LIGHTWAVE: A REAL-TIME APPROACH OF ESTIMATING INDIRECT ILLUMINATION USING WAVES OF LIGHT ON THE GPU

A Thesis

Presented to the

Faculty of

California State University, Fullerton

---

in Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer Science

By

Michael Robertson

Approved by:

Dr. Michael Shafae, Committee Chair Department of Computer Science	Date
Dr. Kevin Wortman, Committee Member Department of Computer Science	Date
Dr. Bin Cong, Committee Member Department of Computer Science	Date

ABSTRACT

With the growth of computers and technology, so to has grown the desire to accurately recreate our world using computer graphics. However, our world is very complex and in many ways beyond our comprehension. Therefore, in order to perform this task, we must consider multiple disciplines and areas of research including physics, mathematics, optics, geology, and many more to at the very least approximate the world around us. The applications of being able to do this are plentiful as well, including the use of graphics in entertainment such as movies and games, in science such as weather forecasts and simulations, in medicine with body scans, or used in architecture, design, and many other areas. In order to recreate the world around us, an important task is to accurately recreate the way light travels and affects the objects we see. Rendering lighting has been a heavily researched area since the 1970's and has gotten more sophisticated over the years. Until recent developments in technology, realistic lighting of scenes has only been achievable offline taking seconds to hours or more to create a single image, however, due to advances in graphics technology, realistic lighting can be done in real-time. To achieve real-time rendering, we must make trade offs between scientific accuracy and performance, but as will be discussed later, scientific accuracy may not be necessary after all.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

BACKGROUND

1.1 LIGHT AS A PARTICLE AND A WAVE

Before discussing current topics of research in the field of realistic light rendering in computer graphics, we first need a basic understanding of optics, a branch of physics that studies the behavior and properties of light. Prior to the nineteenth century, light was regarded as a stream of particles and with this particle theory of light in mind, certain light phenomena such as refraction and reflection could be explained. However in 1801, the first evidence of light acting as a wave was found when light rays were found to interfere with one another. Later, in 1873, light was compared to a form of a high-frequency electromagnetic wave. In 1905, Einstein proposed a theory that explained the photoelectric effect (ejection of electrons from a metal surface when hit by light) that used the concept of quantization and stated that the energy of light waves are present in particles called photons which are quantized. (Serway and Jewett 2004) Therefore, light can have the behavior of a wave and a particle. These findings are important to consider in our research to accurately depict light as it travels throughout a scene. As will be discussed in the related work section, many lighting techniques use the idea that light consists of particles in order to simulate proper lighting conditions. Some even use the term photons such as the lighting technique photon mapping. This paper, however, will try to explore a hybrid approach that considers light as both a particle and as a wave.

1.2 TERMINOLOGY

Lastly, before jumping into the related work section, we must cover some terminology used in current papers covering lighting in computer graphics. Lighting or illumination, is broken down into different components. Direct illumination is the rendering of a scene as it would appear with the light rays from the light sources terminating at the first surface it hits. Scenes made with this type of illumination are high performance but low realism due to attributes such as hard shadows, no reflection or refraction, and the fact that only surfaces in view of the light source will be illuminated with everything else black. In order to add realism, indirect illumination needs to be added to our prior directly illuminated scene. Indirect illumination considers what happens to the light after it comes into contact with a surface. Indirect illumination takes into account the reflection and refraction of light and produces realistic scenes depending on the extent of indirect illumination captured. Indirect illumination produces the illumination seen on surfaces that are not directly viewable by the light source such as behind another object. The performance of indirect illumination varies greatly due to the extent of indirect illumination captured. Noting that indirect illumination can be recursively calculated to infinity, indirect illumination can be expressed as a Neumann series and would be calculated by calculating the sum of light incident on a surface due to the reflection of light n-times all the way up to infinity. This fact will be explored further in the related works section, but the main idea is that the performance of calculating the indirect illumination depends on how large n is along with other factors discussed later, but is typically much slower than calculating direct illumination. Once we have direct illumination and indirect illumination calculated, we can say we have calculated the global illumination of the scene. This paper's technique will try to exploit the high performance of direct illumination by trying to approximate the indirect illumination of a single light source by using the direct illumination of multiple virtual light sources.

CHAPTER 2

PREVIOUS WORK

2.1 REAL-TIME VERSUS OFFLINE RENDERING TECHNIQUES

Rendering techniques can be broken down into two distinct categories: real-time and offline. Offline rendering techniques require anywhere from seconds to many hours to render a single image. Current offline rendering techniques are able to render a ultra-realistic image that takes into account many light sources as well as many types of light principles such as reflections, refraction, sub-surface light scatter, and more in very complicated scenes consisting of millions of triangles and at fairly high resolutions. Real-time rendering techniques render images at a fast enough rate to support multiple frames a second and vary greatly in approach. These techniques can be broken down into dynamic and static. Dynamic scenes allow a user to interact with the scene and actively change the scene such as moving the geometry, the camera, or the light source whereas static scenes do not. As opposed to offline techniques, real-time techniques often have to make sacrifices when rendering the scene and therefore can't be as scientifically accurate as offline techniques. Regardless of whether the technique is offline or real-time, the algorithm can trace it's roots back to a single equation, *The Rendering Equation.*

2.2 THE RENDERING EQUATION

When discussing light transport in computer graphics, the most significant paper is *The Rendering Equation* (Kajiya 1986). In it Kajiya presents an equation that generalizes most rendering algorithms. Such a statement can be confirmed by the fact that all rendering equations try to recreate the scattering of light off of different types of surfaces and materials. The rendering equation is an integral that is adapted from the study of radiative heat transfer for use in computer graphics with an aim at balancing the energy flow between surfaces. The equation, however, is still an approximation because it does not take into account diffraction and it assumes that the space between objects, such as air, is of homogeneous refractive index meaning that light won't refract due to particles in the air.

$$I(x,x') = g(x,x')[\epsilon(x,x') + \int_S \rho(x,x',x'')I(x',x'')dx''] \tag{2.1}$$

The rendering equation is broken down into 4 parts. First, $I(x,x')$ is the intensity of light or energy of radiation passing from point x' to point x measured in energy of radiation per unit time per unit area. Second, the geometry term, $g(x,x')$, indicates the occlusion of objects by other objects. This term is either 0, if x and x' are not visible from one another, or $1/r^2$ where r is the distance between x and x' if x and x' are visible from one another. Third, the emittance term, $\epsilon(x,x')$, measures the energy emitted from point x' that reaches point x. Lastly, the scattering term, $\rho(x,x',x'')$, is the intensity of energy scattered by a surface point x' that originated from point x'' and then ends at point x. As mentioned in the previous section, illumination can be calculated using the Neumann series. A Neumann series is a mathematical series of the form:

$$\sum_{k=0}^{\infty} T^k \tag{2.2}$$

where T is an operator and therefore T^k is a notation for k consecutive operations of operator T.

Furthermore, the rendering equation can also be approximated using the Neumann series. This is done by rewriting the rendering equation above (2.1) as:

$$I = g\epsilon + gMI \tag{2.3}$$

where M is the linear operator given by the integral in the rendering equation. Next, we rewrite equation 2.3 as:

$$(1 - gM)I = g\epsilon \tag{2.4}$$

so that we can invert it to get:

$$I = g\epsilon + gMg\epsilon + gMgMg\epsilon + g(Mg)^3\epsilon + ... \tag{2.5}$$

Equation 2.5 is a Neumann series of the form:

$$I = g\epsilon \sum_{k=0}^{\infty} (Mg)^k \tag{2.6}$$

Equation 2.6 indicates that the rendering equation (equation 2.1) is the final intensity of radiation transfer as a sum of a direct term, a once scattered term, a twice scattered term, and so on. Therefore, as mentioned in the previous section, indirect illumination can be calculated by summing light incident on a surface due to the reflection of light n-times all the way up to infinity. The more scattered terms we include in the calculation, the better the approximation will be but with worse performance. Therefore, in real-time applications, this needs to be avoided.

Next, we show how the rendering equation can be seen has a generalization of most rendering algorithms, but first we must cover some other rendering equations. A good place to start is with offline rendering techniques.

2.3 OFFLINE RENDERING TECHNIQUES

Key examples of offline rendering techniques are ray tracing, radiosity, and photon maps. We begin with ray tracing.

2.3.1 RAY TRACING

Ray tracing is a technique for rendering an image of a three-dimensional scene by casting rays from a camera positioned somewhere in the scene. These rays are shot into the scene and register the first surface it hits. From this surface point, additional rays go to each of the light sources to determine occlusion from the light sources as well as to other surfaces to calculate reflections. These rays can also be used to calculate other lighting phenomena such as refractions. The rays from the camera can be cast into the scene using different sampling patterns and techniques such as 1 per pixel or many per pixel. Also, the rays can be cast through the center of each pixel or through the use of stochastic sampling can be cast through non-uniformly spaced locations in each pixel to avoid aliasing artifacts or jaggies. Ray tracing is able to recreate ultra-realistic scenes but at a high cost. Examples of ray tracing techniques include (Whitted 1980), (Cook 1986), and (Ward, Rubinstein, and Clear 1988). With adaptations to ray tracing techniques and advances in technology, there now exist some interactive ray-tracing techniques mentioned in section 2.4.

Ray tracing can also be related to the rendering equation. (Whitted 1980) describes a new approximation for ray tracing by rewriting the Phong illumination model in order to improve the quality of specular reflections. The Phong illumination model is a way of calculating lighting on a surface through the combination of three components: ambient, diffuse, and specular. Diffuse is the reflection of light from rough surfaces, specular is the reflection of light on shiny surfaces, and the ambient component accounts for the amount of light that is scattered throughout the scene. The ambient term is most similar to indirect lighting, but is a user-specified constant amount distributed uniformly throughout the scene to avoid any actual calculations. The improved model from (Whitted 1980) is written:

$$I = I_a + k_d \sum_{j=1}^{j=ls} (\bar{N} \cdot \bar{L}_j) + k_s S + k_t T \tag{2.7}$$

where S is the intensity of light incident from the specular reflection direction, k_t is the transmission coefficient, and T is the intensity of light from the transmitted light direction. k_s and k_t are coefficients that are to be used to try to accurately model the Fresnel reflection law. Equation 2.7 is in the form of equation 2.5 from (Kajiya 1986) with Mo as a scattering model which is the sum of the reflection and refraction as well as a cosine term that is the diffuse component. Also, the term $g\epsilon o$ consists of shadows with point radiators and the ambient term can be approximated by the ϵ term. Lastly, M is approximated by summing over all the light sources rather than using integration.

2.3.2 RADIOSITY

Radiosity is a type of rendering technique that was adapted for use in computer graphics from thermal engineering techniques. The method is based on the fundamental Law of Conservation of Energy within a closed area. This law states that energy can neither be created nor destroyed and that the energy in this closed area is to remain constant over time. It provides a global solution for the intensity of light incident on each surface by solving a system of linear equations that describes the transfer of energy between each surface in the scene. Examples of radiosity are seen in (Immel, Cohen, and Greenberg 1986) and (Goral, Torrance, Greenberg, and Battaile 1984).

Radiosity is a natural extension from the rendering equation (equation 2.1) since its focus is on balancing the flow of energy. The only difference is that radiosity makes assumptions about the reflectance characteristics of the surface material. The radiosity in the scene is found by taking the hemispherical integral of the energy leaving the surface called flux which can be found using the following equation from (Goral, Torrance, Greenberg, and Battaile 1984):

$$B_j = E_j + \rho_j H_j \tag{2.8}$$

where B_j is the rate of energy leaving the surface j measured in energy per unit time per unit area, E_j is the rate of direct energy emission, ρ_j is the reflectivity of surface j, and H_j is the incident radiant energy arriving at surface j per unit time per unit area. Equation 2.8 can be derived using our rendering equation (equation 2.1) (Kajiya 1986) by integrating over all surfaces in the scene to calculate the hemispherical quantities, calculating the contribution of the emittance and reflectance terms by checking for occlusions, and by using those calculations the rendering equation becomes:

$$dB(x') = \pi[\epsilon_0 + \rho_0 H(x')]dx' \tag{2.9}$$

where ϵ_0 is the hemispherical emittance of the surface element dx', ρ_0 comes from the reflectance term, and H is the hemispherical incident energy per unit time per unit area. This adaptation of the rendering equation (equation 2.9) is the same as the radiosity equation shown above (equation 2.8).

2.3.3 PHOTON MAPS

Photon maps originally introduced in (Jensen 1996) is a two pass global illumination method. As mentioned in the Background section, Einstein coined the term photon to be the particles present in the energy of light waves. In the method of photon mapping, the term photon is used in a similar context. The first pass of the method consists of making two photon maps by emitting packets of energy called photons from the light sources and storing where they hit surfaces in the scene. The second pass of the method calls for the use of a distribution ray tracer that is optimized using the data gathered in the photon maps. Photon maps are able to render complex lighting principles such as caustics.

Photon maps are an extension to the rendering equation (equation 2.1) as well. During the second pass of the method, the scene is rendered by calculating the radiance by tracing a ray from the eye through the pixel and into the scene using ray tracing, and the radiance is computed at the first surface that the ray hits. The surface radiance leaving the point of intersection x in some direction is computed using the equation from (Jensen 1996):

$$L_s(x, \psi_r = L_e(x, \psi_r + \int_{\Omega} (f_r(x, \psi_i; \psi_r) L_i(x, \psi_i) \cos(\theta_i) d\omega_i) \tag{2.10}$$

where L_e is the radiance emitted by the surface, L_i is the incoming radiance in the direction ψ_i, and f_r is the BRDF or bidirectional reflectance distribution function, which is a four-dimensional function that describes how light is reflected at a surface point. Lastly, Ω is the sphere of incoming directions. This can be broken down into a sum of four components:

$$\begin{aligned} L_r = &\int_\Omega (f_r L_{i,l} \cos(\theta_i) d\omega_i) + \int_\Omega (f_{r,s}(L_{i,c} + L_{i,d}) \cos(\theta_i) d\omega_i) \\ &+ \int_\Omega (f_{r,d} L_{i,c} \cos(\theta_i) d\omega_i) + \int_\Omega (f_{r,d} L_{i,d} \cos(\theta_i) d\omega_i) \end{aligned} \tag{2.11}$$

where the first term of equation 2.11 is the contribution by direct illumination, the second term is the contribution by specular reflection, the third term is the contribution by caustics, and the fourth term is the contribution by soft indirect illumination. Both equations 2.10 and 2.11 are direct adaptations from the rendering equation in (Kajiya 1986) (equation 2.1).

2.3.4 OTHER OFFLINE RENDERING TECHNIQUES

Many recent offline techniques have been influenced by real-time techniques most notably the idea of using virtual point lights or VPL's as introduced in *Instant Radiosity* (Keller 1997). This technique will be discussed in detail in the real-time rendering techniques section, but the main idea is that we can render indirect light through the use of a set of VPL's where we accumulate the contributions of each of these lights in multiple rendering passes. Examples of using this technique are used for rendering illumination from area lights, high dynamic range (HDR) environment maps or sun/sky models, single/multiple subsurface light scattering in participating media, and most importantly for our purposes indirect illumination. For offline purposes, techniques typically call for the use of upwards of millions of VPL's to achieve higher quality renderings, however, many techniques attempt to mitigate the cost of using such a large number of lights.

VIRTUAL POINT LIGHTS:

In *Lightcuts: A Scalable Approach to Illumination* (Walter, Fernandez, Arbree, Bala, Donikian, and Greenberg 2005), it is discussed that when using many lights, as in the VPL method, the cost to render the scene scales linearly with the number of lights used. This limits the number of VPL's we can use without serious performance impact. Therefore, the Lightcuts method is introduced as a way to reduce the rendering cost of using VPL methods by making it strongly sub-linear with the number of lights used without noticeable impact on quality. Using this method, hundreds of thousands of point lights can be accurately rendered using only a few hundred shadow rays. Lightcuts does this by clustering a group of VPL's together to form a single brighter light thereby reducing the cost of rendering the group of lights present in the cluster group to a single light. This is done by using a global light tree which is a binary tree that has individual VPL's as the leaves and the interior nodes are the light clusters that contain the individual light below it in the tree. A cut of this tree then is a set of nodes such that every path from the root of the tree to a leaf will contain exactly one node from the cut and will represent a valid clustering of the lights. This approach was further advanced in *Multidimensional Lightcuts* (Walter, Arbree, Bala, and Greenberg 2006) for use with visual effects such as motion blur, participating media, depth of field, and spatial anti-aliasing in complex scenes and used again in *Bidirectional Lightcuts* (Walter, Khungurn, and Bala 2012) to support low noise rendering of complicated scenes with glossy surfaces, subsurface BSSRDF's, and anisotropic volumetric models.

<u>MATRIX SAMPLING</u>:

Another adaption of the VPL method in offline rendering techniques is with the use of mathematical matrix sampling to reduce the rendering cost. In *Matrix Row-*

Column Sampling for the Many-Light Problem (Hašan, Pellacini, and Bala 2007), it is shown that many point lights can be seen as a large matrix of sample-light interactions. The final image is then the sum of all of the columns of the matrix. Therefore, by sampling this matrix and using a small number of the rows and columns, we can approximate the final image at a fraction of the rendering cost. The matrix used in this technique is comprised of the columns representing all of the surface points to be lit by each light and each row being comprised of all of the lights in the scene. Then it can often be shown that this matrix is of low rank meaning that the row and/or columns of the matrix can be linearly combined to form a smaller matrix that would approximate the original larger matrix. Therefore, this method approaches the problem in a similar way as Lightcuts, but instead of using a tree, we use a matrix. This method is comprised of four primary steps. First, we sample r randomly selected rows using shadow maps on the GPU. Shadow maps will be discussed in the next section. Next, we partition the reduced columns into clusters on the CPU. Third, we pick a representative from each cluster to be scaled appropriately to account for the entire cluster on the CPU. Last, we accumulate each of these representatives using shadows maps on the GPU. By using this method, the render time of shading m surface points using n VPL's is reduced from $O(mn)$ to $O(m+n)$. Additionally, *LightSlice: Matrix Slice Sampling for the Many-Lights Problem* (Ou and Pellacini 2011) uses a similar approach as in (Hašan, Pellacini, and Bala 2007) and (Walter, Fernandez, Arbree, Bala, Donikian, and Greenberg 2005), but found that neither technique optimally exploits the structure of the matrices in scenes with large environments and complex lighting so some modifications were made to reduce render time and improved overall quality.

SHADOW MAPS:

Shadow maps is a technique typically used to render shadows, but as shown in the real-time rendering techniques section, they have additional uses. They are created in a render pass where the view of the scene is computed from the light source's point of view and the distances from the light to the nearest surface for each pixel is stored in a texture or buffer to be used for comparisons when rendering the scene from the camera's point of view to determine whether a surface point is in shadow (Williams 1978), (Reeves, Salesin, and Cook 1987). Further discussion on shadow maps will be done in the real-time rendering techniques section as well as in the implementation section.

VIRTUAL RAY LIGHTS:

Lastly, (Novák, Nowrouzezahrai, Dachsbacher, and Jarosz 2012) describes a technique to render scenes with single/multiple light scattering in participating media in *Virtual Ray Lights for Rendering Scenes with Participating Media.* The technique modifies the idea of using VPL's by instead using virtual ray lights or VRL's inside the participating media. So instead of evaluating each VPL at discrete locations, we calculate the contribution of each VRL with an efficient Monte Carlo sampling technique. Monte Carlo sampling involves the use of randomly selecting a point based off a probability distribution and is often used for sampling in a wide variety of techniques. The reason for using VRL's over VPL's is that VPL's can be negatively affected by singularities in the scene. These singularities can cause artifacts in the scene such as spikes of high intensity which can be eliminated through the use of clamping or blurring. But by using a VRL, we compute the contribution of the light by distributing the energy over a line segment reducing singularities. In this case, instead of having spikes of high intensity, we have a uniform noise distributed evenly over the image which would be more pleasing to the eye.

2.4 REAL-TIME RENDERING TECHNIQUES

With the advancement of technology and the expanded use of the GPU, many offline rendering techniques have been adapted to work on the GPU to run in real-time usually by making trade offs and approximations. This includes interactive ray tracing (Wald, Kollig, Benthin, Keller, and Slusallek 2002), approximate ray tracing on the GPU (Szirmay-kalos, Aszdi, Laznyi, and Premecz 2005), image space photon mapping (McGuire and Luebke 2009), and *Instant Radiosity* (Keller 1997). The most significant advancement coming from the idea of virtual point lights as in *Instant Radiosity*.

2.4.1 *INSTANT RADIOSITY*

As discussed in prior sections, *Instant Radiosity* introduces the technique of rendering indirect illumination approximated with the use of virtual points lights where each individual light acts independently as a producer of indirect illumination and behaves like a normal point light source. In the original technique, these VPL's are generated using a quasi-random walk technique created at the hit points of the photons as they are traced into the scene from the primary light source. The contributions of these VPL's are then summed through the use of multiple rendering passes. This implementation had a few limitations such as requiring many rendering passes to support dynamic objects or lights as well as being limited to simple environments due to the high cost of computing shadows for each of the VPL's. These shadows had to be computed by using shadow volumes or shadow maps and were the bottleneck of the technique. There were also issues with low sampling rates which would result in weak singularities when the VPL got increasingly close to the illuminated surface point and each VPL has a high influence or contribution on the overall color of the image. Also, temporal flickering can be seen if there are not enough VPL's being used. Despite these issues, real-time performance was attainable in 1997 through the use of this technique.

2.4.2 ADAPTATIONS OF *INSTANT RADIOSITY*

In addition to the techniques already mentioned in section 2.3.4 which adapted the idea of using VPL's for the offline rendering of multiple types of lighting phenomena, many real-time techniques also adapted the use of VPL's.

SHADOW MAP ALTERATIONS:

As teased in section 2.3.4, in many real-time rendering techniques shadow maps have been expanded for use beyond just creating shadows. In *Translucent Shadow Maps* (Dachsbacher and Stamminger 2003) extended the binary shadow map look-up to a shadow map filter to assist in the implementation of real-time sub-surface scattering. Translucent Shadow Maps extend shadow maps by having additional information stored such as depth and incident light information. Therefore, each pixel in the Translucent Shadow Map stores the 3D position of the surface sample, the irradiance entering the object at that sample, and the surface normal. Although this implementation was for sub-surface scattering, it would lead to a development in rendering indirect illumination as well.

From this idea of extending shadow maps came further developments in *Reflective Shadow Maps* or RSM (Dachsbacher and Stamminger 2005). This technique extended a shadow map such that each pixel in the shadow map would be thought of as an indirect light source. The reflective shadow map then stored information such as depth value, world space position, normal, and reflected radiant flux. With all this information available, we can then approximately calculate the indirect irradiance at a surface point by summing the illumination due to all pixel lights as seen in equations 2.12 and 2.13.

$$E_p(x,n) = \Phi_p max(0, n_p \cdot (x - x_p)) max(0, n_p \cdot (x_p - x)) \div \|x - x_p\|^4 \tag{2.12}$$

$$E(x,n) = \sum_{pixels p} E_p(x,n) \tag{2.13}$$

where x is the surface point, n is the normal at x, n_p and x_p is the pixel light normal and position, and Φ_p is the reflected radiant flux of the visible surface point. This technique had the same singularity issue near the boundary of two adjoining walls. Other limitations includes ignoring occlusion and visibility for the indirect light sources as well as having to restrict the number of VPL's to around 400 meaning that sampling had to be done to chose the 400 best VPL's. Also, screen-space interpolation had to be performed by computing the indirect illumination using a low-resolution shadow map and interpolating using multiple low-res samples. With these restrictions in play, this technique renders approximate indirect illumination for dynamic scenes. Reflective shadow maps were then used in many other real-time rendering techniques that followed. This included *Splatting Indirect Illumination* (Dachsbacher and Stamminger 2006) which rendered the VPL's contributions through a splatting technique in a deferred shading process which reduced the effect of scene complexity on the rendering time. It also allowed for efficient rendering of caustics and reduced scene artifacts.

Additionally, RSM's are used in other applications such as indirect illumination for area light sources as in *Direct Illumination from Dynamic Area Lights With Visibility* (Nichols, Penmatsa, and Wyman 2010). Here we use RSM's to generate VPL's, but add visibility to the technique by computing occlusion by marching rays towards each VPL through a voxel buffer. The technique then checks the visibility for each VPL and adds in its illumination provided the VPL is visible.

Imperfect Shadow Maps for Efficient Computation of Indirect Illumination (Ritschel, Grosch, Kim, Seidel, Dachsbacher, and Kautz 2008) altered shadow maps for their purposes by making them imperfect shadow maps or ISM's. These ISM's were low resolution shadow maps with a simplified point-representation of the scene such that some of the depth values could be incorrect. This simplified scene is done by approximating the 3D scene by a set of points with a near uniform density which is then used to create an ISM. The point-based representation of the geometry is used to allow the creation of hundreds of ISM's in parallel in a single pass to support dynamic scenes. These hundreds of ISM's are stored in a single large texture and used as approximate visibility for the hundreds or thousands of VPL's used for indirect illumination. These VPL's are generated through the use of RSM's. ISM's can also be built on top of reflective shadow maps to create imperfect reflective shadow maps to allow for multiple bounces of light. With the use of ISM's, indirect illumination scales well with an increase in scene complexity, however, it has no effect on the increase of rendering costs for the direct illumination component. Limitations include the traditional VPL method issues as discussed in section 2.4.1 as well as that indirect shadows cannot be generated for smaller geometry. Although, this technique calls for inaccurate visibility, it is an upgrade to the strategy of ignoring visibility as in RSM's and it results in a very minimal impact on the final scene but allows for good performance with real-time global illumination.

SCREEN SPACE ALGORITHMS:

Additional VPL variants include *Hierarchical Image-Space Radiosity for Interactive Global Illumination* (Nichols, Shopf, and Wyman 2009) where instant radiosity is combined with multiresolution techniques such as (Nichols and Wyman 2009). This is done by accumulating indirect illumination at a variety of different resolutions in image space depending upon singularities. When no singularities are nearby, a lower resolution can be used whereas when there are singularities, a higher resolution should be used to avoid artifacts. Similar to many of the previous methods mentioned, this techniques ignores visibility for indirect light.

Lastly, VPL approaches can introduce bias. When a VPL is close to a surface, it will introduce a singularity that appears as a high intensity peak in the image. Most techniques solve this by clamping a VPL's contribution to a surface if it is nearby, however, this removes energy from the system and therefore introduces a bias. This bias results in darkening of the image near singularity areas such as wall boundaries and edges of objects. In order to prevent this, screen-space bias compensation (Novák, Engelhardt, and Dachsbacher 2011) was introduced as a post-processing step to recover the clamped energy. This step involves applying a residual operator to the direct illumination and clamped indirect illumination as computed through the use of the rendering equation (equation 2.1) and a new residual operator.

2.4.3 SPHERICAL HARMONICS AND LATTICE-BASED METHODS

Spherical harmonics are the angular portion of a set of solutions to Laplace's equation represented in a system of spherical coordinates. In (Nijasure, Pattanaik, and Goel 2005) they are used as a way to represent the incident light at each sample point on a regular grid comprised of both direct and indirect light that arrives from all surface points visible to that sample point in a compact way. This compact representation is comprised of a small number of coefficients that are computed through the use of a spherical harmonics transformation. The incident radiance is approximated through the use of a cube map at each grid point, stored as spherical harmonics coefficients, and then the indirect illumination is calculated by interpolating the radiance at the nearest grid points. This techniques allows for indirect occlusion through the use of shadow cube maps, but is expensive for complicated dynamic scenes. (Papaioannou 2011) combined the grid-based radiance caching of (Nijasure, Pattanaik, and Goel 2005) and the use of spherical harmonics along with the reflective shadow maps discussed in the above section. Instead of using cube maps to sample the visibility, this technique sampled the RSM to increase performance. (Kaplanyan and Dachsbacher 2010) used a volume-based method with lattices and spherical harmonics to represent the spatial and angular distribution of light in the scene where VPL's from a RSM are inserted into a volume texture. This light propagation volume allows for iterative propagation of energy among voxels as well as accounting for fuzzy occlusion by storing depth information and RSM's in a separate occlusion volume to compute indirect shadows that is limited to surfaces larger than the grid size. These light propagation volumes allow for the use of many more VPL's than other methods due to not calculating the contribution of each of the VPL's individually.

The occlusion calculations are also view-dependent which leads to problems such as popping artifacts.

CHAPTER 3

IMPLEMENTATION

3.1 IMPLEMENTATION GOALS

The goal of this implementation is to propose a real-time rendering technique that will render global illumination on the GPU through the use of OpenGL and GLSL. We will use the idea of virtual point lights from *Instant Radiosity* (Keller 1997) to render the indirect illumination. Shadow maps will be used as in the spirit of (Williams 1978) and (Reeves, Salesin, and Cook 1987) to render shadows for direct illumination. The VPL's will be structured outward from a primary light source in a lattice. These VPL's will be structured in specific distances and angles around the primary light source in hemispheres. The goal is to simulate multiple waves of VPL's flowing outward from the primary light source in order to simulate the light transport as wave-like in order to simulate indirect illumination using only the direct illumination of many VPL's while ignoring the bouncing of light among surfaces. This way, we simulate light as a particle by using VPL's and as a wave in the way the VPL's will be organized. This will allow the wrapping of light around objects in order to illuminate surfaces that may not be directly visible from the light source. An additional goal of this implementation is to have the resulting technique be scalable. The organization of the VPL's will allow us to use more or less VPL's depending on quality and performance desires. Specific implementation details will follow in section 3.3. Lastly, in support of this technique and its ignorance of calculating indirect illumination and instead use a direct illumination approximation of indirect illumination, we discuss the importance of scientifically correct rendering versus visually appealing rendering.

3.2 SCIENTIFICALLY CORRECT VERSUS VISUALLY APPEALING

All of the previous work mentioned in section 2 used approximations when computing indirect illumination to varying degrees. This is because as mentioned in section 2.2, the rendering equation (equation 2.1) is calculated using a Neumann series and in order to get an exact calculation, an infinite number of iterations would need to be carried out to fully calculate the result. These iterations account for the infinite number of indirect light bounces that take place. Since this is unfeasible, approximations are made based off our demands for performance or realism. For real-time applications, we must err on the side of performance, but since the presence of indirect light has been shown to be perceptually important we can't just simply ignore it either (Stokes, Ferwerda, Walter, and Greenberg 2004). Also, according to (Stokes, Ferwerda, Walter, and Greenberg 2004), diffuse indirect illumination is perceptually the most important component to consider. However, as the notion of realism is merely whether the rendering is visually pleasing to the human eye, major approximations can be made to increase performance provided the result is visually pleasing. As such, we ignore the bouncing of light and try to approximate it with just cheap direct lighting that will be able to reach surfaces that otherwise only indirect light would reach in order to simulate the feeling of indirect light. Proof of the idea that visually pleasing is sufficient is provided next.

In *Perceptual Influence of Approximate Visibility in Indirect Illumination* (Yu, Cox, Kim, Ritschel, Grosch, Dachsbacher, and Kautz 2009) it is proven that accurate visibility is not required and that certain approximations may be introduced. When a person sees lighting rendered in a scene or in real-life, the most revealing or obvious lighting is the direct illumination. When it is approximated or incorrect, it is apparent right away due to the high-frequency nature of direct lighting. Indirect lighting, however is usually low-frequency and therefore has smooth graduations in changes of intensity. This allows for more leeway for approximations and interpolations. The most expensive calculation being visibility determination for indirect illumination. To research the effects of this on the viewer's perception of the rendering, (Yu, Cox, Kim, Ritschel, Grosch, Dachsbacher, and Kautz 2009) conducted a psychophysical study involving two different experiments. The first experiment involved estimating the difference between the approximation rendering and the reference rendering based off of a number scale ranging from one to five corresponding to not similar and extremely similar respectively. This experiment had 14 participants. The second experiment involved ranking 10 approximated renderings and 1 reference rendering in order of least realistic to most realistic. This experiment had 18 participants. Over half of the participants in both experiments were computer scientists with a background in imaging. The approximated renderings consisted of four different categories of indirect visibility. These were imperfect visibility (Ritschel, Grosch, Kim, Seidel, Dachsbacher, and Kautz 2008), ambient occlusion (Zhukov, Iones, and Kronin 1998), directional ambient occlusion (Sloan, Govindaraju, Nowrouzezahrai, and Snyder 2007) (Ritschel, Grosch, and Seidel 2009), and no visibility ie. visibility for indirect illumination is completely neglected (Dachsbacher and Stamminger 2005) and (Dachsbacher and Stamminger 2006). All the renderings were prerendered 5-second video sequences and consisted of four different test scenes.

The results showed that the imperfect visibility approximations were very much similar or moderately similar to the reference video in all scenes. The ambient occlusion approximations were also very much similar or moderately similar to the reference video in most scenes. Directional ambient occlusion was considered very much similar or moderately similar to the reference video in most scenes. Lastly, the no visibility approximations were considered moderately similar to the reference video. Also, there was found to be a connection between the amount of indirect illumination and the perceived similarity to the reference. In terms of the perceived realism of the renderings, the reference video as well as the imperfect visibility, ambient occlusion, and directional ambient occlusion approximations were all considered equally realistic leaving just the no visibility approximations as less realistic than the rest. Overall, the imperfect visibility approximations as in (Ritschel, Grosch, Kim, Seidel, Dachsbacher, and Kautz 2008) were considered the most realistic of the approximations.

This study shows that visibility approximations can be made when rendering indirect illumination while remaining perceptually similar or as realistic as a reference rendering. This provides validity to the approximations already done as well as support further approximations to come. Having the imperfect visibility methods declared as the most realistic leads to the assumption that randomly corrupted visibility is more pleasing than incorrect visibility. Lastly, it is shown that although all the approximations had noticeable differences in the renderings, many of them were still thought of as being realistic.

3.3 IMPLEMENTATION DETAILS

As stated in section 3.1, the goal of this implementation is to propose a real-time rendering technique that will render global illumination on the GPU through the use of OpenGL and GLSL. The primary focus is to render indirect illumination with the findings on scientifically accurate versus perceptually pleasing in mind. As such, we will ignore the indirect bouncing of light (the infinite series portion of the rendering equation), but try to simulate it using the VPL technique as introduced in (Keller 1997). We will accomplish this simulation through the method of VPL placement and handling. The VPL's will be structured in hemispheres around the primary light source. The default original implementation will include VPL's at every 5 degrees around the y-axis of the light source resulting in 360 degrees / 5 degrees per ray = 72 rays of VPL's forming a circle around the light source. See figure 3.1.

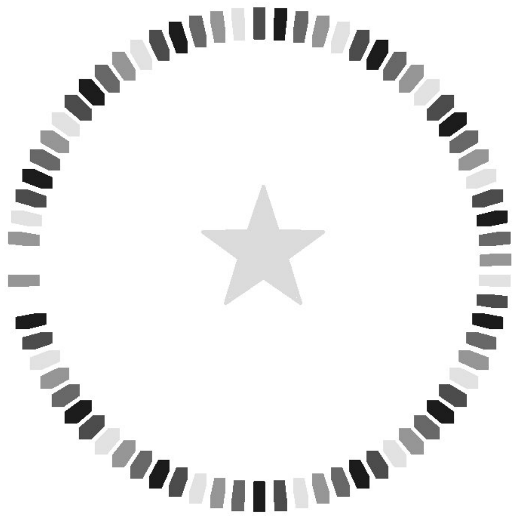

Figure 3.1 The light source (star) is facing the reader (ie. direction/normal pointing out of the paper.

Next, the VPL's will be structured at every 5 degrees around the z-axis of the light source resulting in 90degrees/5degrees per ray = 18 rays of VPL's for every one of the 72 rays around the z-axis. This totals in 1296 rays plus the vertical ray along the z-axis resulting in 1297 rays to form a hemisphere around the light. See figures 3.2 and 3.3.

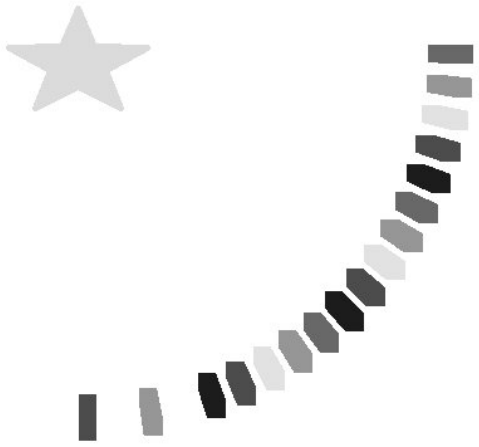
Figure 3.2 The light source (star) is facing downwards. Angles are drawn to scale.

Next, we will have multiple VPL's on each of these rays. This implementation will have 5 VPL's per ray. This results in having 5 stacked hemispheres of increasing radii and a total of 6485 VPL's. See figure 3.4.

The distance from the primary light to each of the VPL's on each ray will be calculated based off of the depth of the scene. The distance between VPL's on each ray will be logarithmic as shown below and the attenuation will be exponential. See figure 3.5.

The goal of this structure is to simulate multiple waves of VPL's flowing outward from the primary light source in order to simulate the light transport as wave-like in order to simulate indirect illumination using only the direct illumination of these VPL's while ignoring the bouncing of light among surfaces. An additional goal of this implementation is to have the resulting technique be scalable. This organization of the VPL's will allow us to use more or less VPL's depending on quality and performance desires. As such, we can increase or decrease the corresponding angles

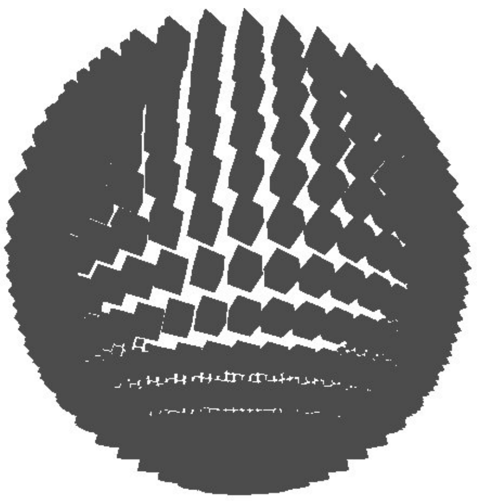

Figure 3.3 Figures 3.1 and 3.2 together form a hemisphere. View is looking slightly up at the hemisphere.

to reduce or increase the number of VPL's used as well as increase or decrease the number of hemisphere shells used. Also, we can use stochastic sampling in order to turn off or on certain VPL's in order to introduce noise in the implementation as well as decrease the number of VPL's required. As shown in the previous section, noise can be perceptually pleasing provided it is in smooth graduations.

The VPL's contributions will computed as shown in figure 3.6. The VPL's will act similar to directional lights in that the normal of the VPL will be used in computing the radiance that it provides. The major difference, however, is that the VPL will be able to contribute radiance to surfaces that are slightly behind the VPL's normal. Instead of doting the surface normal with the VPL normal and only allowing for contribution when the dot product is between 0 and 1, we will allow the dot product to be between -0.5 and 1. This will mean that the VPL's viewable range is 240 degrees instead of 180 degrees. This further depicts each VPL as wave-like.

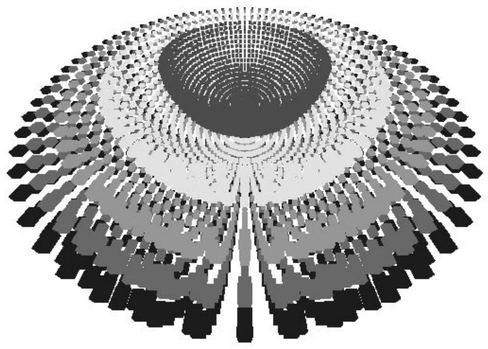

Figure 3.4 5 stacked hemispheres formed by the VPL's around the primary light source. Each hemisphere is a different shade to make distinguishing easier.

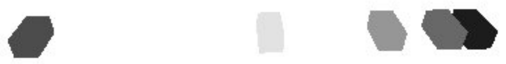

Figure 3.5 Showing the logarithmic distances between the VPL's on each ray. The first box on the left is the first VPL on the ray with successive VPL's going towards the right. Distance shown to scale. (Light source is not shown)

The calculation to account for the expanded dot product range will be discussed along with the code explanations of chapter 4. See figure 3.6.

Occlusion will be handled with a shadow map as in (Williams 1978) and (Reeves, Salesin, and Cook 1987) to render shadows for the direct illumination based solely off of the primary light source. This method is very cheap and provides sufficient shadows for the purposes of this implementation. The indirect occlusion will be handled through the use of regular shadow maps on stochastically sampled VPL's. This implementation will use 20 randomly chosen VPL's to derive indirect shadows. The specific implementation details will be discussed in chapter 4 with code excerpts present.

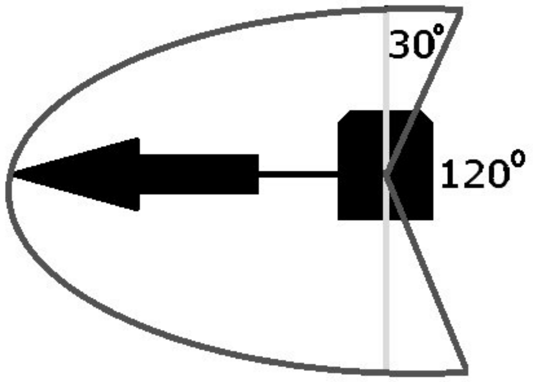

Figure 3.6 Showing the wave-like radiance contribution of a VPL. We add an additional 30 degrees of viewable range on the top and bottom limiting the blind spot of the VPL to 120 degrees rather than 180. The arrow signifies the VPL's direction/normal.

3.3.1 GPU AND CPU FUNCTIONS

This implementation will be designed to use both the CPU and the GPU. The CPU will handle all preliminary tasks such as setting up the window, initializing variables and the shaders, setting up the scene including position for the light source and objects, handling interactive actions including keyboard callbacks, initializing and filling textures, and initializing the VPL's position, direction, and attenuation. The GPU will calculate shading and illumination based off vertices inputted along with the VPL texture and calculate shadows based off the provided shadow maps passed in from the CPU.

3.3.2 OPENGL

OpenGL or Open Graphics Library is a multi-platform API for graphics that will be used for this implementation. The technique is being designed on a machine with OpenGL 4.2.0, however, OpenGL 2.1 and above will be supported. Also, the technique will run on both Windows and Macintosh.

3.3.3 GLSL

GLSL or OpenGL Shading Language is a high-level C-syntax shading language to be used with OpenGL in order to give the developer control over what instructions are executed in the vertex and fragment shaders of the graphics pipeline. In this implementation, all illumination and shading calculations are computed in the vertex and fragment shaders programmed manually through the use of GLSL. The technique is being desgned on a machine with GLSL 4.20 support, but the shaders are written with GLSL 1.20 support.

3.3.4 WORKING ENVIRONMENT

The implementation is being designed and tested on the following hardware and software using the following libraries. Any and all results such as rendered images and fps results will be gathered while running the method on the following.

HARDWARE

- CPU: AMD Athlon 62 X2 Dual Core 5200+ 2.61Ghz
- GPU: Nvidia GeForce GTX 465 (1GB memory)
- Memory: 6.00GB

All other hardware specifications are irrelevant.

SOFTWARE

- OS: Windows 7 64-Bit

- Languages: C++, OpenGL 4.2.0 (2.1 tested/supported), OpenGL Shading Language 4.20 (written using version 1.20)
- Developer Tools: Microsoft Visual Studio 2008

LIBRARIES

The following libraries are used in this implementation and provided in the repository.

- OpenGL Easy Extension Library (GLEE)
- OpenGL Extension Wrangler Library (GLEW)
- OpenGL Utility Toolkit (GLUT)

CHAPTER 4
SAMPLE CODE AND EXPLANATION

4.1 MAIN PROGRAM (C++ CODE)

The main program is written in C++ using OpenGL and is run exclusively on the CPU. It's responsibilities include setting up the window, initializing the shaders, and initializing and updating the textures and variables passed to the shaders. After setting up the window, the first major step for the program is to generate the VPL's. As discussed in section 3.3 and in the associated figures, the default step up will include the use of 6485 VPL's. The VPL's will be structured outward in hemispheres from the primary light source in the direction of the primary light source. Assuming that the direct of the primary light source is pointing downward (negative y-axis), there will be a VPL at every 5 degrees around the y-axis for a total of 72 VPL's (360/5). Then VPL's will be at every 5 degrees around the the z-axis resulting in 18 VPL's per 90 degree angle. With this, our hemisphere is almost complete except for the VPL on the y-axis which is not included in the previous calculations. Therefore, we will have 1297 VPL's per hemisphere and 1297 outward rays. Next, we will chose to have 5 stacked hemispheres flowing outward from the primary light source resulting in 6485 VPL's total.

The locations of these VPL's will be calculated on start-up based off of the location of the primary light source and the direction it is looking at. The VPL's will only be updated whenever the primary light source is moved and at no other times reducing overhead. The VPL's are calculated with an equation similar to below:

$$vpl[x,y,z] = lightPosition[x,y,z] + normal[x,y,z] * (maxDistance - (maxDistance/2^i)) \tag{4.1}$$

Equation 4.1 calculates the position of the VPL by taking the position of the primary light source and moving in the direction of the primary light source by a particular distance. This distance is calculated by using the maximum distance allowable (varies based off the dimensions of the scene) and the distances between each VPL on each outward ray is logarithmic which is achieved using 2^i where i ranges from 1 to the number of hemispheres or 5 as is default with 1 being the innermost hemisphere and 5 being the outermost hemisphere. Similarly, we get the VPL direction from the primary light source direction and we get the attenuation from this exponential equation:

$$vplAttenuation = 0.05 * pow(2.0, i) \tag{4.2}$$

Equation 4.2 results in us getting the following attenuation levels for a VPL in each of the 5 hemispheres: $5\%, 10\%, 20\%, 40\%, 80\%$.

Next, in order for the GPU to have access to the VPL data we have generated above, we store them somehow. This is done by using 2 1D textures. The VPL position data and normal data each receive their own texture. The VPl position data texture is a RGB texture which stores float values of each of the xyz position values. This is done in C++ with OpenGL by the following statement:

```
glTexImage1D( GL_TEXTURE_1D, 0, GL_RGB, numLights, 0, GL_RGB,
        GL_FLOAT, &vplDataPos[0]);
```

where numLights will be 6485 in this case and vplDataPos is the address for the VPL position data. Similarly with the normal data we use a 1D texture but this time with RGBA since we will also store the attenuation on top of the xyz values.

```
glTexImage1D( GL_TEXTURE_1D, 0, GL_RGBA, numLights, 0, GL_RGBA,
        GL_FLOAT, &vplDataNor[0]);
```

One problem arises in that the texture will clamp the values stored between 0 and 1. This is a problem because our VPL data may be negative and will likely be larger than 1 (depending on the dimensions of our scene). Therefore, prior to storing our data, we must encode our data such that all values will be between 0 and 1 and we can then knowingly decode the data once the texture is in the shaders so our original values are preserved. This can be done by the following equation:

$$vpl[data] = (vpl[data]/(4 * maxDistance)) + 0.5 \tag{4.3}$$

Equation 4.3 normalizes the data to be between -0.5 and 0.5 and then adds 0.5 to it to reach the required 0 to 1 range. The normalization is primarily important to the VPL position data since the VPL normals are already between -1 and 1 and the attenuation is already between 0 and 1. Then once in the shader we can decode the data with the following:

$$vpl[data] = (vpl[data] - 0.5) * maxDistance * 4.0; \tag{4.4}$$

Once again, the above steps will only be done at initialization and whenever the primary light source is moved.

The next step is to generate our shadow maps. As discussed in section 2, shadow maps are generated by viewing the scene from the perspective of the light source in question. Then we calculate the distance to the first surface in each pixel to find the depths of the scene. Using this depth, we can compare the value with the depth of other surfaces and determine whether a point lies in shadow. In the default set-up, we use 21 shadow maps. One for the primary light source for direct shadows and 20 for the randomly chosen VPL's for the indirect shadows. In order to do this, we use a single 2D texture array which consists of 21 layers. This is done in OpenGL by using the following line of code:

```
glTexImage3D(GL_TEXTURE_2D_ARRAY, 0, GL_DEPTH_COMPONENT32, SMWidth,
SMHeight, numShadowMaps, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
```

The above line of code uses the depth component since we are only interested in the depth value at each pixel and nothing else with shadow maps. Next, we need the width and height of the shadow map and we declare that we will be storing floats. Next, in order to capture the depth values of the scene we use OpenGL's frame buffer capabilities. We render the scene 21 times from the perspective of each light in question to generate the shadow maps and then use these to determine which points lie in shadow from each light's perspective for the final rendering which the user sees. While generating the shadow maps we are using the fixed function pipeline to perform the rendering from each perspective. Each time we render the scene, we attach the frame buffer to our shadow map texture using the following:

```
glFramebufferTextureLayer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
            shadowMapTexture, 0, i);
```

where *i* corresponds to which layer of the texture to attach it. Next, in order for this shadow map to be useful we need a way to transform any given vertex to a coordinate in the shadow map. We do this by storing the associated modelview and projection matrices used to render the scene each time in a uniform matrix variable that is pass over to the shaders. This way we will be able to take any vertex, transform it using our transformation matrix and find the correct coordinate in the shadow map to compare it's depth. This can be done in OpenGL by:

```
glUniformMatrix4fv(lightMatrix, numShadowMaps, GL_FALSE,
        (GLfloat*)textureMatrix);
```

where lightMatrix corresponds to a OpenGL variable that stores the location of the variable, GL_FALSE tells OpenGL to not transpose the matrix, and textureMatrix is where we are storing our modelview and projection matrices for each shadow map.

These shadow maps will be generated at initialization and for every frame thereafter.

Next, the main thing left for the main program to perform is to render the final scene with the assistance of the shaders, which will be discussed in the following sections.

4.2 VERTEX SHADER CODE

The vertex shader is run exclusively on the GPU and is programmed using GLSL or OpenGL Shading Language, which is very similar to C. The vertex shader is ran for each individual vertex from our scene. The primary tasks performed in the vertex shader for our program is transforming the input vertices, reading some of the textures and all of the uniform variables from the CPU, calculating the per-vertex variables for direct lighting, calculating the VPL color contributions to indirect illumination, and calculating the corresponding shadow map coordinates.

Transforming each input vertex is an easy one-line statement:

$$gl\_Position = gl\_ModelViewProjectionMatrix * gl\_Vertex; \tag{4.5}$$

Each of the above variables used in the line above are preallocated variables used in GLSL.

Next, we calculate the per-vertex variables for the direct lighting. For our diffuse shading, we need the light direction which can be calculated using the following line:

$$lightDir = vec3(masterLightPosition) - gl\_Vertex.xyz; \tag{4.6}$$

For the specular lighting used in our direct illumination, we need the direction of the light reflected as well as the viewing direction of our camera.

$$lightDirRef = reflect(-lightDir, gl\_Normal); \tag{4.7}$$

$$camDir = vec3(cameraPosition) - gl\_Vertex.xyz; \tag{4.8}$$

The reflect function is a built-in function that is part of the GLSL language.

The next task is to calculate the VPL indirect lighting contributions. For all 6485 VPL's, we first access our VPL data from our 2 textures using the following lines of code:

```
vec3 vplPosition = texture1D(vplPosTex,texCoord).rgb;
vec3 vplNormal = texture1D(vplNorTex,texCoord).rgb;
float vplAttenuation = texture1D(vplNorTex,texCoord).a;
```

where vplPosTex and vplNorTex are our vpl position and normal/attenuation textures and texCoord is the location in that texture we need to access. We have to calculate texCoord by using the line:

```
float texCoord = (float(i)/numLights);
```

where i ranges from 1 to 6485 and numLights is 6485. Once we have our data, we must decode as discussed earlier using equation 4.4.

Now that we have our original VPL data, we must calculate the vector from our vertex to each VPL similar to lightDir above. Next, we calculate the diffuse terms of the object reflection and of the directional light. This is done by taking the dot product between the normal of the vertex and the vector from the vertex to the VPL and then taking the dot product between the normal of the VPL and the vector from the light to the vertex. All of these vectors must be normalized prior to these calculations.

Next, as discussed in section 3.3 and shown in figure 3.6, we will widen the viewable angle of the VPL from 180 degrees to 240 degrees. We do this by taking our diffuse term from directional light and normalize it to allow for a wider contributing angle by the following lines of code:

$$DiffuseTermLight = (DiffuseTermLight + 0.5)/1.5; \tag{4.9}$$

$$DiffuseTermLight = clamp(DiffuseTermLight, 0.0, 1.0); \tag{4.10}$$

This allows the dot product result to contribute to the color by mapping the range $[-0.5, 1.0]$ to the range $[0.0, 1.0]$ thus widening our VPL viewable contributions from 180 degrees to 240 degrees.

We then calculate the specular contributions of each VPL using the reflection ray from the normal of the vertex and the vector from the light to the vertex which is dotted with the view direction of the camera.

Lastly, we accumulate all of the VPL contributions to the color of that vertex by the following line:

$$\begin{aligned} indirect\_color += gl\_Color * DiffuseTermObj * DiffuseTermLight \\ * (1 - vplAttenuation) + SpecularTerm; \end{aligned} \tag{4.11}$$

where gl_Color is the color of the input vertex. We then divide this by the number of VPL's to get our final indirect color for that vertex.

Lastly, the vertex shader must compute the corresponding coordinate for the vertex in each of the shadow maps. This is done by multiplying our vertex by the input modelview and projection matrices for each of the rendered views corresponding to each of our 21 shadow maps.

The vertex shader then passes control on to the fragment shader. It also must pass variables calculated over by using varying variables. These variables include: light direction, light direction reflected ray, camera direction, and vertex normal for direct lighting, as well as our indirect color contributions per-vertex and the corresponding coordinates for the vertex in each of our 21 shadow maps.

4.3 FRAGMENT SHADER CODE

The fragment shader is similar to the vertex shader in operation except that it perform operations on fragments of our primitives rather than vertices. The fragment shader will perform these operations using the varying variables passed in from the vertex shader as well as our 2D texture array shadow map.

The first task the fragment shader has is to calculate whether our fragment is in shadow. We do this with the line:

```
float shadow = shadow2DArray(ShadowMap,
        vec4(ShadowCoord.xy / ShadowCoord.w, i ,
        ShadowCoord.z / ShadowCoord.w)).r;
```

where we take our shadow map and index using our coordinates calculated in the vertex shader that is divided by the 4 component of the vector know as perspective divide. This is necessary in order to index into our shadow map, because our texture is in the range $[0.0, 1.0]$ and our coordinates are not. The i corresponds to the layer of the shadow map to index into. Our direct shadow map then will have $i = 0$ and our indirect shadow maps will range from 1 to 20. We do this for all 21 shadow maps. From this we get a float value that will range from 0 to 1 and will correspond to percentage of shadow with 0 meaning not lighting whatsoever and 1 meaning no shadows whatsoever for that particular fragment. This resulting value from the direct shadow map is then multiplied with our direct lighting and the resulting 20 values from our indirect shadow maps are normalized and multiplied with our indirect lighting.

Next, we finish our direct lighting calculations by calculating the diffuse and specular terms similar to our indirect lighting in the vertex shader and add them for our direct lighting contributions.

Lastly, we set the color of the fragment using the line:

$$gl\_FragColor = (direct\_color * shadow) + (indirect\_color * INDshadow); \quad (4.12)$$

where shadow and INDshadow are the float values we calculated above from our shadow maps, direct color is our direct lighting we have just calculated and indirect color is the VPL contributions we calculated in our vertex shader. From this we now have the rendered scene with an estimated global illumination using our VPL's.

CHAPTER 5

RESULTS

CHAPTER 6

CONCLUSION

Bibliography

Cook, R. L. (1986, January). Stochastic sampling in computer graphics. *ACM Trans. Graph. 5*(1), 51–72.

Dachsbacher, C. and M. Stamminger (2003). Translucent shadow maps. In *Proceedings of the 14th Eurographics workshop on Rendering*, EGRW '03, Aire-la-Ville, Switzerland, Switzerland, pp. 197–201. Eurographics Association.

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

Dachsbacher, C. and M. Stamminger (2006). Splatting indirect illumination. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, I3D '06, New York, NY, USA, pp. 93–100. ACM.

Goral, C. M., K. E. Torrance, D. P. Greenberg, and B. Battaile (1984). Modeling the interaction of light between diffuse surfaces. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '84, New York, NY, USA, pp. 213–222. ACM.

Hašan, M., F. Pellacini, and K. Bala (2007, July). Matrix row-column sampling for the many-light problem. *ACM Trans. Graph. 26*(3).

Immel, D. S., M. F. Cohen, and D. P. Greenberg (1986, August). A radiosity method for non-diffuse environments. *SIGGRAPH Comput. Graph. 20*(4), 133–142.

Jensen, H. W. (1996). Global illumination using photon maps. In *Proceedings of the eurographics workshop on Rendering techniques '96*, London, UK, UK, pp. 21–30. Springer-Verlag.

Kajiya, J. T. (1986). The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '86, New York, NY, USA, pp. 143–150. ACM.

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

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

McGuire, M. and D. Luebke (2009). Hardware-accelerated global illumination by image space photon mapping. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, New York, NY, USA, pp. 77–89. ACM.

Nichols, G., R. Penmatsa, and C. Wyman (2010). Direct illumination from dynamic area lights with visibility. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games: Posters*, I3D '10, New York, NY, USA, pp. 21:1–21:1. ACM.

Nichols, G., J. Shopf, and C. Wyman (2009). Hierarchical image-space radiosity for interactive global illumination. *Computer Graphics Forum 28*(4), 1141–1149.

Nichols, G. and C. Wyman (2009). Multiresolution splatting for indirect illumination. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, I3D '09, New York, NY, USA, pp. 83–90. ACM.

Nijasure, M., S. Pattanaik, and V. Goel (2005). Real-time global illumination on gpus. *Journal of Graphics, GPU, and Game Tools 10*(2), 55–71.

Novák, J., T. Engelhardt, and C. Dachsbacher (2011). Screen-space bias compensation for interactive high-quality global illumination with virtual point lights. In *Symposium on Interactive 3D Graphics and Games*, I3D '11, New York, NY, USA, pp. 119–124. ACM.

Novák, J., D. Nowrouzezahrai, C. Dachsbacher, and W. Jarosz (2012, July). Virtual ray lights for rendering scenes with participating media. *ACM Trans. Graph. 31*(4), 60:1–60:11.

Ou, J. and F. Pellacini (2011, December). Lightslice: matrix slice sampling for the many-lights problem. *ACM Trans. Graph. 30*(6), 179:1–179:8.

Papaioannou, G. (2011). Real-time diffuse global illumination using radiance hints. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, New York, NY, USA, pp. 15–24. ACM.

Reeves, W. T., D. H. Salesin, and R. L. Cook (1987, August). Rendering antialiased shadows with depth maps. *SIGGRAPH Comput. Graph. 21*(4), 283–291.

Ritschel, T., T. Grosch, M. H. Kim, H.-P. Seidel, C. Dachsbacher, and J. Kautz (2008, December). Imperfect shadow maps for efficient computation of indirect illumination. *ACM Trans. Graph. 27*(5), 129:1–129:8.

Ritschel, T., T. Grosch, and H.-P. Seidel (2009). Approximating dynamic global illumination in image space. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, I3D '09, New York, NY, USA, pp. 75–82. ACM.

Serway, R. A. and J. W. Jewett (2004). *Physics for Scientists and Engineers*, Volume 2. Brooks Cole.

Sloan, P.-P., N. K. Govindaraju, D. Nowrouzezahrai, and J. Snyder (2007). Image-based proxy accumulation for real-time soft global illumination. In *Proceedings of the 15th Pacific Conference on Computer Graphics and Applications*, PG '07, Washington, DC, USA, pp. 97–105. IEEE Computer Society.

Stokes, W. A., J. A. Ferwerda, B. Walter, and D. P. Greenberg (2004). Perceptual illumination components: a new approach to efficient, high quality global illumination rendering. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, New York, NY, USA, pp. 742–749. ACM.

Szirmay-kalos, L., B. Aszdi, I. Laznyi, and M. Premecz (2005). Approximate ray-tracing on the gpu with distance impostors. computer graphics forum (eurographics 05.

Wald, I., T. Kollig, C. Benthin, A. Keller, and P. Slusallek (2002). Interactive global illumination using fast ray tracing. In *Proceedings of the 13th Eurographics workshop on Rendering*, EGRW '02, Aire-la-Ville, Switzerland, Switzerland, pp. 15–24. Eurographics Association.

Walter, B., A. Arbree, K. Bala, and D. P. Greenberg (2006). Multidimensional lightcuts. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, New York, NY, USA, pp. 1081–1088. ACM.

Walter, B., S. Fernandez, A. Arbree, K. Bala, M. Donikian, and D. P. Greenberg (2005). Implementing lightcuts. In *ACM SIGGRAPH 2005 Sketches*, SIGGRAPH '05, New York, NY, USA. ACM.

Walter, B., P. Khungurn, and K. Bala (2012, July). Bidirectional lightcuts. *ACM Trans. Graph. 31*(4), 59:1–59:11.

Ward, G. J., F. M. Rubinstein, and R. D. Clear (1988, June). A ray tracing solution for diffuse interreflection. *SIGGRAPH Comput. Graph. 22*(4), 85–92.

Whitted, T. (1980, June). An improved illumination model for shaded display. *Commun. ACM 23*(6), 343–349.

Williams, L. (1978). Casting curved shadows on curved surfaces. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '78, New York, NY, USA, pp. 270–274. ACM.

Yu, I., A. Cox, M. H. Kim, T. Ritschel, T. Grosch, C. Dachsbacher, and J. Kautz (2009, October). Perceptual influence of approximate visibility in indirect illumination. *ACM Trans. Appl. Percept. 6*(4), 24:1–24:14.

Zhukov, S., A. Iones, and G. Kronin (1998). An ambient light illumination model. In *Proceedings of the Eurographics Workshop on Rendering*, pp. 45–56.