Project Number: EOA-0017

RENDERING LEAVES DYNAMICALLY IN REAL-TIME

A Major Qualifying Project Report
submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Bachelor of Science
by

Madison Dickson
John Sandbrook
Michael Oliver

April 28, 2011

Approved:

Professor Emmanuel O. Agu, Advisor

Abstract

Methods of Global Illumination, which model inter-reflections between objects in a scene, are key to creating realistic images. A recent technique that approximates Global Illumination is Light Propagation Volumes (LPVs), which runs at interactive frame rates, or 'realtime'. This report details the design and implementation of a realtime Global Illumination approximation using LPVs, simulating both reflected and transmitted diffuse light. It employs 4^{th} degree polynomial Spherical Harmonics on separate color channels to represent the flow of light, and its complexity is independent of resolution or geometry. A modular framework using OpenGL version 3.3 was built to support the task and implements a deferred rendering system to facilitate the multiple passes required by the technique employed. A Reflective Shadow Map is used to build a model of the lighting environment inside the LPV, which is then processed to propagate light through the scene. The program runs at an average of 40 FPS on modern graphics hardware and uses less than 50MB of system memory and 70MB of GPU memory, making it a viable option for realtime rendering applications. The scene displayed is a leafy bush and ground plate.

Keywords: OpenGL, Light Propagation Volumes, Spherical Harmonics, Global Illumination, Indirect Lighting, real-time, realtime

Contents

List of Equations

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The Problem

Achieving realtime photorealism is the ultimate goal of computer graphics. Empirical lighting models[1] are insufficient for this task; physically based algorithms must be devised with the limitations of current hardware in mind. How humans perceive light and interpret images are also major factors in the believability of an image. Shadows and numerous other visual cues are used to understand images and the spatial relationships of the objects within a scene. These cues can be exploited to render realistic images even when the underlying model is not a strictly accurate representation.

Many important visual cues are modeled by Global Illumination (GI), which refers to any technique that takes into account light interactions between objects in a scene. GI methods include numerous shadowing techniques, Radiosity to simulate the way surfaces illuminate each other, Ambient Occlusion (AO) to portray how close surfaces mutually darken each other, and various ways to calculate reflections, refractions, and transmittance[3].

The development of powerful hardware-based graphics acceleration continually increases the potential for realism in realtime graphics: dedicated graphics hardware can perform millions of floating point calculations per second and support custom algorithms that can approximate all

[1] explained in Section 2.3 on page 5

types of GI, bringing realtime photorealism closer to reality.

Light Propagation Volumes (LPVs) is one such GI method that has recently gained notoriety. First used by Crytek, LPV coarsely models the way light flows through a scene by separating a given scene volume into cubes, and iteratively propagating the light through that volume[16].

1.2 Project Goal

Accounting for the full lighting environment of a scene is essential to achieving realism. This project renders in realtime a visually accurate approximation of leaves lit with a single source (the sun) using GI techniques. We consider 'realtime' to be at least 30 frames per second (FPS). The level of realism achieved will be compared with photographic references and with images rendered on commercial rendering platforms and game engines. We want to model the way light is reflected off and transmitted through leaves, as shown by the mulberry leaves in Figure 1.1. Additional subtle effects will be used to enhance the realism of the final image; surfaces behind a leaf should receive color from the light transmitted through it, while shadows cast on upper surfaces should affect the transmitted light seen through the bottom surface.

We will incorporate LPVs using Spherical Harmonics (SH) to describe the lighting environment of a complex scene that incorporates translucent media. Lambertian shading and an ambient sphere map will be used for additional environment lighting. Because common shader libraries and demo packages do not support the level of control needed to implement these advanced methods, particularly the dynamic generation of three dimensional textures, we will design a custom framework in C.

Figure 1.1: Sun shining behind mulberry leaves[9]

Chapter 2

Graphics Background

2.1 General Shading

Shading refers to any method of ”determining the effect light has on a material”[3]. The goal of shading is to produce an image that ‘makes sense’ to the viewer.

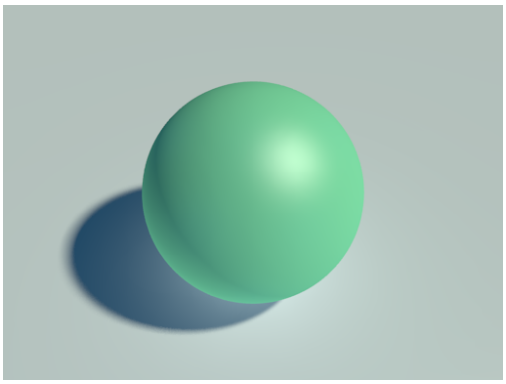


Figure 2.1: Lambertian reflectance, Cook-Torrance specularity, indirect lighting and shadows give a simple sphere shape and depth. Rendered in Blender[8].

2.2 Technical Shading

Shading in computer graphics involves altering an object's color and brightness based on the properties of the object. This commonly includes the color of the material, the angle to, distance from, and color of light sources, and the viewing angle as well. Shading is independent of the representation of a surface as long as the needed data for any particular shading algorithm can be derived from the surface representation.

Current models of the process of realtime shading on computers fall into two broad categories. These are Forward Shading and Deferred Shading. Note that either model can be used for non-realtime, or offline, pipelines, but we will discuss them in terms of realtime. Offline implementations of either model do not necessarily work in the same way.

Two terms that are essential to distinguish these two models of rendering are the concept of fragments and the technique of depth testing. A fragment is a rasterized piece of the image that can receive a color. Fragments are related to pixels; after geometry is transformed into a coordinate space that can be displayed, it is cut up into pieces that eventually determine the color pixels, and these pieces are called fragments. Fragments do not need to directly correspond to display pixels. Shading on a pixel based device is applied to fragments[3].

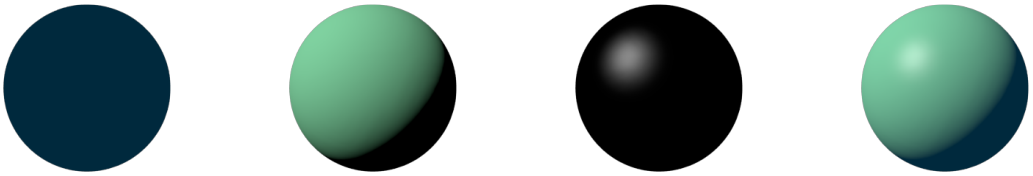
Depth testing is related to the fact that graphics hardware usually takes advantage of parallelization, running multiple processes or threads of execution, sometimes on multiple separate processing cores. In addition to this, many systems do not strictly order this processing relative to actual scene space: parts of a scene may be processed out of order. To avoid objects being drawn over each in the wrong order, depth testing is used to sort fragments that have already been colored so that only the fragments closest to the viewer are seen[3].

2.3 A General Shading Model

Shading is a process only concerned with assigning the outgoing radiance along a view direction from a surface given its properties and the properties of the light that illuminates it. It does not in itself deal with more complex issues such as global illumination[3].

The general model of shading involves three generalizations of the lighting environment of an object. These are the ambient, diffuse, and specular terms, which each describe a component of how a surface is lit. Observe Figure 2.2 for a visual representation of the differences. This division is not physically accurate; it is instead informed by the perceptual components of an image that are easy to control and provide believable results.

Understanding how these three components are put together creates a foundation for understanding how more advanced techniques and methods work. For the purpose of these examples, color variables are represented as three component vectors containing the value of the red, green, and blue color channels .



(a) Ambient term

(b) Diffuse Term (Lambert)

(c) Specular Term (Blinn-Phong)

(d) Sum of basic shading terms

Figure 2.2: Examples of the three basic lighting terms and their sum. Rendered in Blender[8]

Ambient light

Ambient light is a base approximation of the environmental light in a setting. It accounts for the light that has reflected off all objects in a scene in such a manner that it "seems to come from everywhere" [7]. The Lighting Design Lab describes ambient light as "lighting throughout a space that produces uniform illumination"[18]. Without applying other shading models, every object in the scene with an ambient term will have the same color, defined by the color and intensity of the ambient light source. Equation 2.1 defines the typical ambient term found within many general shading models[3].

EQUATION 2.1

$$\mathcal{C}_A = \mathcal{C}_{ambient}$$

$\mathcal{C}_A$: shaded ambient color

$\mathcal{C}_{ambient}$: the ambient color

Note that we do not include any intensity term for the ambient light in this formulation. We assume that any such factor is accounted for in the term $\mathcal{C}_{ambient}$, a simplifying convention we use for the following formula.

Diffuse light

Diffuse shading describes the interaction of a finely rough surface and a light source. At each point of calculation, diffuse terms often only deal with the direction from the point on the surface to the light source, and so are generally agnostic to the type of light source. As an approximation, diffuse shading assumes an infinitely rough surface such that it scatters and reflects light uniformly and independently of the viewing direction. The intensity of the resulting color depends on the intensity and color of the light source and the angle between the direction to the light and the surface normal. The diffuse intensity will be greatest when it is parallel to the normal, and the least when it is perpendicular.

The classic model is called Lambertian Reflectance, and is mathematically defined as the dot-product[1] of the surface normal and the direction to the light multiplied by the material's diffuse color and the light color. This is shown in equation 2.2[3].

[1]We are using the dot product directly, whereas the proper definition uses the cosine of the angle between the normal and light direction. Using the angle between the vectors is required when the normal and light direction are not unit vectors.

EQUATION 2.2

$$\mathcal{C}_D = \mathcal{C}_{diffuse} * \mathcal{C}_{light} * (\hat{\mathbf{N}} \odot \hat{\mathbf{L}}) * \frac{1}{\pi}$$

$\mathcal{C}_D$: shaded diffuse color

$\mathcal{C}_{diffuse}$: the diffuse color of the surface

$\mathcal{C}_{light}$: the color of the light source

$\hat{\mathbf{N}}$: the normal to the surface

$\hat{\mathbf{L}}$: the direction to the source of light

$*$: per component vector multiplication

$\odot$: the dot product clamped to the range $[0, 1]$

Specular light

Specular reflection (or specular highlight) is a type of shading that models reflected light, like diffuse shading. However, specular shading is also dependent on the viewer's location. It provides the glossy, shiny look of plastics, wet materials, and other reflective surfaces that can reflect non-scattered light directly from the light source to the viewer's eye. The color of specular reflection at a given point on the surface is dependent on the strength of the light source, the vector from the point to the viewer, the vector from the point to the light source, the surface normal, and the material's specular color. In equation 2.3, we show the mathematically complete specular term[3].

EQUATION 2.3

$$\mathcal{C}_\mathcal{S} = \mathcal{C}_{specular} * \mathcal{C}_{light} * (\hat{\mathbf{N}} \odot \hat{\mathbf{L}}) * \frac{m+8}{8\pi}(\hat{\mathbf{N}} \odot \hat{\mathbf{H}})^m$$

$\mathcal{C}_S$: shaded specular color

m : the specular power; larger values result in brighter and smaller highlights

$\hat{\mathbf{H}}$: the vector halfway between the direction to the eye and the light direction

$\mathcal{C}_{specular}$: the specular color of the surface

General Shading

Combining these three shading terms provides a simple parameterized model of the behavior of a surface with some physical basis. However, most general shading models drop the extra factors which involve π and instead use intensity factors to adjust the effect of each term directly. Equation 2.4 shows this sum mathematically and Figure 2.2 shows it visually. This general equation is very similar to the Blinn-Phong shading model, which is shown in Equation 2.5 for comparison[7].

EQUATION 2.4

$$\begin{aligned}
\mathcal{G}_{color} &= \mathcal{C}_A + \mathcal{C}_D + \mathcal{C}_S \\
\mathcal{G}_{color} &= \mathcal{C}_{ambient} \\
&\quad + \mathcal{C}_{diffuse} * \mathcal{C}_{light} * (\hat{\mathbf{N}} \odot \hat{\mathbf{L}}) * \frac{1}{\pi} \\
&\quad + \mathcal{C}_{specular} * \mathcal{C}_{light} * (\hat{\mathbf{N}} \odot \hat{\mathbf{L}}) * \frac{m+8}{8\pi} (\hat{\mathbf{N}} \odot \hat{\mathbf{H}})^m
\end{aligned}$$

$\mathcal{G}_{color}$: General Model shaded color

EQUATION 2.5

$$\begin{aligned}
\mathcal{P}_{color} &= \mathcal{C}_A + \mathcal{C}_D + \mathcal{C}_S \\
\mathcal{P}_{color} &= \mathcal{C}_{ambient} \\
&\quad + \mathcal{C}_{diffuse} * \mathcal{C}_{light} * (\hat{\mathbf{N}} \odot \hat{\mathbf{L}}) \\
&\quad + \mathcal{C}_{specular} * \mathcal{C}_{light} * (\hat{\mathbf{N}} \odot \hat{\mathbf{H}})^m
\end{aligned}$$

$\mathcal{P}_{color}$: Blinn-Phong Model shaded color

2.4 Forward Shading

Forward shading[2] is the traditional model of rendering a scene in realtime whereby depth testing and shading happen together[3]. When forward shading, geometry is processed and pixels colored directly in the output image. All shading calculations happen in this single pass, even on fragments

[2]This term is not strictly defined except to contrast it from Deferred Shading. We list it first because it is by far more common.

that will later be discarded by depth testing. In the general case, forward shading output goes directly to the display device, or to the back buffer of a double buffered system. The Fixed Function Pipeline (FFP) of OpenGL is specified as a forward shading system[3], as are traditional graphics engines used for video games. Primitives are given to the pipeline with certain parameters and they are rendered to the main framebuffer.

2.5 Deferred Shading

Deferred shading is a method of rendering a scene usually contrasted with forward shading. Under a deferred framework, shading operations are set aside for a later processing step and a number of output buffers[4] are used to store relevant shading data from the scene. This data may include surface normals, albedo, and any other information deemed necessary. These images are then used as the input data for shading algorithms.

By deferring the expensive shading calculations the rendering system does not waste time shading fragments that will never be seen. Instead, depth testing selects only one set of source data and actual shading occurs just once for every pixel of the output image. The process is analogous to a complex PhotoShop® file, allowing a great deal of flexibility in the possible effects that the shading pass can include. With access to all this data through texture lookups, a variety of post-processing techniques such as depth of field and motion blur can be easily applied to the scene[3]. Crytek's CryENGINE[5] uses deferred shading to include a scene-independent customizable color grading pass to help set the mood in their games[5].

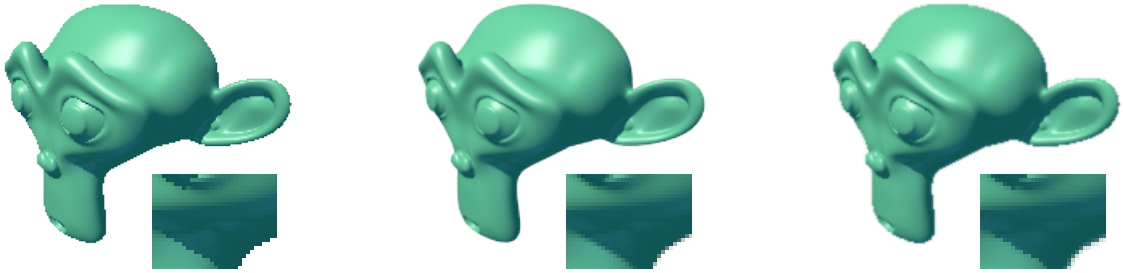
Deferred shading does have drawbacks, beyond the bandwidth involved with using g-buffers. For instance, anti-aliasing of primitive edges does not function properly. It is thus necessary to handle the 'jaggie'[6] artifacts that result, as shown in Figure 2.3. Crytek uses a blur on areas

[3]The FFP can be used for Deferred Shading, but it is complicated and involves expensive memory operations.

[4]Referred to as g-buffers in the literature.

[5]CryENGINE is actually a hybrid system. Crytek has combined several methods, including many pre-processing and post-processing algorithms, in a non-trivial manner which is beyond the scope of this paper[5].

[6]A 'jaggie' describes the jagged edges caused by a rasterized object's edge that falls in-between two pixels, resulting in an edge clamped to the nearest whole pixel[3].



(a) Jaggies resulting from no anti-aliasing

(b) Jaggies fixed with Mitchell-Netravali anti-aliasing

(c) Jaggies fixed in post-processing with edge blur

Figure 2.3: Jaggies; anti-aliasing does not work with deferred shading because multi-sampling cannot occur across separate output buffers. Images 2.3b and 2.3c use post processing in PhotoShop by blurring on edges detected from normals. Rendered in Blender[8].

identified with an edge detection algorithm[5].

Also, it is difficult to use different algorithms on different objects, since the granularity of the scene has been reduced to one large collection of shading input data. Using color object IDs with control branching in the final shader is a naïve solution, but adds conditionals to the shader code which will affect performance[3]. Some techniques also cannot easily benefit from deferred frameworks. The more robust versions of relief mapping require direct access to unmapped textures to accurately warp texture coordinates. If one of the g-buffers contains diffuse color information, these operations must happen in the first stage of deferred shading to produce correct results.

The popular game Killzone uses deferred shading, designating several g-buffers for various special effects, such as accurate motion blur[3]. Crytek manages to pack almost all of the data their algorithms need into only two g-buffers[5]. Since Crytek has extensively documented their methods[5, 16], which complement our need to render various components of the lighting model in separate passes, much our code is inspired by their work.

2.6 Graphics packages

2.6.1 Graphics Application Programming Interfaces (APIs)

Graphics APIs provide an interface to graphics hardware from within a user application, allowing it to set up a graphics context and initiate draw calls. The two common graphics APIs, OpenGL and and portions of Microsoft's DirectX, are in fact published standards which describe how an implementation should behave, much like how C and C++ are defined.[7] Hardware vendors implement these standards through a combination of hardware, microcode, and drivers. We chose to use OpenGL because it works on any Operating System (OS).

OpenGL

OpenGL[8] is a graphics standard maintained by the Khronos Group, an association of graphics software and hardware vendors including NVIDIA, Intel, Texas Instruments, and Epic Games[11]. Originally, OpenGL was an open substitute for the proprietary scripting language Iris GL, which was used for programming graphics applications on Silicon Graphics Inc. workstations.

Over the four major versions of the standard[9], OpenGL has been altered drastically from a strict Fixed Function Pipeline to a highly customizable graphics processing interface. A programmer may alter key stages of the pipeline to achieve nearly any effect desired. The Khronos Group has since created the similar OpenGL ES API, an embedded hardware and mobile platform based version of OpenGL which shares aspects of both the new and old OpenGL standards[12].

2.6.2 Programmable Graphics Pipeline

In terms of paradigm shifts, one of the most significant recent developments in graphics has been the introduction of programmable stages to the graphics pipeline as described by DirectX and OpenGL. Generally speaking, the Graphics Processing Unit (GPU) of dedicated graphics hardware

[7] Also like C and C++, OpenGL implementations can have compliance issues. See Section 6.2.3 on page 68.

[8] The Open Graphics Language

[9] The majority of the OpenGL revisions have been released in the last five years[28].

is a highly specialized processor with an emphasis on parallelized floating point operations and, in particular, common linear algebra operations such as dot products. The rendering pipeline of these devices is essential to their speed and power[3].

Consequently, only a few key areas in the calculation of pixels can actually be altered, namely the processing of raw geometry, the amplification of geometry, and the shading of rasterized primitives. Before 2001, all hardware was fixed-function and the graphics pipeline was straightforward and inflexible[7, 33]. There was essentially one model for shading with only a few user controllable features. The graphics cards of today allow for programmable instructions to change how an image is rendered, and can even be used for non-rendering purposes such as fluid simulations and character animations[7]. The programs that do this, called shaders, are loaded onto the GPU at runtime. Figure 2.4 provides a graphical representation of the standard programmable pipeline.

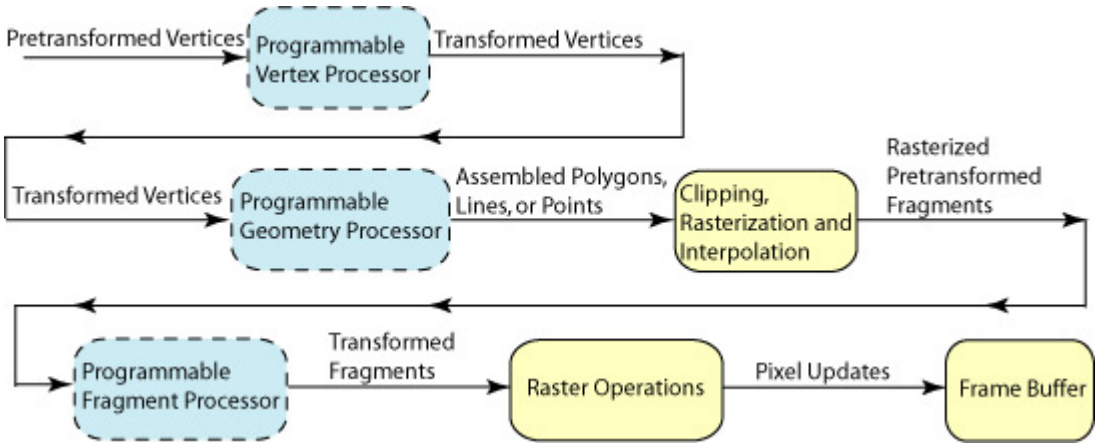


Figure 2.4: A generalized diagram of the typical graphics pipeline[26]

Vertex Shaders

Vertex shaders handle operations on incoming vertices into the pipeline; what primitives these vertices describe is not known by a vertex shader. Vertex shaders can only perform operations on the specific vertex and attributes passed in. The vertex shader is analogous to the automatic transformation of vertices using the ModelView and Projection matrices of the OpenGL Fixed

Function Pipeline (FFP).

Geometry Shaders

Unlike vertex shaders, geometry shaders have access to the primitive assembly stage of the pipeline. This means that, once raw inputed vertices are processed and transformed into normalized device coordinates, geometry shaders may emit or 'amplify' them to create geometric primitives such as triangles and lines which may not contain the same number of vertices. The Geometry shader can be used to duplicate a simple but often repeated object in large quantities without having to store them all in memory, or to generate dynamic geometry such as terrain[22].

Geometry shaders are a recent addition[10], and because of the complexity of the primitive assembly process are not yet well suited to amplification factors of more than two dozen[27]. There is no analogue for this stage in the FFP since primitive assembly is handled internally.

Fragment Shaders

After primitives are assembled, they are rasterized into fragments that are mapped to individual pixels on the display device. The fragment shader's job, in broad terms, is to give each fragment a color.

Like vertex shaders, fragment shaders have no knowledge of the context of the fragment being shaded. The values of attributes which were assigned on a per-vertex basis are interpolated between vertices across primitives and these values are made available to the fragment shader.

This stage also has few analogous portions in the OpenGL FFP. Under the FFP, the programmer's only interaction with this stage is via the setting of lighting and material parameters in the OpenGL rendering context before a vertex was loaded into the pipeline.

2.6.3 Shading Languages

Both OpenGL and DirectX implement graphics shading languages, designed to give control over the graphics pipeline using a high-level C-like language.

[10] Geometry shaders were added to the core OpenGL in version 3.2, which was released in 2009 [27, 28]

Graphics Language Shading Language (GLSL)

GLSL is based on C, and was created by the OpenGL Architecture Review Board[33]. GLSL can use geometry, vertex, and fragment shaders, which are compiled at runtime.[11] Benefits of GLSL include cross platform compatibility on Windows, Mac OSX, and most Linux distributions, and is hardware independence; it works on all OpenGL GLSL enabled graphics chips[33]. Another added benefit is that each hardware vendor includes the GLSL compiler with their drivers, allowing for architecture specific optimizations. While it cannot use pointers, support for structured data was recently added[28].

High-Level Shading Language (HLSL)

HLSL is the proprietary shading language for the Microsoft Direct3D Application Programming Interface (API), and parallels OpenGL's GLSL.

C for Graphics (Cg)

NVIDIA's proprietary graphics language Cg was developed closely with Microsoft as the first high level shading language, and thus Cg and HLSL have many similarities[7].

2.6.4 Shader Tools & Graphics Engines

Writing the entire framework to support shader development is tedious and time consuming if the goal is is to quickly design and test shader effects. As a result, several packages have been designed to provide the backbone and front end, and sometimes even include a Graphical User Interface (GUI) to design the effect programing. The following is a list of the options most relevant to our project, but is not exhaustive.

[11] GLSL version 4.1 introduced the ability to load pre-compiled shaders to reduce loading time and increase overall performance[28].

GLSLDemo

GLSLDemo is a small package designed to quickly test GLSL shaders on a variety of models. It is limited to Vertex and Fragment shaders, but provides an XML interface for variable input with slider GUI elements. Originally developed by 3DLabs, it was dropped in 2006 when the company changed its direction[2].

FX Composer

NVIDIA's most recent graphics package is called FX Composer, currently on version 2.5[23]. It is a robust shader designer with advanced features and user-friendly wizards and templates. It has cross-API support, including DirectX 9, 10, and OpenGL, and can export GLSL, HLSL, and CgFX code natively. Shader FX even has a built-in particle system for effects programming.

Unity Game Engine

The Unity game engine was designed to make game development quick and easy while still providing a powerful engine[38]. It can export to both Mac and PC, mobile devices, consoles like the Wii and Xbox 360, and even the web via their browser plug-in.

Blender Game Engine

The Blender Game Engine is included with the 3D modeling and animation program Blender. It uses Python for advanced control of nearly every aspect of the game and rendering, and supports all OpenGL lighting modes and GLSL shaders[8]. Support is included for the open source Bullet Physics Library, originally developed for the Playstation 3, for use in collision detection and rigid body dynamics[8, 19].

2.6.5 OpenGL Extension Wrangler (GLEW)

Native OpenGL implementations are frozen at version 1.1 on Windows and up to 1.4 on the latest versions of Linux. This limits the functionality of OpenGL programs which use only the Operating

Systems (OSs) built-in support, which does not even include buffer objects[12] to store vertex information. Furthermore, the majority of the functionality in these versions has since been deprecated. In order to render anything with modern OpenGL the available extensions and core functionality must be loaded at runtime.

There are hundreds of functions in OpenGL to check for driver support. GLEW performs this task for the programmer by testing everything in the OpenGL specification on the current hardware and loading all available parts of the API. Version information is exposed for querying so precautions can be taken in an application's code to accommodate, cope, or simply remove advanced functionality if older hardware is being used [13].

2.6.6 OpenGL Utility Toolkit (GLUT)

Every windowing system Application Programming Interface (API) has its own way to interface with OpenGL. The details of these interfaces are largely Operating System (OS) dependent and by definition not portable.

To facilitate more portable OpenGL applications, GLUT hides the platform-dependent details of actually displaying OpenGL images with a uniform API. It contains routines for opening windows, detecting and interpreting input, timing, and several advanced primitives significantly more easy to use than those available in OpenGL utilities[36].

However, GLUT is not part of the OpenGL standard and as such has related disadvantages. The last time the original GLUT library was updated was in 1997, ostensibly with bugs remaining. The primary implementation used now is called FreeGLUT which replicates the original's functionality[25] and adds a few new features such as a new primitive type[25].

2.6.7 OpenGL Shader Wrangler (GLSW)

Shaders in OpenGL are comprised of two to three shader steps: vertex and fragment or vertex, geometry, and fragment. Each is treated as a separate block of shader code, compiled separately, and finally linked together into a single shader program. Since the shader steps reference each

[12] Buffers were introduced in OpenGL 1.5

other, making sure variables are named properly and outputs are aligned to inputs becomes a maintenance liability. GLSW takes care of this by providing string handling functions that allow a single file to containing all the shader code to be cut up into its constituent shader source. These source blocks can then be loaded into OpenGL separately[31]. Having related shaders in the same file makes shader development easier. For a library that has only six functions, GLSW is very flexible and effective. [13]

[13]There is a small bug in the library: due to an optimization concerning file loading, substitution of code blocks from one part of a file into another part does not work.

Chapter 3

Lighting

3.1 What is light?

Before discussing the methods used to accurately simulate light, it is important to understand what light is and how it behaves in the real world. Light is a flow of electromagnetic radiation which is carried by photons. Because of the quantum effects on the scale of photons, light has both particle and wave properties. For instance, the color of visible light is related its wavelength, a wave-like property, whereas the 'brightness' of light is related to the amount of photons in the beam of radiation, a particle-like property. The wave properties and secondary properties like polarization are usually ignored in computer graphics as they contribute little to the majority of scenes[3].

The study of electromagnetic radiation in general is called Radiometry; this includes radiation outside the narrow band visible to the human eye[29], which only spans wavelengths of 400nm to just over 750nm[3]. The measurement and study of this narrow band and how humans perceive it is called Photometry[42]. To the human eye, the appearance of an object is determined by the interaction between the light illuminating it and the object's surface properties.

The particle-like properties of light imply that when a photon hits a surface, it bounces off and continues on its way. However, surfaces absorb different wavelengths of light at different rates, and the microscopic details of surfaces are complex. Absorption properties will alter the wavelength

and thus the color of the light, whereas surface topology and optical properties will alter its path[1]. Light is modified by every surface it encounters until it finally exhausts its energy; a very small fraction reaches a viewer's eye and is then observed. In general terms, this means a computer can simulate a scene by summing the interaction of all light sources and objects in it. Modeling this interaction to produce images that mimic what the eye expects is the practical challenge of computer graphics[3].

Light does not bounce around a scene in a uniform manner, however. Even seemingly flat or smooth surfaces have microscopic details that affect the way light reflects and refracts off them. Standard light properties still apply: a ray of light will bounce off a surface at an incident angle equal to the incoming angle reflected about the surface's normal, but these tiny details change the outgoing angle. There are several components of reflection models, which are broadly categorized into "diffuse, glossy specular, perfect specular, and retro-reflective" reflections[30].

In addition to modeling direct lighting, our project focuses on the use of indirect lighting, or, any effect that a light ray has after its first reflection. Once a photon hits a surface, calculating its new direction, intensity, and wavelength is a complex problem with innumerable solutions for diverse situations. Though some solutions simulate the physics of lighting with high accuracy, we chose to find a solution that could run in a real time setting with dynamic geometry.

3.1.1 Abstraction of Light

Explicitly simulating photon paths quickly becomes intractable for realtime graphics, even on the best of current hardware. For instance, a $100W$ incandescent light bulb at 2% efficiency releases on the order of 10^{32} photons a second, as shown in Equation 3.1.

EQUATION 3.1

$$100W \times 2\% = 10^{32}\ photons/s$$

[1]The effects of optical properties are usually wavelength dependent, so even optical properties can alter the color of an object.

Tracking each photon's direction, intensity, position, and wavelength takes a minimum of seven floating point numbers. Using 32 bit floats, we would need to store $7 \times 32 \times 10^{32}$ bits of information, or approximately 2.3×10^{21} terabytes.

This is an impossible amount of information to deal with, requiring graphics algorithms to use generalized models of light that do not rely on full light simulation. Individual photons are ignored and instead are often modeled as rays that transport color and intensity information.

3.1.2 Representations of Light

The direct implementation of this model propagates light rays out from light sources and throughout a scene, using an eye model to accumulate the image. This quickly becomes intractable in a realtime context; the eye model only sees the very small fraction of all rays that reach it, requiring a very large number of rays to produce an image. Ideal solutions would involve reflecting each ray around the scene indefinitely, or until its energy was spent, further complicating this issue.

Since this is very inefficient, realtime solutions typically use the surface properties of objects in the scene in relation to the position and properties of lights to color surfaces. This method corresponds to finding a solution to the Rendering Equation(see Equation 3.2)[3]. The equation is a general relation between light, surface geometry, and material properties. It is a mathematical model of the ground truth of the interaction of surfaces and light sources. Many rendering algorithms, as discussed in Section 2.3, are possible solutions to the Rendering Equation, and all shading methods henceforth discussed are in some way an approximation of this ideal behavior.

EQUATION 3.2

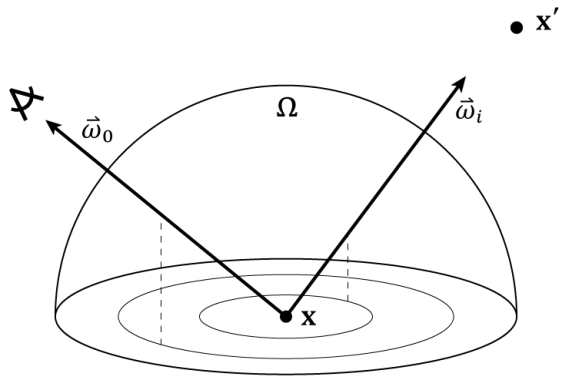


Figure 3.1: The parameters of the Rendering Equation visualized; Ω is the set $\vec{\omega}_i$ for all i.

$$L(\mathbf{x}, \vec{\omega}_0) = L_e(\mathbf{x}, \vec{\omega}_0) + \int_S f_r(\mathbf{x}, \vec{\omega}_i \rightarrow \vec{\omega}_o)\ L(\mathbf{x}', \vec{\omega}_i)\ G(\mathbf{x}, \mathbf{x}')\ V(\mathbf{x}, \mathbf{x}')\ d\omega_i$$

where:

$L(\mathbf{x}, \vec{\omega}_0)$: the radiance reflected from position $\mathbf{x}$ in direction $\vec{\omega}_o$

$L_e(\mathbf{x}, \vec{\omega}_0)$: the light emitted from $\mathbf{x}$ by the material itself

$f_r(\mathbf{x}, \vec{\omega}_i \rightarrow \vec{\omega}_o)$: the BRDF of the surface at point $\mathbf{x}$, transforming incoming light $\vec{\omega}_i$ to reflected light $\vec{\omega}_o$

$L(\mathbf{x}', \vec{\omega}_i)$: light from $\mathbf{x}'$ on another object arriving along $\vec{\omega}_i$

$G(\mathbf{x}, \mathbf{x}')$: the geometric relationship between $\mathbf{x}$ and $\mathbf{x}'$

$V(\mathbf{x}, \mathbf{x}')$: a visibility test: returns 1 if $\mathbf{x}$ can see $\mathbf{x}'$, 0 otherwise

3.2 Indirect Lighting

Indirect Lighting, part of Global Illumination, is often broken into a separate topic in light representation methods. Indirect Lighting occurs when diffusely reflected light illuminates other surfaces. When light reflects off a surface, other surfaces nearby may receive some of the original surface's reflected light. This brightens the image and helps increase its photorealism [21].

We will use the ideas of primary and secondary light sources to discuss Indirect Lighting.

Primary light sources include points that light originate from, and secondary light sources can include any point in the entire scene. Light reflecting off of any surface effectively creates another light source, caused by a diffuse reflection from the surface. This reflection creates a secondary light source which can illuminate other surfaces, possibly creating more light sources.[2] The light reflects from the surface in a diffuse manner and affects the surfaces around it.

However, calculating these effects in an efficient manner becomes increasingly complicated. Using secondary light sources, the total number of lights a scene contains exponentially grows with each light bounce. Methods implementing aggregation [16], splatting , and pre-computation all attempt to reproduce the effects of global illumination through approximations that reduce the number of bounces. Though they may not strictly adhere to the way light travels, the diffuse nature of Indirect Lighting allows these shortcuts to approach photorealism.

3.3 Mathematical Models

There are many parameters to the Rendering Equation, generally too many to use for fast solutions. Depending on the requirements of a scene, some properties may be more important than others.

For this reason, different types of lighting and shading allow different approaches which can be optimized in different ways. For instance, specular materials require a higher granularity of lighting information, like the viewing direction properties, whereas dull objects can use simpler approximations without sacrificing realism. Indirect Lighting presents different challenges: it needs full scene information including unseen surfaces, while transparency models only needs information about objects away from the eye model.

3.3.1 Bi-directional Distribution Functions

For more complex shading approximations, Bi-directional Distribution Functions can be used to mathematically store surface shading information. These are a class of functions that describe in

[2]These *tertiary* light sources can be thought of as secondary light sources relative to the secondary source that illuminates them.

detail how light is reflected from or transmitted through the surface, through a set of parameters.

There are two basic types: Bi-directional Reflectance Distribution Function s (BRDFs) and Bi-directional Transmittance Distribution Functions (BTDFs), as seen in Figure 3.2.

A BRDF describes how light is reflected off a surface; both Lambertian reflectance and Phong specularity are examples of a simple BRDF. Similarly, BTDFs describe how light is transmitted through a material and how it may exit on the other side. Since Indirect Illumination relies on diffuse reflections of light, designing a good BRDF is important.

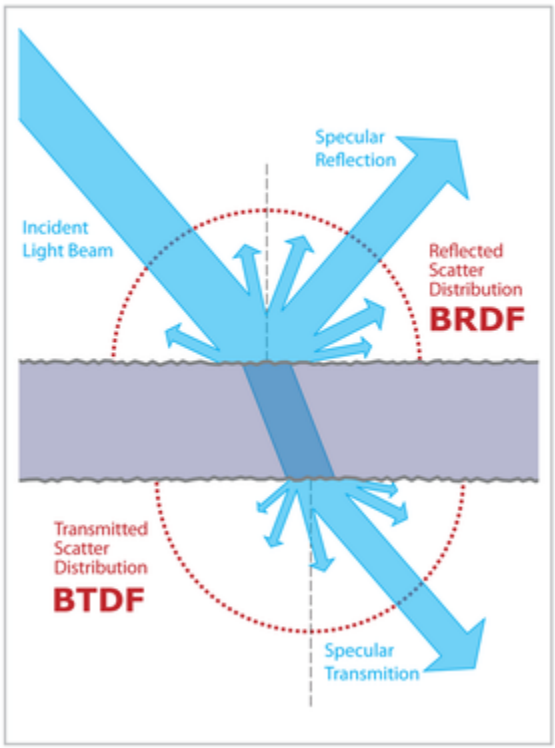


Figure 3.2: Demonstrated light reflecting and refracting via BRDF/BTDF[1]

BxDFs can be directly measured from real life materials by using a custom device called a Gonioreflectometer[24]. Several other derivation methods exist; sometimes merely observing a given surface to define its parameters phenomenologically can produce realistic results[15]. Approximations for essentially any real surface can be generated through appropriate measurements.

However, accurate BRDFs and BTDFs for specific surfaces are very complex and difficult to simply parameterize and thus are not suitable for a general lighting model. Storing the data for

these complex surfaces must also be taken in consideration. Pre-computed methods such as Pre-computed Radiance Transfer (PRT) are one way to store large amounts of shading information, often by using Spherical Harmonics (SH) to compress or approximate the lighting information[41, 3].

These methods are unsuitable for fully dynamic situations, however, since a BRDF is discretely defined for a specific surface and pre-calculating lighting information assumes a static mesh or lighting environment, or both. If an object were to change shape or material, all pre-computed values wold need to be updated. Storing any large quantity of values of a highly defined BRDFs will also quickly consume memory[41].

3.3.2 Spherical Harmonics

Spherical Harmonics are a special set of solutions to Laplace's equation, shown in equation 3.3.

$$\nabla^2 \rho = 0 \qquad \text{EQUATION 3.3}$$

It is a second order partial differential equation which is comprised of Laplace's operator, ∇^2, applied to a scalar function, ρ, set equal to zero. All solutions to Laplace's equations are termed harmonic functions. The gradient, a representation of a scalar field's rate of change, in spherical coordinates is shown applied to a spherical scalar field in Equation 3.4 . The result after taking the gradient's divergence is shown in Equation 3.5. Spherical harmonic functions are solutions to this final equation in the spherical coordinate space.

$$\frac{\delta f}{\delta r} r + \frac{1}{r}\frac{\delta f}{\delta \Theta}\Theta + \frac{1}{r sin\theta}\frac{\delta f}{\delta \Phi}\Phi \qquad \text{EQUATION 3.4}$$

$$\frac{1}{r^2}\frac{\delta}{\delta r}(r^2\frac{\delta f}{\delta r}) + \frac{1}{r^2 sin\Theta}\frac{\delta}{\delta \Theta}(sin\Theta\frac{\delta f}{\delta \Theta}) + \frac{1}{r^2 sin^2\Theta}\frac{\delta^2 f}{\delta \phi^2} \qquad \text{EQUATION 3.5}$$

We use the spherical coordinates because they more natively represent how light travels. A point in three dimensional spherical coordinates is defined with a radius r, an azimuth angle θ, and

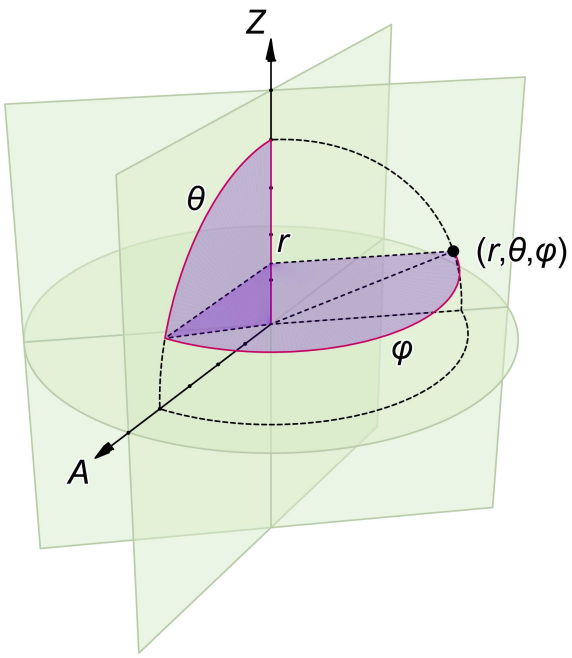


Figure 3.3: The spherical coordinate system[1]

a zenith (elevation) angle ϕ, as shown in Figure 3.3. If the light source is at the origin, these coordinates define how far away a point is from the light source and in what direction, key information for determining the direction and intensity of light at a point in space.

Spherical harmonics are linearly independent and can be calculated to an arbitrary complexity, and this qualifies them as a set of functional bases. Basis functions for a coordinate space are a mathematical series of linearly independent functions which can define any function located within that space to an arbitrary accuracy, one proportional to the size of the basis series. The full theory behind functional bases is beyond the scope of this paper; what is important is that we will use spherical harmonics as a set of bases to approximate the distribution of a light source's intensity.

In order to provide an example, we will speak of the use of spherical harmonics in two stages: encoding and decoding. For the encoding step, intensity information is placed into a spherical harmonic approximation via the generation of coefficients that most closely amplify the bases to represent the fully realized, 'ideal' light source. Decoding involves retrieving the intensity data from this approximation in the same form as the original light.

Without geometry and neglecting any differential attenuation, the radiance distribution of a point light is a sphere and could be perfectly represented by a single spherical harmonic function. The example, nonetheless, uses four bases to demonstrate how more complex sets of bases work, using two bands of spherical harmonics. We will also define the radiant flux of this area light source as 0.8 on the range$[0, 1]$.

The first step in encoding is to take a series of samples of the light source's radiant flux. The granularity of the sampling is dependent on the complexity of the original intensity distribution and the number of bases used; more variation in the light source or more bands of spherical harmonics require more samples. For a perfect sphere, we will take samples in the six directions of 3D Cartesian space.

EQUATION 3.6

$$(1,0,0) = .8 \quad (-1,0,0) = .8$$
$$(0,1,0) = .8 \quad (0,-1,0) = .8$$
$$(0,0,1) = .8 \quad (0,0,-1) = .8$$

Since we are using a point light, all values are the same. It should be stressed that this is rarely always the case, especially when representing secondary light sources.

Now that we have sample values, we calculate coefficients for the first two bands of spherical harmonic functions. Our approximation of the light source's distribution function is then a linear combination of these functions multiplied by their coefficients.

EQUATION 3.7

$$Y_0^0 = \frac{1}{2} \times \sqrt{\frac{1}{\pi}}$$
$$Y_1^{-1} = \sqrt{\frac{3}{8\pi}} \times e^{-i}\phi \sin\theta$$
$$Y_1^0 = \sqrt{\frac{3}{4\pi}} \times \cos\theta$$
$$Y_1^1 = -\sqrt{\frac{3}{8\pi}} \times e^{i}\phi \sin\theta$$

The functions, transformed from complex spherical coordinates to real Cartesian coordinates, are below. We use the real-valued functions for simplicity and speed of computation on a computer.

EQUATION 3.8

$$
\begin{aligned}
s &= \frac{1}{2} \times \sqrt{\frac{1}{\pi}} \\
p_x &= \sqrt{\frac{1}{2}} \times (Y_1^{-1} - Y_1^1) = \sqrt{\frac{3}{4\pi}} \times \frac{x}{r} \\
p_y &= i\sqrt{\frac{1}{2}} \times (Y_1^{-1} + Y_1^1) = \sqrt{\frac{3}{4\pi}} \times \frac{y}{r} \\
p_z &= Y_1^0 = \sqrt{\frac{3}{4\pi}} \times \frac{z}{r}
\end{aligned}
$$

The next step, in effect, determines the light source's radiance lobe's similarity to the spherical harmonic basis functions by actually generating the coefficients, c_l^m [10].

EQUATION 3.9

$$c_l^m = \int_S f(s) \times y_l^m(s) \delta S$$

The integral evaluates a function, $f(s)$, at points around the sphere, S, multiplying the results by the evaluation of the current spherical harmonic band in the sample's direction. Using the integral notation alludes to the idea of infinite sampling. Technically, if an infinite number of samples were taken for a light source, we would be able to perfectly approximate any light source's shape using this integral. However, an inifinte number of samples is infeasible, especially for real time solutions.

Monte Carlo evaluation uses a finite number of samples to approximate the same integral. Our original sample information evaluates the function in each direction, s_j, around the light source's lighting function. Using Monte Carlo estimation, the function becomes:

EQUATION 3.10

$$c_l^m = \frac{4\pi}{N} \sum_{j=1}^{N} f(s_j) y_l^m(s_j)$$

We now have all the information needed to generate a spherical harmonic representation of our light source. Evaluating for each coefficient, we get:

EQUATION 3.11

$$\begin{aligned}
c_0^0 &= \frac{4\pi}{6}\{6 \times 0.8 \times s([1,0,0])\} = 4\pi \times 0.8 \times .282094792 = 2.835926164 \\
c_1^{-1} &= 0.8 \times [p_x([1,0,0]) + p_x([-1,0,0])] = 0 \\
c_1^0 &= 0.8 \times [p_y([0,1,0]) + p_y([0,-1,0])] = 0 \\
c_1^1 &= 0.8 \times [p_z([0,0,1]) + p_z([0,0,-1])] = 0
\end{aligned}$$

Writing out the entire spherical harmonic representation including it's functions and coefficients gives us a final linear combination, ϕ, of:

EQUATION 3.12

$$\begin{aligned}
\phi = &\left[2.835926164 \times \frac{1}{2} \times \sqrt{\frac{1}{\pi}}\right] + \\
&\left[0 \times \sqrt{\frac{3}{8\pi}} \times \frac{x}{r}\right] + \left[0 \times \sqrt{\frac{3}{4\pi}} \times \frac{y}{r}\right] + \left[0 \times \sqrt{\frac{3}{8\pi}} \times \frac{z}{r}\right] \\
\phi = &2.835926164 \times \frac{1}{2} \times \sqrt{\frac{1}{\pi}} = 0.8
\end{aligned}$$

Constructing this spherical harmonic representation for more complex radiance lobes, such as the one produced by groups of secondary point lights, results in more interesting calculations, but the method is entirely the same.

In this example, only four bands are used. But, as previously mentioned, spherical harmonics are an infinite set of functions. As more are used, the representation approaches the actual function until, at an infinite number of functions, it is a perfect representation of the original . Later we discuss that, given the diffuse, low-frequency nature of indirect lighting, only the first two bands of spherical harmonics are needed to achieve a believable model of the phenomenon.

3.4 Rendering Techniques

3.4.1 Raytracing

Ray tracing is a technique for rendering an image that traces the path of light backwards from the eye, through the pixels of the rendering window, and into the scene, calculating the effects of its encounters with virtual objects. Ray tracing is capable of simulating a wide range optical effects, such as reflection and refraction, scattering, and chromatic aberration, but is computationally slow and grows exponentially more complex with larger screen sizes, increased numbers of objects, and more ray bounces.

3.4.2 Photon Mapping

Photon mapping is a two-pass Global Illumination (GI) algorithm developed by Henrik Wann Jensen. Rays from the light source and rays from the camera are traced independently until some termination criterion is met, then they are visualized in a second step to calculate a radiance value. Photon Mapping is used to simulate the interaction of light with different objects, most often the caustic properties of glass and liquids.

3.4.3 Radiosity

Radiosity is one of the oldest GI techniques for indirect lighting, originally stemming from 1950's heat transfer methods. Like Photon Mapping, Radiosity tracks reflected light before being captured by the camera, creating view independent indirect lighting.

It accomplishes this by iteratively running several passes with increasing light bounces, essentially tracking light as it 'spills' into the scene. Note the way the color from the floor bleeds onto the ceiling and walls for the Radiosity image in Figure 3.4. While it historically has been a still frame rendering solution, there are several methods for simple Radiosity solutions in realtime[35].

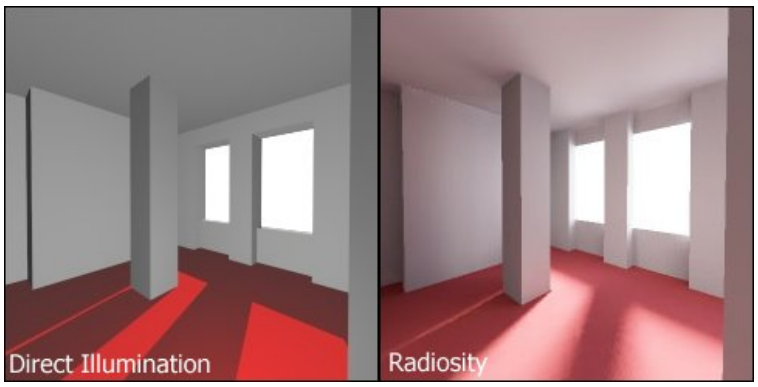


Figure 3.4: Comparison of Radiosity to Direct Illumination[4]

3.4.4 Precomputed Radiance Transfer

The PRT algorithm [41] presents a real time solution to indirect lighting that utilizes a low frequency spherical harmonic basis to represent radiance transfer. Though able to fill our needs of a real time solution, the PRT method cannot cope with a dynamically changing objects.

Before the scene is rendered, the algorithm iterates across a scene's geometry and splits objects into equal sized patches. At the vertices of each of these patches, a radiance transfer function is defined using spherical harmonics. As a volumetric equivalent to a Fourier series approximation, this spherical harmonic representation emulates the function that defines how light transfers between an incoming and outgoing angle after hitting the object. The incident light hitting an object (from a scene's light source) is also represented using spherical harmonics. This allows the previously necessary integrals across a scene's lighting to be represented as a summation of four, nine, or 25 basis coefficients instead.

Further research was completed by Teemu Mäki-Patola on using PRT to render self-occlusion, glossy, and diffuse lighting surfaces, but these techniques are still pre-computed[21]. Though real time speeds are reached, if a scene's geometry were to change, the algorithm would need to define new surface patches and new radiance transfer functions across even unchanged parts of the scene. Though the use of spherical harmonics bases relates to our work, PRT is not suitable for our use.

3.4.5 Light Propagation Volumes

A method of indirect lighting called Light Propagation Volumes (LPVs), first defined by Anton Kaplanyan working for Crytek, calculates first hop radiance information using a three-step, fully dynamic process [16]. It does not use any pre-calculated values from a scene's geometry, thus allowing the geometry to change within the scene with no loss of performance. It is highly down-scaled in nature, independent of scene geometry, and uses spherical harmonics to further compress its lighting information.

In a general sense, the method of LPV uses a three dimensional texture to represent highly down-sampled first bounce lighting information through a scene. Radiant flux is first sampled across the scene then injected in the down-sampled volumetric texture. This lighting information is propagated through the three dimensional texture giving a final, volumetric representation of the entire scene's indirect lighting. This texture can then be used as a three dimensional lookup table to determine the effect global illumination has on a single fragment [16].

It should be noted that LPV only calculates the first bounce of indirect lighting. Information past the first bounce has been shown to make little difference to the scene's final look and feel and only adds to the computation time of the algorithm [16].

Reflective Shadow Map Generation

This first step generates a reflective shadow map (RSM) from the lights perspective. The Reflective Shadow Map (RSM) map contains standard shadow map information but also includes values describing a scene's radiant flux. Performed for each light source, it is computationally cheap and can be done in a single pass each time the scene is rendered. Each time this shader runs, it generates world space position, normal vector, depth, and radiant flux information for each point in the scene. This output is stored within two OpenGL textures and is passed to the next stage of LPV.

The resulting map represents an array of 'secondary light sources', storing all the necessary information needed to calculate second hop lighting information. Using radiant flux instead of other measurements of light, such as radiance or radiant intensity, allows us to disregard the area the secondary light source is affecting during subsequent calculations[6]. We only store information

regarding the direction and intensity of light bouncing off the current surface. Once generated, we can calculate the light intensity, I, at any point in any direction, $\hat{\omega}$, by using equation 3.13.

EQUATION 3.13

$$I(\hat{\omega}) = \Phi_R * (\hat{\mathbf{N}} \odot \hat{\omega})$$

Volume Injection

The key component of LPV is the volume representation of the secondary point light sources. A cube is defined, encompassing the geometry currently visible by the camera. Inside of this cube, smaller cubes are delineated, each representing an individual region of the main volume. Each smaller cube will eventually contain a spherical harmonic representation of the average radiant flux the pixels it contains output. Using spherical harmonic representations allows lighting information, including direction, to be stored using only a small number of float values.

Iterating across each cube in the volume, the algorithm samples the RSM at a dynamically specified sample rate and determines if that pixel of the RSM resides within the current smaller cube through depth testing. If the pixel lies within the cube, the direction and radiant flux from that point of the RSM is used as a single sample in the final smaller cube's spherical harmonic representation. Increasing the sample rate, will create a finer tuned the spherical harmonic function at a performance loss only directly proportional to the increase.

The injection stage is repeated for each light source but the results are aggregated into a single volumetric texture. This is possible because of the additive nature of spherical harmonic coefficients. It also means this is the last step that requires iteration across each light source, increasing the algorithm's performance [16].

Light Propagation

This step takes into account the way indirect lighting from the secondary light sources spreads. In the previous step, a volume is generated, representing the initial state of the scene's lighting. However, each secondary point light inside one of the smaller cubes can transfer light into other

cubes as well. Propagation takes into account the distance light can travel from a secondary light source.

Given the coefficients of a spherical harmonic representation and a direction, the intensity the function represents in that direction is found by evaluating a linear combination of basis functions. To conduct the propagation stage and finalize the volumetric indirect lighting texture, each smaller cube's radiant flux in the direction of its adjacent cubes is calculated. Once again, the additive nature of spherical harmonic coefficients allows us to simply add to each set of coefficients in order to represent the propagation of light.

Once propagated, the volumetric representation is ready to be used for lighting the scene as a whole. When running the final lighting shader, the total indirect lighting a pixel receives can be found by adding the coefficients multiplied by their respective spherical harmonic basis function together from the just the smaller cube the current pixel is located within. No additional information from surrounding pixels needs to be calculated [16].

Chapter 4

Methodology

4.1 Chosen Platforms

While there are several platforms for developing graphics, none of the game engines, development packages, or shader design packages had the flexibility or advanced support we needed for our project. Microsoft's C# based XNA has unwanted overhead from the .NET framework while still requiring us to build the entire backend. Other packages like the Blender and the Unity game engine are specifically targeted at rapid game development rather than advanced graphics, and can often be buggy. No packages offered the control of input and output necessary to create or use a Light Propagation Volume (LPV), so we decided to simply program everything ourselves.

Our Application Programming Interface (API) options were OpenGL and DirectX, and we chose OpenGL for its open standard and implementation on more systems than the proprietary DirectX. We were limited to OpenGL version 3.3 because of the capabilities of our personal hardware. Fortunately, this version of OpenGL has core support for framebuffers, a feature we found necessary for the generation and manipulation of LPVs.

For external images, the Targa (TGA) image format was used[40]. This format, originally created by Truevision for use on graphics workstations, is a popular choice when simple parsing of files is desired since the actual color information in an uncompressed TGA is stored linearly in memory. Due to storing its channels in blue-green-red-alpha (BGRA) order, TGAs are ideally suited

to Graphics Processing Unit (GPU)s which prefer incoming color information in BGRA. Writing the importer for TGA images is thus easier than other formats that support alpha channels, which we used for passing information other than color into the program[40].

Keeping with the open source theme, we started coding in Eclipse and compiling with MinGW, but quickly ran into library issues: there is no precompiled library for OpenGL Extension Wrangler (GLEW) for use with MinGW, and compiling it requires a cross-compilation from GNU/Linux to Windows. This was beyond our skill set and we subsequently switched to Visual Studio 2008, but ran into compiling issues again. This time it was because Visual Studio does not support the C99 standard[20]. Compiling our C code with the C++ compiler fixed these issues, with possible loss of C compliance.

The 3D models and textures in the project were created entirely by our team.

4.2 GSUITE Practical Overview

The framework we built to support the project is an OpenGL based low level graphics engine that we named GSUITE, short for 'graphics suite'. Since the OpenGL standard itself provides a lowest common denominator API, GSUITE was built to encapsulate the raw functionality into abstractions that are easier to deal with and more interoperable. As a library, we provide few features not present in OpenGL already: apart from mathematical constructs common in graphics, GSUITE mostly exposes easier-to-use versions of the constructs that OpenGL specifies, sometimes mirroring the deprecated OpenGL functionality. These include buffer objects, shader programs, textures, and framebuffers. In general, it is meant to be used as an aid to programming OpenGL demos and experiments. For that reason, we sought to mask as little OpenGL functionality as possible and only fold features together that would most usually be used together. The result is a library that served our goals and took care of the boilerplate aspects that results from complex OpenGL tasks, like using shaders.

4.3 GSUITE Technical Overview

GSUITE is a collection of almost entirely original code written in C. It relies on several external libraries that use OpenGL: GLEW, GLSW, and FreeGLUT. These libraries provide extension loading, shader source management, and platform independent OpenGL context and windowing control, respectively. For ease, we will refer to the combination of the GSUITE code and these support libraries collectively as GSUITE.

The combination of libraries and code in GSUITE manages the details and complexities of dealing with the low-level aspects of OpenGL. This allowed us to focus on implementing graphical effects instead of OpenGL extensions and platform-specific APIs or Graphical User Interface (GUI)s. The framework has allowed the implementation and addition of new shaders to the rendering pipeline, changes to color formats, and alterations of core functionality with isolated errors and highly localized changes to code outside of the altered packages. It also makes changes much faster. Refer to Figure 4.1 for the full Module Diagram.

We built GSUITE to serve our needs within this project, but in the interests of programming best practices we also designed it with extensibility, portability, and scalability in mind. How well we achieved this remains to be seen in a real-world context, though it has proved fairly easy to work with despite some quirks. Modules such as GCanvas[1] were built and incorporated in only a few hours. The highly encapsulated design also keeps relations among modules clear to that bug tracking only deals with a shallow dependency hierarchy. We point out that the positive experiences we had were under limited scope on only four platforms and one Operating System.

4.3.1 Language

The choice of programming in C was motivated by the assumption that performance would be an issue we would have to deal with, and we wanted to be able to maximize processing power and speed to provide ample space for the shader work. C's exposed memory management would allow us to tightly control allocation and deallocation and produce code that, theoretically, had

[1] A module designed to show picture-in-picture style images in the window.

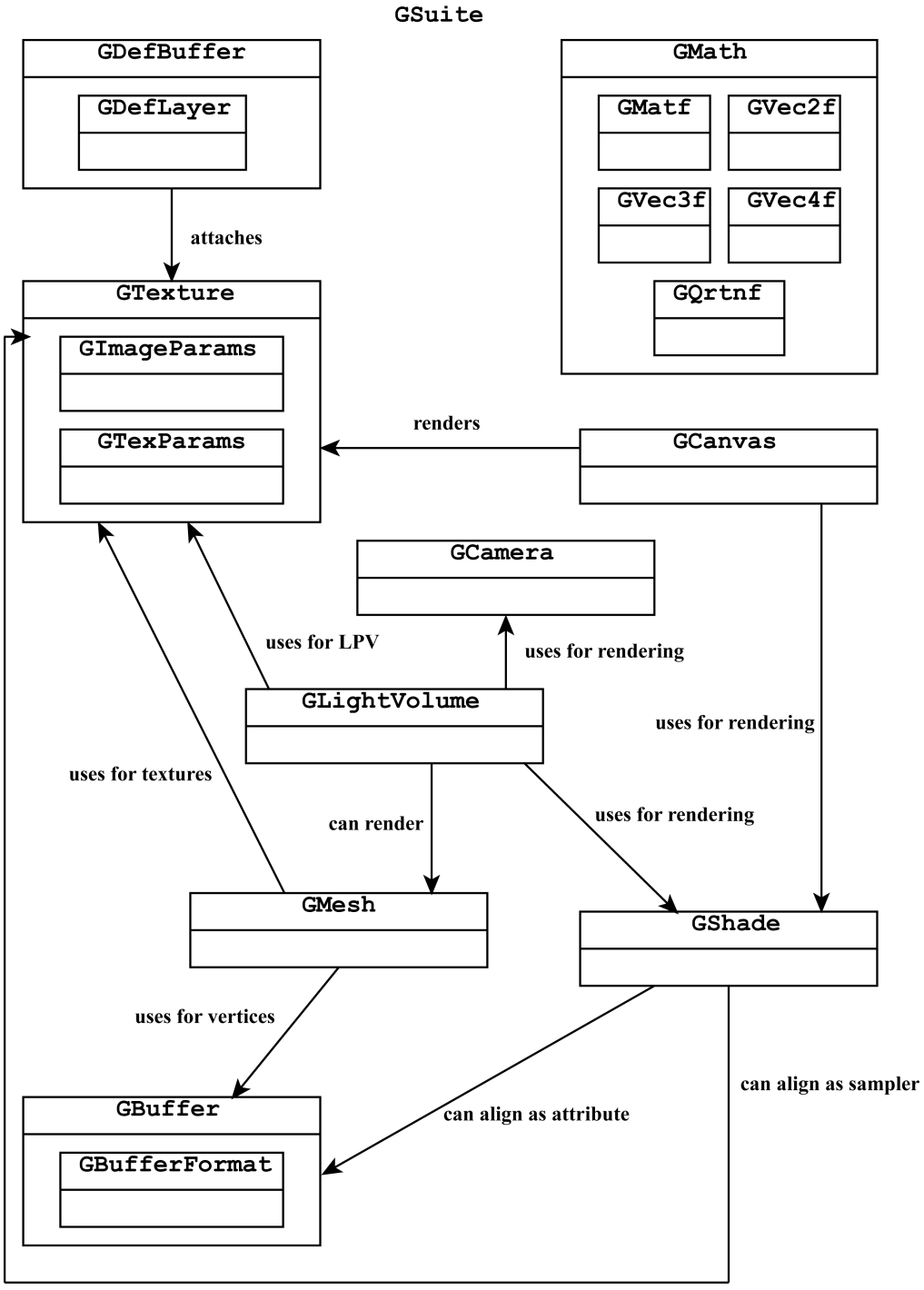


Figure 4.1: The relationships between GSuite's various modules; GMath is not shown connected because practically all modules use it.

little overhead to do exactly what it needed to do. We originally chose C89 as our programming standard but because of compiler issues[2] and expediency we switched to using C99 features.

Also, OpenGL has always been designed from a procedural perspective because it seeks to be a standard that any language can have bindings to. We felt C was a natural fit for this paradigm.

The choice of language was also affected by the team's familiarity with C. Two members are well versed in the language and had few issues with pointer management. Using a more object-oriented language such as C++, though beneficial in terms of reuse and abstraction with similar performance, would have slowed down our work and likely caused subtle bugs from features we did not fully understand. It is very hard to debug the interaction of dozens of objects when the details of memory are unknown to the programmer due to a lack of experience with the language.

4.3.2 Object Model

Without objects but with a desire for their benefits, we had to implement a limited object system in C in order to avoid an intractable jumble of ad-hoc code. Object data is simply represented by a STRUCT, forward declared in its header file and defined in its implementation file. This was necessary to both avoid name collisions from inclusions that preprocessor directives cannot resolve and to provide data hiding by making every field in an object effectively private.

The collection of functions that act upon a given object are declared in the header file and implemented in the implementation file and each takes a pointer to the object the specific call applies to as the first parameter. Since this is the sole location where the actual fields of an object are defined, only class functions can mutate class fields directly. This assurance greatly simplified debugging since it is (mostly) impossible for code in one module to alter data from another without passing through a controlled interface, barring the use of bad memory management.

Likewise, allocation and deallocation of objects can only happen in code that exists in the implementation file, so the use of MALLOC and FREE is restricted to a very tight and small collection of code that is isolated from the rest of the program. This helped avoid memory issues.

None of these features are novel or original. C++ and Java have the same functionality built-

[2]These issues mostly deal with the arbitrary declaration of local variables within a function block.

in with far more powerful features than our simple system and they are the common techniques employed in C's standard library. However, this system was central to our ability to make the program work and do so with minimal volatile behavior or development. Even though it is not a strictly object-oriented system in terms of 'real' features, we will use the standard terminology of classes and objects to discuss it.

4.3.3 Modules

For most classes, there is a header file and implementation file. The exceptions are Textures and Renderbuffers, which share a source file because of the analogous and brevity of the Renderbuffer class, as well as any support classes for the main classes. We refer to the two files together a module. In general, the classes provided by each module provide an interface to or abstraction of an OpenGL object. A few are unique and incorporate elements of non-OpenGL specific functionality as well as some OpenGL interaction. For instance, GSUITE GBUFFER objects interface directly with OpenGL buffers objects, whereas GCANVAS provides a GUI widget that draws a texture to the screen, something that OpenGL does not support natively. Each module masks OpenGL interaction as much as is practical. Since GSuite is not full-featured or prefect, there are still numerous OpenGL facilities that are used directly by the code.

GMATH

Because OpenGL versions past 3.0 removed the Fixed Function Pipeline (FFP) from the core features, all application space linear algebra needs to be implemented by the programmer. These include the construction of perspective and transformation matrices, and any vector operations. The GMATH module provides this faculty.

GMATH defines structured data that mimic primitive data types. We used Graphics Language Shading Language (GLSL)'s primitives as a guide and implemented C analogs. Thus GMATH provides two, three, and four component vectors as well as all combinations of these dimensions for matrices[3].

[3]Since the data type has arbitrary dimensions, these types were mimicked through constructors.

The vector types are defined as arrays of the length of their respective size; typedefs and the assumption that only allocation of vectors via the support in GMATH will be used keep this system type safe. In addition, we included a quaternion type for orientations. These are modeled exactly as four component vectors are, except with a separate name and interface.

The matrix type, GMATF, is implemented similarly to other GSUITE classes. Unlike vectors, matrices support arbitrary size since it would be cumbersome and unmanageable to have separate explicitly dimensioned types. Matrices can also be initialized explicitly or from vectors, and various built-in functions can create and mutate matrices in expected ways, such as inversion, adjunction, and transposition.

A large collection of converters between types are supported when appropriate. These include operations such as conversion between quaternions and rotation matrices.

GSHADE

OpenGL's support for shader programs is low-level and modeled on the C/C++ compilation model. Shader source code needs to be uploaded using C-strings and each stage compiled separately. Using the string handling functionality of OpenGL Shader Wrangler (GLSW), the GSHADE module handles all interaction with shader objects in OpenGL.

Creating a GSHADER involves a filename and path and parameters which GLSW uses to slice the source file up into its component stages. The stages can then be compiled, aligned to output buffers, and linked. GSHADERS also define an interface for the alignment of attributes, uniforms, and textures to the shader program which is far simpler than the raw OpenGL interface. Since the interface is more type aware, aligning inputs consists of a single bind operation and a call to one of the five alignment functions, instead of the several dozen in the OpenGL standard.

GTEXTURE

Three classes encapsulate the large list of parameters and options of OpenGL's texture objects: acrnGTexture, GIMAGEPARAMS , and GTEXPARAMS . GTEXTURE is a container for all the texture options and the image data for each level of a texture itself, which are managed by the two other

support classes.

In addition, GTEXTURE tracks the current framebuffer attachment, layer, target, and texture unit of the texture it represents. This values are used when the corresponding operations are performed. A GTexture also has a default attachment, layer, and unit, usually the one it was created with but this can be changed. It is important to know what the creation parameters were sicne OpenGL implementations are free to use such information to optimize performance. The current context information can be reset to any of the default values.

GDEFBUFFER

GDEFBUFFER is a class that models a more complex conception of a framebuffer than that of its OpenGL counterpart. In addition to managing an OpenGL framebuffer, GDEFBUFFERS register and track all textures that can be attached to them, indexed by a unique integral name retrieved with GETGDEFLAYERENUM. These layers can be attached and detached by this name, allowing a collection of render targets to be managed as a group and selectively used. In terms of performance, this organization is crucial because switching attachments is far faster than switching framebuffers. For practical reasons, however, few modules makes use of this ability and no centralized framebuffer exists.

GCAMERA

GCAMERAS deal with the projection of geometry onto the screen. In many ways the class behaves like a concrete version of the GLUT camera functions, except the projection and model view matrices must be extracted and loaded by the programmer into shaders. Cameras are initialized by the point it looks at, the up vector, and the position of the camera and a collection of projection parameters.

Because of the accuracy of floating point and the math needed to create a quaternion from a description of an orientation related to a rotation matrix, the initialization of the camera can be less than ideal. Implementation tricks ensure that the camera will look at the appropriate location as specified, but the actual angle of the camera may differ by several degrees and the location may

be off by a few hundredths. Subsequent transformations work as expected, as the conversion from quaternions to rotation matrices is much more reliable.

GBUFFER

GBUFFERS are similar to GTEXTURES in that they manage a large amount of options and parameters. However, GBUFFERS track attribute formats that describe how data is packed into memory. These are created then stored in a list, indexed by the string name of the input they attach to in the shader. GSHADERS are can use the GBUFFER type directly to align attributes by name.

GCANVAS

The canvas is a simple display widget which renders a texture, including 3D textures, to the window in a specific location. We have used this extensively to debug various textures.

GLIGHTVOLUME

GLIGHTVOLUME manages our LPV implementation. After initializing the object, successively calling three functions in order is all that is needed to produce a lighting volume. Using a collection of internal textures, a camera, and three shaders, as well as transformation code from other modules, the light volume collapses a huge amount of code into a very small main program footprint. The specifics of how the algorithm works are described in the following section.

4.4 Implementing Light Propagation Volumes

Implementing LPVs introduced new challenges to developing an extensible coding framework around OpenGL. The LPV algorithm relies on multiple passes and the use of OpenGL structures in unusual ways, which proved difficult to develop especially in the absence of extant OpenGL examples and the novelty of the technique.

4.4.1 Transformations

In order to render to the LPV we needed to be able to systematically transform from points defined in the volume's space to points in the light's space. This allowed depth testing to occur in light space such that surfaces could be identified and their radiance modeled by spherical harmonics.

4.4.2 Generating the Reflective Shadow Map (RSM)

The RSM is a projection of the scene from the point of view of the light source. We used an orthographic projection to model exterior sunlight, which is essentially parallel in nature. Our RSM (Figure 4.2) stores the albedo, depth, and surface normal in three distinct textures.

Figure 4.2: RSM Depth Map

4.4.3 Building the LPV

Building the initial LPV requires rendering to a 3D texture. However, graphics APIs are designed to only render to 2D arrays of values. This presented a problem: printing a 3D texture using 2D outputs. Only a single layer of a 3D texture may be attached as an output of a shader, and the number of 2D slices in even a small volume is prohibitive to attach all at once, because the limited

number of attachments allowed.[4] Furthermore, any number of outputs more than one would require us to alter the shader any time a change in the resolution of the volume was needed.

Printing to a 3D Volume

Thus, we wrote a shader that executed on a single slice of the volume. The geometry passed to the shader is in fact a single quadrilateral mapped to the render window, run with depth testing turned off. When the shader exits the vertex stage, the fragment stage receives its location in volume space, with each fragment representing a single voxel in the LPV. This location is mapped exactly the same way the normalized device coordinates are, on the range $[0, 1]$ on all axes. Using the transformation from the LPV space and the transformation to the RSM allows points to be put into the light's space so that samples may be accumulated and injected into the LPV. This means that only a few matrix multiplications are used to get the base position of each fragment to use for identifying primary reflectors and transmitters.

Injection of Secondary Light Sources

The fragment stage uses the information received from the the vertex shader to gather samples of nearby secondary light sources and transmitters from the RSM. Because the LPV is guaranteed to be coarser[5] than the RSM, we sample an area around the fragment's location for reflector data.

The number of samples to take within each voxel is passed as a shader input and the actual offset from the center of a voxel is taken from a lookup of a 1D texture of random colors. These offsets are transformed so that they always lie within the volume of the voxel, then the final sample location is transformed to light space.

With this information, the RSM's depth texture is sampled and the current sample's z component compared. If the RSM depth contains a value that lies within half a voxel's width from the sample location, the RSM contains a primary reflector and transmitter that is within the current voxel.

At this point, the shader looks up the surface normal and albedo at the sample location in the RSM. These are used to generate two bands of per-channel spherical harmonic coefficients.

[4] The hardware we used, for example, only allows 8 draw buffers.

[5] See Section 5 for why.

The average of these coefficients is saved in the three LPV textures, one each for the three color channels.

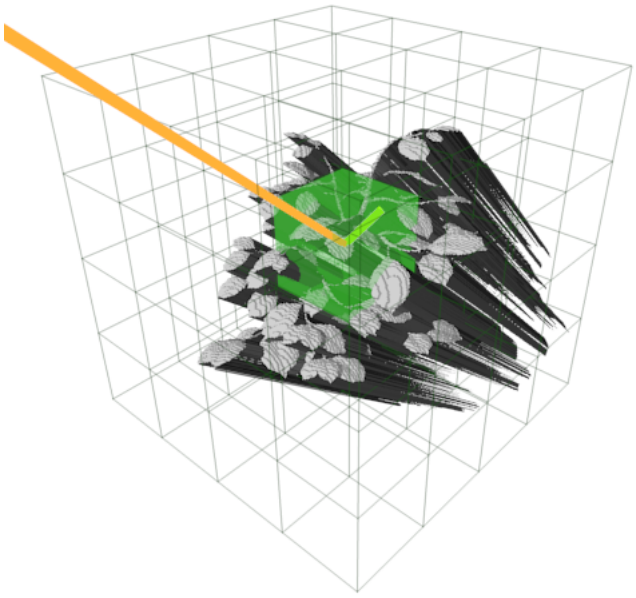


Figure 4.3: RSM Injection and LPV Seeding model

4.4.4 Propagation

To propagate the lighting data through the volume, a similar design to the volume generation shader is used. Three 3D textures are taken as input, which are the LPV channels from the previous propagation step.[6] Subsequent passes swap the outputs for the inputs, thus overwriting the propagations from two passes previously. This is necessary because, though the standard allows it, writing to a texture which is currently being sampled has vendor dependent and undefined behavior. [7]

Given its position in the volume, the shader pulls coefficients from adjacent voxels in the volume using the inputs and adds it to the current location's coefficients, multiplied by an attenuation factor which makes sure light smoothly 'smears out' across the volume. This value is then output

[6] On the first propagation pass, the shader uses the LPV channels from the injection step.

[7] Extensions do exist to allow this to work but we were more concerned with portability and support.

to the the slice of the LPV the fragment is part of.[8] The propagation process is executed for each slice and the whole process repeated as many times as desired to increase the number of 'hops' of light.

4.4.5 Illumination

Once the volume is generated, it is plugged into the compositing shader of the deferred framework. At composite time, each fragment's surface normal and position are transformed into the volume's texture space are used to sample the volume and calculate the illumination at that point due to indirect lighting. This effect is compounded with the more standard diffuse and specular techniques to produce an image that displays the transmissive properties of the material.

[8] This alignment is ensured in application space since outputs must be two dimensional.

Chapter 5

Results

5.1 Benchmarks

Our benchmarks were primarily made on computers in the Worcester Polytechnic Institute (WPI) IMGD lab, which are Intel Core2 Duo 3.0GHZ processor computers with 4GB of RAM and two Nvidia GeForce 9800GT graphics cards in SLI configuration, running Windows 7 [14].[1] Additional user experience tests were conducted on a similar machine with a single GeForce 9500 (Machine 1), an Intel Core2 1.80GHZ processor, 2GB RAM desktop with an nVidia GeForce 8500GT (Machine 2), and a Lenovo Thinkpad w510 i7 quad-core 1.6GHZ processor, 3GB RAM laptop with a Quadro FX 880M graphics card (Machine 3).

The test mesh used was "bigbush.obj", created in Blender, and has 22,100 faces. For qualitative tests, a ground plane object "groundplane.obj" was included as well, bringing the total scene face count to 24,191. Both objects use separate 512x512 texture maps and normal maps.

The first test compares the effects of Reflective Shadow Map (RSM) map size to Light Propagation Volume (LPV) cube resolution, measuring the resulting FPS while noting the visual differences in image quality. The results are shown in Tables 5.1 and 5.2. The observed results were fairly expected, showing decreasing frame rates as both the voxel density and RSM size increased, but

[1] While many programs and games can greatly benefit from an SLI configuration, there was no noticeable FPS difference when running our benchmark tests with SLI on or off, so those results have been omitted.

RSM Map size v.s. LPV Voxel Cube size	8^3	16^3	32^3	64^3	128^3
128x128 px	47.95	52.65	42.27	12.20	1.89
256x256 px	49.55	52.73	41.44	11.68	1.83
512x512 px	49.55	52.71	39.64	11.25	1.80
1024x1024 px	34.10	47.07	34.17	8.84	1.32
2048x2048 px	17.7	38.22	28.55	7.91	1.31

Table 5.1: Frame rendering speed (FPS), 512x512 px window

RSM Map size v.s. LPV Voxel Cube size	8^3	16^3	32^3	64^3	128^3
128x128 px	44.15	37.71	29.32	11.20	1.60
256x256 px	44.44	37.24	29.00	10.87	1.57
512x512 px	42.04	35.65	27.96	10.67	1.58
1024x1024 px	33.50	31.61	24.68	7.74	0.83
2048x2048 px	8.97	26.07	20.71	6.70	0.81

Table 5.2: Frame rendering speed (FPS), 1024x1024 px window

the increase in FPS from LPV voxel densities 8^3 to 16^3 was unusual.

Small LPV densities resulted in a smooth but very generic lighting, and caused jagged shadow approximations when specifically hard edges are present. Small RSM values also resulted in granular shadows, due to the loss of edge precision caused by the small resolution of the RSM map. Lower resolutions of both LPV and RSM decreases processing time, which helped increase the frame rate.

Overall, the best results for both qualitative measurement and rendering speed were for RSM sizes between 128 and 512 pixels, and an LPV voxel density of 16^3.[2] There was little noticeable difference between RSM map sizes above 512 pixels squared, and large LPV cube densities created very granular and spotted illumination passes, and were thus not a good representation of Global Illumination (GI).

A second test was conducted to measure the program's effect on graphics hardware.[3] A stand-alone executable, GPU-Z, was used to measure the change in Graphics Processing Unit (GPU) load and memory usage[39] (see Figure 5.1). Variations of RSM and LPV sizes were used to stress the graphics card. The results are shown in Table 5.3.

Because the GPU is used for rendering the Operating System (OS) as well, the graphics card's memory is always in use. To calculate the memory used by our program, data was measured before and while the program was running, and the difference recorded. [4] CPU and GPU processes were observed for 10 seconds after the program fully loaded and displayed on screen, and the average recorded.

The results show that this implementation has a very small memory footprint, but is rather processor intensive for both the CPU and the GPU. Being GPU bound is a good thing, however; it narrows down where optimizations could be made and means that using faster GPUs will create higher frame rates.

[2]It is worth noting that while our shader was designed to handle virtually any RSM map or LPV size, graphics cards prefer texture sizes of binary powers [28].

[3]Administrator Privileges were required to run the program, so the test was conducted on a personal computer (Machine 1); the same one used for the qualitative assessment.

[4]The settings of 2048px, 256^3 for RSM and LPV sizes, respectively, crashed the GPU, so data was unable to be retrieved for that value pair. Since the graphics card had 1GB of onboard memory, and the average memory load was around 50MB, it was likely a processing choke; it couldn't handle the exponentially increased calculations.

RSM & LPV sizes	GPU (%)	GPU Memory (MB)	CPU (%)	Computer RAM (MB)
256px, 16^3	98	30	50	31
512px, 32^3	99	34	50	33
1024px, 64^3	98	30	60	68
2048px, 256^3	NA	NA	NA	NA
2048px, 128^3	99	74	97	122
2048px, 64^3	99	80	95	63
256px, 32^3	98	46	50	33
256px, 128^3	99	76	97	106
512px, 16^32	99	72	50	31
64px, 8^3	97	36	44	44

Table 5.3: GPU Utilization - Scene with 24,919 faces

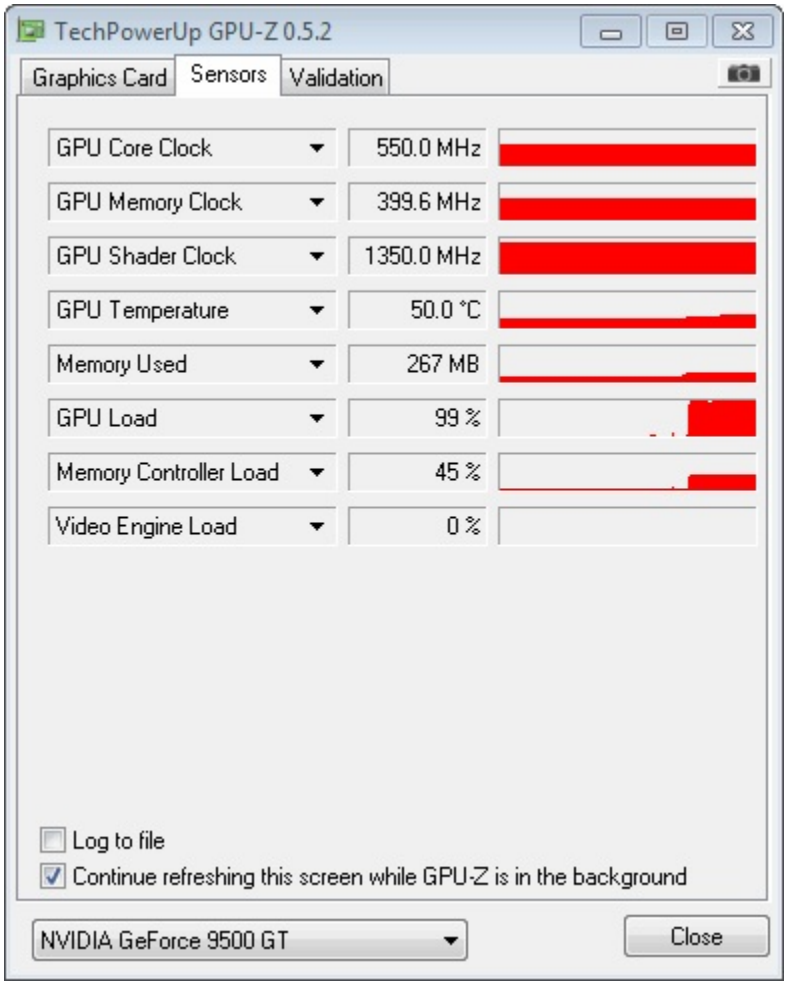


Figure 5.1: Screenshot of GPU-Z program while collecting data[39]

5.2 Objects and Textures

All objects and textures used in this project were created by the project team, using Blender 2.5 for models and Photoshop CS5 for image creation and manipulation. Only the skymap was used from an outside source.

5.2.1 Big Bush

The bigbush.obj consists of 22,100 faces.

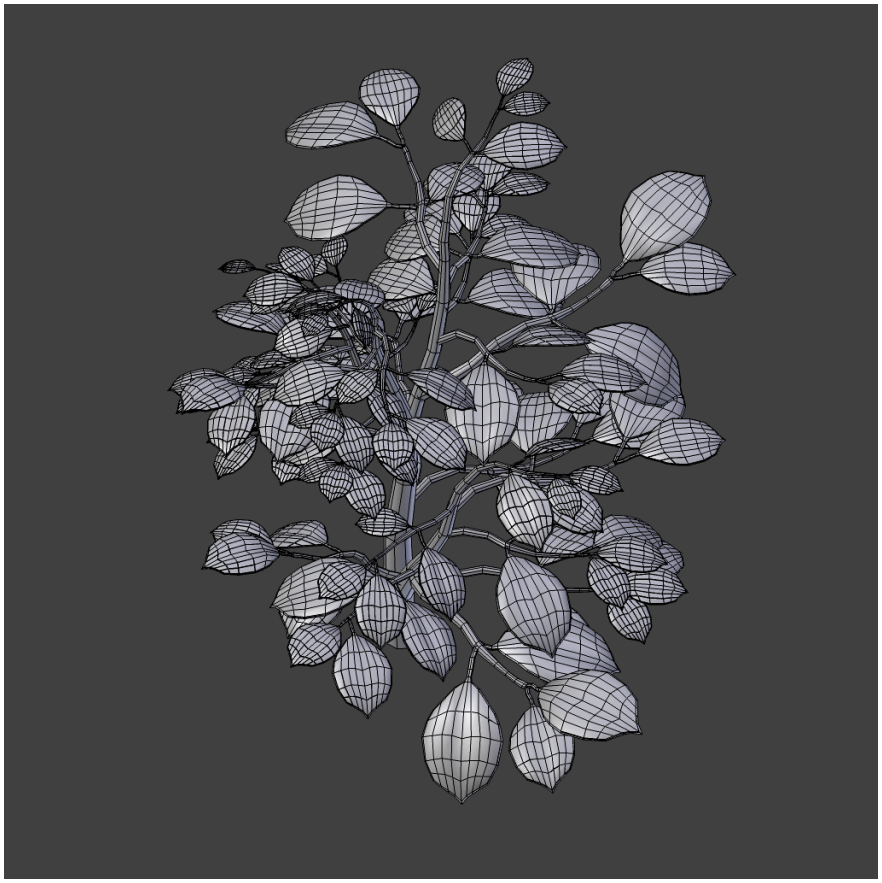
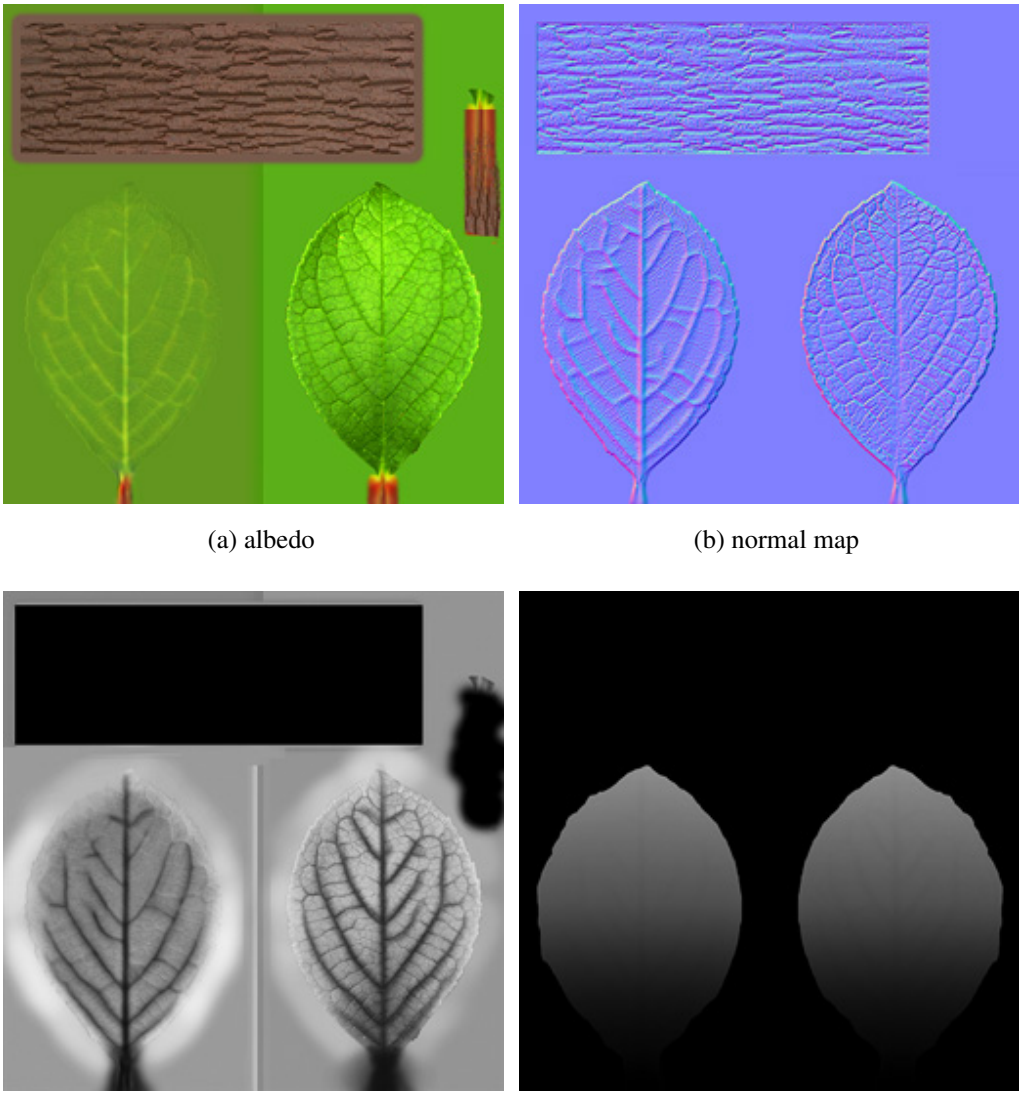


Figure 5.2: “bigbush.obj” object mesh, modeled after real leaves

Bush Textures



(a) albedo

(b) normal map

(c) transmittance value

(d) movement weights

Figure 5.3: Bush Textures

5.2.2 Ground Plane

The groundplane.obj consists of 2,819 faces.

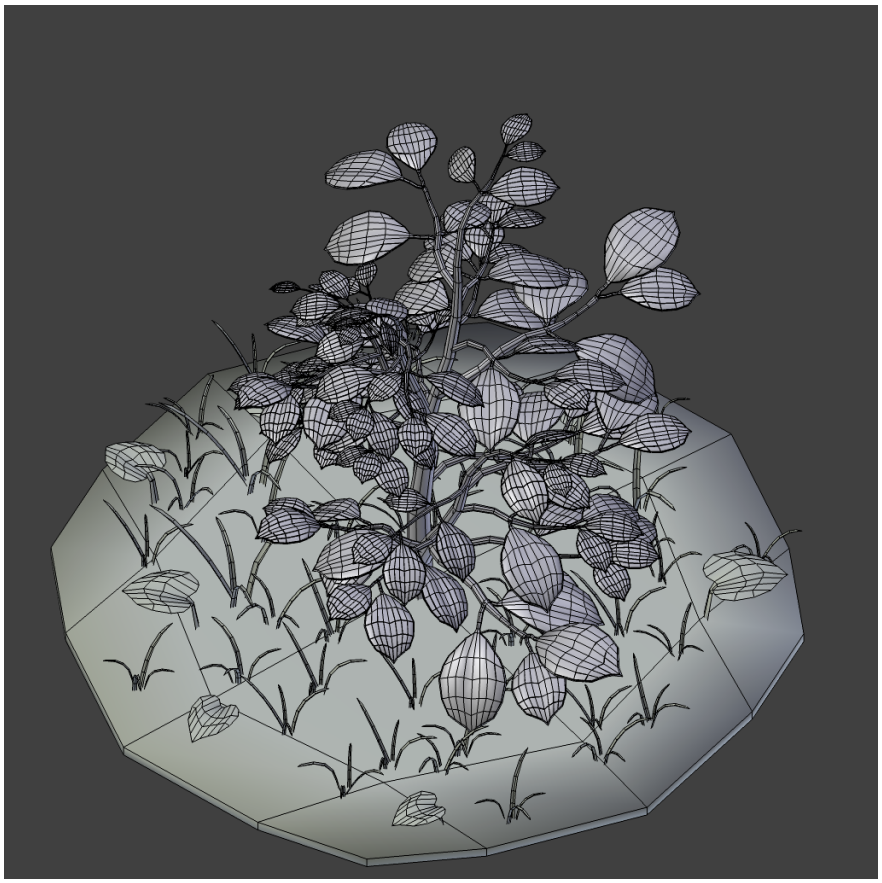
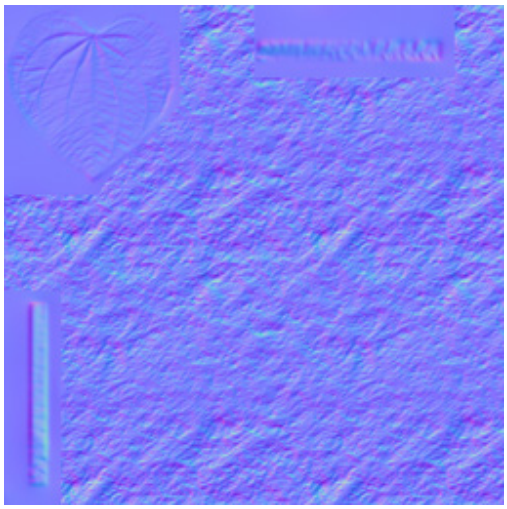


Figure 5.4: “groundplane.obj” object mesh

Groundplane Textures

(a) albedo



(b) normal map

(c) movement weights

Figure 5.5: Groundplane Textures

5.3 Qualitative Assessment

The qualitative assessment involved explaining the purpose of the project to the subject, then showing them, in increasing complexity, the various rendering stages of the program.[5]

This involved rotating the bush object, as well as the light around it, to fully show the effect of the implemented LPV GI approximation. The subject's reaction and opinion on the look of the final image were observed. The test was rendered in a 1024x1024 px window, using an RSM size of 256x256 px and an LPV voxel size of 32^3. The subject pool consisted of seven WPI Senior students, with a diverse set of graphics knowledge and experience: ranging from having seen at least one 3d animated movie to someone currently coding a game-oriented graphics engine.

All subject responses were positive[6], noting how the lighting was smoother and the ability to see the shadows from the underside of the leaves was 'realistic'.[7] These results demonstrate the visual effectiveness of the lighting techniques.

[5]The four stages used were Albedo (Figure 5.8a), Specular and Diffuse5.6a, Specular and Diffuse with Shadow Map (Figure 5.6b), and the final composite stage with LPVs (Figure 5.7).

[6]Many subjects exclaimed "oh wow!" and "oooh!" when the final composite layer was shown.

[7]A couple subjects complained about the light flickering as it rotated, but concluded it was still realistic overall.

5.4 Direct Lighting vs. LPV vs. Ground Truth

The following are screenshots of the final program,[8] and then breakdowns of all the shader information that is used to create the final composite image.

(a) Lambertian Diffuse

(b) Diffuse + Shadow

(c) LPV Composite

(d) Blender Raytrace, 2 minutes

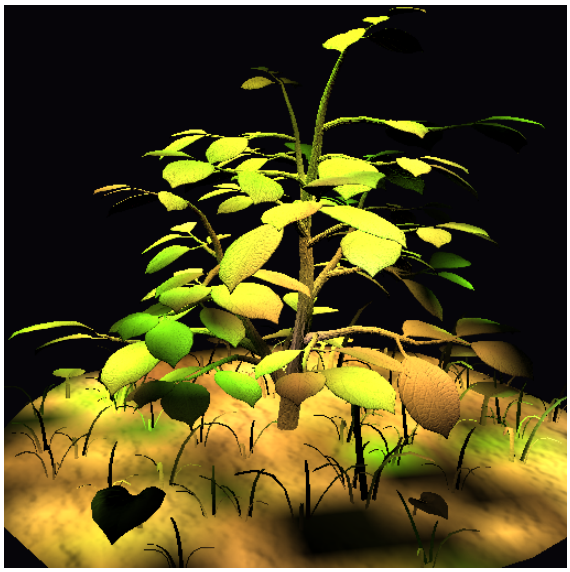
Figure 5.6: The project results, compared to raytraced ground truth, rendered in Blender

[8] An odd, noted bug that results in a less than perfect ground plane: regardless of input mesh, the ground plane object's normals are always inverted, which affects the resulting reflected GI term.

Figure 5.7: Full composite

(a) Texture color; one of the deferred inputs

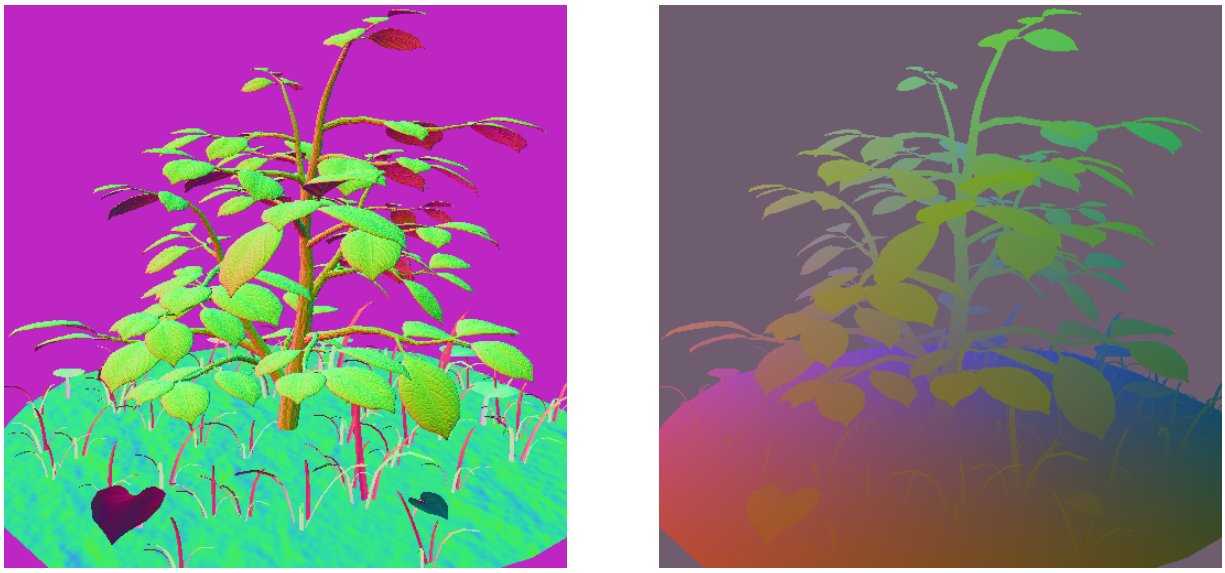
(b) Ambient sky term with shading and shadows



(c) Non-normalized global illumination term

(d) Non-normalized transmitted illumination term

Figure 5.8: Shading layers that result in the final composite image in figure 5.7 on page 59



(a) World normals; one of the deferred inputs

(b) Position of fragment in the LPV; based on the deferred input of world position

Figure 5.9: Graphical Representation of Object Data, used in the shader calculations

5.5 Additional Effects

Subsurface Scattering (SSS) is loosely approximated using our LPV method through the translucence/thickness map, which is packed into the alpha channel of the normal map. See Appendix 5.2 for the actual texture. The values from this map not only scatter the propagated light in a slightly more realistic way, but serve to selectively darken the underside of the leaves as well, to emphasize the internal plant structure.

An additional improvement to the scene was the inclusion of subtle leave movement. This helps bring the scene to life while effectively proving the validity of the dynamic qualities of the LPV lighting method. The vertices are moved via the vertex shader of the initial deferred shader pass, using a sin wave offset by the vertices' xyz position to randomize the movement, a time value to smoothly loop between the movement extremes, and a vertex weight texture which is packed into the alpha map of texture map. This texture-based method is derived from one of Crytek's papers

in GPU Gems 3[37]. In their example, they have an entire texture dedicated to the movement of palm tree leaves, and use animated wind vectors to realistically blow the branches around.

Initially, the object's normal map was too dense, and caused significant specular noise on the final rendered image. A normal map filter was devised to reduce the noise and smooth the look of the leaves, using the 8 rooks sampling kernel [3] and an original convolution method.[9] It works great for far away and dense materials, but for surfaces that are close to the camera, it starts to lose detail.

The original black background hindered the visuals, so we added a spherical sky map. The sky texture is mapped dynamically using the camera's orientation, avoiding the addition of geometry to the scene and any depth testing issues.

5.6 Fudgefactors and Assumptions

Shadows

There is a notable margin of error with the shadow map due to the shadow bias, that can cause thin walls to self-shadow and light at certain angles to illuminate correctly, but only on one side.

Edges

One unfortunate side-effect of deferred shading is that traditional antialiasing techniques do not work since the final fragment values are determined from textures whose finite resolution cannot easily yield the multi-sampling information needed to properly disguise edges. This problem can be difficult to deal with; the popular solution is to use a blur filter on areas located with an edge detection algorithm[3]. We did not implement this for reasons of time and priority.

[9]The way we smoothed normals was not based on any literature, only experiment. It may be attested to before our implementation.

Chapter 6

Conclusion

Combining our results, benchmarks, user studies, and final product, it is evident that real-time, dynamic, global illumination is possible on todays graphics hardware. Using methods such as Light Propagation Volumes (LPVs) is a feasible solution to the problem of realtime Global Illumination (GI), and can be implemented using OpenGL and Graphics Language Shading Language (GLSL).

Though it does not cleanly reproduce the effects of a complex and complete Bi-directional Reflectance Distribution Function (BRDF) and Bi-directional Transmittance Distribution Function (BTDF) pair, nor the physical accuracy of a raytracer, our solution produces a scalable representation of GI that is perceptually similar and runs in realtime.

6.1 Light Propagation Volume Trade Offs

Our method of LPV contains multiple stages where implementation could differ, producing different results. Some of these differences were tested in our benchmarks, for example, Reflective Shadow Map (RSM) resolution or the number of voxels contained in the final volume. Others exist and can be changed as well.

6.1.1 Advantages

The primary advantage of using LPV is that it is geometry independent and consequently incurs a constant performance cost. It has a very small memory footprint, and is GPU-bound, so future technology can run it even faster. Also, the diffuse nature of the indirect lighting it traditionally simulates and the transmittance we added acts on a scale large enough that the LPV can be quite coarse and still produce the desired visuals. The same principle applies to the spherical harmonics themselves, since two bands are enough to reasonably simulate the direction of the scattered light.

6.1.2 Reflective Shadow Map Sampling

This trade off played a less significant role in altering our approximation of GI yet significantly reduced our number of calculations per frame. In order to create a Spherical Harmonics (SH) approximation to a scene's lighting function, you must take a discrete number of samples of that lighting function. However, when creating the SH for a voxel to represent the global lighting inside it, choices can be made on how those samples are taken.

Initially, our plan was to pick a number of points from the RSM that were located inside the current voxel. At each point, we would chose random directions using a buffer of randomly generated noise to sample in. The lighting values in each direction would be a single sample and would contribute to the SH coefficients for that point inside the voxel. In the end, the coefficients for all points would be added and normalized to represent the final SH function.

This proved to be inefficient. Creating an accurate SH function for a single point on the RSM required at least 20 sample points for the simplest scenes and many more for complex geometry. Then, to have an accurate representation of the geometry inside a voxel, an increasing amount of sample RSM points were required, raising the number of calculations exponentially. Thus, the number of calculations necessary to form an accurate representation of GI was limiting.

To increase run speeds with little detraction from our volume's accuracy, we changed our method of sampling from the RSM. Instead of creating SH functions for each point we sampled, we used the points as samples themselves. Each point on the RSM contained radiant flux and position values. Using the fragment position we found a given point's direction from the center of a voxel

and the radiant flux served as our sample value.

Now, forming an SH required less calculations on orders of ten to 1000. Though more complex scene geometry still requires a higher number of samples to accurately portray it's lighting information, the relationship became linear as opposed to exponential.

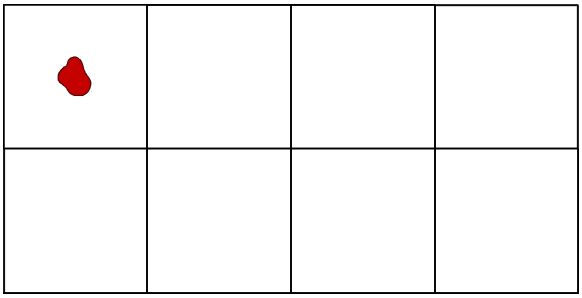
6.1.3 Propagation

The way our propagation stage is implemented, one run of the shader pulls Indirect Lighting from a voxel's 12 immediate neighbors. The shader is run multiple times to further spread light throughout the entire volume. However, this places more weight on the initial seed illumination. Light reaches further distances through the volume by multiple runs of the shader, yet each shader run propagates the initial seeded light an additional time. This method was mainly chosen due to the limitations today's graphics cards have on the number of input and output buffers for shaders.

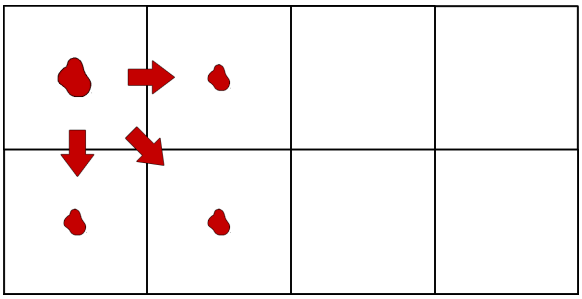
As seen in Figure 6.1, spreading a voxel's light past its immediate neighbors inevitably increases the light in it's neighbors cells a second time. This also inherently limits the number of voxels light can travel. For light to travel four voxels away from its original position, the propagation shader must be run four times. That also means a voxel's original neighbors receive light four times, blowing their illumination out of proportion.

Limiting the number of propagations ends up limiting our GI algorithm's performance as well. The number of voxels in a light volume can be easily increased or decreased to produce different results. However, as the number of voxels increases, the percentage of a scene lit by the same number of propagations decreases, sidestepping the fluid, globular nature of SH functions and causing our GI to become jagged and defined, as you can see in Figure 6.2.

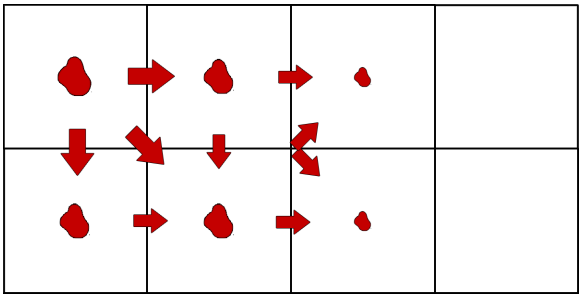
While this method provides us with a fairly accurate approximation of GI propagation through a scene, it does not model the true behavior of light. A different method may have produced more accurate results while keeping within the bounds of our hardware limitations. Instead of propagating only from immediate neighbors, the entire light volume could be passed to the propagation shader. Iterating over each voxel in the volume, we could determine the distance away from our current voxel, reconstruct the lighting function for that voxel in the correct direction, then propa-



(a) Initial Seed



(b) First Propagation



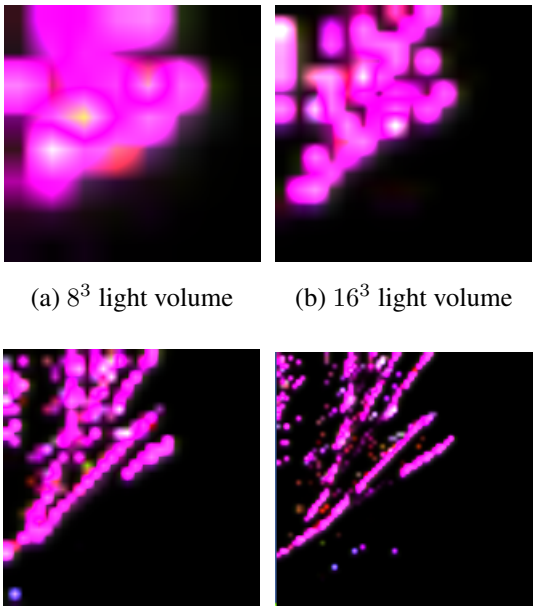
(c) Second Propagation

Figure 6.1: Visualization of our propagation scheme

gate the GI using an attenuation value based on the distance. In this way, the shader would only need to be run one time and the initial seed light would only be spread once.

6.2 Implementation

Our implementation is written in OpenGL, meaning that a working LPV algorithm can be created for any platform, possibly using our methods as an example. It is also written in C but compiled



(a) 8^3 light volume (b) 16^3 light volume

(c) 32^3 light volume (d) 64^3 light volume

Figure 6.2: Spherical harmonic coefficients using different voxel densities

in C++ mode, which means that using the code directly in an application is not easy nor advisable since most such programs will be written in actual C++. The benefits of object oriented programming lend themselves well to graphics applications, and as such, we found ourselves implementing object-like functionality in our C code. Even with Visual Studio's recent improvements to standards compliance, the code is potentially unportable as well.

6.2.1 Using OpenGL

Some parts of our implementation are not designed with the best performance in mind. For instance, it is far cheaper to alter the attachments of a framebuffer than it is to swap entire framebuffers. The fact that we use multiple framebuffers instead of one or two with more use of attachment swaps is an artifact of how we wrote earlier modules and abstracted the objects involved; a centralized framebuffer is simply harder to build and test after the fact. This incurs an unknown performance cost.

Our methods also assume a version of OpenGL of 3.1 or better, which fully updated drivers

on up to four year old nVidia cards support, but this is not universal. All our test machines had NVIDIA cards with up-to-date drivers, so the project's compatibility with other brands is questionable due to a few small details of NVIDIA's GLSL compliance.

6.2.2 Coding Shortcomings

There is a known memory leak that is triggered at some point after the camera is rotated, which consumes about three matrices-worth of memory per frame. Given the quantity of math and data involved, this is a small hole, but it remains an open issue.

Also, our shaders have not been scrutinized for optimizations, and given their length it is likely large improvements can be made. In particular the final compositing shader has several different methods of shading combined into one, complex system which probably has some redundancies and has factors incorporated to make the final product more visually pleasing and a more dynamic demo.

6.2.3 Additional Problems and Challenges

Documentation

The vast difference in design and complexity between older OpenGL and the OpenGL version 3.3 we used creates a problem when it comes to documentation: much of the openGL documentation is based on older standards and small, one-off demos. A huge supply of Fixed Function Pipeline (FFP) information exists, but relatively little is targeted at a thorough understanding of the latest features of OpenGL, particularly the issues that come with shaders and the complex relationship they have with buffers, framebuffers, renderbuffers, and application space data. There is a contrast between core features and the extensions that inspired them and new standards depricate many common functions in favor of more programmable options at the cost of complexity.

Standards and Portability

An open standards based Application Programming Interface (API) has the advantage of anyone being able to implement it, free of charge, and the decoupling of the standard from specific platforms. This is also a weakness; there is no consequence for providing a working implementation that is not standards compliant. C and C++ suffer from this, and some of their features are very dependent on the compiler being used. OpenGl has similar issues.

Very often, implementing OpenGL based applications requires careful consideration for the hardware they will run on. Being gaming enthusiasts, we managed to avoid this issue of hardware support for the features we desired; the computers used for developing the project had relatively new NVIDIA cards. This allowed us to ignore programming support for discrepancies between hardware capabilites.

6.3 Recommendations / Further Study

There are several future directions this project could take. The propagation step is a significant bottleneck that warrants improving. In particular, we would like to work on better methods of propagation as the interaction of two dimensional output and three dimensional input is difficult to resolve.

The quality of the lighting is rather sensitive to small changes in settings, and one way to find a balance between the extensive parameters would be to add graphical sliders to change settings during runtime. There is a significant amount of untested configurations that could affect the end result.

The issue of color flickering has been a long standing and open ended issue for this project. Numerous fixes were attempted but resulted in only marginal success. It may have to do with the relative size of the volume to the scene, the resolution of the depth map, how we sample the RSM or LPV, or how many spherical harmonic bands we use. We switched to using three LPVs for each color channel in an attempt to resolve color flickering artifacts. Prior versions propagated a single spherical harmonic model of intensity and the light color separately. The method used for

propagating the color without loosing its fidelity was identified as the source of the flickering, so we switched to per channel harmonics. These three volumes consume more memory because they use three, four channel, 3D 16-bit float textures instead of only one 16-bit float texture and one three channel 8-bit color texture. Devising a more compact method for representing the harmonics or further investigating separating intensity and color propagation would benefit performance.

A significant change that would improve the GSUITE API would be to overhaul the math module. An example of its shortcomings is the camera module: the current implementation does not store the camera's true location because of fixes to rotation matricies within the camera.[1] We did not have the time to resolve this issue, so attempts to get the camera's position return a constant value. A side effect is that specular terms are not dynamic since the camera appears to be in the same place to the shader. Issues like this expose more fundamental issues related to numerical stability and accuracy in GMATH, and the huge number of operations required makes the module's functions nearly illegible under C's required unique function names. Either a change of language or a better design are needed to improve these problems.

Crytek combines their LPV method into a Cascaded LPV, nesting 3 volumes to provide extra detail to objects closest to the camera[16]. This is a possible avenue to pursue. On the flipside, including an Ambient Occlusion (AO) pass in the final GI approximation could also help improve realism. There are several papers on this [16, 32, 34].

Since the seeding of the LPV is done using straightforward depth testing, a technique which has its own set of similar issues when applied to shadows, it may be possible to use solutions from the extensive shadow rendering literature on the LPV. In particular, using variance shadow mapping techniques may be a way to improve the way the lighting environment is modeled in the initial volume since it allows spatially related samples to be convolved in a rational and non-destructive manner.[2]

Since LPV simulates a complex propagation of properties through a volume using a 3D grid, it shares features with the field of fluid simulation and techniques applied there may be useful. [3]

[1] See Section 4.3.3 on page 42.

[2] Such as *Summed-Area Variance Shadow Maps*[22].

[3] Such as *Dynamic Particle Coupling for Graphics Processing Unit (GPU)-based Fluid Simulation*[17].

Bibliography

[1] (2011). Creative commons.
URL http://creativecommons.org/

[2] 3Dlabs (2006). Legacy graphics cards.
URL http://www.3dlabs.com/content/legacy/

[3] Akenine-Moller, T., Haines, E., & Hoffman, N. (2008). *Real-Time Rendering, 3rd edition*. Wellesly, MA: A K Peters, Ltd.

[4] budda165 (2011). radiosity_comparison.jpg.
URL http://budda165.files.wordpress.com/2008/01/radiosity_comparison.jpg

[5] Crytek (2009). Deferred Shading.
URL http://www.crytek.com/sites/default/files/A_bit_more_deferred_-_CryEngine3.ppt

[6] Dachsbacher, C., & Stamminger, M. (2005). Reflective shadow maps.
URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.87.6346&rep=rep1&type=pdf

[7] Fernando, R., & Kilgard, M. J. (2003). *The Cg Tutorial*. Boston, MA: Pearson Education, Inc.

[8] Foundation, T. B. (2011). Blender 2.56a.
URL http://www.blender.org/

[9] foxhavenjournal.com (2011). mulberry-leaves-in-sun.jpg.
URL http://foxhavenjournal.com/wp-content/uploads/2008/07/mulberry-leaves-in-sun.jpg

[10] Greene, R. (2003). Spherical harmonic lighting - the gritty details.
URL www.cs.columbia.edu/~cs4162/slides/spherical-harmonic-lighting.pdf

[11] Group, K. (2011). Opengl.
URL http://www.khronos.org/opengl/L

[12] Group, K. (2011). Opengl apis.
URL http://www.khronos.org/apis

[13] Ikits, M., & Magallon, M. (2011). The OpenGL Extension Wrangler Library.
URL http://glew.sourceforge.net/

[14] Institute, W. P. (2011). Interactive media & game development - imgd lab.
URL http://imgd.wpi.edu/imgdlab.html

[15] Jafolla, J., Stokes, J., & Sullivan, R. (1998). Phenomenological brdf modeling for engineering applications.
URL http://mi-projekte.hs-harz.de:8800/trac/brdf/export/10/trunk/literatur/jafolla.pdf

[16] Kaplanyan, A. (2009). Light propagation volumes in cryengine 3.
URL http://www.crytek.com/sites/default/files/Light_Propagation_Volumes.pdf

[17] Kol, A., & Cuntz, N. (2011). Dynamic particle coupling for gpu-based fluid simulation.
URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.89.2285&rep=rep1&type=pdf

[18] Lab, L. D. (2011). Lighting glossary.
URL www.lightingdesignlab.com/library/glossary.htm

[19] Library, B. P. (2011). Game physics simulation.
URL http://bulletphysics.org/wordpress/

[20] Microsoft (2009). Support c99.
URL http://connect.microsoft.com/VisualStudio/feedback/details/485416/support-c99

[21] Mki-Patola, T. (2003). Precomputed radiance transfer.
URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.131.6778

[22] NVIDIA (2007). Gpu gems 3.
URL http://http.developer.nvidia.com/GPUGems3/gpugems3_pref01.html

[23] NVIDIA (2009). FX Composer 2.5.
URL http://developer.nvidia.com/object/fx_composer_home.html

[24] of Computer Graphics, C. U. P. (2001). Reflectance data.
URL http://www.graphics.cornell.edu/online/measurements/reflectance/index.html

[25] Olszta, P. W., Umbach, A., & Baker, S. (2009). The Free OpenGL Utility Toolkit.
URL http://freeglut.sourceforge.net/

[26] OpenGL (2011). pipeline.gif.
URL http://www.opengl.org/sdk/docs/tutorials/ClockworkCoders/pipeline.gif

[27] OpenGL.org (2011). Geometry shaders.
URL http://www.opengl.org/wiki/Geometry_Shader

[28] OpenGL.org (2011). History of opengl.
URL http://www.opengl.org/wiki/History_of_OpenGL

[29] Palmer, J. M. (2003). Radiometry and photometry faq.
URL http://www.optics.arizona.edu/Palmer/rpfaq/rpfaq.pdf

[30] Pharr, M., & Humphreys, G. (2004). San Francisco, CA: Morgan-Kaufmann.

[31] Rideout, P. (2010). The OpenGL Shader Wrangler.
URL http://prideout.net/blog/?p=11

[32] Ritschel, T., Grosch, T., & Seidel, H.-P. (2009). Approximating Dynamic Global Illumination in Image Space. In *Proceedings ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D) 2009*.
URL http://www.mpi-inf.mpg.de/~ritschel/Papers/SSDO.pdf

[33] Rost, R. J. (2006). *OpenGL Shading Language, Second Edition*.

[34] Shanmugam, P., & Arikan, O. (2007). Hardware Accelerated Ambient Occlusion Techniques on GPUs. In *Proceedings ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D) 2009*, (pp. 73–80).
URL http://doi.acm.org/10.1145/1230100.1230113

[35] Shanmugam, P., & Arikan, O. (2007). Incremental Instant Radiosity for Real-Time Indirect Illumination. In *Eurographics Symposium on Rendering (2007)*.
URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.109.1355&rep=rep1&type=pdf

[36] Shreiner, D., Woo, M., Neider, J., & Davis, T. (2007). *OpenGL Programming Guide, Sixth Edition*. Boston, MA: Pearson Education, Inc.

[37] Sousa, T. (2008). *Vegetation Procedural Animation and Shading in Crysis*. Boston, MA: Pearson Education, Inc.

[38] Technologies, U. (2011). Unity: Game development tool.
URL http://unity3d.com/unity/

[39] TechPowerUp (2011). Gpu-z video card gpu information utility.
URL http://www.techpowerup.com/gpuz/

[40] Truevision (1991). Truevision TGA File Format Specification.
URL http://www.dca.fee.unicamp.br/~martino/disciplinas/ea978/tgaffs.pdf

[41] Wang, L., Wang, W., Dorsey, J., Yang, X., Guo, B., & Shum, H.-Y. (2005). Real-time rendering of plant leaves.
URL graphics.cs.yale.edu/julie/pubs/Leaf.pdf

[42] Zakia, R. D., & Stroebel, L. (1996). Woburn, MA: Butterworth-Heinemann.

Appendix A

Compiling & Running GSUITE

A.1 Hardware Requirements

Our code requires a graphics card capable of OpenGL version 3.2/3.3 commands. This allows a higher flexibility with textures and samplers sent to our shaders as well as support for GLSL 1.5.

A.2 Build Environment

Our environment is built in Windows, requires the OpenGL Extension Wrangler (GLEW) and FREEGLUT includes and libraries to compile, their respective binaries to run, and the latest drivers for your specific graphics hardware. The binaries and lib files are included with this project, but can also be found at `http://www.transmissionzero.co.uk/software/freeglut-devel/` and `http://glew.sourceforge.net/`

The binaries need to be copied to

`C:\Windows\System32`

and for x64 bit systems, to

`C:\Windows\SysWOW64`

Copy the lib and include folders to

`C:\Program Files\Microsoft Visual Studio 9.0\VC`

`C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC`

for 32 bit and 64 bit computers, respectively. [1]

The code was compiled using Visual Studio 2008[2]. If this environment is not available we have included a standalone version in the Release folder. To change the window, RSM and LPV sizes, recompile changing their respective global variables found in `defshade.c`.

[1] Note: Your installation directory may be different depending on your system setup.

[2] If there are glew.c compile errors, make sure the project's C/C++ properties "Additional Include Directories" points to the actual location of the project folder. For whatever reason, local references did not work.

A.3 Running & Hotkeys

- Escape: Terminate the program
- 0-9: Switch between individual render layers
- n: Toggle normalmap filter modes
- w, s, a, d : Translate camera position (x,y)
- t: Rotate the camera around the up vector
- v: Scroll through the LPV volume layers
- l, k: Rotate light
- p: Three level toggle; Spin object continuously, spin light continuously, stop.
- M, m: Increase and decrease the number of shadow samples

Appendix B

Shader Code

B.1 rsmGen.glsl

```
--Vertex

# version 150

in vec4 position;
in vec4 tangent;
in vec4 bitangent;
in vec4 texCoord;
in vec3 color;

out vec4 interpNormal;
out vec4 interpAlbedo;
out vec2 interpTexCoord;

uniform mat4 perspective;
uniform mat4 lightView;
uniform mat4 normalMatrix;

void main() {
  // Calculating stuff
  vec4 lightProjection = perspective * lightView * position;

  // Setting outputs
  gl_Position = lightProjection; // Position in light space
  interpTexCoord = texCoord.st;
  interpAlbedo = vec4(color, 1.0);
  interpNormal = normalMatrix * vec4(cross(tangent.xyz, bitangent.xyz), 0.0);
}
--Fragment

# version 150
/*
 *  For each pixel, this outputs:
 *    depth
 *    world space position
 *    surface normal
 *    radiant flux
 */

 // Radiant intensity (I) in direction (w) from flux (F) per
 //   normal = (n)
 //   <_|_> = dot product
 //
 //   I(w) = F * max(0, <n|w>)
 //
```

```
 // This will happen in another shader

in vec4 interpNormal;
in vec4 interpAlbedo;
in vec2 interpTexCoord;

out vec4 albedoFlux;
out vec4 normalDepth;

uniform sampler2D testTexture;

void main() {
  float radiantFlux = 1.0;
    albedoFlux = texture(testTexture, interpTexCoord);

    albedoFlux.a = radiantFlux;
    // linearly interpolated normals are not unit
    vec4 packedNormal = normalize(interpNormal);
    // to pack normal data into a color texture, we need to
    // remap [-1,1] to [0,1]
    packedNormal = packedNormal * 0.5 + 0.5;
    // same goes for depth
  normalDepth = vec4(packedNormal.xyz, 1.0 - gl_FragCoord.z);

}
```

B.2 volumeGen.glsl

```
-- Vertex

# version 150

in vec2 position;

uniform mat4 lpvTransform;    // Transform to lpv space
uniform mat4 lightTransform;  // Transform to light space
uniform mat4 lightProjection; // Transform for the projection of light space into cannonical ↩
    volume
                              // or < [-1,1], [-1,1], [-1,1] >
uniform float nSlices;
uniform float currentSlice;

out vec4 volumePos;
out mat4 volumeToLight; // Composed transform amtrix for moving sample offsets into
                        // RSM light space


void main() {


    float sliceZ = (currentSlice + 0.5 ) / nSlices;
    sliceZ = (sliceZ * 2.0) - 1.0;

    vec4 volumePosition = vec4( position.xy, sliceZ, 1.0 );
    gl_Position = volumePosition;

    mat4 volToRSM = lightProjection * lightTransform * inverse(lpvTransform);
    volumePos = volumePosition;

    volumeToLight = volToRSM;

}

-- Fragment

# version 150

in mat4 volumeToLight;

in vec4 volumePos;

out vec4 redCoeffs;
out vec4 greenCoeffs;
out vec4 blueCoeffs;

// Half the square root of the total number of sample points per cubie (NEEDS TO BE AN INT)
uniform int nEdgeSamples;
// This is the size of one side of a cubie divided by the number of samples
// (red, green, blue, radiance)
uniform float stepSize; uniform sampler2D rsmAlbedoFlux;
uniform sampler2D rsmNormal; // (x, y, z, depth)
uniform sampler2D rsmDepth;
uniform sampler2D randomOffsets;

uniform float nSlices;
uniform float depthBias;
uniform vec3 lightDirection;

const float pi = 3.141592594535;
```

```
const float band2Factor = sqrt(3.0 / (4.0 * pi));
int nEdgeSquared = nEdgeSamples * nEdgeSamples;
vec3 normedLightDir = normalize( lightDirection );

// Calculate the radius for a spherical vector
float calcRadius(vec3 sphericalVec) {
  return sqrt(dot(sphericalVec, sphericalVec) );
}

// First band spherical harmonic
float calcY00() {
  return 0.5 / sqrt(pi);
}

// Second band spherical harmonics
float calcY10(vec3 sampleDir) {
  return band2Factor * sampleDir.x * calcRadius(sampleDir);
}

float calcY11(vec3 sampleDir) {
  return band2Factor * sampleDir.y * calcRadius(sampleDir);
}

float calcY12(vec3 sampleDir) {
  return band2Factor * sampleDir.z * calcRadius(sampleDir);
}

// Third band SHs
float calcY20(vec3 sampleDir) {
  float constant = .25 * sqrt(5 / pi);
  float rad = calcRadius(sampleDir);
  float poly = pow(-1 * sampleDir.x, 2) - pow(-1 * sampleDir.y, 2) + (2 * pow(sampleDir.z, 2));

  return constant * (poly / rad);
}

float calcY21(vec3 sampleDir) {
  return .5 * sqrt(15 / pi) * ((sampleDir.y * sampleDir.z) / pow(calcRadius(sampleDir), 2));
}

float calcY22(vec3 sampleDir) {
  return .5 * sqrt(15 / pi) * ((sampleDir.z * sampleDir.x) / pow(calcRadius(sampleDir), 2));
}

float calcY23(vec3 sampleDir) {
  return .5 * sqrt(15 / pi) * ((sampleDir.x * sampleDir.y) / pow(calcRadius(sampleDir), 2));
}

float calcY24(vec3 sampleDir) {
  return .5 * sqrt(15 /pi) * ((pow(sampleDir.x, 2) - pow(sampleDir.y, 2)) / pow(calcRadius(↩
      sampleDir), 2));
}

// Fourth band spherical harmonic
float calcY30(vec3 sampleDir) {
  float constant = .25 * sqrt(7 / pi);
  float radius = calcRadius(sampleDir);
  float poly = sampleDir.z * ((2 * pow(sampleDir.z, 2)) - (3 * pow(sampleDir.x, 2)) - (3 * pow(↩
      sampleDir.y, 2)));

  return constant * (poly / pow(radius, 3));
}

float calcY31(vec3 sampleDir) {
  float constant = .25 * sqrt(35 / (2 * pi));
```

```
  float radius = calcRadius(sampleDir);
  float poly = ((3 * pow(sampleDir.x, 2)) - pow(sampleDir.y,2)) * sampleDir.y;

  return constant * (poly / pow(radius, 3));
}

float calcY32(vec3 sampleDir) {
  float constant = .25 * sqrt(35 / (2 * pi));
  float radius = calcRadius(sampleDir);
  float poly = (pow(sampleDir.x,2) - (3 * pow(sampleDir.y,2))) * sampleDir.z;

  return constant * (poly / pow(radius, 3));
}

float calcY33(vec3 sampleDir) {
  float constant = .25 * sqrt(105 / pi);
  float radius = calcRadius(sampleDir);
  float poly = (pow(sampleDir.x,2) - pow(sampleDir.y,2)) * sampleDir.z;

  return constant * (poly / pow(radius, 3));
}

float calcY34(vec3 sampleDir) {
  float constant = .5 * sqrt(105 / pi);
  float radius = calcRadius( sampleDir );
  float poly = sampleDir.x * sampleDir.y * sampleDir.z;

  return constant * (poly / pow(radius, 3));
}

float calcY35(vec3 sampleDir) {
  float constant = .25 * sqrt(21 / (2 * pi));
  float radius = calcRadius( sampleDir );
    // yuo forgot a ".y" on the last pow(...) call there...I think
  float poly = sampleDir.y * ((4 * pow(sampleDir.z, 2)) - pow(sampleDir.x, 2) - pow(sampleDir.y↩
      ,2));

  return constant * (poly / pow(radius, 3));
}

float calcY36(vec3 sampleDir) {
  float constant = .25 * sqrt(21 / (2 * pi));
  float radius = calcRadius(sampleDir);
  float poly = sampleDir.x * ((4 * pow(sampleDir.z, 2)) - pow(sampleDir.x, 2) - pow(sampleDir.y↩
      ,2));

  return constant * (poly / pow(radius, 3));
}

// Each time this code runs, it runs on one cubie
// of one slice of the entire light propagation volume
void main() {
  int x = 0, y = 0, z = 0;
    float actualSamples = 0.0;
    int index = 0;

  vec3 sampledNormal;
    float sampledDepth; // The Alpha component of the NormalDepth texture
    float volDepth;

    float nEdgeSamplesf = nEdgeSamples;

    vec4 sampledAlbedoFlux;
    float radiantFlux; // The Alpha component of the AlbedoFlux texture
```

```
vec4 rgbaTemp = vec4(1.0,1.0,1.0,1.0);
  //vec4 coeffsTemp = vec4(0.0,0.0,0.0,0.0);
  vec4 redCoeffsTemp = vec4(0.0,0.0,0.0,0.0);
  vec4 greenCoeffsTemp = vec4(0.0,0.0,0.0,0.0);
  vec4 blueCoeffsTemp = vec4(0.0,0.0,0.0,0.0);

  vec4 sampleOffset;
  vec4 samplePos = volumePos;

  int nSamples = nEdgeSamples;// * nEdgeSamples * nEdgeSamples;
 // int nEdgeSquared = nEdgeSamples * nEdgeSamples;

  // Interate across the cubie
  for( index = 0; index < nSamples; index++){

      // A little nifty math saves two loops, which are expensive
      // (seriously, it ran smoother this way)
      //x = index % nEdgeSamples;
     // y = (index / nEdgeSamples) % nEdgeSamples;
      //z = (index / nEdgeSquared) % nEdgeSamples;
      // So we have the integral index of a pixel in the slice stating at the
      // lower left-hand corner, [x, y].
      // Now we shift that into _cubie_ volume coordinates

      float offsetLookup = index;
      offsetLookup /= 64.0;

      sampleOffset = texture( randomOffsets, vec2( offsetLookup, 0.5 ));
      sampleOffset = (sampleOffset * 2.0) - 1.0;
      sampleOffset /= 4.0 * nSlices;
      sampleOffset.w = 0.0;

      samplePos += sampleOffset;
      // And now transform that volume location into RSM/light space
      samplePos = volumeToLight * samplePos;

      // We use that sample position to lookup the albedo RSM map so we can get
      // the depth, being careful to shift the RSM/Light space coordinates to
      // Texture Space

      // We test this by seeing if the depth component is near enough to our
      // sample position to count.
      sampledDepth = texture( rsmDepth, samplePos.xy * 0.5 + 0.5).r;
      volDepth = 1.0 - (samplePos.z * 0.5 + 0.5);

      // Checking for zero removes erroneous seeding of the back of the depth map as a ↩
          surface
      // which causes a thick band of light to slice the scene
      if( sampledDepth != 0.0
          && sampledDepth + depthBias <= volDepth + stepSize
          && sampledDepth + depthBias >= volDepth - stepSize ){

              sampledAlbedoFlux = texture( rsmAlbedoFlux, samplePos.xy * 0.5 + 0.5 );
              sampledNormal = texture( rsmNormal, samplePos.xy * 0.5 + 0.5 ).xyz;
              sampledNormal = sampledNormal.xyz * 2.0 - 1.0;
              normalize( sampledNormal );
              vec3 bounceDir = reflect( -normedLightDir, sampledNormal );

              vec4 bounceCoeff = vec4( calcY00(),
                                       calcY10( bounceDir ),
                                       calcY11( bounceDir ),
                                       calcY12( bounceDir ) );

              redCoeffsTemp += sampledAlbedoFlux.r * bounceCoeff;
              greenCoeffsTemp += sampledAlbedoFlux.g * bounceCoeff;
```

```
                blueCoeffsTemp += sampledAlbedoFlux.b * bounceCoeff;

                float fudge = 0.75;
                vec4 transmitCoeff = vec4( calcY00(),
                                           calcY10( normedLightDir ),
                                           calcY11( normedLightDir ),
                                           calcY12( normedLightDir ) );

                redCoeffsTemp += fudge * sampledAlbedoFlux.r * transmitCoeff;
                greenCoeffsTemp += fudge * sampledAlbedoFlux.g * transmitCoeff;
                blueCoeffsTemp += fudge * sampledAlbedoFlux.b * transmitCoeff;

                actualSamples += 2.0;
            }
        }

        redCoeffs = vec4( 0.0, 0.0, 0.0, 0.0);
        greenCoeffs = vec4( 0.0, 0.0, 0.0, 0.0);
        blueCoeffs = vec4( 0.0, 0.0, 0.0, 0.0);

        if( actualSamples != 0){
            redCoeffs = 4.0 * pi * redCoeffsTemp / actualSamples;
            greenCoeffs = 4.0 * pi * greenCoeffsTemp / actualSamples;
            blueCoeffs = 4.0 * pi * blueCoeffsTemp / actualSamples;
        }

}
```

B.3 volumeProp.glsl

```
-- Vertex

# version 150

in vec2 position;

out vec3 volPos;

uniform float nSlices;
uniform float currentSlice;

void main() {

    float sliceZ = ( currentSlice + 0.5 ) / nSlices;
    sliceZ = ( sliceZ * 2.0 ) - 1.0;

    gl_Position = vec4( position, sliceZ, 1.0) ;
    volPos = vec3( position, sliceZ ) * 0.5 + 0.5;

}

-- Fragment

# version 150

in vec3 volPos;

out vec4 redProp;
out vec4 greenProp;
out vec4 blueProp;

uniform float stepSize;

uniform sampler3D redHarmonics;
uniform sampler3D greenHarmonics;
uniform sampler3D blueHarmonics;

const float pi = 3.141592594535;
const float band2factor = sqrt(3.0 / (4.0 * pi));

float calcRadius(vec3 sphericalVec) {
    return sqrt(dot(sphericalVec, sphericalVec));
}

// First band spherical harmonic
float calcY00() {
    return 0.5 / sqrt(pi);
}

// Second band spherical harmonics
float calcY10(vec3 sampleDir) {
    return band2factor * sampleDir.x / calcRadius(sampleDir);
}

float calcY11(vec3 sampleDir) {
    return band2factor * sampleDir.y / calcRadius(sampleDir);
}

float calcY12(vec3 sampleDir) {
    return band2factor * sampleDir.z / calcRadius(sampleDir);
}
```

```
float reconstructSH(vec3 dir, vec4 coeffs) {
    float result = coeffs.x * calcY00();
    result += coeffs.y * calcY10(dir);
    result += coeffs.z * calcY11(dir);
    result += coeffs.w * calcY12(dir);

    return result;
}

vec4 calculateBands_1_2( vec3 direction, float weight )
{
        vec4 result = vec4(0.0,0.0,0.0,0.0);
        result.x = 0.5 / sqrt(pi);
        result.y = calcY10(direction);
        result.z = calcY11(direction);
        result.w = calcY12(direction);
        return result * weight;
}

/*
* Given three LPV slices, propagate between them
*/
void main() {

    // f for falloff
    float f = 0.65;

    int nRedProps = 1;
    int nGreenProps = 1;
    int nBlueProps = 1;

    vec4 redPropStore = vec4(0.0, 0.0, 0.0, 0.0);
    vec4 greenPropStore = vec4(0.0, 0.0, 0.0, 0.0);
    vec4 bluePropStore = vec4(0.0, 0.0, 0.0, 0.0);

    vec4 redPropCoeffs = vec4(0.0);//texture( redHarmonics, volPos );
    vec4 greenPropCoeffs = vec4(0.0);//texture( greenHarmonics, volPos );
    vec4 bluePropCoeffs = vec4(0.0); //texture( blueHarmonics, volPos );

    vec3 offset = vec3( stepSize, 0.0, 0.0 );
    vec3 direction = vec3( -1.0, 0.0, 0.0 );

    vec4 redCoeffsSample = texture( redHarmonics, volPos + offset );
    vec4 greenCoeffsSample = texture( greenHarmonics, volPos + offset );
    vec4 blueCoeffsSample = texture( blueHarmonics, volPos + offset );

    float RedL = max(0.0, reconstructSH( direction, redCoeffsSample ));
    float GreenL = max( 0.0, reconstructSH( direction, greenCoeffsSample ));
    float BlueL = max(0.0, reconstructSH( direction, blueCoeffsSample ));

    redPropCoeffs += calculateBands_1_2( direction, f * RedL );
    greenPropCoeffs += calculateBands_1_2( direction, f * GreenL);
    bluePropCoeffs += calculateBands_1_2( direction, f * BlueL );

    offset = vec3( -stepSize, 0.0, 0.0 );
    direction = vec3( 1.0, 0.0, 0.0 );

    redCoeffsSample = texture( redHarmonics, volPos + offset );
    greenCoeffsSample = texture( greenHarmonics, volPos + offset );
    blueCoeffsSample = texture( blueHarmonics, volPos + offset );

    RedL = max(0.0, reconstructSH( direction, redCoeffsSample ));
    GreenL = max( 0.0, reconstructSH( direction, greenCoeffsSample ));
    BlueL = max(0.0, reconstructSH( direction, blueCoeffsSample ));
```

```
redPropCoeffs += calculateBands_1_2( direction, f * RedL);
greenPropCoeffs += calculateBands_1_2( direction, f * GreenL);
bluePropCoeffs += calculateBands_1_2( direction, f * BlueL);

offset = vec3(  0.0, stepSize, 0.0 );
direction = vec3( 0.0, -1.0, 0.0 );

redCoeffsSample = texture( redHarmonics, volPos + offset );
greenCoeffsSample = texture( greenHarmonics, volPos + offset );
blueCoeffsSample = texture( blueHarmonics, volPos + offset );

RedL = max(0.0, reconstructSH( direction, redCoeffsSample ));
GreenL = max( 0.0, reconstructSH( direction, greenCoeffsSample ));
BlueL = max(0.0, reconstructSH( direction, blueCoeffsSample ));

redPropCoeffs += calculateBands_1_2( direction, f * RedL);
greenPropCoeffs += calculateBands_1_2( direction, f * GreenL);
bluePropCoeffs += calculateBands_1_2( direction, f * BlueL);

offset = vec3(  0.0, -stepSize, 0.0 );
direction = vec3( 0.0, 1.0, 0.0 );

redCoeffsSample = texture( redHarmonics, volPos + offset );
greenCoeffsSample = texture( greenHarmonics, volPos + offset );
blueCoeffsSample = texture( blueHarmonics, volPos + offset );

RedL = max(0.0, reconstructSH( direction, redCoeffsSample ));
GreenL = max( 0.0, reconstructSH( direction, greenCoeffsSample ));
BlueL = max(0.0, reconstructSH( direction, blueCoeffsSample ));

redPropCoeffs += calculateBands_1_2( direction, f * RedL );
greenPropCoeffs += calculateBands_1_2( direction, f * GreenL );
bluePropCoeffs += calculateBands_1_2( direction, f * BlueL );

offset = vec3(  0.0, 0.0, stepSize );
direction = vec3( 0.0, 0.0, -1.0 );

redCoeffsSample = texture( redHarmonics, volPos + offset );
greenCoeffsSample = texture( greenHarmonics, volPos + offset );
blueCoeffsSample = texture( blueHarmonics, volPos + offset );

RedL = max(0.0, reconstructSH( direction, redCoeffsSample ));
GreenL = max( 0.0, reconstructSH( direction, greenCoeffsSample ));
BlueL = max(0.0, reconstructSH( direction, blueCoeffsSample ));

redPropCoeffs += calculateBands_1_2( direction, f * RedL );
greenPropCoeffs += calculateBands_1_2( direction, f * GreenL );
bluePropCoeffs += calculateBands_1_2( direction, f * BlueL );

offset = vec3(  0.0, 0.0, -stepSize );
direction = vec3( 0.0, 0.0, 1.0 );

redCoeffsSample = texture( redHarmonics, volPos + offset );
greenCoeffsSample = texture( greenHarmonics, volPos + offset );
blueCoeffsSample = texture( blueHarmonics, volPos + offset );

RedL = max(0.0, reconstructSH( direction, redCoeffsSample ));
GreenL = max( 0.0, reconstructSH( direction, greenCoeffsSample ));
BlueL = max(0.0, reconstructSH( direction, blueCoeffsSample ));

redPropCoeffs += calculateBands_1_2( direction, f * RedL );
greenPropCoeffs += calculateBands_1_2( direction, f * GreenL );
bluePropCoeffs += calculateBands_1_2( direction, f * BlueL );
```

```
//////////// Second order neighbors (12) /////////////////

    offset = vec3(  stepSize, stepSize, 0.0 );
    direction = vec3( -0.707107, -0.707107, 0.0 );

    redCoeffsSample = texture( redHarmonics, volPos + offset );
    greenCoeffsSample = texture( greenHarmonics, volPos + offset );
    blueCoeffsSample = texture( blueHarmonics, volPos + offset );

    RedL = max(0.0, reconstructSH( direction, redCoeffsSample ));
    GreenL = max( 0.0, reconstructSH( direction, greenCoeffsSample ));
    BlueL = max(0.0, reconstructSH( direction, blueCoeffsSample ));

    redPropCoeffs += calculateBands_1_2( direction, f * RedL * 0.5 );
    greenPropCoeffs += calculateBands_1_2( direction, f * GreenL * 0.5 );
    bluePropCoeffs += calculateBands_1_2( direction, f * BlueL * 0.5 );

    offset = vec3(  -stepSize, stepSize, 0.0 );
    direction = vec3( 0.707107, -0.707107, 0.0 );

    redCoeffsSample = texture( redHarmonics, volPos + offset );
    greenCoeffsSample = texture( greenHarmonics, volPos + offset );
    blueCoeffsSample = texture( blueHarmonics, volPos + offset );

    RedL = max(0.0, reconstructSH( direction, redCoeffsSample ));
    GreenL = max( 0.0, reconstructSH( direction, greenCoeffsSample ));
    BlueL = max(0.0, reconstructSH( direction, blueCoeffsSample ));

    redPropCoeffs += calculateBands_1_2( direction, f * RedL * 0.5 );
    greenPropCoeffs += calculateBands_1_2( direction, f * GreenL * 0.5 );
    bluePropCoeffs += calculateBands_1_2( direction, f * BlueL * 0.5 );

    offset = vec3(  -stepSize, -stepSize, 0.0 );
    direction = vec3( 0.707107, 0.707107, 0.0 );

    redCoeffsSample = texture( redHarmonics, volPos + offset );
    greenCoeffsSample = texture( greenHarmonics, volPos + offset );
    blueCoeffsSample = texture( blueHarmonics, volPos + offset );

    RedL = max(0.0, reconstructSH( direction, redCoeffsSample ));
    GreenL = max( 0.0, reconstructSH( direction, greenCoeffsSample ));
    BlueL = max(0.0, reconstructSH( direction, blueCoeffsSample ));

    redPropCoeffs += calculateBands_1_2( direction, f * RedL * 0.5 );
    greenPropCoeffs += calculateBands_1_2( direction, f * GreenL * 0.5 );
    bluePropCoeffs += calculateBands_1_2( direction, f * BlueL * 0.5 );

    offset = vec3(  stepSize, -stepSize, 0.0 );
    direction = vec3( -0.707107, 0.707107, 0.0 );

    redCoeffsSample = texture( redHarmonics, volPos + offset );
    greenCoeffsSample = texture( greenHarmonics, volPos + offset );
    blueCoeffsSample = texture( blueHarmonics, volPos + offset );

    RedL = max(0.0, reconstructSH( direction, redCoeffsSample ));
    GreenL = max( 0.0, reconstructSH( direction, greenCoeffsSample ));
    BlueL = max(0.0, reconstructSH( direction, blueCoeffsSample ));

    redPropCoeffs += calculateBands_1_2( direction, f * RedL * 0.5 );
    greenPropCoeffs += calculateBands_1_2( direction, f * GreenL * 0.5 );
    bluePropCoeffs += calculateBands_1_2( direction, f * BlueL * 0.5 );
//////////////////////////////
    offset = vec3(  stepSize, 0.0, stepSize );
    direction = vec3( -0.707107, 0.0, -0.707107 );
```

```
redCoeffsSample = texture( redHarmonics, volPos + offset );
greenCoeffsSample = texture( greenHarmonics, volPos + offset );
blueCoeffsSample = texture( blueHarmonics, volPos + offset );

RedL = max(0.0, reconstructSH( direction, redCoeffsSample ));
GreenL = max( 0.0, reconstructSH( direction, greenCoeffsSample ));
BlueL = max(0.0, reconstructSH( direction, blueCoeffsSample ));

redPropCoeffs += calculateBands_1_2( direction, f * RedL * 0.5 );
greenPropCoeffs += calculateBands_1_2( direction, f * GreenL * 0.5 );
bluePropCoeffs += calculateBands_1_2( direction, f * BlueL * 0.5 );

offset = vec3( -stepSize, 0.0, stepSize );
direction = vec3( 0.707107, 0.0, -0.707107 );

redCoeffsSample = texture( redHarmonics, volPos + offset );
greenCoeffsSample = texture( greenHarmonics, volPos + offset );
blueCoeffsSample = texture( blueHarmonics, volPos + offset );

RedL = max(0.0, reconstructSH( direction, redCoeffsSample ));
GreenL = max( 0.0, reconstructSH( direction, greenCoeffsSample ));
BlueL = max(0.0, reconstructSH( direction, blueCoeffsSample ));

redPropCoeffs += calculateBands_1_2( direction, f * RedL * 0.5 );
greenPropCoeffs += calculateBands_1_2( direction, f * GreenL * 0.5 );
bluePropCoeffs += calculateBands_1_2( direction, f * BlueL * 0.5 );

offset = vec3( -stepSize, 0.0, -stepSize );
direction = vec3( 0.707107, 0.0, 0.707107 );

redCoeffsSample = texture( redHarmonics, volPos + offset );
greenCoeffsSample = texture( greenHarmonics, volPos + offset );
blueCoeffsSample = texture( blueHarmonics, volPos + offset );

RedL = max(0.0, reconstructSH( direction, redCoeffsSample ));
GreenL = max( 0.0, reconstructSH( direction, greenCoeffsSample ));
BlueL = max(0.0, reconstructSH( direction, blueCoeffsSample ));

redPropCoeffs += calculateBands_1_2( direction, f * RedL * 0.5 );
greenPropCoeffs += calculateBands_1_2( direction, f * GreenL * 0.5 );
bluePropCoeffs += calculateBands_1_2( direction, f * BlueL * 0.5 );

offset = vec3( stepSize, 0.0, -stepSize );
direction = vec3( -0.707107, 0.0, 0.707107 );

redCoeffsSample = texture( redHarmonics, volPos + offset );
greenCoeffsSample = texture( greenHarmonics, volPos + offset );
blueCoeffsSample = texture( blueHarmonics, volPos + offset );

RedL = max(0.0, reconstructSH( direction, redCoeffsSample ));
GreenL = max( 0.0, reconstructSH( direction, greenCoeffsSample ));
BlueL = max(0.0, reconstructSH( direction, blueCoeffsSample ));

redPropCoeffs += calculateBands_1_2( direction, f * RedL * 0.5 );
greenPropCoeffs += calculateBands_1_2( direction, f * GreenL * 0.5 );
bluePropCoeffs += calculateBands_1_2( direction, f * BlueL * 0.5 );
//////////////////////
offset = vec3( 0.0, stepSize, stepSize );
direction = vec3( 0.0, -0.707107, -0.707107 );

redCoeffsSample = texture( redHarmonics, volPos + offset );
greenCoeffsSample = texture( greenHarmonics, volPos + offset );
blueCoeffsSample = texture( blueHarmonics, volPos + offset );

RedL = max(0.0, reconstructSH( direction, redCoeffsSample ));
```

```
    GreenL = max( 0.0, reconstructSH( direction, greenCoeffsSample ));
    BlueL = max(0.0, reconstructSH( direction, blueCoeffsSample ));

    redPropCoeffs += calculateBands_1_2( direction, f * RedL * 0.5 );
    greenPropCoeffs += calculateBands_1_2( direction, f * GreenL * 0.5 );
    bluePropCoeffs += calculateBands_1_2( direction, f * BlueL * 0.5 );

    offset = vec3(  0.0, -stepSize, stepSize );
    direction = vec3( 0.0, 0.707107, -0.707107 );

    redCoeffsSample = texture( redHarmonics, volPos + offset );
    greenCoeffsSample = texture( greenHarmonics, volPos + offset );
    blueCoeffsSample = texture( blueHarmonics, volPos + offset );

    RedL = max(0.0, reconstructSH( direction, redCoeffsSample ));
    GreenL = max( 0.0, reconstructSH( direction, greenCoeffsSample ));
    BlueL = max(0.0, reconstructSH( direction, blueCoeffsSample ));

    redPropCoeffs += calculateBands_1_2( direction, f * RedL * 0.5 );
    greenPropCoeffs += calculateBands_1_2( direction, f * GreenL * 0.5 );
    bluePropCoeffs += calculateBands_1_2( direction, f * BlueL * 0.5 );

    offset = vec3(  0.0, -stepSize, -stepSize );
    direction = vec3( 0.0, 0.707107, 0.707107 );

    redCoeffsSample = texture( redHarmonics, volPos + offset );
    greenCoeffsSample = texture( greenHarmonics, volPos + offset );
    blueCoeffsSample = texture( blueHarmonics, volPos + offset );

    RedL = max(0.0, reconstructSH( direction, redCoeffsSample ));
    GreenL = max( 0.0, reconstructSH( direction, greenCoeffsSample ));
    BlueL = max(0.0, reconstructSH( direction, blueCoeffsSample ));

    redPropCoeffs += calculateBands_1_2( direction, f * RedL * 0.5 );
    greenPropCoeffs += calculateBands_1_2( direction, f * GreenL * 0.5 );
    bluePropCoeffs += calculateBands_1_2( direction, f * BlueL * 0.5 );

    offset = vec3(  0.0, stepSize, -stepSize );
    direction = vec3( 0.0, -0.707107, 0.707107 );

    redCoeffsSample = texture( redHarmonics, volPos + offset );
    greenCoeffsSample = texture( greenHarmonics, volPos + offset );
    blueCoeffsSample = texture( blueHarmonics, volPos + offset );

    RedL = max(0.0, reconstructSH( direction, redCoeffsSample ));
    GreenL = max( 0.0, reconstructSH( direction, greenCoeffsSample ));
    BlueL = max(0.0, reconstructSH( direction, blueCoeffsSample ));

    redPropCoeffs += calculateBands_1_2( direction, f * RedL * 0.5 );
    greenPropCoeffs += calculateBands_1_2( direction, f * GreenL * 0.5 );
    bluePropCoeffs += calculateBands_1_2( direction, f * BlueL * 0.5 );

    redProp = redPropCoeffs  / 18.0;
    greenProp = greenPropCoeffs / 18.0;
    blueProp = bluePropCoeffs / 18.0;

    redProp += texture( redHarmonics, volPos );
    greenProp += texture( greenHarmonics, volPos );
    blueProp += texture( blueHarmonics, volPos );

}
```

B.4 normalShade.glsl

```
--Vertex

void main(void){

gl_Position     = ftransform();

}

--Fragment

void main(void){
 gl_FragColor = vec4(.2,.8,.8,.7);
}
```

B.5 defShader.glsl

```
--Vertex

# version 150

in vec4 position;
in vec4 texCoord;

in vec4 tangent;
in vec4 bitangent;

in vec3 color;

out vec4 interpColor;
out vec2 interpTexCoord;
out vec3 interpBitangent;
out vec3 interpTangent;
out vec3 interpNormal;
out vec4 fragPos;

uniform mat4 perspective;
uniform mat4 modelView;
uniform mat4 normalMatrix;
uniform float time;

uniform sampler2D testTexture;

void main()
{
    vec4 deltaY = vec4( 0.0,
                        cos( time / 1000.0 *2 + (position. x + position.z)/20 -(position.y*↩
                            position.y/70) + position.z*position.z/300) * texture(testTexture,↩
                             texCoord.xy).a,
                        0.0, 0.0 );

    vec4 movingPos = position + (deltaY * 0.3);
    vec4 projection = perspective * modelView * movingPos;

    // to pack W into a color channel, we need to ensure it is in the range [-1,1]
    // first by adding or subtracting one to ensure it has a magnitude greater than one
    // the absolute value of a number divided by the number is either -1 or +1
    fragPos = movingPos;
    gl_Position = projection;
    interpColor = vec4(color, 1.0);
    interpTexCoord = texCoord.st;

    mat3x3 normalMat = mat3(normalMatrix[0].xyz,normalMatrix[1].xyz,normalMatrix[2].xyz);

    interpBitangent = (modelView * (bitangent + deltaY * 0.1)).xyz;
    interpTangent = (modelView * (tangent + deltaY * 0.1)).xyz;
}

--Fragment

# version 150

in vec4 interpColor;
in vec2 interpTexCoord;
in vec3 interpBitangent;
in vec3 interpTangent;
in vec4 fragPos;
```

```
out vec4 albedo;
out vec4 outPosition;
out vec4 normal;
out float depth;

uniform int normalMode;

uniform sampler2D testTexture;
uniform sampler2D testNormalTexture;

void main(){

    vec4 normalTransmit = texture(testNormalTexture, interpTexCoord);
    vec3 normalPrime = normalTransmit.xyz * 2.0 - 1.0;

    if( normalMode == 0){
        float spread = 0.35;
        vec3 sample00 = texture( testNormalTexture,
                                 interpTexCoord.xy + spread * vec2( 0.001953125, 0.0078125 )).↩
                                     xyz;
        sample00 = normalize(sample00 * 2.0 - 1.0);
        vec3 sample10 = texture( testNormalTexture,
                                 interpTexCoord.xy + spread * vec2(  0.005859375, 0.001953125 )↩
                                     ).xyz;
        sample10 = normalize(sample10 * 2.0 - 1.0);
        vec3 sample20 = texture( testNormalTexture,
                                 interpTexCoord.xy + spread * vec2(  0.0078125,  -0.00390625 ))↩
                                     .xyz;
        sample20 = normalize(sample20 * 2.0 - 1.0);
        vec3 sample30 = texture( testNormalTexture,
                                 interpTexCoord.xy + spread * vec2( 0.00292969,  -0.005859375 )↩
                                     ).xyz;
        sample30 = normalize(sample30 * 2.0 - 1.0);

        vec3 sample01 = texture( testNormalTexture,
                                 interpTexCoord.xy + spread * vec2( -0.001953125, -0.0078125 ))↩
                                     .xyz;
        sample01 = normalize(sample01 * 2.0 - 1.0);
        vec3 sample11 = texture( testNormalTexture,
                                 interpTexCoord.xy + spread * vec2(  -0.005859375, -0.001953125↩
                                      )).xyz;
        sample11 = normalize(sample11 * 2.0 - 1.0);
        vec3 sample21 = texture( testNormalTexture,
                                 interpTexCoord.xy + spread * vec2(  -0.0078125,  0.00390625 ))↩
                                     .xyz;
        sample21 = normalize(sample21 * 2.0 - 1.0);
        vec3 sample31 = texture( testNormalTexture,
                                 interpTexCoord.xy + spread * vec2( -0.00292969,  0.005859375 )↩
                                     ).xyz;
        sample31 = normalize(sample31 * 2.0 - 1.0);

        sample00 *= dot( sample00, normalPrime );
        sample10 *= dot( sample10, normalPrime );
        sample20 *= dot( sample20, normalPrime );
        sample30 *= dot( sample30, normalPrime );

        sample01 *= dot( sample01, normalPrime );
        sample11 *= dot( sample11, normalPrime );
        sample21 *= dot( sample21, normalPrime );
        sample31 *= dot( sample31, normalPrime );

        normalPrime +=   sample00
                       + sample10
                       + sample20
                       + sample30
```

```
                        + sample01
                        + sample11
                        + sample21
                        + sample31;

    }

    normalPrime = normalize(normalPrime);



    vec3 normedTangent = normalize(interpTangent);
    vec3 normedBitangent = normalize(interpBitangent);
    vec3 normedNormal = normalize(cross(interpTangent, interpBitangent));

    mat3x3 surfaceBasis = mat3x3(normedTangent,
                                 normedBitangent,
                                 normedNormal);

    vec3 outNormal = normalize(surfaceBasis * normalPrime);

    vec4 textureColor = texture(testTexture, interpTexCoord );


    albedo = textureColor;

    normal = vec4(outNormal  * 0.5+.5, normalTransmit.a);

    outPosition = fragPos;

    depth = 1.0 - gl_FragCoord.z;

}
```

B.6 defCompositor.glsl

```
--Vertex

# version 150

in vec4 viewPosition;

uniform mat4 modelView;
uniform mat4 perspective;

out vec2 texCoord;
out mat4 toWorldSpace;
out mat3 normalToWorld;



void main()
{
    gl_Position = viewPosition;

    texCoord = viewPosition.xy * vec2(0.5) + vec2(0.5);
    toWorldSpace = inverse( modelView );

    mat4 normModelView = inverse( modelView );
    normalToWorld = mat3(normModelView[0].xyz, normModelView[1].xyz, normModelView[2].xyz);
}

--Fragment

# version 150

in vec2 texCoord;
uniform vec3 cameraLocation;
uniform int layer;
in mat4 toWorldSpace;
in mat3 normalToWorld;

uniform mat4 volumeTransform;
uniform mat4 toShadowMap;

uniform sampler2D albedo;
uniform sampler2D position;
uniform sampler2D normal;
uniform sampler2D shadowMap;
uniform sampler2D skySphere;
uniform sampler1D randomOffsets;
uniform float shadowBias;
uniform int nShadowSamples;
uniform float specularPower;
uniform float nearPlane;
uniform float projPlaneHalfHeight;
uniform float width;
uniform float height;

uniform sampler3D redHarmonics;
uniform sampler3D greenHarmonics;
uniform sampler3D blueHarmonics;

uniform vec3 lightDirection;

out vec4 fragColor;
```

```
float pi = 3.141592594535;

float calcRadius(vec3 sphericalVec) {
  return sqrt(length(sphericalVec));
}

// First band spherical harmonic
float calcY00() {
  return 0.5 / sqrt(pi);
}

// Second band spherical harmonics
float calcY10(vec3 sampleDir) {
  return sqrt(3 / (4 * pi)) * sampleDir.x * calcRadius(sampleDir);
}

float calcY11(vec3 sampleDir) {
  return sqrt(3 / (4 * pi)) * sampleDir.y * calcRadius(sampleDir);
}

float calcY12(vec3 sampleDir) {
  return sqrt(3 / (4 * pi)) * sampleDir.z * calcRadius(sampleDir);
}

float reconstructSH(vec3 dir, vec4 coeffs) {
    float result = coeffs.x * calcY00();
    result += coeffs.y * calcY10(dir);
    result += coeffs.z * calcY11(dir);
    result += coeffs.w * calcY12(dir);

    return result;
}

void main(){

    float aoSpread = 1.0/64.0;

    vec3 toLightDir = -normalize(lightDirection);
    vec4 textureColor = texture(albedo, texCoord);
    textureColor.a = 1.0;

    vec4 cameraPosColor = texture(position, texCoord);
    cameraPosColor.w = 1.0;

    vec4 normalTransmit = texture(normal, texCoord);
    float transmitance = normalTransmit.a;
    vec3 normalColor = normalize(normalTransmit.xyz * 2.0 - 1.0);
    normalColor = normalToWorld * normalColor;
    normalColor = normalize( normalColor );

    vec4 volumePos = (volumeTransform * cameraPosColor) * 0.5 + 0.5;

    // generate the spherical angles of the world space normal
    float phi = acos( normalColor.y );
    float theta = atan( normalColor.z, normalColor.x );

    // calculate the normalized radius of the latitude mapped to a spherical texture
    float r = phi/pi;
    // calculate the texture coordinates of the normal on the sky sphere texture
    float s = r * cos(theta) * 0.5 + 0.5;
    float t = r * sin(theta) * 0.5 + 0.5;

    vec4 skyIlluminant = vec4( 0.0,0.0,0.0,0.0 );
    if( cameraPosColor.r != 0.0
        && cameraPosColor.g != 0.0
```

```
    && cameraPosColor.b != 0.0){
    skyIlluminant = texture( skySphere, vec2( s, t ) );
}
skyIlluminant.a = 1.0;

vec4 shadowMapPos = toShadowMap * cameraPosColor;
shadowMapPos /= shadowMapPos.w;
shadowMapPos = shadowMapPos * 0.5 + 0.5;
float biasedFragDepth = - shadowBias - (1.0 - shadowMapPos.z);

// Seventeen RSM fragdepth difference samples
float shadowFactor = max(0.0,texture( shadowMap, shadowMapPos.xy).r + biasedFragDepth);
float depthDiff = min( 1.0,shadowFactor);

int i = 0;
float lookUp = 0.0;
vec2 offset;
for( i = 0; i < nShadowSamples; i++){
    lookUp = i;
    lookUp /= nShadowSamples;
    offset = texture( randomOffsets, lookUp ).rb;
    offset = offset * 2.0 - 1.0;
    offset /= 256.0;
    shadowFactor += max(0.0,texture( shadowMap, shadowMapPos.xy + offset).r + ↩
        biasedFragDepth);
}

shadowFactor = 1.0 - clamp(shadowFactor, 0.0, 1.0);

vec3 sampleOffset = normalColor * 0.001;

float redLightGI = clamp( reconstructSH( -normalColor, texture( redHarmonics, volumePos.xyz↩
     + sampleOffset )), 0.0, 1.0 );
float greenLightGI = clamp( reconstructSH( -normalColor, texture( greenHarmonics, volumePos↩
    .xyz + sampleOffset )), 0.0, 1.0 );
float blueLightGI = clamp( reconstructSH( -normalColor, texture( blueHarmonics, volumePos.↩
    xyz + sampleOffset )), 0.0, 1.0 );

float backScatter = 0.05 + min( 2.0, max(dot( normalColor, -toLightDir), 0.0) / max(0.1,( ↩
    1.0 - transmitance )));

float redLightTI = backScatter * clamp( reconstructSH( -toLightDir,  texture( redHarmonics,↩
     volumePos.xyz + sampleOffset )), 0.0, 1.0 );
float greenLightTI = backScatter * clamp( reconstructSH( -toLightDir,  texture( ↩
    greenHarmonics, volumePos.xyz + sampleOffset )), 0.0, 1.0 );
float blueLightTI = backScatter * clamp( reconstructSH( -toLightDir,  texture( ↩
    blueHarmonics, volumePos.xyz + sampleOffset )), 0.0, 1.0 );

float shadeFactor = clamp(dot( normalColor, toLightDir ), 0.0, 1.0);

vec4 GIColor = vec4( redLightGI, greenLightGI, blueLightGI, 1.0 );
vec4 TIColor = vec4( redLightTI, greenLightTI, blueLightTI, 1.0 )
               * shadowFactor;

float diffuse = shadeFactor * shadowFactor;
vec4 illuminants = vec4( diffuse ) + vec4( 1.0 - diffuse ) * GIColor;
illuminants.r += (1.0 - illuminants.r ) * TIColor.r;
illuminants.g += (1.0 - illuminants.g ) * TIColor.g;
illuminants.b += (1.0 - illuminants.b ) * TIColor.b;

illuminants.a = 1.0;

vec3 viewVector = cameraLocation;
viewVector -= cameraPosColor.xyz;
viewVector = normalize(viewVector);
```

```
    vec3 halfVector = normalize( viewVector + toLightDir );
    float cosTh = clamp( dot( normalColor, halfVector ), 0.0, 1.0 );
    float cosTi = clamp( dot( normalColor, toLightDir ), 0.0, 1.0 );
    float specularFactor = (specularPower + 0.8)/(8.0 * pi) * pow(cosTh, specularPower) * cosTi↩
        ;

    // The default fragColor is the full shading equation
    fragColor = textureColor * illuminants
                   + skyIlluminant * 0.2
                   + specularFactor * shadowFactor;

    vec2 fragPos = vec2( gl_FragCoord.x, gl_FragCoord.y );

    fragPos /= vec2( width, height );

    fragPos = fragPos * 2.0 - 1.0;

    vec4 viewProjectionVector = vec4( fragPos.x, fragPos.y, -1.0, 0.0 );

    viewProjectionVector = toWorldSpace * viewProjectionVector;

    if( layer < 5 && textureColor.rgb == vec3( 0.0, 0.0, 0.0 ) ){

        // color the sky as the sky
        vec3 normedView = normalize(viewProjectionVector.xyz);
        float phi2 = acos( normedView.y );
        float theta2 = atan( normedView.z, normedView.x );

        float r2 = phi2/pi;
        // calculate the texture coordinates of the normView on the sky sphere texture
        float lat = r2 * cos(theta2) * 0.5 + 0.5;
        float long = r2 * sin(theta2) * 0.5 + 0.5;

        fragColor = texture( skySphere, vec2( lat, long ) );

    } else {
        // If we aren't in default mode, try these other layers
    if( layer != 0 ){
      // Layer 1 has no GI, TI, shading, or sky
      if(layer == 1){
        fragColor = textureColor;
      }
      // Layer 2 has the standard shading
      if(layer == 2){
        fragColor = textureColor * shadeFactor + specularFactor;
      }
      // Layer 3 has shadows as well
      if(layer == 3){
        fragColor = (textureColor * shadeFactor + specularFactor) * shadowFactor;
      }
      // Layer 4 has the sky color
      if(layer == 4){
        fragColor = (textureColor * shadeFactor + specularFactor) * shadowFactor
              + skyIlluminant * 0.2;
      }
      // Layer 5 is the GI factor
      if(layer == 5){
        fragColor =  GIColor;
      }
      // Layer 6 is the TI factor
      if(layer == 6){
        fragColor =  TIColor;
      }
      // Layer 7 is the normal layer
      if(layer == 7){
```

```
        fragColor = vec4(normalColor.xyz * 0.5 + 0.5, 1.0);
      }
      // Layer 8 is the position layer
      if(layer == 8){
        fragColor = vec4(volumePos.xyz, 1.0);
      }
      // Layer 9 has some transmittence, floppiness, and shadow as different RGB values
      if(layer == 9){
        float floppiness = texture(albedo, texCoord).a;
        fragColor = vec4( transmitance, floppiness, shadowFactor, 1.0);
      }
    }
    }

}
```