal* that VERTEX sees from the purple pentagon in figure 7.9, we follow a very similar procedure as for finding the *diffuse normal*. We will cast rays from VERTEX x and sum a form-factor weighted average normal. However, there are a couple of differences between the *specular normal* and the *diffuse normal*. First, we **do not** weigh each ray's contribution in finding the *specular normal* by the ambient occlusion of the point hit P. We have a better way to measure if P will send a specular reflection or not: We test *if the reflection direction is open* about the point hit. If the mirror reflection direction $\vec{R}$ in figure 7.11 is not open, then the ray is discarded.

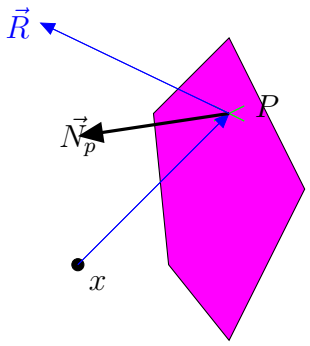


Figure 7.11: ***Finding the specular normal vector***: Make sure that the specular reflection direction about P is *open*, otherwise discard the ray. For more complex shapes and scenes, the *specular normal* may be quite different from the *diffuse normal*

Computing DS lighting inside the VO vertex shader

The diffuse-from-specular (DS) lighting term is found using a modified Phong specular lighting formula, but as if x were the eye.

To add in the DS lighting code to the VO vertex shader, we need to know *if the occluder passes on a specular reflection* to x. In other words, in the diagram of figure 2.2b, we are considering x as the eye. Referring to figure 7.12, we reflect the vector from x to the *average position* of OCCLUDER ($\overrightarrow{xP_{\text{avg}}}$) about the *average specular normal* at OCCLUDER (called $\vec{N}_{\text{S}}$ in figure 7.12) to find the average specular reflection direction $\vec{S}$. Find the dot product of $\vec{S}$ with the vector from $\vec{P_{\text{avg}}}$ to L (ie find $\textbf{SpecPower} = \vec{S} \cdot \overrightarrow{P_{\text{avg}}L}$). This dot product tells you "how lined up" $\vec{S}$ and $\overrightarrow{P_{\text{avg}}L}$ are. The more lined up $\vec{S}$ and $\overrightarrow{P_{\text{avg}}L}$ are, the stronger your diffuse-from-specular caustic. We may raise **SpecPower** to an exponent to sharpen the highlight. This method of computing the DS term borrows from Phong's model for specularity [40]. Other methods may exist and should be explored.

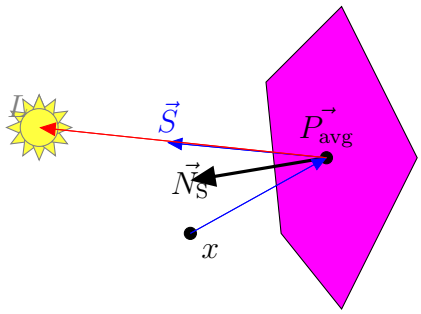


Figure 7.12: Lighting the diffuse-from-specular term that x would receive from the purple pentagon. Here x behaves like an "eye" and wants to know if it receives a specular reflection from L after reflection at point $\vec{P}_{\text{avg}}$ over specular normal direction $\vec{N}_{\text{S}}$.

Computing SS lighting inside the VO vertex shader

To find the specular-from-specular term we must reflect the vector from the eye to the vertex x, and see how much that reflected vector "lines up" (via dot product) with the vector from x to $\vec{P}_{\text{avg}}$. This is illustrated in figure 7.13.

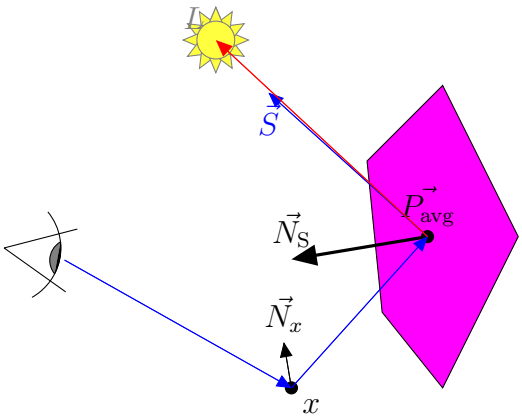


Figure 7.13: Lighting the specular-from-specular term that x would receive from the purple pentagon. It requires two specular reflections: one about the normal $\vec{N}_x$ at the vertex x, and the second about the *specular normal* $\vec{N}_S$ at the purple polygon

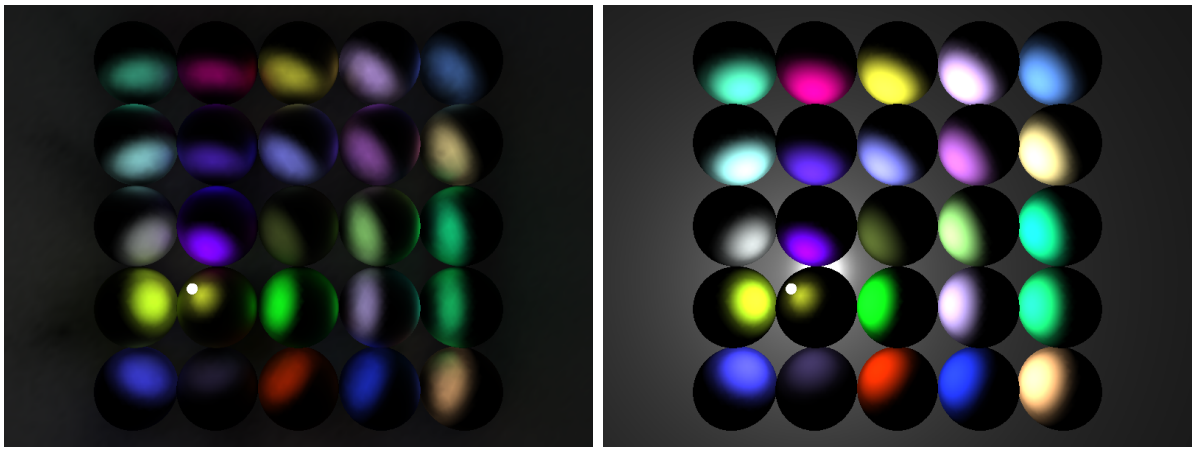
In a nutshell that is how VO works. In a preprocess, we "feel out" the groups of OCCLUDERs that surround *every* vertex in the scene. Each vertex will see multiple OCCLUDERs. There are 8 vectors stored for each OCCLUDERs seen at each VERTEX.

These 8 vectors describe all the properties of the occluder that we need to know to compute the shadow that occluder will send, the interreflection, and the caustic.

7.3 Special effects

Because vectors to occluders are just vectors, it is straightforward to bend and twist these vectors in real-time to achieve underwater or other-worldly special effects. We can do this using a wavetrain of arbitrarily oriented plane waves in 3-space. By using random transverse and longitudinal axis directions for each plane wave, we can get some neat effects[30].

7.3.1 Point lights



(a) VO, single point light @ 250 fps (b) Phong, single point light @ 2500 fps

Figure 7.14: VO is able to easily handle local lights just as easily as Phong does, adding shadows and interreflections in real-time.

[30] One optimization we are missing here is to make the wavetrain periodic, so position and normal displacement values only need to be computed once. I couldn't figure out how to do that, however and it may not be possible to find the temporal period for a group of plane waves with arbitrary transverse and longitudinal axes.

Because VO is vector based, VO easily handles point lights, where the spherical harmonic and wavelet approaches described in [50] and [38] use infinitely distant lights. It should be noted that [27] discusses a method for local illumination in spherical harmonics, which is somewhat complicated, however.

7.3.2 Modulating underwater caustics

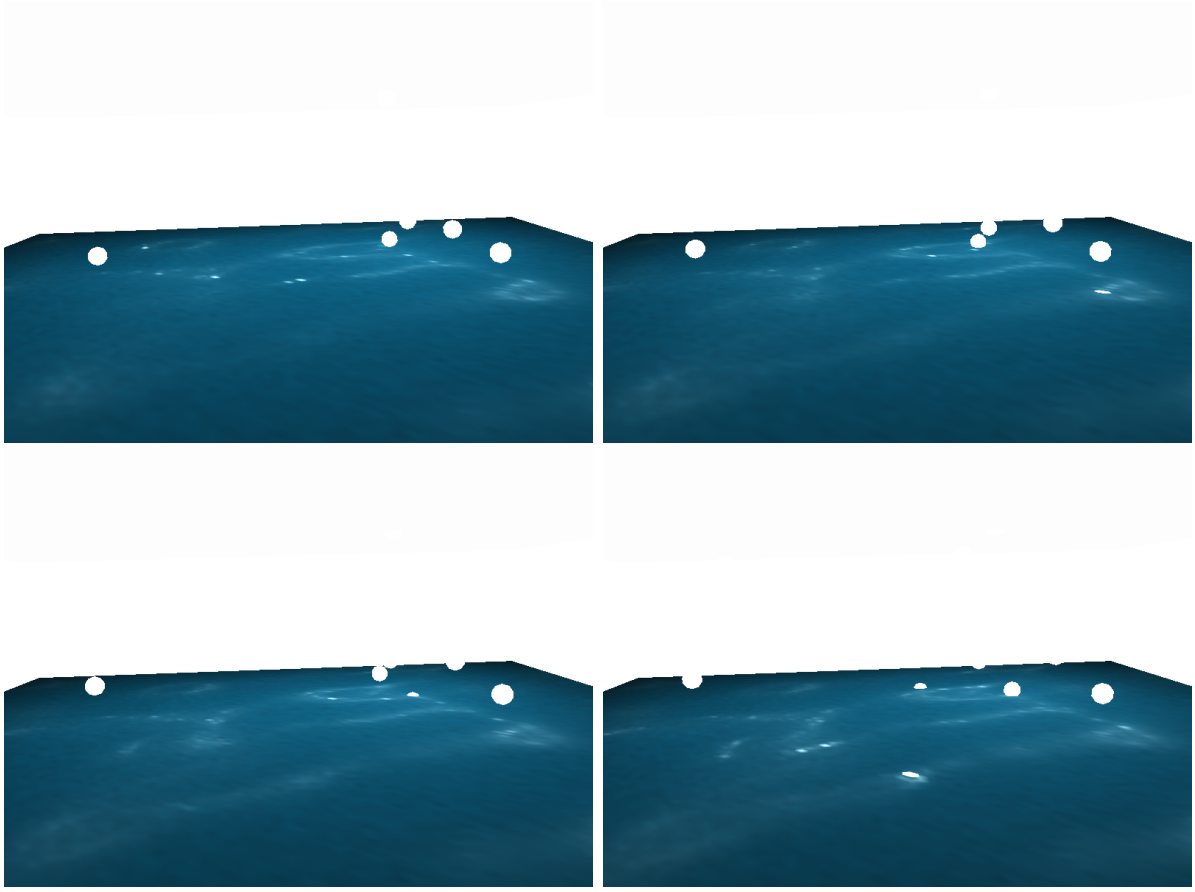


Figure 7.15: Real-time caustics computed by bending the caustic normal. We intentionally show the light sources very close to the surface – this technique doesn't create an ocean surface, but it can modulate light *as if* an ocean surface was above. Artists can use adjustable parameters to manipulate the effect to their liking.

Caustics are easy to form, in fact they are just the opposite of shadows. Instead of geometry blocking incident light in certain directions, the geometry *intensifies* it. In

the case of being under a surface with a shifting normal, such as being under the surface of water, the refraction angle of the light is continuously changing in time. If we can model the directions that will focus light varying in time realistically, we can create real-time caustics.

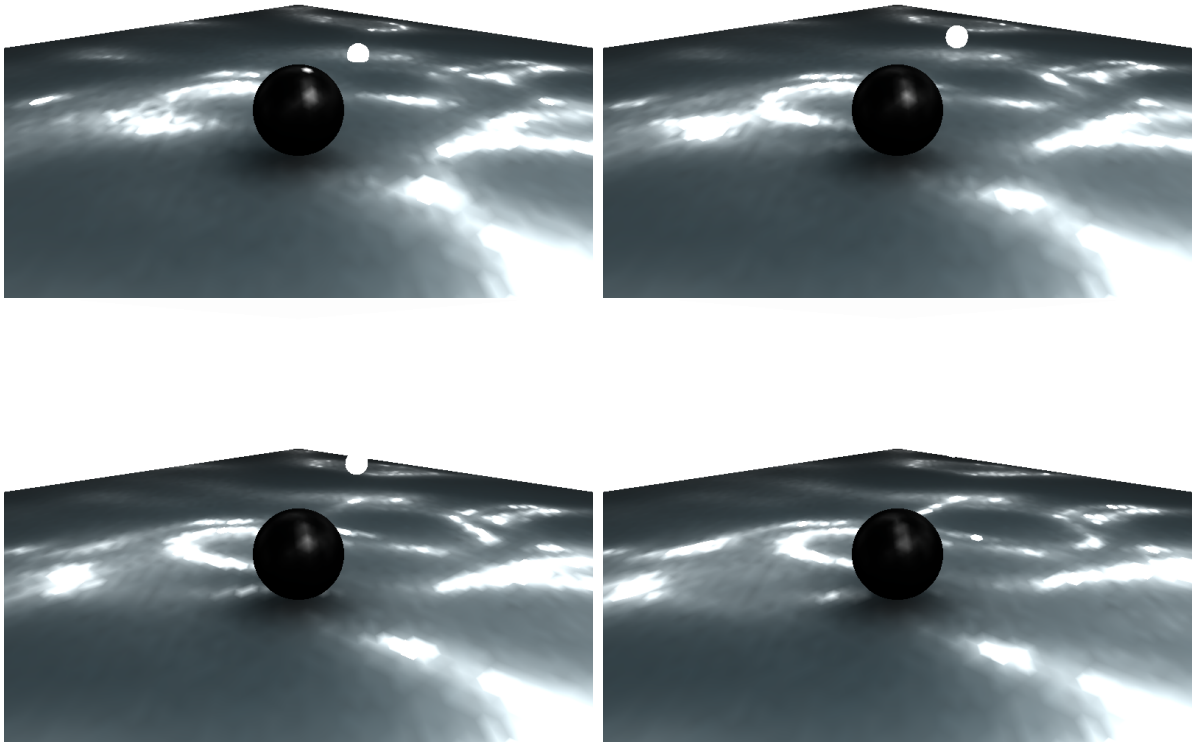


Figure 7.16: Real-time caustics computed by bending the caustic normal. Note the modulated shadow near the black pearl at the ocean floor.

7.3.3 Color absorption

An artist and some experimentation may yield some even more interesting effects. If we multiply the *diffuse normal* by a negative value, then the interreflections absorb color from neighbouring surfaces instead of sending them.

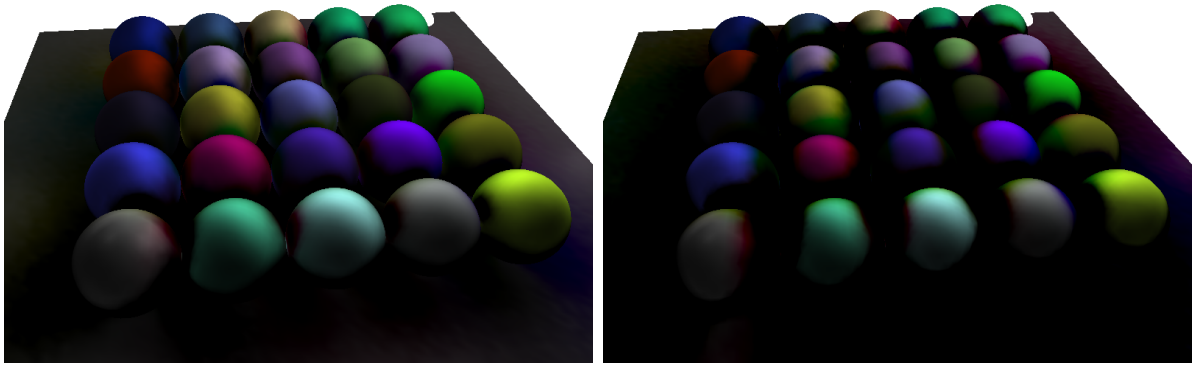


Figure 7.17: Another fun effect: color drain and disease. The diffuse-from-diffuse normal vector is multiplied by a negative scalar value, so color is drained from neighbouring geometry instead of shared.

7.4 Implementation on modern graphics cards

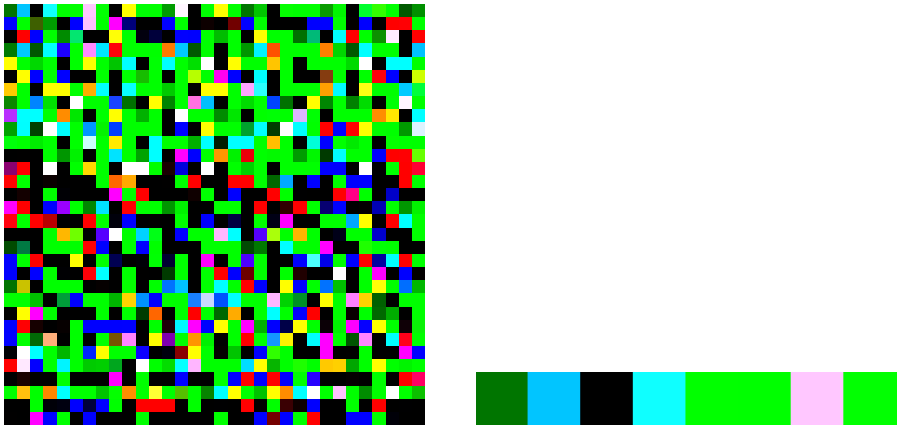


Figure 7.18: Vector occluder vector texture for a large number of vertices (left), and one vertex (right)

We use a floating point RGB texture to save the groups of 8 vectors. Recall that each vertex requires 8 vectors *per occluder it can see* in the scene. For a scene with 100,000 vertices, if we assume that each vertex can see an average of 4 distinct occluders, then

we would need to store $100,000 \times 4 \times 8 = 3,200,000$ vectors.

A 2048×2048 96 bit-per-pixel RGB floating point texture would consume 48 MB of video memory and have space to store $4,194,304$ vectors, more than enough for this scene.

7.4.1 Limiting the number of occluders

The algorithm as described above computes the full set of eight vectors described above *for every mesh that each vertex can see.*

One must consider that shadow shape and quality will be influence by the number of occluders that a particular model is made up of. If a statue with outspread arms, a large torso, and outspread legs is considered as *one occluder*, then the shadow cast by that statue can be expected to be quite blobby for points on a floor, for example. A large statue might be broken into five mesh groups, one group for the head and torso, and one group for each of its arms and its legs. Choosing a wise subdivision of each model into individual occluders for shadowing is likely a task best left to the artist, but devising algorithms for creating VO occluder groups at precompute time is suggested as future work.

One may discard mesh groups that have less than a certain threshold form factor with the vertex being considered, to reduce storage requirements as well as runtime cost.

7.5 VO Comparisons

We would like to see how VO compares with the other techniques studied. We present a qualitative comparison of the renders, because the differences between VO and other techniques are large enough that they can be easily observed by the human eye.

7.5.1 VO vs Radiosity

To check the accuracy of VO's shadows and diffuse interreflections, we first compare a simple VO render with a radiosity render.

(a) VO (b) Radiosity

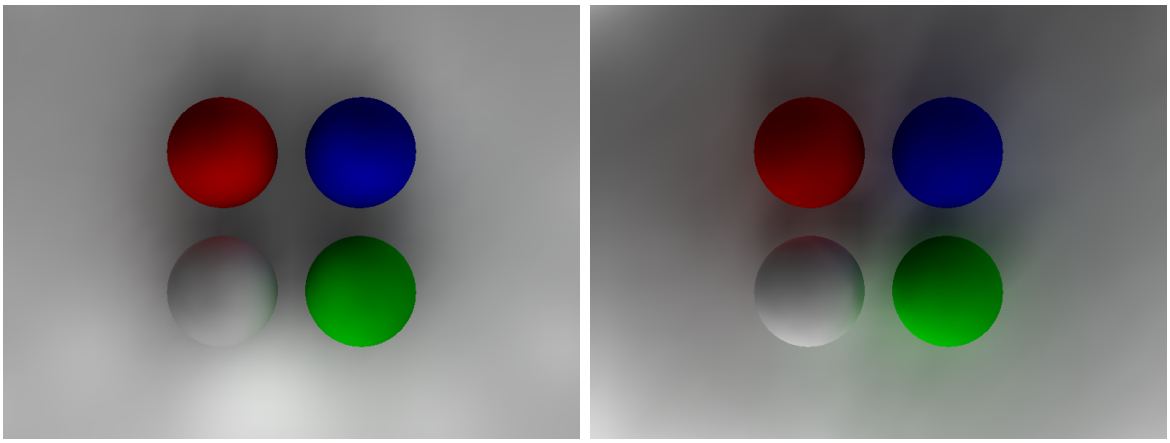


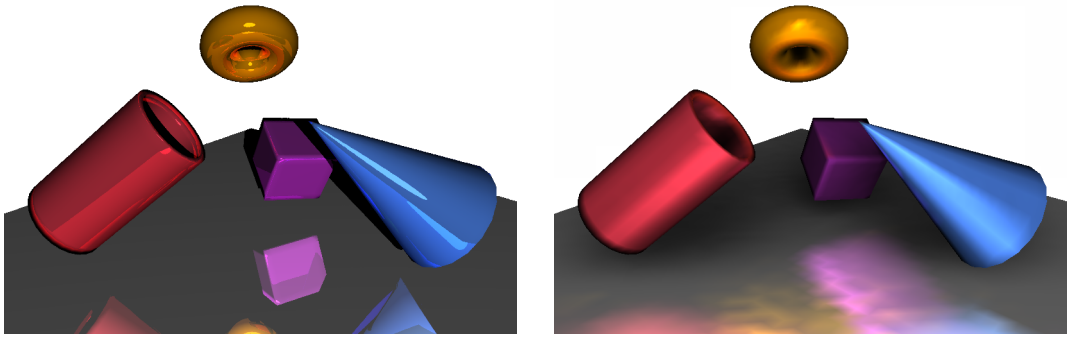
Figure 7.19: Vector occluders and radiosity rendering a scene with 21k vertices, 20 bright lights overhead. Notice VO has the diffuse interreflections on the white ball looking very similar to those computed by radiosity.

We rendered a purely diffuse scene in figure 7.19. We do not expect VO to be exactly the same as a radiosity render. However in figure 7.19, we can see VO generating some reasonable soft shadows and diffuse-from-diffuse interreflections. Note especially the diffuse interreflections between the white ball and the red and green balls. While the shadowing patterns do differ in some portions of the images, in the areas between the spheres and on the spheres themselves, the shadowing patterns are quite similar.

Recall that the VO solution *will update shadows and interreflections in real-time as the light source moves*, without reduction to frame rate, as a major advantage over traditional radiosity.

7.5.2 VO vs Whitted Ray Tracing

To evaluate the specular from specular interreflection component, we use Whitted ray tracing. We are interested to verify that the specular from specular interreflection component looks somewhat similar to a Whitted ray trace.



(a) Whitted ray trace (b) VO, SS term only

Figure 7.20: VO vs Whitted ray tracing

Figure 7.20 shows that VO's SS term does a decent job of estimating a glossy specular from specular term in real-time. Of note are each shape's specular reflection in the floor, as well as the red tube's specular interreflection from the bottom of the torus.

7.6 VO vs SH vs CH runtime

We now provide a 3-way comparison precompute time, runtime speed, and storage of VO, SH and CH. This serves both to compare VO against SH and CH as well as to show how similar SH and CH are to each other with respect to runtime.

Technique	Precompute (s) (pass 1+pass 2)	Render speed * (fps)	Storage (vectors)	Storage (MB)
Phong (REF T&L)	0	5500	0	0
AO	58	5500	$\frac{1 \text{ float}}{\text{vertex}}$	0.019
VO/DD	58 + 60†	4700	67,244	1.03
VO/DD+DS+SS	58 + 85	3500	134,464	2.05
SH3/DD	60 + 63‡	5100	135,351	2.07
CH3/DD	56 + 61	5100	135,351	2.07
SH6/DD	62 + 68	2500	541,404	8.26
CH6/DD	60 + 66	2500	541,404	8.26
SH3/DD+SS	105 + 70	114	1,353,510	20.7
CH3/DD+SS	103 + 70	97	1,353,510	20.7
SH6/DD+SS	142 + 175	25	20,031,948	306
CH6/DD+SS	142 + 173	22	20,031,948	306

* Fraps was used to measure frame rates.

† VO uses AO pass + VO pass

‡ SH and CH use shadow pass + interreflection pass

Table 7.1: Shapes scene (see figure 7.22), 1 light, 9818 tris, 5013 verts, 10000 rays. For the SH/CH entries DD is diffuse interreflection only, while DD+SS is diffuse and specular interreflection.

Interpreting the data in table 7.1 has a number of caveats, which we shall discuss shortly. First, we should discuss the naming convention for each of the techniques in table 7.1. SH3/DD stands for "spherical harmonics, 3 bands, with diffuse from diffuse interreflections". CH6/DD+SS stands for "cpherical harmonics, 6 bands, diffuse from diffuse interreflections and specular interreflections". VO/DD stands for "vector

occluders, diffuse from diffuse only", while VO/DD+DS+SS is "vector occluders, with diffuse from diffuse, diffuse from specular, and specular from specular interreflection components".

7.6.1 SH/DD vs SH/DD+SS render speed apparent large disparity

First of all, [48] (the program written to develop and evaluate these algorithms) does not have a GPU implementation for specular SH or specular CH. As such the SH/DD+SS and CH/DD+SS entries were evaluated on the CPU only. This is the reason for the extremely large disparity in frame rates for SH/DD and SH/DD+SS cases. As such SH/DD vs SH/DD+SS frame rates should not be compared. The useful information that we derive is SH/DD+SS and CH/DD+SS have similar frame rates.

7.6.2 SH vs CH

SH and CH (both diffuse and specular cases) have identical storage requirements, and both run similarly. We cannot fairly conclude one is faster than the other based on these timings, nor was this our goal. This experiment simply shows that SH and CH are similar. One must keep in mind that there is always a jitter in CPU speed and performance based on heat, and other uncontrollable factors such as O/S service behavior, which may explain the slight difference in precompute time for SH and CH.

7.6.3 VO vs SH+CH

The ***precompute*** times are very similar for AO, VO, SH/DD and CH/DD. This leads us to hypothesize that precompute time for these methods is dominated by *ray cast intersection code*, rather than the VO *data aggregation code* or the Legendre or Chebyshev polynomial evaluation. Indeed, inspection with a code profiler reveals ***over***

90% of the AO, VO, SH and CH precompute times are spent in ray-polygon intersection code. We note here that [48] does not perform the specular reflection ray casts when running SH/DD, CH/DD or VO/DD, which explains the difference between DD and DD+SS precompute times for both SH and CH.

With regard to ***storage*** and ***runtime***, table 7.6 confirms that SH and VO are actually quite different in both these regards.

Storage

The number of coefficients that SH and CH store is independent of scene complexity. The number of coefficients that VO stores, on the other hand, depends on the number of occluders each vertex in the scene can see.

SH/DD and CH/DD need to store ℓ^2 coefficients per vertex, where ℓ is the number of SH bands used. Specular SH or CH requires $\ell^4 + \ell^2$ coefficients *per vertex*, which is considerably more than diffuse SH or CH and also considerably more than VO for this particular scene. Note that Sloan already used clustered principal component analysis [49] to reduce the amount of data that SH uses.

VO/DD+DS+SS stored 134,464 vectors for this particular scene. The worst case for this scene for VO/DD+DS+SS (5 shapes, 5013 vertices) was $5013 \times 5 \times 8 = 200,520$ VO vectors[31]. Areas like the inside of the red tube or the top of the orange torus would help reduce the total number of vectors used by VO (the top of the orange torus won't see any occluders at all!)

VO/DD had less data (67,244 vectors), and as a result a faster frame rate than VO/DD+DS+SS. VO/DD compares nicely in runtime with SH3/DD, while requiring less storage for this particular scene.

[31] That is, 5013 vertices × 5 occluders × 8 vectors per occluder seen = 200,520 VO vectors total. The worst case for VO involves every vertex seeing every occluder in the scene.

Runtime

The runtime of each of these methods is related to the data size, in that the more data a technique has to crunch through each frame, the longer the frame processing time.

Although SH3/DD and CH3/DD have similar data size for this particular scene, the runtime for SH3/DD or CH3/DD is better than VO. This is expected, as the simplicity of the 9-component dot product evaluation required per vertex for SH3/DD and CH3/DD of SH and CH is much simpler than the reflect vector operations required by VO for the VO-DS and VO-SS terms.

VO's runtime depends on the number of occluders seen at each vertex ***and*** on the number of point lights in the scene. The time complexity to compute VO at a particular vertex for VO/DD or VO/DD+DS+SS is $O(mn)$, where m is the average number of occluders seen at each vertex and n is the number of lights in the scene.

Figure 7.21 shows fps decay for increasing number of lights for Phong (almost no decay), VO/DD and VO/DD+DS+SS. The parallel nature of the GPU is the reason we do not see much decay in runtime for Phong, even as we go up to 32 lights.

SH uses only a single SH vector to represent an arbitrarily complex light source, since an SH vector represents a function on a sphere. This remains an advantage for SH, and it is the reason why SH is able to evaluate complex looking lighting with only a small number of multiplies – SH always has just "one light source". The time complexity to compute SH lighting at a vertex for SH/DD or CH/DD, $O(\ell^2)$, where ℓ is the number of bands used.

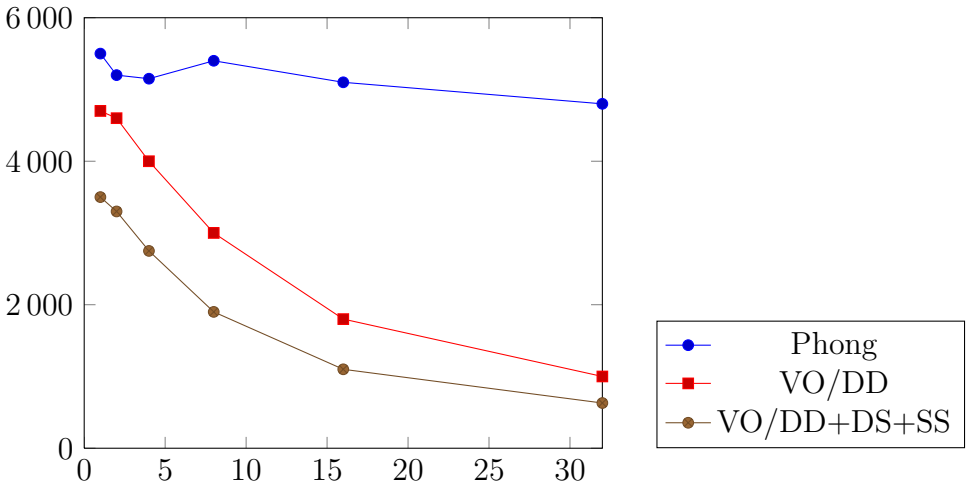


Figure 7.21: Decay in frame rates for VO/DD and VO/DD+DS+SS as compared to Phong by number of lights. Uses the Shapes scene in figure 7.22.

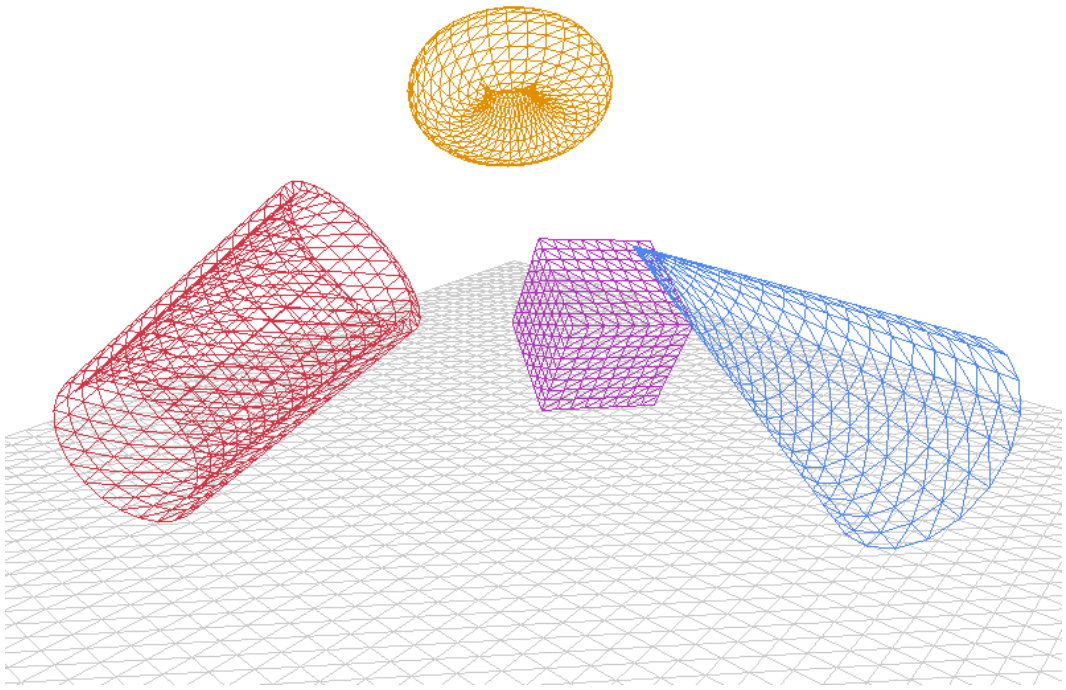
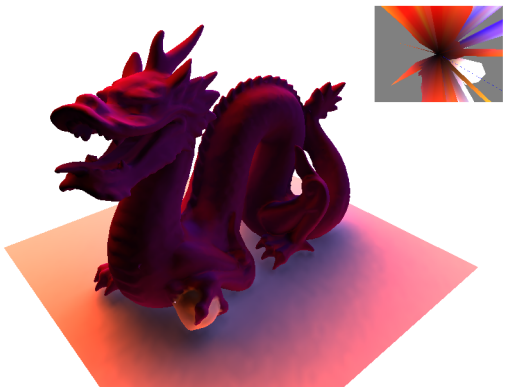
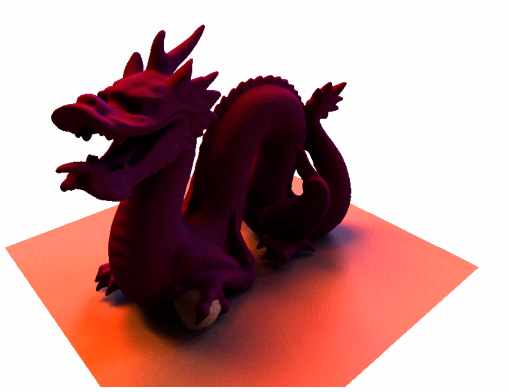


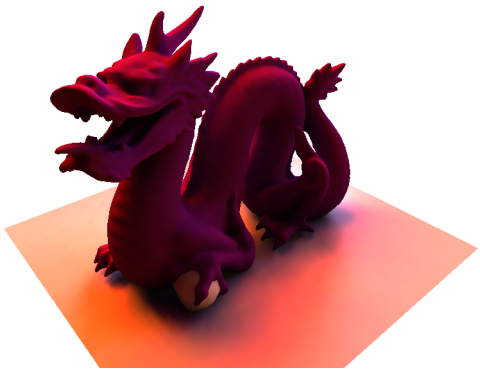
Figure 7.22: Shapes scene used for runtime evaluation in table 7.1. 9818 triangles, 5013 vertices, 5 shapes/occluder primitives.



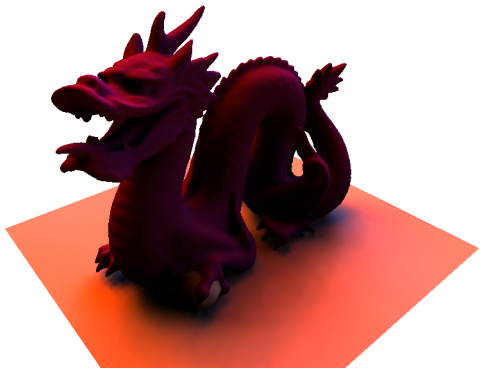
(a) Vector occluders @ 75 fps



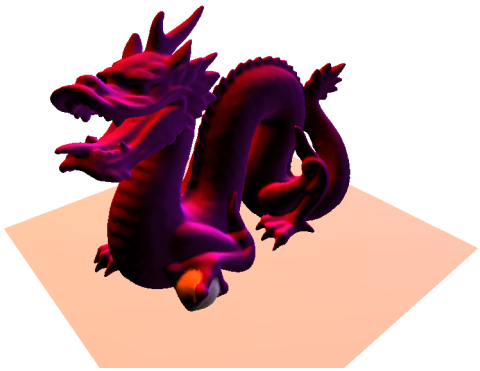
(b) Path traced (not real-time in our impl.)



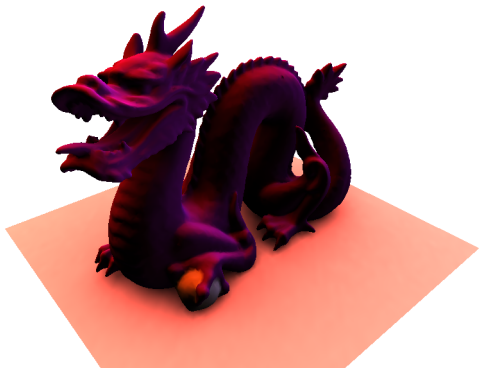
(c) Wavelet @ 2 fps



(d) 6 band spherical harmonics @ 190 fps



(e) Phong direct illumination only @ 650 fps



(f) Ambient occlusion @ 450 fps

Figure 7.23: VO versus path tracing and the real-time GI techniques studied

A more complex render is presented in figure 7.23. VO must work from point lights, so for this render, a point light approximation of the Grace Cathedral cubemap was computed first, and used to generate the two images below. The point light

approximation is shown in figure 7.24.

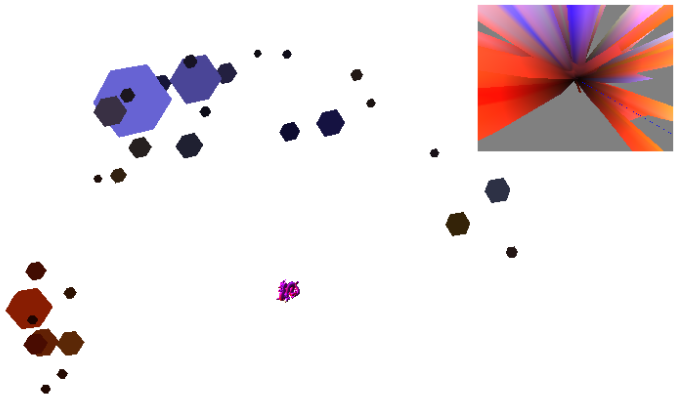


Figure 7.24: 32-point light approximation of the Grace Cathedral cubemap. The size of each point light represents its intensity.

Looking at figure 7.23, vector occluders presents a reasonable real-time approximation to what global illumination should look like as compared to the reference path-traced, wavelet and spherical harmonics renders. Above AO, it offers directionality for the shadowing, and coloured diffuse interreflections. A quick glance at the basic, unshadowed Phong render in 7.23e tells us just how far VO takes us towards global illumination effects, using only simple vectors.

On average, VO presents a much better frame rate than wavelets, but slower than 6-band SH. Future work could improve the formula for creating shadows, to make them more closely match the reference render.

7.7 Summary

7.7.1 Advantages of VO

VO produces an empirical approximation to global illumination. The results are reasonable and provide depth cues to the viewer that make the computer graphics scene appear more realistic to the human viewer. VO works with the same point lights that traditional local illumination models work with. VO's closest *performance* competitors are ambient occlusion and spherical harmonics. Spherical harmonics offers better frame rates and more accurate results, however VO differs from SH in two unique ways:

i) VO can use point lights with arbitrary space position. SH generally uses infinitely distant directional lights

ii) Special effects that have to do with wobbling the shadow or caustic directions (mimicing participating media or other such things) can be applied to the caustic and shadow vectors, which can be manipulated by a skilled artist to simulate underwater caustics or smoke flying overhead

7.7.2 Problems with VO and future work

We think that VO as presented is promising. The use of basic vectors for shadowing makes it interesting to consider empirical solutions to problem of global illumination with deformable geometry, as we shall suggest below.

Shadow accuracy

As shown in 7.23, the shadows that VO produces could be improved. Specifically, we see the shadows under the dragon's chest as not being dark enough. We also see excessive attenuation of the red light from above on the white platform on the right

side of the dragon. We also found in other experiments that shadows that are cast a long distance tend to get "pointy" in our tests as shown in figure 7.25.

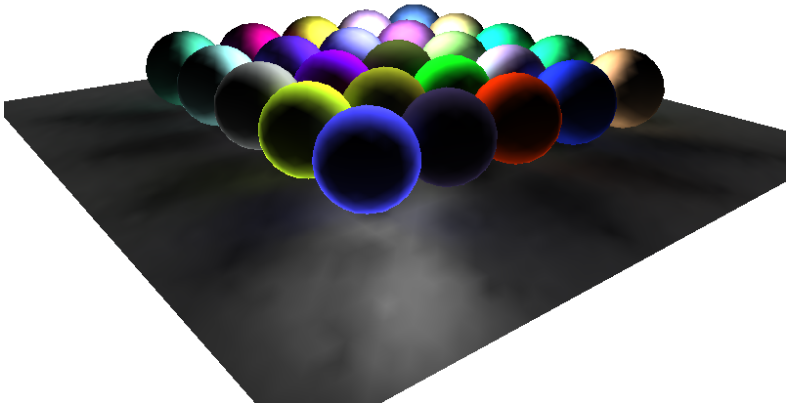


Figure 7.25: VO's pointy-shadows problem. The spheres on the left and bottom edges of this image cast pointy shadows.

Improving the shadow shape by using different formulae could be interesting future work.

Specular from specular component

The specular from specular component requires the largest number of rays to appear stable. Alternative methods for estimating the specular interreflection direction with fewer rays cast should be explored. Alternative formulations for finding light specular interreflection light output due to the double reflection should be explored as well.

Grouping blockers

A systematic way to group blockers may be useful. There currently is no algorithm for selecting an "occluder" in VO – it is left to the artist to appropriately partition the

geometry for VO. There may be a method for subdividing occluders so that shadows cast are sharper or more accurate.

Updating VO data for dynamic occluders

It may be possible to make occluders at least partially dynamic if a way can be found to update the shadow, caustic, and interreflection vector groups in real-time, such that the approximation remains reasonable through occluder translations and rotations. It may be possible, for example, to update vector occluders using distance as a scaling factor for form factor estimation and the centroid of the occluder as the new direction for the shadow vector.

Principal Component Analysis

We would also like to optimize the storage and runtime of VO. Perhaps using principal component analysis (PCA) as Sloan did for SH [49] may enable us to reduce the total number of vectors we are storing in the VO texture.

Alternative reflectance models

Models for per-vertex lighting other than Phong can be employed such as Cook-Torrance or Oren-Nayar.
Even if these problems aren't resolved, still VO remains a decent approximation to global illumination. It could also see use on low powered platforms such as portable devices.

Runtime cost

The runtime cost of VO changes with the number of surfaces that are allowed to send shadows. VO will always only care about surfaces ***that a point can see***. Even if

there are a large number of objects in a scene, VO's cost will not be very high if we limit the the number of objects that can send shadows or caustics per-vertex.

7.8 VO in real-time

VO is a precomputed, real-time method. The entire idea behind PRT and real-time is to leverage the fact that ***finding form factors is what's expensive***, not re-lighting. If you don't change the form factors, then re-lighting the scene with a different light source is actually a very light operation in terms of compute time.

Chapter 8

Conclusions and future work

What's current in real-time global illumination has been surveyed. Real-time versions of the classic techniques radiosity and ray tracing are becoming viable on modern systems. Current global illumination approximation techniques are tending to work in screen-space, which makes the global illumination computation step independent of the amount of scene geometry. PRT techniques, particularly spherical harmonics, are still in widespread use.

In our work, we made two contributions. First, we substituted the Chebyshev polynomial in the place of the Legendre polynomial in the spherical harmonics definition to create the cpherical harmonics. Cpherical harmonics had similar precompute time, runtime performance, and image quality to spherical harmonics. Second, ***vector occluders*** was a novel, empirical technique for computing global illumination on static geometry with dynamic light sources.

8.1 Cpherical harmonics

We did not find any particular advantage or disadvantage to using the cpherical harmonics formulation. Still, the Chebyshev polynomials can be evaluated in many different ways, including using recurrence relations or a trigonometric expression, so it

may be possible to optimize cpherical harmonics further, and possibly make its evaluation faster than spherical harmonics. This warrants further study. The implementation code for cpherical harmonics as provided in appendix A.1 is on par in terms of runtime with the spherical harmonics implementation found in Numerical Recipes [41]. A GPU implementation should be attempted.

8.2 Vector occluders

The problems with VO outlined in 7.7.2 can be addressed by future work.
Some next steps for VO work could include a mesh dividing algorithm that optimizes shadow quality with the smallest number of vectors. Cutting a mesh into chunks that will produce good shadows with VO is a time-consuming process for the artist, so automating this process will be very good for workflow.
It may be possible to transform VO vectors as the source geometry moves and deforms to allow VO to track global illumination effects through mesh deformation.
The formulae for generating shadows and caustics should be experimented with to make VO more accurate as compared to reference renders.

Appendix A

Source listing

A.1 Cpherical harmonics

cphericalHarmonics.cpp

```
// The chebyshev polynomial
static inline real T( int n, real x ) {
  return cos( n*acos(x) ) ;
}

// ith derivative of nth chebyshev polynomial, helper
static inline real Tb( int i, int n, int k ) {
  int s = (k+n-i)/2;
  int c = n==0 ? 2 : 1 ;
  return ((1<<i) * k * facs[ s-n+i-1 ] * facs[ s+i-1 ]) /
         ( facs[i-1]*c * facs[s] * facs[ s-n ] ) ;
}

// Gets you the "ith" derivative of a chebyshev polynomial
```

```
// "k" at x, due to Elbarbary et al
static inline real TDiff( int i, int k, real x ) {
  real sum = 0 ;
  for( int n = 0 ; n <= k-i ; n++ )
    if( ! ((n+k-i)&1) ) // isEven
      sum += Tb( i,n,k ) * T( n, x ) ;
  return sum ;
}

// Associated Chebyshev polynomial
static inline real Tlm( int l, int m, real x ) {
  return sqrt( pow( 1 - x*x, m ) ) * TDiff( m, l, x ) ;
}

// precomputed K values for each (l,m) band.
static real KcCoeffs[] =
{
  // These come from MuPad by integrating the (l,m) cpherical
  // harmonic multiplied by itself.
  // If the (l,m) harmonics are multiplied by these factors,
  // then the inner product of
  // an (l,m) harmonic with itself is 1.  With any other band,
  // it'd always be 0 anyway.
  2.820947918e-1, //L=0
  4.886025119e-1, 4.886025119e-1, //L=1
  4.129444918e-1, 2.731371076e-1, 1.365685538e-1, //L=2
  4.047665634e-1, 1.854328105e-1, 6.022107172e-2,
```

```
2.458514958e-2, //L=3
4.021466875e-1, 1.399411247e-1, 3.414213846e-2,
9.219431093e-3, 3.259561122e-3, //L=4
4.009725341e-1, 1.122723096e-1, 2.200197653e-2,
4.562170666e-3, 1.081074122e-3, 3.418656546e-4, //L=5
4.003445423e-1, 9.370452984e-2, 1.535720259e-2,
2.607284268e-3, 4.802311784e-4, 1.027178456e-4,
2.965208790e-5, //L=6
3.999691606e-1, 8.039264056e-2, 1.132511831e-2,
1.633723697e-3, 2.490256634e-4, 4.173679601e-5,
8.203003044e-6, 2.192344781e-6, //L=7
3.997268284e-1, 7.038582157e-2, 8.695055776e-3,
1.092352436e-3, 1.427899302e-4, 1.994605450e-5,
3.089600665e-6, 5.649543187e-7, 1.412385797e-7, //L=8
3.995612758e-1, 6.259091642e-2, 6.884843506e-3,
7.667497016e-4, 8.798356395e-5, 1.060573155e-5,
1.376251441e-6, 1.992021591e-7, 3.420248310e-8,
8.061602577e-9, //L=9
3.994431498e-1, 5.634839052e-2, 5.586011607e-3,
5.589858279e-4, 5.725452568e-5, 6.091881619e-6,
6.852855807e-7, 8.342077800e-8, 1.137633131e-8,
1.847145202e-9,  4.130342235e-10, //L=10
3.993559065e-1, 5.123694925e-2, 4.622648211e-3,
4.201140561e-4, 3.889712274e-5, 3.712222653e-6,
3.701090444e-7, 3.919400144e-8, 4.509096539e-9,
5.830959106e-10, 9.003829770e-11, 1.919622957e-11, //L=11
3.992896399e-1, 4.697496858e-2, 3.888462059e-3,
```

```
  3.237425149e-4, 2.736552222e-5, 2.371042625e-6,
  2.128144456e-7, 2.003750505e-8, 2.011003346e-9,
  2.199307652e-10, 2.710808555e-11, 3.999214493e-12,
  8.163362401e-13, //L=12
  3.992381219e-1, 4.336708685e-2, 3.316150644e-3,
  2.547591851e-4, 1.981907608e-5, 1.573796100e-6,
  1.286657036e-7, 1.093755829e-8, 9.784679097e-10,
  9.356368862e-11, 9.773181415e-12, 1.153056687e-12,
  1.631464435e-13, 3.199564996e-14 //L=13
} ;

// Normalization constants Kc for Chebyshev
static inline real Kc( int l, int m )
{
  int entry = (l)*(l+1)/2 + m ; // pyramid of values.
  return KcCoeffs[ entry ] ;
}
```

A.2 Vector occluders shader code

vo.hlsl

```
#define VOVECTORSPERBLOCKER 8
#define MAX_LIGHTS 32
// VO DATA TEXTURE
Texture2D<float4> VOTEX : register( t3 );

// Per vertex vo shader
vvncOut //->TO pxvncVO
vGVectorOccluders_vnc( GENERALVERTEX input )
{
  vvncOut output ;
  output.pos = transform(input.pos) ;
  output.untransformedPos = input.pos ;
  output.norm = input.norm ;

  // Need these vectors everywhere, compute them now
  float3 eyeToV = normalize( input.pos.xyz - eyePos.xyz ) ;
  float3 eyeRefl = reflect( eyeToV, input.norm ) ;

  //  AO ONLY || THERE ARE NO BLOCKERS
  // I tested it and this if statement does not cost much.
  if( vo.x==0 || input.tVOIndex.z == 0 )
  {
    // primary diffuse and specular only
    float3 primaryDiffuseColor = {0,0,0} ;
```

```
    float3 primarySpecularColor = {0,0,0} ;

    for( int i = 0 ; i < activeLights[1] ; i++ )
    {
      float3 vToLight = normalize( diffuseLightPos[i].xyz -
        input.pos ) ;
      primaryDiffuseColor +=
        max( dot( input.norm, vToLight ), 0 ) *
          input.cDiffuse.rgb * diffuseLightColor[i].rgb *
          pow( input.tVOIndex.w, vo.y ) ;
          // exaggerate AO with vo.y.
          // if AO is 1, then nothing happens.

      float spower = max( dot( eyeRefl, vToLight ), 0 ) ;
      spower = pow( spower, input.cSpecular.w ) ;
      primarySpecularColor +=
        spower * specularLightColor[i].rgb * input.cSpecular.rgb ;
    }

    output.color = float4(
      (1-input.cTrans.rgb)*tonemap(primaryDiffuseColor +
       primarySpecularColor), 1-avg( input.cTrans.rgb ) ) ;
    return output ;
}

//////////
// VO
```

```
float3 fColor = {0,0,0};
float width,height;
VOTEX.GetDimensions( width, height ) ;
float2 texIndex = input.tVOIndex.xy ;

int nBlockers = input.tVOIndex.z / VOVECTORSPERBLOCKER ;
// VOVECTORSPERBLOCKER is # vectors used per blocker, default 8
float3 p[ MAX_LIGHTS ] ; // save primary colors for EVERY LIGHT,
// which we will attenuate later with VO shadows
// max # lights=default 32

// First save-in the primary (1st bounce/direct illum)
// colors for EVERY LIGHT on vertex
for( int lNo = 0 ; lNo < activeLights[1] ; lNo++ )
{
  float3 vToLight = normalize( diffuseLightPos[lNo].xyz -
    input.pos ) ;
  p[lNo] = max( dot( input.norm, vToLight ), 0 ) *
    input.cDiffuse.rgb * diffuseLightColor[lNo].rgb ;
  p[lNo] += pow( max( dot( eyeRefl, vToLight ), 0 ),
    input.cSpecular.w ) * specularLightColor[lNo].rgb *
    input.cSpecular.rgb ;
}

// Next look at all sets of VO vectors at this vertex
// (1 set of 8 per occluder seen)
for( int k = 0 ; k < input.tVOIndex.z ; k += VOVECTORSPERBLOCKER )
```

```
{
  // 1 POS
  float3 slPos = VOTEX[ texIndex ] ;
  nextIndexOffset( texIndex, width ) ;

  // 2 SHADOW DIR
  float3 slSurfNShadow = VOTEX[ texIndex ] ;
  nextIndexOffset( texIndex, width ) ;

  // 3 CAUSTIC DIR // different from shadow due to light bending
  float3 slCausticDir = VOTEX[ texIndex ] ;
  nextIndexOffset( texIndex, width ) ;

  // 4 CAUSTIC COLOR // rgb translucency/opacity
  float3 slTransColor = VOTEX[ texIndex ] ;
  nextIndexOffset( texIndex, width ) ;

  // 5 DIFFUSE NORMAL // remoteSurfaceDiffuseNormal
  float3 slSurfNDiffuse= VOTEX[ texIndex ] ;
  nextIndexOffset( texIndex, width ) ;

  // 6 DIFFUSE COLOR // remoteSurfaceDiffuseColor
  float3 slSurfDiffuseColor = VOTEX[ texIndex ] ;
  nextIndexOffset( texIndex, width ) ;

  // 7 SPECULAR NORMAL // remoteSurfaceSpecularNormal
  float3 slSurfNSpecular = VOTEX[ texIndex ] ;
```

```
float slSurfNSpecularPower = length( slSurfNSpecular ) ;
// any vector you reflect about must be normalized.
slSurfNSpecular /= slSurfNSpecularPower ;
nextIndexOffset( texIndex, width ) ;

// 8 SPECULAR COLOR
float3 slSurfSpecularColor = VOTEX[ texIndex ] ;
nextIndexOffset( texIndex, width ) ;

// If using 4-component vectors, can put this in .w
float blockageFF = length( slSurfNShadow ) ; // this
// is the FORM FACTOR of the occluding object.
float3 blockageDirection = slSurfNShadow / blockageFF ;
float causticFF = length( slCausticDir ) ;
float3 causticDirection = slCausticDir / causticFF ;

// I tested it and this if statement does not cost much.
if( vo.x > 1 ) {
  float3 dummy;
  for( int waveNo = 0 ; waveNo < 50 ; waveNo++ )
    wave2( vo.z, dot( input.pos.xyz, longAxis[waveNo].xyz ),
      timePhaseFreqAmp[waveNo].z, timePhaseFreqAmp[waveNo].w,
      longAxis[waveNo].xyz, transverseAxis[waveNo].xyz, dummy,
      causticDirection, blockageDirection ) ;
}

float3 secondaryDiffuseFromDiffuse   = {0,0,0} ;
```

```
float3 secondaryDiffuseFromSpecular  = {0,0,0} ;
float3 secondarySpecularFromSpecular = {0,0,0} ;

for( int lNo = 0 ; lNo < activeLights[1] ; lNo++ )
{
  float3 vToLight = ( diffuseLightPos[lNo].xyz - input.pos ) ;
  float distToLight = length( vToLight ) ;
  vToLight /= distToLight ;

  // sin^2 angle will be 1.0 if no blockage
  float blockageStrength = pow( getSin2AcuteAngle(
    blockageDirection, vToLight ),
    vo.y*blockageFF*distToLight ) ;
  float causticStrength  = pow( getSin2AcuteAngle(
    causticDirection, vToLight ), -vo.w*causticFF ) ;
  float3 shadowAtten = blockageStrength * (1-slTransColor) ;
  // atten. by blockage*(OPACITY). If not opaque no blockage.
  float3 causticAmp  = causticStrength * slTransColor ;
  // mult by amp*TRANSLUCENCY. If not trans no amp.

  // Amplification/attenuation due to THIS BLOCKER.
  // caustic+shadow fight each other
  p[lNo] *= ( shadowAtten + causticAmp ) ;
  // amp/reduce primary color

  // SECONDARY, DD
  float3 surfToLight = normalize( diffuseLightPos[lNo].xyz -
```

```
    slPos ) ;
  float dpower = max( dot( surfToLight, slSurfNDiffuse ), 0 ) ;
  // find power remote surface can send me
  secondaryDiffuseFromDiffuse += dpower *
    diffuseLightColor[lNo].rgb * slSurfDiffuseColor *
    input.cDiffuse.rgb ;

  // DS
  float3 vToSurf = normalize( slPos - input.pos.xyz ) ;
  float3 vToSurfRefl = reflect( vToSurf, slSurfNSpecular ) ;
  float spower = slSurfNSpecularPower * max( dot( vToSurfRefl,
    surfToLight ), 0 ) ;
  spower = pow( spower, voI.w ) ; // "hurt it" (reduce)
  // proportionally to material exponent.
  // spower is already proportional to "how much
  // vToSurfRefl lines up with surfToLight."
  secondaryDiffuseFromSpecular += spower *
    slSurfSpecularColor * input.cDiffuse.rgb ;

  // SS:
  // want eyeRefl==vToSurf, and eyeReflRefl==surfToLight
  // eyeRefl must POINT TO slPos.
  float ssLineup = max( dot( vToSurf, eyeRefl ), 0.0 ) ;
  ssLineup = pow( ssLineup, 1-slSurfNSpecularPower ) ;
  // WIDEN (0.4 nominal.. use 1-ff) smaller=WIDER band
  // forgiveness in lineup depends on SIZE of remote surface.
```

```
      // eyeReflRefl must POINT TO LIGHT to get SS refln
      float3 eyeReflRefl = reflect( eyeRefl, slSurfNSpecular ) ;
      // eye Reflect vertex Reflect remoteSurface
      spower = ssLineup *
        max( dot( eyeReflRefl, surfToLight ), 0 ) ;
      spower = pow( spower, voI.w ) ;
      // voI.w is the exponent to be used in SS reflections
      secondarySpecularFromSpecular += spower *
        slSurfSpecularColor * input.cSpecular.rgb ;
    }

    // add in the secondary contributions FROM ALL LIGHTS
    // due to this blocker
    fColor +=
      +secondaryDiffuseFromDiffuse * voI.x
      +secondaryDiffuseFromSpecular * voI.y
      +secondarySpecularFromSpecular * voI.z
    ;
  }

  // Add in all 8 p lights
  for( int lNo = 0 ; lNo < activeLights[1] ; lNo++ )
  {
    fColor += p[lNo] ;
  }

  output.color = float4( input.cEmissive.rgb + fColor,
```

```
    1-avg(input.cTrans.rgb) ) ;
  output.color.rgb = tonemap( (1-input.cTrans.rgb) *
    output.color.rgb ) ;

  return output ;
}

// Between 0 and 1
// If the angle is obtuse you get 1.0
// Function works this way on purpose
float getSin2AcuteAngle( float3 v1, float3 v2 )
{
  // if angle >90 deg then you get just 1 back.
  float cosAngle = max( dot( v1, v2 ), 0 ) ;
  float cos2Angle = cosAngle*cosAngle ;
  return 1 - cos2Angle ; // will be 1.0 if no blockage
}

// Bends 2 normals (shadow,caustic) instead of just 1
void wave2( float time, float phase, float freq, float amp,
  float3 longAxis, float3 transverseAxis, inout float3 pos,
  inout float3 norm1, inout float3 norm2 )
{
  // the displacement to the pos variable is function of
  // amp and angle in the direction of the transverseAxis
  pos += amp*sin( time*freq + phase ) * transverseAxis ;
```

```
  // Can displace along longitudinal axis as well if wish
  //pos += amp*sin( time*freq + phase ) * longAxis ;

  // the change to the normal components
  float slope = freq*amp*cos( time*freq + phase ) ;
  float t = atan( slope ) ;

  float3 zAxis = cross( longAxis, transverseAxis ) ;

  // rotation about "z" axis
  norm1 = normalize( mul( rotation( zAxis, t ), norm1 ) ) ;
  norm2 = normalize( mul( rotation( zAxis, t ), norm2 ) ) ;
}
```

Appendix B

Hardware

B.1 Benchmark Machine

The benchmark machine is as follows:

OS Version: Microsoft Windows 7 Professional

System RAM: 8173 MB

CPU Name: Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz

CPU Speeds: 3.4 GHz

Physical CPUs: 1

Virtual CPUs: 8

Video Card Description: NVIDIA GeForce GTX 550 Ti

VRAM: 1024 MB

Primary Display Resolution: 1600x1200

Multi-Monitor Desktop Resolution: 3520x1200

Windows Experience Index Rating: 7.2

All simulations are run in DirectX 11 over Windows 7.

Bibliography

[1] James Arvo, Philip Dutré, Alexander Keller, Henrik Wann Jensen, Art Owen, Matt Pharr, and Peter Shirley. Monte carlo ray tracing. In *ACM SIGGRAPH 2003 Course Notes*, SIGGRAPH '03, New York, NY, USA, 2003. ACM.

[2] Ian Ashdown. *Radiosity: a programmer's perspective*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[3] Louis Bavoil, Miguel Sainz, and Rouslan Dimitrov. Image-space horizon-based ambient occlusion. In *ACM SIGGRAPH 2008 talks*, SIGGRAPH '08, pages 22:1–22:1, New York, NY, USA, 2008. ACM.

[4] Louis Bavoil, Miguel Sainz, and Rouslan Dimitrov. Image-space horizon-based ambient occlusion. In *ACM SIGGRAPH 2008 talks*, SIGGRAPH '08, pages 22:1–22:1, New York, NY, USA, 2008. ACM.

[5] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, December 2008.

[6] Gregory Nichols Brent Burley and Jared Johnson of Walt Disney Animation Studios. BRDF Explorer. http://www.disneyanimation.com/technology/brdf.html.

[7] Michael F. Cohen, John Wallace, and Pat Hanrahan. *Radiosity and realistic image synthesis*. Academic Press Professional, Inc., San Diego, CA, USA, 1993.

[8] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. *SIGGRAPH Comput. Graph.*, 18(3):137–145, January 1984.

[9] NVIDIA Corporation. Optix. https://developer.nvidia.com/optix.

[10] Carsten Dachsbacher and Jan Kautz. Real-time global illumination for dynamic scenes. In *ACM SIGGRAPH 2009 Courses*, SIGGRAPH '09, pages 19:1–19:217, New York, NY, USA, 2009. ACM.

[11] Cyrille Damez and François X. Sillion. Space-Time Hierarchical Radiosity. In D. Lischinski and G.W. Larson, editors, *Rendering techniques '99 (Proceedings of the 10th Eurographics Workshop on Rendering)*, volume 10, pages 235–246, Grenade, Espagne, June 1999. Eurographics, Springer-Verlag/Wien.

[12] Michael Deering, Stephanie Winner, Bic Schediwy, Chris Duffy, and Neil Hunt. The triangle processor and normal vector shader: a vlsi system for high performance graphics. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '88, pages 21–30, New York, NY, USA, 1988. ACM.

[13] Elsayed M. E. Elbarbary and Salah M. El-Sayed. Higher order pseudospectral differentiation matrices. *Appl. Numer. Math.*, 55(4):425–438, December 2005.

[14] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[15] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modeling the interaction of light between diffuse surfaces. In *Proceedings*

of the 11th annual conference on Computer graphics and interactive techniques, SIGGRAPH '84, pages 213–222, New York, NY, USA, 1984. ACM.

[16] H. Gouraud. Continuous shading of curved surfaces. *Computers, IEEE Transactions on*, C-20(6):623 – 629, june 1971.

[17] Robin Green. Gritty details of spherical harmonic lighting. http://www.research.scea.com/gdc2003/spherical-harmonic-lighting.pdf, 2003.

[18] Paul S. Heckbert. *Simulating Global Illumination Using Adaptive Meshing*. PhD thesis, EECS Department, University of California, Berkeley, Jun 1991.

[19] Sebastian Herholz, Timo Schairer, and Wolfgang Stra. Screen space spherical harmonics occlusion (S_3HO) sampling. In *ACM SIGGRAPH 2011 Posters*, SIGGRAPH '11, pages 76:1–76:1, New York, NY, USA, 2011. ACM.

[20] Joseph Ivanic and Klaus Ruedenberg. Rotation matrices for real spherical harmonics. direct determination by recursion. *Journal of Physical Chemistry*, 100(15):6342–6347, 1996.

[21] Joseph Ivanic and Klaus Ruedenberg. Rotation matrices for real spherical harmonics. direct determination by recursion. *The Journal of Physical Chemistry A*, 102(45):9099–9100, 1998.

[22] Arne Jensen and Anders la Cour-Harbo. *Ripples In Mathematics*. Springer, 2001.

[23] Henrik Wann Jensen. Global illumination using photon maps. In *Proceedings of the eurographics workshop on Rendering techniques '96*, pages 21–30, London, UK, 1996. Springer-Verlag.

[24] James T. Kajiya. The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '86, pages 143–150, New York, NY, USA, 1986. ACM.

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

[26] Janne Kontkanen and Samuli Laine. Ambient occlusion fields. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games*, I3D '05, pages 41–48, New York, NY, USA, 2005. ACM.

[27] Anders Wang Kristensen, Tomas Akenine-Möller, and Henrik Wann Jensen. Precomputed local radiance transfer for real-time lighting design. In *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, pages 1208–1215, New York, NY, USA, 2005. ACM.

[28] Jaroslav Křivánek, Jaakko Konttinen, Sumanta Pattanaik, Kadi Bouatouch, and Jiří Žára. Fast approximation to spherical harmonics rotation. In *ACM SIGGRAPH 2006 Sketches*, SIGGRAPH '06, New York, NY, USA, 2006. ACM.

[29] Samuli Laine, Hannu Saransaari, Janne Kontkanen, Jaakko Lehtinen, and Timo Aila. Incremental instant radiosity for real-time indirect illumination. In *Proceedings of Eurographics Symposium on Rendering 2007*, pages 277–286. Eurographics Association, 2007.

[30] Christian Lessig and Eugene Fiume. Soho: Orthogonal and symmetric haar wavelets on the sphere. *ACM Trans. Graph.*, 27(1):4:1–4:11, March 2008.

[31] Sam Martin. Enlighten real-time radiosity. In *ACM SIGGRAPH 2011 Computer Animation Festival*, SIGGRAPH '11, pages 97–97, New York, NY, USA, 2011. ACM.

[32] Sam Martin. Film: Enlighten real-time radiosity. In *ACM SIGGRAPH 2011 Real-Time Live!*, SIGGRAPH '11, pages 3:1–3:1, New York, NY, USA, 2011. ACM.

[33] Sam Martin and Per Einarsson. A real-time radiosity architecture for video games. In *ACM SIGGRAPH 2010 Courses*, SIGGRAPH '10, 2010.

[34] M. McGuire. Ambient occlusion volumes. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 47–56, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[35] Phil Miller. GPU Ray Tracing. http://developer.nvidia.com/siggraph-2011, 2011.

[36] Robert B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, AFIPS '68 (Fall, part I), pages 267–277, New York, NY, USA, 1968. ACM.

[37] Martin Mittring. Finding next gen: Cryengine 2. In *ACM SIGGRAPH 2007 courses*, SIGGRAPH '07, pages 97–121, New York, NY, USA, 2007. ACM.

[38] Ren Ng, Ravi Ramamoorthi, and Pat Hanrahan. All-frequency shadows using non-linear wavelet lighting approximation. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pages 376–381, New York, NY, USA, 2003. ACM.

[39] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005.

[40] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18:311–317, June 1975.

[41] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 3 edition, 2007.

[42] Ravi Ramamoorthi and James O'Brien. Cs 294-13, advanced computer graphics. http://inst.eecs.berkeley.edu/~cs294-13/fa09/.

[43] RIGIDGEMS. http://www.rigidgems.sakura.ne.jp/.

[44] Tobias Ritschel, Thorsten Grosch, and Hans-Peter Seidel. Approximating dynamic global illumination in image space. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, I3D '09, pages 75–82, New York, NY, USA, 2009. ACM.

[45] Volker Schönefeld. Spherical harmonics. http://limbicsoft.com/volker/prosem_paper.pdf, 2005.

[46] Peter Schröder and Pat Hanrahan. On the form factor between two polygons. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '93, pages 163–164, New York, NY, USA, 1993. ACM.

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

[48] William Sherif. Graphics test project source code. https://github.com/superwills/gtp/, 2012.

[49] Peter-Pike Sloan, Jesse Hall, John Hart, and John Snyder. Clustered principal components for precomputed radiance transfer. *ACM Trans. Graph.*, 22(3):382–391, July 2003.

[50] Peter-Pike Sloan, Jan Kautz, and John Snyder. 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*, SIGGRAPH '02, pages 527–536, New York, NY, USA, 2002. ACM.

[51] Peter-Pike Sloan, Ben Luna, and John Snyder. Local, deformable precomputed radiance transfer. In *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, pages 1216–1224, New York, NY, USA, 2005. ACM.

[52] Eric Stollnitz, Tony Derose, and David Salesin. *Wavelets for Computer Graphics*. Morgan Kaufmann, 1996.

[53] Wim Sweldens. The lifting scheme: A new philosophy in biorthogonal wavelet constructions. In *in Wavelet Applications in Signal and Image Processing III*, pages 68–79, 1995.

[54] Sinje Thiedemann, Niklas Henrich, Thorsten Grosch, and Stefan Müller. Voxel-based global illumination. In *Symposium on Interactive 3D Graphics and Games*, I3D '11, pages 103–110, New York, NY, USA, 2011. ACM.

[55] Robert F. Tobler, Laszlo Neumann, Mateu Sbert, and Werner Purgathofer. A new form factor analogy and its application to stochastic global illumination algorithms. Technical Report TR-186-2-98-33, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, June 1998.

[56] Eric Veach and Leonidas J. Guibas. Metropolis light transport. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '97, pages 65–76, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[57] Rui Wang, Ren Ng, David Luebke, and Greg Humphreys. Efficient wavelet rotation for environment map rendering. In *Proceedings of the 17th Eurographics conference on Rendering Techniques*, EGSR'06, pages 173–182, Aire-la-Ville, Switzerland, Switzerland, 2006. Eurographics Association.

[58] Eric W. Weisstein. Integral equation Neumann series. from mathworld–a wolfram web resource. http://mathworld.wolfram.com/IntegralEquationNeumannSeries.html.

[59] Mick West. Programming responsiveness. http://cowboyprogramming.com/2008/05/27/programming-responsiveness/.

[60] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23:343–349, June 1980.

[61] Kun Zhou, Yaohua Hu, Stephen Lin, Baining Guo, and Heung-Yeung Shum. Precomputed shadow fields for dynamic scenes. In *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, pages 1196–1201, New York, NY, USA, 2005. ACM.

[62] Sergej Zhukov, Andrej Iones, and Grigorij Kronin. *An Ambient Light Illumination Model*, pages 45–56. Springer Wien, 1998.