LIGHTWAVE: AN INTERACTIVE ESTIMATION OF INDIRECT ILLUMINATION USING WAVES OF LIGHT

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. Bin Cong, Committee Member Department of Computer Science	Date
Dr. Kevin Wortman, 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. An important aspect of realistic lighting involves the inclusion of indirect illumination. However, to achieve a real-time rendering with indirect illumination, 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

Chapter Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

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 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 is quantized in particles called photons. Quantization is the idea of constraining elements that are continuous into discrete sets. (Serway & Jewett, 2004) Therefore, light can have the behavior of a wave and a particle. These findings are important to consider in our attempt 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 have used the term photons such as in 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 basic 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 if the light rays from the light sources terminated or ended at the first surface they hit. 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 are illuminated with everything else black. In order to make the scene more realistic, we need to add indirect illumination. Indirect illumination considers the behavior of light after it hits a surface. It features the reflection and refraction of light and produces realistic scenes depending on the extent of indirect illumination captured. It illuminates the surfaces that are not directly visible by the light source such as behind another object. Light can be interpreted as bouncing through a room infinitely until the energy of the light reaches zero and is completely absorbed by the surfaces in the room. Noting this, indirect illumination can be recursively calculated to infinity and therefore can be expressed using a Neumann series where we calculate the sum of light hitting a surface point due to the reflection of light from other surfaces n-times with n approaching infinity. Then 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 that we have the global illumination of the scene. This paper's goal is 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 ultra-realistic images that take into account many light sources as well as many types of lighting principles such as reflection, refraction, sub-surface light scattering, and others in very complex scenes consisting of millions of triangles and at 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 by actively moving the geometry, the camera, or the light source whereas static scenes do not. As opposed to offline techniques, real-time techniques have to make sacrifices and therefore can't be as scientifically accurate as offline techniques. The goal of this paper is to create a real-time technique that supports a fully dynamic scene. Regardless of whether the technique is offline or real-time, the algorithm used 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* by 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 the 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, global 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 chapter, indirect illumination can be calculated by summing the light incident on a surface due to the reflection of light n-times as n approaches 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 techniques beginning 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

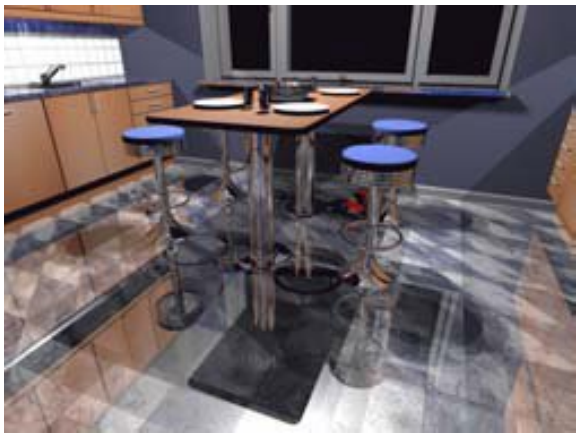


Figure 2.1: Image produced using ray tracing.

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. Each ray is shot into the scene and it registers the first surface it hits. From this surface point, additional rays go to each of the light sources to determine visibility to render shadows as well as to other surfaces to render reflections as shown in figure 2.2 from Ward, Rubinstein, and Clear (1988). These rays can also be used to calculate other lighting phenomena such as refractions.

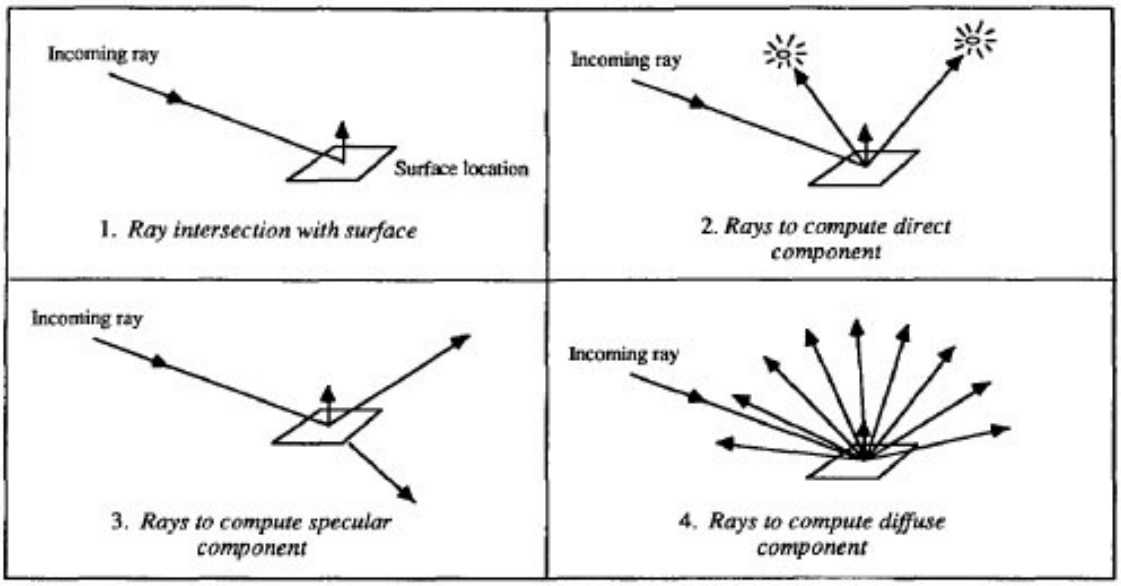


Figure 2.2: Steps of ray tracing.

The rays from the camera can be cast into the scene using different sampling patterns and techniques such as one ray per pixel or many rays 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 as shown in figures 2.3 and 2.4 from Reeves, Salesin, and Cook (1987).

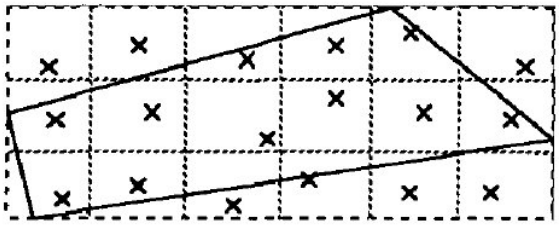


Figure 2.3: Sampling non-uniformly spaced locations in each pixel.

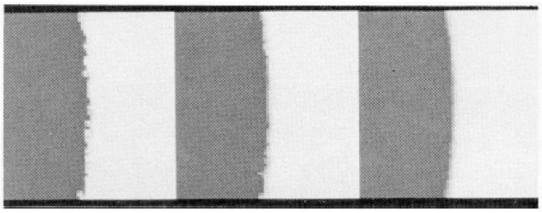


Figure 2.4: Image of aliasing artifacts known as jaggies. Better sampling results in softer edges.

Ray tracing is able to recreate ultra-realistic scenes such as figure 2.1 from Wald, Benthin, and Slusallek (2003) but at a high cost. Examples of ray tracing techniques can be seen in the work of Whitted (1980), Cook (1986), and Ward et al. (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, which is an equation that describes how light behaves when moving from one medium to another with a different refractive index. Equation 2.7 is in the form of equation 2.5 from Kajiya (1986) with M as the sum of the reflection and refraction terms as well as the diffuse component. Then the term g considers shadows 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

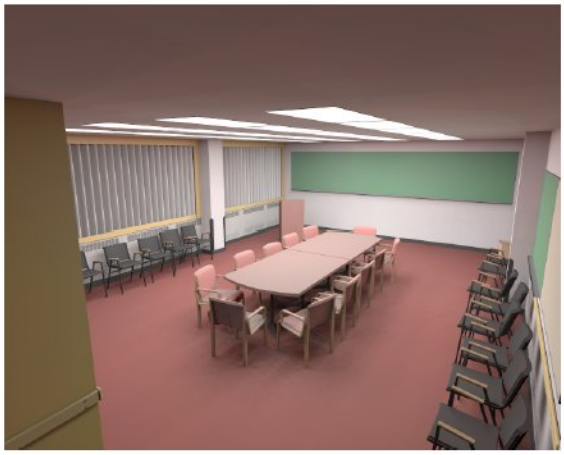


Figure 2.5: Image rendered using radiosity.

Radiosity is a type of rendering technique that was adapted for use in computer graphics from the thermal engineering field. 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 can be found in the works of Immel, Cohen, and Greenberg (1986) and Goral, Torrance, Greenberg, and Battalie (1984) and shown in figure 2.5 from Keller (1997).

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 seen in figure 2.6 from Immel et al. (1986). This can be found using the following equation from Goral et al. (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 as depicted in figure 2.7 from Goral et al. (1984).

Equation 2.8 can be derived using our rendering equation (equation 2.1) by 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

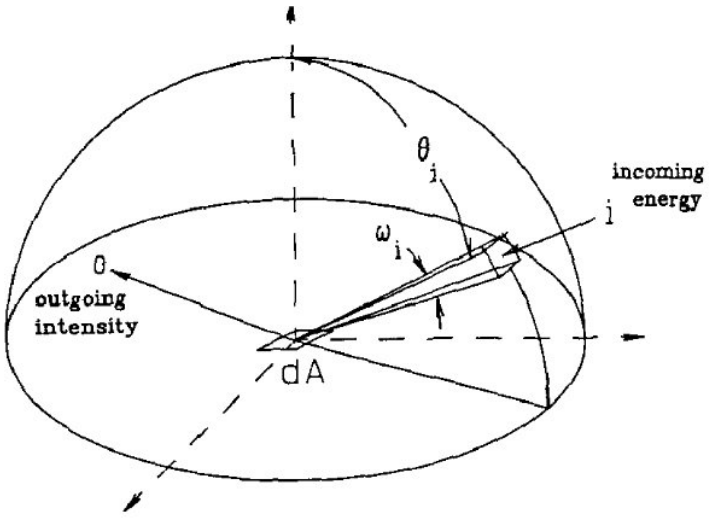


Figure 2.6: Hemispherical integral of the energy leaving the surface.

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 by Jensen (1996) is a two pass global illumination method that produces images such as that seen in figure 2.8. As mentioned in the Background section, Einstein coined the term photon for 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

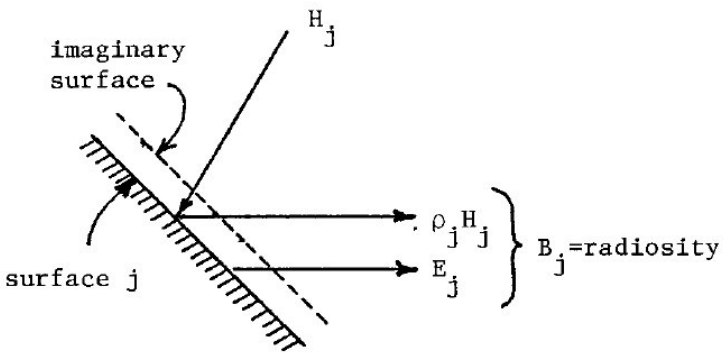


Figure 2.7: Illustration of equation 2.8.

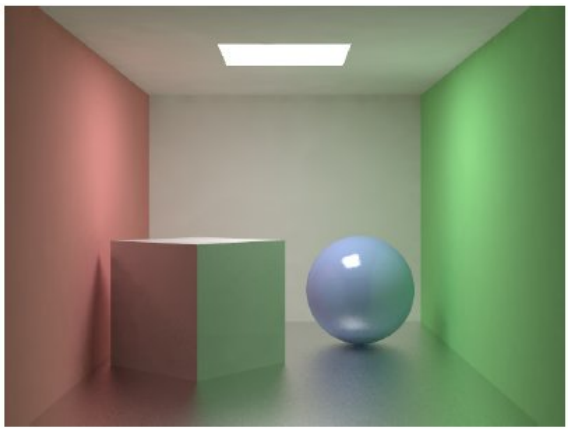


Figure 2.8: Image rendered using photon maps.

light sources and storing where they hit surfaces in the scene as seen in figure 2.9. 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 the caustics seen in figure 2.10.

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

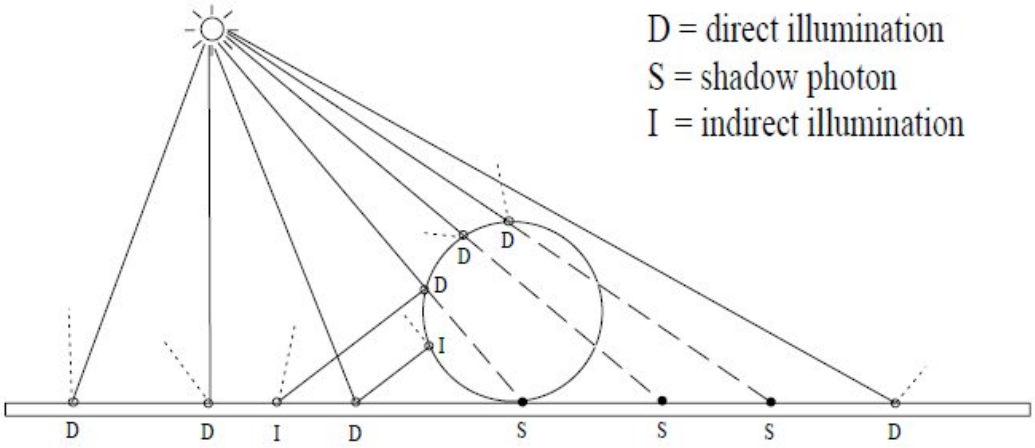


Figure 2.9: Process of casting photons into the scene.

Figure 2.10: Example of caustics.

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 this 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 as introduced by 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* by 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. This technique is 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 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* by Walter et al. (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 interior nodes as the light clusters that contain the individual lights below it in the tree as seen in figure 2.11 shown in Walter et al. (2005). 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* by 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* by Walter, Khungurn, and Bala (2012) to support low noise rendering of complicated scenes with glossy surfaces, subsurface BSSRDF's, and anisotropic volumetric models.

Matrix Sampling: Another adaption of the VPL method in offline rendering techniques is with the use of mathematical matrix sampling to reduce the rendering

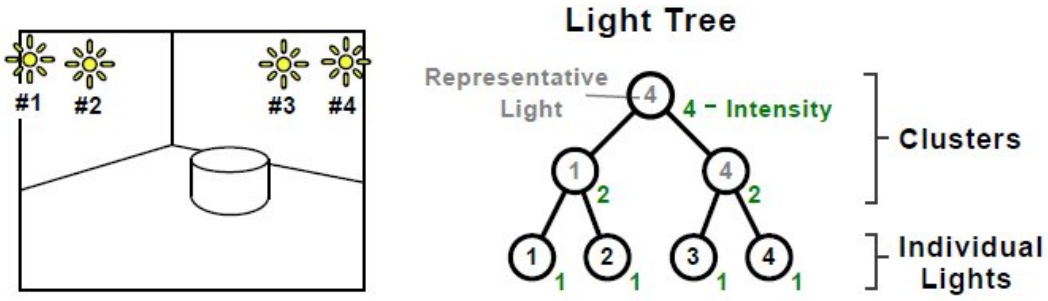


Figure 2.11: Example of a light tree.

cost. In *Matrix Row-Column Sampling for the Many-Light Problem* by 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 $O(m+n)$ as seen in figure 2.12 by Hašan et al. (2007). Additionally, *LightSlice: Matrix Slice Sampling for the Many-Lights Problem* by Ou and Pellacini (2011) uses a similar approach as in Hašan et al. (2007) and Walter et al. (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.

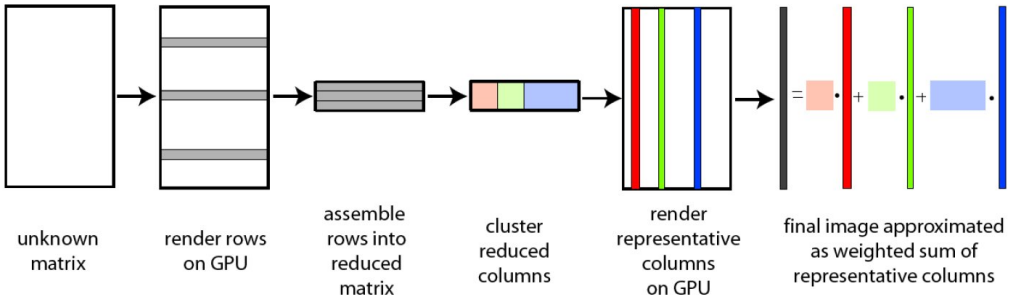


Figure 2.12: Overview of matrix sampling algorithm.

Shadow Maps: Shadow maps are textures that are 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 this 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 as presented by Williams (1978) and by Reeves, Salesin, and Cook (1987). See figures 2.13 and 2.14 by Dachsbacher and Stamminger (2005) for an image of a shadow map and the resulting final image, respectively. Further discussion on shadow maps will be done in the real-time rendering techniques section as well as in the implementation section.

Figure 2.13: Image of a shadow map. Scene is from the view of the light source.

Virtual Ray Lights: Lastly, Novák, Nowrouzezahrai, Dachsbacher, and Jarosz (2012) describe 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. A singularity is when a given value or location is undefined in our equation. For example, the equation $1/x$ is undefined for $x = 0$. Singularities are common along boundaries of walls as seen in figure 2.15 by Dachsbacher and Stamminger (2005). These singularities can cause artifacts in the scene such as spikes of high intensity which

Figure 2.14: Resulting image using the shadow map from figure 2.13.

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 as shown in figure 2.16 by Novák et al. (2012). 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 by Wald, Kollig, Benthin, Keller, and Slusallek (2002), approximate ray tracing on the GPU by Szirmay-Kalos, Aszodi, Lazanyi, and Premecz (2005), image space photon mapping by McGuire and Luebke (2009), and

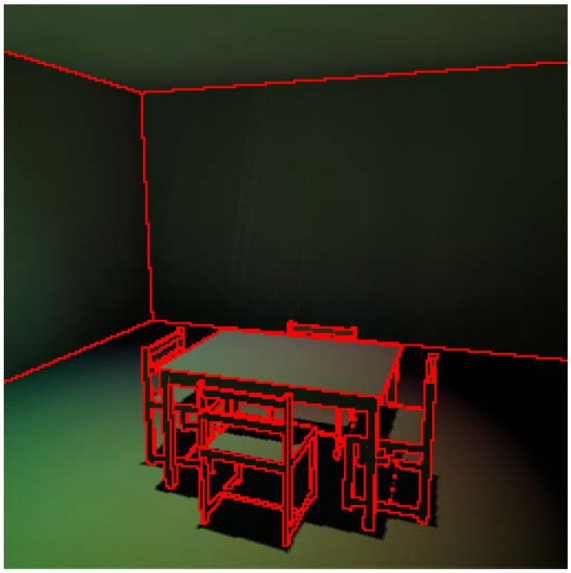


Figure 2.15: Common locations where singularities exist.

Instant Radiosity by 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 approximating indirect illumination by using 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 seen in figure 2.16. 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

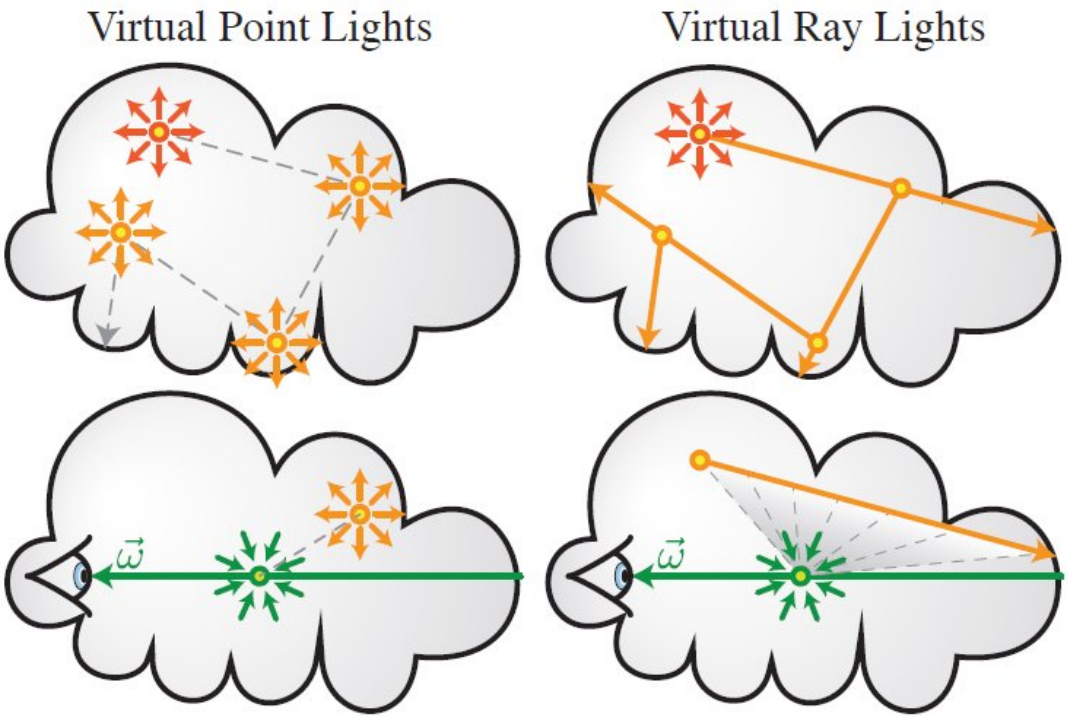


Figure 2.16: VRL versus VPL.

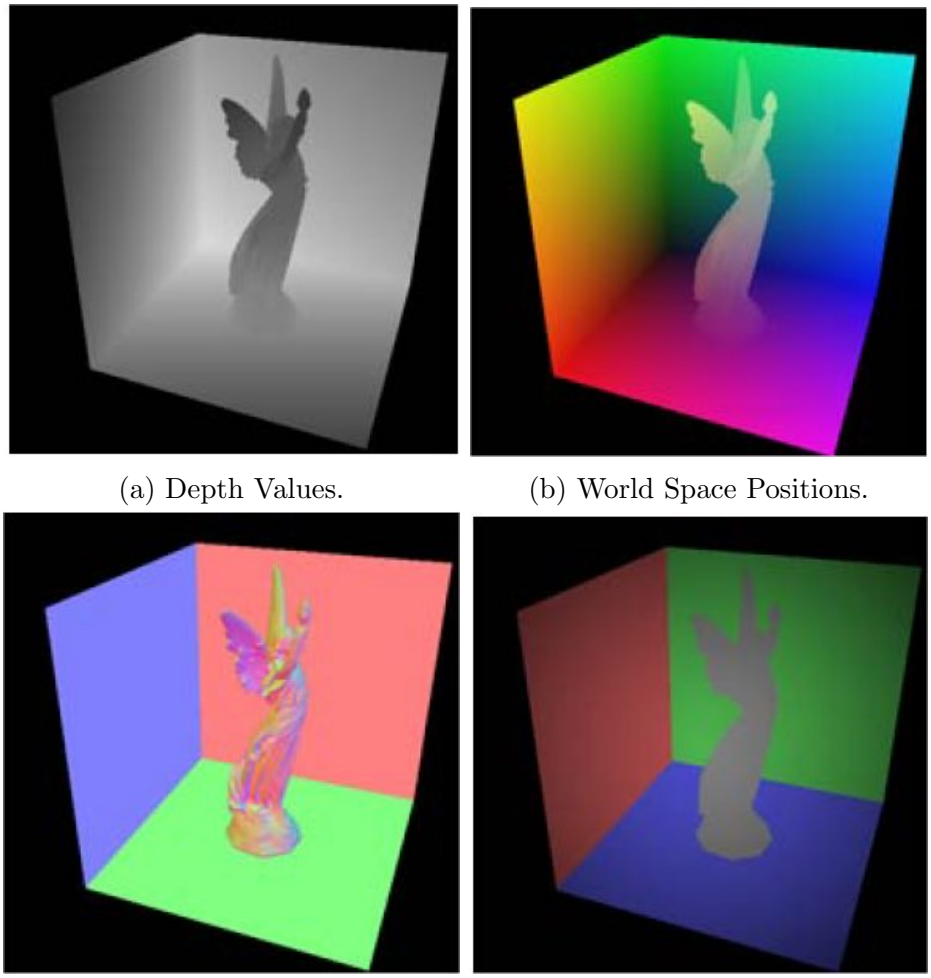
for each of the VPL's. These shadows had to be computed by using shadow volumes or shadow maps and were the bottleneck of the technique. There were also issues with low sampling rates which would result in weak singularities when the VPL got increasingly close to the illuminated surface point and each VPL had a high influence or contribution on the overall color of the final image. Also, temporal flickering could be seen if there were 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 offline techniques already mentioned which adapted the idea of using VPL's for the offline rendering of a variety of lighting phenomena, many real-time techniques also adapted the use of VPL's.

Shadow Map Alterations: As teased in section 2.3.4, many real-time rendering techniques expanded the use of shadow maps beyond just creating shadows. *Translucent Shadow Maps* by 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* abbreviated RSM by 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 shown in figure 2.17 by Dachsbacher and Stamminger (2005). 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



(a) Depth Values. (b) World Space Positions.

(c) Normal Values. (d) Reflected Radiant Flux.

Figure 2.17: Data stored in the RSM.

seen in equations 2.12 and 2.13.

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

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

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 as shown in figure 2.18 by Dachsbacher and Stamminger (2005). This technique

had the same singularity issue near the boundary of two adjoining walls shown earlier in figure 2.15. Other limitations includes having to ignore 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.

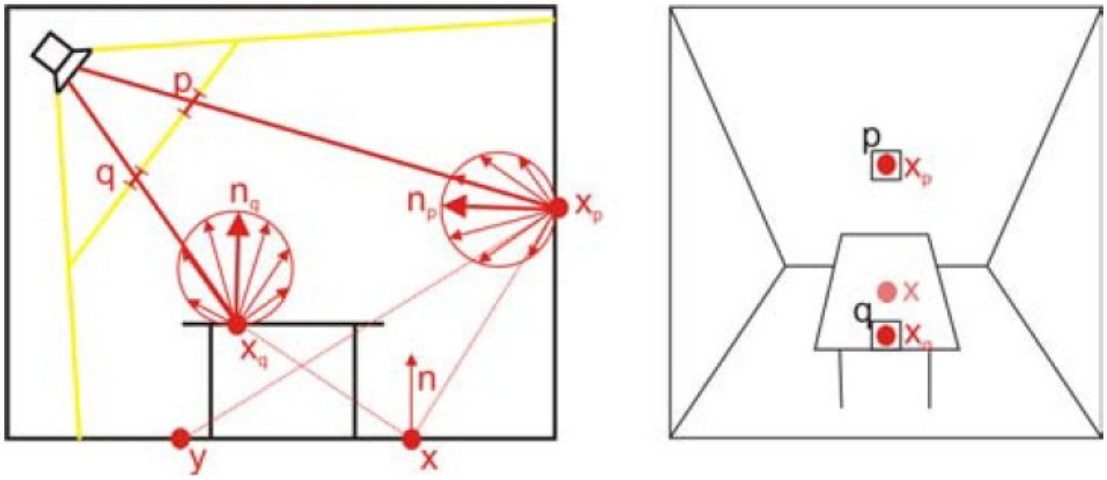


Figure 2.18: Illustration of equations 2.12 and 2.13.

Reflective shadow maps were then used in many other real-time rendering techniques that followed. This included *Splatting Indirect Illumination* by 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* by 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 by Ritschel et al. (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 are used for 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. Also, 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 shown in figure 2.19 by Ritschel et al. (2008).

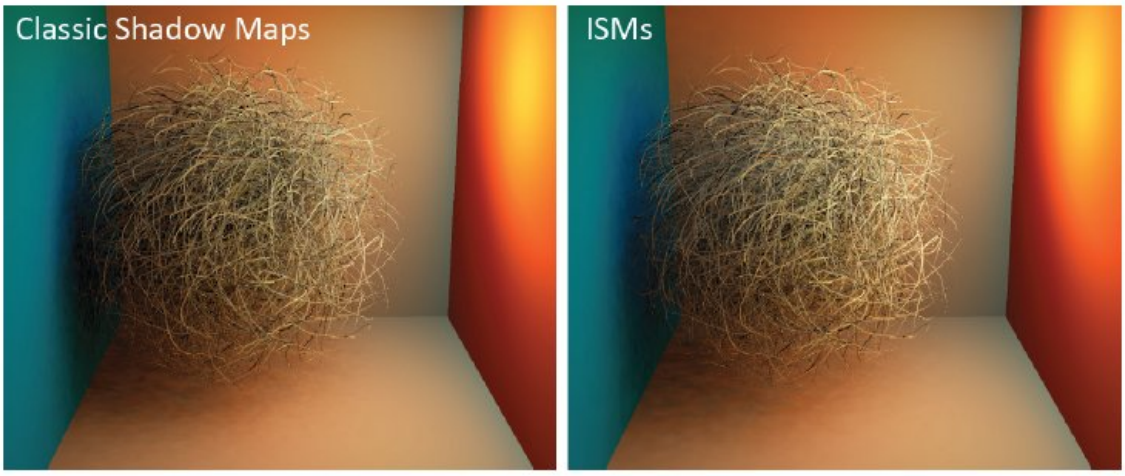


Figure 2.19: Comparison of normal shadow maps and ISM's.

Screen Space Algorithms: Additional VPL variants include *Hierarchical Image-Space Radiosity for Interactive Global Illumination* by Nichols, Shopf, and Wyman (2009) where instant radiosity is combined with multiresolution techniques by 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 by 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. Nijasure, Pattanaik, and Goel (2005) used them 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 et al. (2005) and the use of spherical harmonics along with the reflective shadow maps discussed earlier. 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 are 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* by Keller (1997) to render the indirect illumination. Shadow maps will be used as in the spirit of Williams (1978) and Reeves et al. (1987) to render shadows for direct illumination. Shadow maps will also be used to render indirect shadows. The VPL's will be structured outward from the primary light source in hemispheres using specific distances and angles around the primary light source. 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 and therefore simulate indirect illumination using only the direct illumination of many VPL's while ignoring the bouncing of light among surfaces thus simplifying the Neumann series to just one term. 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 along with additional parameters that will provide us with further balancing capabilities. Specific implementation details will follow in section 3.3. Lastly, in support of this technique

and its ignorance of calculating true indirect illumination and instead using a direct illumination approximation of indirect illumination, we discuss the importance of using a scientifically correct rendering versus a visually appealing rendering.

3.2 Scientifically Correct Versus Visually Appealing

All of the previous work mentioned in chapter 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, & Greenberg, 2004). Also, according to Stokes et al. (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 using only direct lighting that will be able to reach surfaces only visible to indirect light 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* by Yu et al. (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. This is important since the most expensive calculation is the visibility determination for indirect shadows as will become apparent in the results section.

To research the effects of this on the viewer's perception of the rendering, Yu et al. (2009) conducted a psychophysical study involving two different experiments. The first experiment involved estimating the difference between the approximation rendering and the reference rendering using 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. First was imperfect visibility which was used in the *Imperfect Shadow Maps* technique discussed in section 2.4 by Ritschel et al. (2008). Next was ambient occlusion which illuminates objects as if the entire hemisphere was a light as described by Zhukov, Iones, and Kronin (1998). Extending that is directional

ambient occlusion which extends ambient occlusion to allow shadows to respond to the lighting direction as described by Sloan, Govindaraju, Nowrouzezahrai, and Snyder (2007) and Ritschel, Grosch, and Siedel (2009). Lastly, no visibility meaning that indirect shadows are not shown such as in the technique described by Dachsbacher and Stamminger (2005,2006). 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 by Ritschel et al. (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 or no 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 by 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 as seen in figure 3.1.

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 y-axis. This totals in 1296 rays plus the vertical ray along the y-axis resulting in 1297 rays to form a hemisphere around the light as seen in figures 3.2 and 3.3.

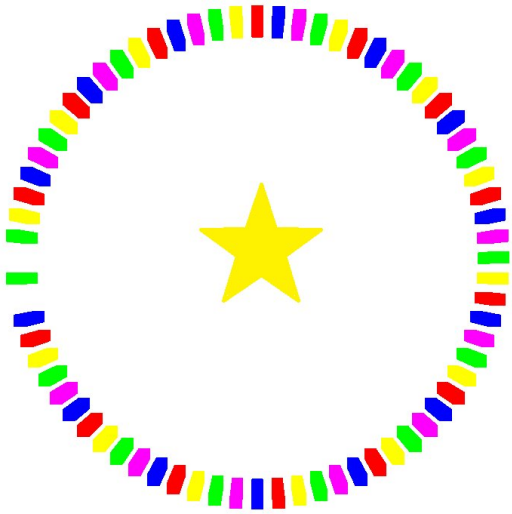


Figure 3.1: The light source (star) is facing the reader (i.e., direction/normal pointing out of the paper).

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 as seen in 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 as seen in 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 and therefore 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

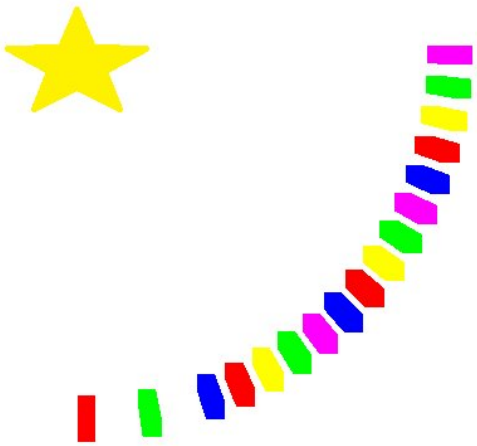


Figure 3.2: The light source (star) is facing downwards. Angles are drawn to scale.

quality and performance desires. As such, we can increase or decrease the corresponding angles to decrease 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

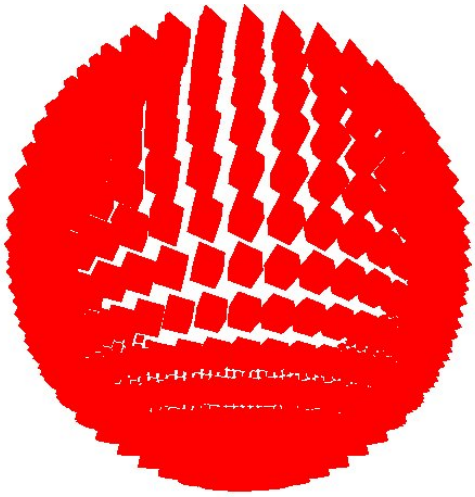


Figure 3.3: Figures 3.1 and 3.2 together form a hemisphere. View is looking slightly up at the hemisphere.

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

Each VPL normal is calculated using the angles required to rotate the primary light source's direction to point to that specific VPL. Figure 3.7 shows the normals of every VPL. The resulting shape is a hemisphere of radius 1 due to the normals being normalized. It also shows that the VPL normals point outward in all negative-y directions from the primary light source.

Lastly, occlusion will be handled with a shadow map as implemented by Williams (1978) and Reeves et al. (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. This

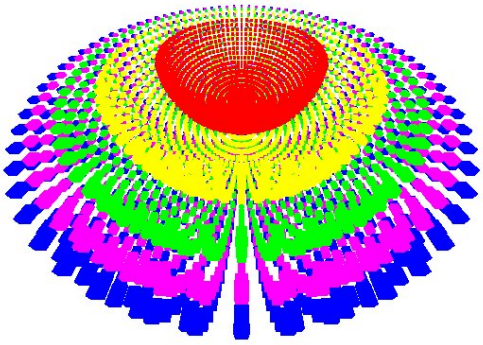


Figure 3.4: 5 stacked hemispheres formed by the VPL's around the primary light source. Each hemisphere is a different shade to make distinguishing easier.

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.)

implementation will try two different techniques of rendering indirect shadows. The first technique will handle indirect occlusion through the use of regular shadow maps on 20 randomly sampled VPL's as seen in figure 3.8 and will render shadows solely on the information gathered in the 20 shadow maps meaning that we will have 1 direct shadow and 20 indirect shadows. Therefore, these shadows are computed using accurate visibility and this method will be referred to as "accurate shadows." The second technique will handle indirect occlusion through the use of regular shadow maps rendered using 5 specific VPL's as seen in figure 3.9 and will render shadows by integrating between each of the 5 shadow maps. The VPL's are equidistant from one another and provide maximum coverage of the scene. By doing this, we will require less shadow maps but will render more indirect shadows.

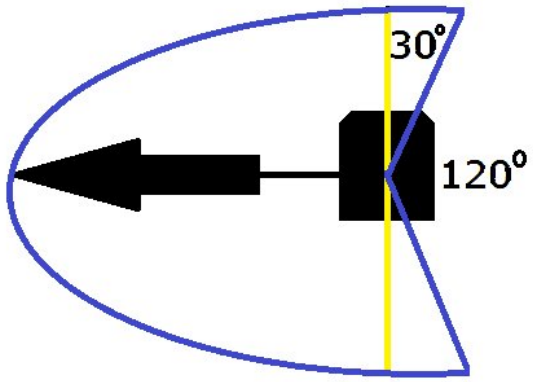


Figure 3.6: Showing the wave-like radiance contribution of a VPL. We add an additional 30 degrees of viewable range on the top and bottom limiting the blind spot of the VPL to 120 degrees rather than 180. The arrow signifies the VPL's direction/normal.

All of the shadows except for the 5 shadows that come directly from the shadow maps do not use accurate visibility. Instead, these shadows come from the act of integrating or interpolating between each of the 5 shadow maps and so this method will be referred to as "integrated shadows." The specific implementation details for both techniques will be discussed in chapter 4 with code excerpts present.

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 positioning 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 CPU will compute all of the necessary VPL data and store it in textures to be passed to the GPU to be used later in the lighting calculations. This action will need to be performed every time the light is moved. The CPU will also

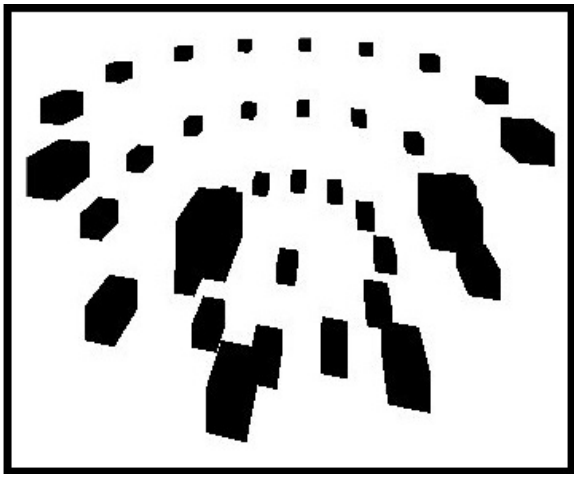


Figure 3.7: Showing the normals of every VPL. Using ray angle of 30 degrees to simplify the graphic.

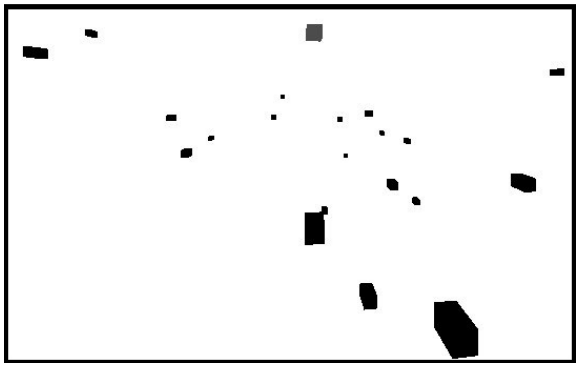


Figure 3.8: Showing 20 randomly sampled VPL's with the primary light source at the top center of the image.

be used to generate the shadow maps for the direct and indirect shadows by rendering the scene from each of the chosen light's view. This action will need to be done every time an object or the light is moved.

The GPU will calculate shading and illumination for the vertices inputted using the VPL data texture and will calculate shadows based off of the provided shadow maps passed in from the CPU. Using the GPU to perform these actions are desired due to the GPU's ability to excel in parallel processing of data. Since the same

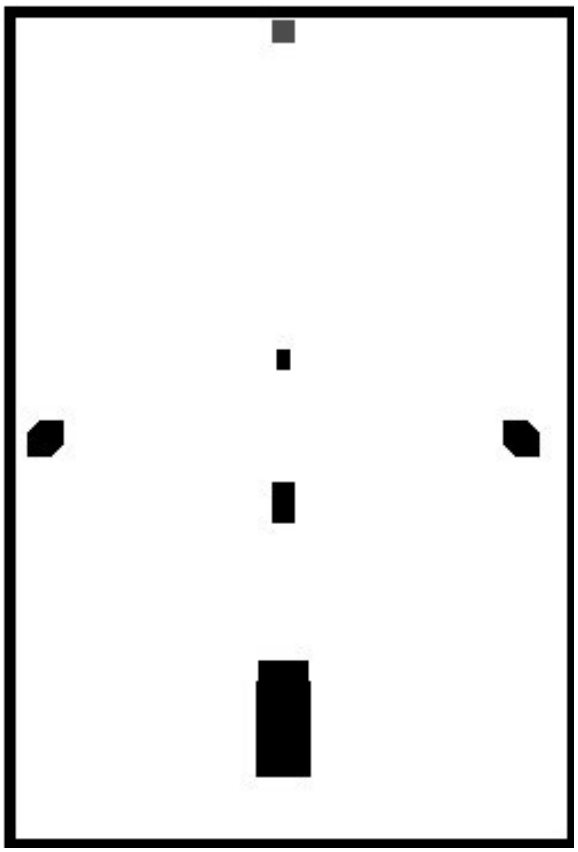


Figure 3.9: Showing 5 specifically chosen VPL's with the primary light source at the top center of the image.

calculations are performed for every vertex in the scene, each vertex illumination can be calculated in parallel in no specific order allowing these calculations to be performed faster than on the CPU. These calculations are programmed using GLSL which will be discussed shortly.

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 designed 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 64 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:

- 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 direction 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 on 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 and then rotating based on the angles required to fill the space in the hemisphere we are forming. These angles are consolidated into the normal variable in equation 4.1. The 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 rotating the primary light source direction to the direction of the corresponding VPL equal to the normal variable of equation 4.1 as calculated by:

```
normal[x,y,z] = primaryLightSourceNormal[x,y,z]
normal[x] = cos(angleZ)*normal[x]-sin(angleZ)*normal[y]
normal[y] = sin(angleZ)*normal[x]+cos(angleZ)*normal[y]
normal[x] = cos(angleY)*normal[x]+sin(angleY)*normal[z]
normal[z] =-sin(angleY)*normal[x]+cos(angleY)*normal[z]
```

Lastly, 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. We use the same statement for the normal data but this time we replace each RGB with RGBA since we will also store the attenuation in addition to the xyz normal values. We also replace the address vplDataPos with vplDataNor.

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 rather than from our camera. 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 first technique that uses accurate shadows, 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. For the second technique that uses integrated shadows, we use 6 shadow maps with 1 being used for direct shadows and the other 5 for indirect shadows. The 5 VPL's chosen lie in the innermost hemisphere equidistant apart to get maximum coverage. In order to store these shadow maps, we use a single 2D texture array which consists of either 6 or 21 layers. This is done in OpenGL by using the following line of code:

```
glTexImage3D(GL_TEXTURE_2D_ARRAY, 0, GL_DEPTH_COMPONENT32, width,
height, 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. The width and height of the shadow maps should be a ratio of the screen size. Ideally, it should be a multiple larger. In our case, we will have the shadow maps be three times larger than the screen size. We want it to be larger in order to minimize any jaggies that would be clearly visible should the shadow maps be the same size or even smaller than the screen size. We do have limitations on how large we want our shadow maps to be because of the amount of memory available on the GPU to store them. Memory considerations and analysis will be discussed in chapter 5.

Next, in order to capture the depth values of the scene we use OpenGL's frame buffer capabilities. We render the scene either 6 or 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 is the only rendering that the user sees. While generating the shadow maps we are using the fixed function pipeline to perform the rendering from each perspective meaning that it is rendered entirely on the CPU using no modified shaders. 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 passed over to the shaders on the GPU. 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 matrix for the shaders, 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 where either an object or the primary light source is moved.

Next, the main thing left for the main program to perform is to render the final scene with the assistance of the shaders on the GPU, 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 run 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 passed in 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;
```

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;
```

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);

camDir = vec3(cameraPosition) - gl_Vertex.xyz;
```

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 for the object reflection and for the directional VPL. 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 VPL 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 the diffuse term for the directional VPL and normalize it to allow for a wider contributing angle by the following lines of code:

```
DiffuseTermLight = (DiffuseTermLight + 0.5)/1.5;

DiffuseTermLight = clamp(DiffuseTermLight, 0.0, 1.0);
```

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 with the vector from the VPL 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 using equation 4.5:

$$indirect\_color+ = gl\_Color * DiffuseTermObj * DiffuseTermLight * (1 - vplAttenuation) + SpecularTerm \tag{4.5}$$

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 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 the direct lighting, as well as our indirect color contributions per-vertex and the corresponding coordinate for each vertex in each of our shadow maps.

4.3 Fragment Shader Code

The fragment shader is similar to the vertex shader in operation except that it performs 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. For the accurate shadows technique, 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 which is then divided by the 4th component of the vector known 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 for our direct shadows that will range from 0 to 1. We will also get a float value for our indirect shadows that after we divide by 20 to normalize will also range from 0 to 1. This float value corresponds to the percentage of shadowing with 0 meaning no lighting whatsoever and 1 meaning no shadowing whatsoever for that particular fragment. This resulting value from the direct shadow map is then multiplied with our direct lighting and the resulting normalized value from our indirect shadow maps is multiplied with our indirect lighting.

For the integrated shadows technique, we use the same line above for our direct shadow ($i = 0$) and for each of our 5 indirect shadow maps. However, the only difference in this technique is that we then integrate or interpolate between each of these 5 indirect shadow maps in order to render additional shadows. This is done by creating a vector that goes from the coordinate of one shadow map to the other. We then divide this vector by the number of between steps we want to take. We will show results based off using 2, 4, 6, and 10 as the number of steps. Then we

step across this vector using the chosen number of steps to calculate the in-between coordinate which then contributes an additional indirect shadow. We use the above line along with our in-between coordinate as the ShadowCoord to get our float value. For the value of i in the above equation, we round to the closest shadow map. So for example, if we are integrating between $i = 1$ and $i = 2$ and using 10 steps, the first 5 coordinates will be using $i = 1$ and the last 5 coordinates will be using $i = 2$.

We then do this for all combinations of the indirect shadow maps. By combinations, we mean the act of selecting a subset of k distinct elements in a set S which can be calculated by using the binomial coefficient (equation 4.6).

$$\begin{pmatrix} n \\ k \end{pmatrix} = \frac{n!}{k!(n-k)!} \tag{4.6}$$

Here n is the number of elements in set S which in our case will be 5 for the number of indirect shadow maps. Then we have $k = 2$ because we want to create a vector between 2 indirect shadow maps. This leads to equation 4.7.

$$\begin{pmatrix} 5 \\ 2 \end{pmatrix} = \frac{120}{2 * 6} = 10 \tag{4.7}$$

Therefore, we create 10 vectors between our 5 indirect shadow maps (1-2, 1-3, 1-4, 1-5, 2-3, 2-4, 2-5, 3-4, 3-5, 4-5). Then we add all of the resulting shadowing values and divide by the number of shadows to normalize, which depends on the number of steps we take:

$$numShadows = (numSteps + 2) * 10 \tag{4.8}$$

where 10 comes from equation 4.7. So for 2 steps, we get 40 indirect shadows and for 10 steps we get 120 indirect shadows. Most of these shadows are not using accurate visibility, but the act of integrating shadow maps that are using accurate visibility provides us with overall smoother shadows while also using less shadow maps and therefore less GPU memory.

Next, we finish our direct lighting calculations by calculating the diffuse and specular terms similar to our indirect lighting calculation in the vertex shader and add them to our direct lighting contributions.
Lastly, we set the color of the fragment using equation 4.9:

$$gl\_FragColor = (direct\_color * shadow) + (indirect\_color * INDshadow) \quad (4.9)$$

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. After performing equation 4.9, we have our scene rendered with approximated global illumination by using our VPL's.

CHAPTER 5

RESULTS

5.1 Accurate Shadows Approach

In order to give a detailed display of the flexibility of this method, we wanted to show the results of the method when changing many of the adjustable parameters. These include varying the resolution of the screen, increasing the angle between VPL rays, decreasing the number of VPL's per ray, and decreasing the number of indirect shadow maps used. Modifying these parameters impact performance by increasing or decreasing the number of frames rendered per second as well as the quality of the rendering. Therefore, section 5.1.1 will detail the performance impact of varying the parameters based off the change in frame rate and section 5.1.2 will detail the impact of varying the parameters on the overall quality of the images rendered by calculating the percentage difference per pixel against the default parameter set-up. As a reminder, the default method which is featured first on each table uses a resolution size of 1280 by 720, an angle of 5 degrees between VPL rays, 5 VPL's per ray, and 20 indirect shadow maps.

5.1.1 Impact of Parameter Choices on FPS

The results will be given in frames per second (FPS). Table 5.1 will display the results of varying the angle between the VPL rays. Table 5.2 will display the results of varying the number of VPL's used on each ray (the total number of hemispheres). Table 5.3 will display the results of varying the resolution used.

Lastly, table 5.4 will display the results of reducing the number of indirect shadow maps used.

Tables 5.1, 5.2, 5.3, and 5.4 show some important characteristics. First, tables 5.1 and 5.2 show the impact of the angle between rays and the number of VPL's per ray on the total number of VPL's used. The equation to calculate the total number of VPL's based off the angle and the number of VPL's per ray used is:

$$numberVPLs = VPLsPerRay * ((90/Angle) * (360/Angle) + 1) \tag{5.1}$$

More interestingly, it shows that a large reduction in the total number of VPL's used has a relatively small affect on the FPS recorded compared to other parameter changes. For example, reducing the number of VPL's used by a factor of 4 (from 6485 to 1625) increases FPS by only 10 whereas reducing the number of VPL's used by a factor of near 76 (from 6485 to 85) increases FPS by 40.

Table 5.3 shows that decreasing the resolution of the screen has a relatively large impact on the FPS recorded. To compare it against decreasing the number of VPL's used, in order to see a similar impact of reducing the resolution by a factor of 1.3 from 1280x720 to 1120x630, we would have to reduce the number of VPL's by a factor of 35 (from 6485 to 185).

Similarly, table 5.4 shows the number of indirect shadow maps having a large impact on the performance. As mentioned often in similar studies on indirect illumination, calculating indirect shadows is computationally expensive as supported in table 5.4. To make a similar comparison as above, reducing the

number of shadow maps used by 5, a factor of 1.33, has a similar impact as reducing the resolution by a factor of 1.3 or reducing the number of VPL's by a factor of 35 (from 6485 to 185). Also, table 5.4 shows that for each indirect shadow map we add, we can expect a decrease in FPS by about 5.

In summary, when choosing parameters of the method to be run on slower machines, a combination of reducing the resolution and reducing the number of indirect shadow maps would result in the most efficient performance gain in regards to FPS increase. Next, we will similarly analyze the impact of these parameter changes to the quality of the image in order to see what changes would result in the most efficient reductions while maintaining quality.

Table 5.1: Varying the Angle Between VPL Rays (Impact on FPS).

Resolution	Angle	VPL's Per Ray	Total #VPL's	#Indirect SM's	FPS
1280x720	5	5	6485	20	55
1280x720	10	5	1625	20	65
1280x720	30	5	185	20	84
1280x720	45	5	85	20	95
1280x720	90	5	25	20	100

5.1.2 Impact of Parameter Choices on Quality

Just as important as performance is the quality of the images rendered. Therefore, this section will detail the quality impact of parameter choice using the default set-up as the reference image. By detailing this information, it will reveal the ultimate flexibility of the method and whether we can use this method on slower machines with limited quality impact. We will use the same parameter

Table 5.2: Varying the Number of VPL's Per Ray (Impact on FPS).

Resolution	Angle	VPL's Per Ray	Total #VPL's	#Indirect SM's	FPS
1280x720	5	5	6485	20	55
1280x720	5	4	5188	20	56
1280x720	5	3	3891	20	61
1280x720	5	2	2594	20	62
1280x720	5	1	1297	20	78
1280x720	5	6	7782	20	52

Table 5.3: Varying the Resolution Size. (Impact on FPS).

Resolution	Angle	VPL's Per Ray	Total #VPL's	#Indirect SM's	FPS
1280x720	5	5	6485	20	55
1120x630	5	5	6485	20	82
960x540	5	5	6485	20	104
800x450	5	5	6485	20	119
640x360	5	5	6485	20	133
1440x810	5	5	6485	20	42

choices as the tables for section 5.1.1, but we will be interested in the percentage difference between the chosen parameters and the default parameters. The percentage difference will be computed using a Python script which takes in two rendered images as input and compares them on a pixel by pixel basis. This comparison is done as follows:

Table 5.4: Varying the Number of Indirect Shadow Maps (Impact on FPS).

Resolution	Angle	VPL's Per Ray	Total #VPL's	#Indirect SM's	FPS
1280x720	5	5	6485	20	55
1280x720	5	5	6485	15	87
1280x720	5	5	6485	10	106
1280x720	5	5	6485	5	128
1280x720	5	5	6485	1	157
1280x720	5	5	6485	0	161

```
similar = 0.0;
for each pixelComponent (R,B,G) in image 1 do
    if pixelComponent_image1 >= pixelComponent_image2 then
        similar += (pixelComponent_image1 +1) /
        (pixelComponent_image2 +1);
    else
        similar += (pixelComponent_image2 +1) /
        (pixelComponent_image1 +1);
    end
end
similar = similar/numPixelComponents;
percentDifference = (1-similar)*100;
```

Algorithm 1: Compute Image Difference

Algorithm 1 adds 1 to the 2 pixel components in order to avoid division by 0. This way our pixel components will range from 1 to 256. For example, if image 1 had a value of 200 for the red component of pixel 1 and image 2 had a value of 100 for the red component of pixel 1, we would say that these two images were 50%

different. We do this calculation for all 3 color components of every pixel in each image and average it out to find the total similarity or difference between the two images.

The purpose of doing such a calculation is simply due to the fact that quantifying the quality of an image or the similarity with another image is rather subjective. This way we can assign an objective value to the comparison and then objectively order the images rendered using different parameters by similarity to the default parameter set-up and declare certain parameter changes as more efficient in preserving quality than others. By efficient, we mean that the parameter changes increase performance with limited quality reductions. Tables 5.5, 5.6, 5.7, and 5.8 show the calculated similarity percentages. The higher percentage the image, the closer to the reference image it is with 100% meaning that it is the same image.

Comparing tables 5.5 and 5.6 reveals that although the number of VPL's used when increasing our VPL ray angles drops more quickly than when we reduce the number of VPL per ray thereby reducing the number of hemispheres, the higher ray angle images (table 5.5) render more similar images to the reference than when we reduce the number of VPL per ray. For example, when we use 185 VPL's with an angle of 30 degrees, we get an image that is 98% similar to the reference, but when we use 1297 VPL's with 1 VPL per ray, we get an image that is only 91.6% similar to the reference. This leads us to believe that increasing the angle to reduce the number of VPL's leads to better quality images than reducing the number of VPL's per ray or hemispheres to reduce the number of VPL's. Such an observation can help us increase performance and maintain a level of similarity to the reference image.

When looking at table 5.7, we see high values of similarity to the reference, however, these can be misleading. These values were calculated after rendering the image at the chosen resolution size and then stretching it to the reference image resolution size. This brings into question the idea of global similarity and local similarity as well as the notion of visually pleasing as discussed in section 3.2. Although these images are very similar to the reference in an overall pixel by pixel basis, there are some things that this number does not show us. It requires looking at the stretched image to realize the effect this has on the image quality. These images show worsening cases of jaggies or artifacts near the edges of boundaries such as the edges of the boxes as the resolution of the image is reduced. So these reduced resolution images are similar to the reference, but when stretched show localized differences that can be distracting to the eye. When comparing the reference to the stretched 640x360 images on a local scale such as on the edges of a box, we get a similarity of 97.4% which is slightly less than the 99.5%, however, a even more localized comparison such as a few pixels on either side of the boundary line would show even more difference.

Regardless of the numbers, a localized difference such as artifacts detract from the overall realism of the image which is not preferred. A slight global difference, however, such as a slightly lighter or darker overall image would result in a larger difference value but more visually pleasing to the human eye. These facts need to be taken into consideration when choosing the method or parameters. This idea will be covered more in section 5.1.3.

Table 5.8 shows the impact of reducing the number of shadow maps used for indirect shadows. An interesting note is that when using 10 shadow maps for

indirect shadows and when completely ignoring indirect shadows we get near identical similarity measurements to our reference. This reinforces the notion that indirect shadows are possibly an unnecessary computational burden that can be ignored especially on slower machines. Recall table 5.4 and the comparison of FPS between the two parameter choices. When choosing to use 10 indirect shadow maps we get 106 FPS, however, when we ignore indirect shadows we get 161 FPS. For nearly identical global similarity values, we can gain 55 FPS by ignoring indirect shadows. This is a major parameter choice to consider when wanting to increase performance.

It is worth noting that if an image is not 100%, this does not mean that it is not a quality or accurate image rendering. It purely means that the image is in some way different to our reference image. For example, the last entry of table 5.6 says that it is 96.5% similar to our reference. However, due to the fact that this rendering indeed uses more VPL's than our reference, this image could be perceived as more accurate or of higher quality than our reference. This calculation does take into account the findings of section 3.2 which stated that there was found to be a connection between the amount of indirect illumination and the perceived similarity to the reference. More on this idea will be discussed in section 5.1.3.

5.1.3 Alternatives Analysis

The aim of this section is to determine any additional sacrifices that could be made in order to improve performance. As shown in table 5.8 and discussed above, indirect shadows are the most computationally expensive scene component that has the least influence on the overall scene. This may lead one to consider whether or

Table 5.5: Varying the Angle Between VPL Rays (Impact on Quality).

Resolution	Angle	VPL's Per Ray	Total #VPL's	#Indirect SM's	% Similar
1280x720	5	5	6485	20	100.000
1280x720	10	5	1625	20	97.945
1280x720	30	5	185	20	97.956
1280x720	45	5	85	20	95.440
1280x720	90	5	25	20	92.676

Table 5.6: Varying the Number of VPL's Per Ray (Impact on Quality).

Resolution	Angle	VPL's Per Ray	Total #VPL's	#Indirect SM's	% Similar
1280x720	5	5	6485	20	100.000
1280x720	5	4	5188	20	92.973
1280x720	5	3	3891	20	93.458
1280x720	5	2	2594	20	90.318
1280x720	5	1	1297	20	91.591
1280x720	5	6	7782	20	96.509

not ignoring indirect lighting could be possible. However, as shown in table 5.9, we compared our default method with a direct lighting only method. Using our similarity calculation, we see the lowest similarity of 84.4%, but the highest performance with 681 FPS. On a machine that is able to run any of the above parameter choices with at least 30 FPS, it is not necessary to sacrifice this much quality for performance, but for the slowest machines, this may be a necessary choice.

Also in table 5.9, we include our indirect lighting without indirect shadows method for comparison with it's respectable 161 FPS and 98.5% similarity. This was the best performing parameter choice from tables 5.1 through 5.4. Should a

Table 5.7: Varying the Resolution Size. (Impact on Quality).

Resolution	Angle	VPL's Per Ray	Total #VPL's	#Indirect SM's	% Similar
1280x720	5	5	6485	20	100.000
1120x630	5	5	6485	20	99.547
960x540	5	5	6485	20	99.541
800x450	5	5	6485	20	99.418
640x360	5	5	6485	20	99.483
1440x810	5	5	6485	20	99.741

Table 5.8: Varying the Number of Indirect Shadow Maps (Impact on Quality).

Resolution	Angle	VPL's Per Ray	Total #VPL's	#Indirect SM's	% Similar
1280x720	5	5	6485	20	100.000
1280x720	5	5	6485	15	99.475
1280x720	5	5	6485	10	98.853
1280x720	5	5	6485	5	93.611
1280x720	5	5	6485	1	90.897
1280x720	5	5	6485	0	98.563

compromise between direct lighting only and indirect lighting with no indirect shadows be needed, we included a few alternative approaches in addition to the previous two in table 5.9. These approaches attempt to fake indirect lighting while achieving the direct only performance. This is done through the use of scaling the color and shadow contributions in the fragment shader. As reminded above and discovered in section 3.2, the amount of indirect light can contribute to the perceived realism of the scene. As such, 5 different scaling approaches are included in table 5.9 to compare against our reference image. The multiplier in parenthesis in each table entry shows the amount of scaling done to the direct lighting of the scene. For example, for the entry with a 1.1 multiplier this means that the final

color is calculated as below:

$$color = (direct\_color * 0.5 * shadow) + (direct\_color * 0.6) \tag{5.2}$$

For the next entry the 0.6 would be 0.7 and so on. This way the direct shadows are only half as dark as before and are not purely black and increase the overall brightness of the scene. As seen in the table, this approach of faking indirect light leads to better similarity numbers than using just the direct lighting approach peaking at the multiplier of 1.3 with a value of 91.8% in this particular case. Therefore, should indirect lighting without indirect shadows approach be too much for a particular machine, a faked version of indirect lighting would be a respectable compromise.

Table 5.9: Alternative Reductions.

Method	FPS	% Similar
Default Method	55	100.000
Direct Lighting ONLY	681	84.426
Indirect Lighting w/out indirect shadows	161	98.563
Faked Indirect Lighting (1.1x)	681	88.217
Faked Indirect Lighting (1.2x)	681	90.675
Faked Indirect Lighting (1.3x)	681	91.847
Faked Indirect Lighting (1.4x)	681	91.393
Faked Indirect Lighting (1.5x)	681	90.161

5.1.4 Final Analysis on Accurate Shadows Approach

Lastly, we will consolidate all of the previous data and tables in order to determine the most efficient reductions and the most influencing factors on the believability of the rendered scene when using accurate shadows. The benefits of

calculating similarities is that it leads to a way of quantifying quality in comparison to our reference image and thus rank parameter choices based off a combination of performance gains and limited quality impact. It is important to note that changing the resolution size and stretching netted the best similarity results, but due to the introduction of artifacts, those parameter choices will be listed last. It is also important to note that should a smaller resolution size be applicable without the need to stretch to our larger default resolution, this should be the very first choice in increasing performance, because the artifacts are only introduced during the stretching process due to naive sampling.

Table 5.10 will use the similarity values calculated along with the FPS gain in order to rank the parameter choices. This will be done using a simple equation:

$$rankValue = fpsGain/(1 - calculatedSimilarity) \tag{5.3}$$

Table 5.10 tells us a few things about using accurate shadows:

- If a smaller resolution size can be used and not stretched, use it.
- Accurate indirect shadows are nice, but they are the next thing to be sacrificed if better performance is needed.
- If we choose to reduce VPL's, do so by increasing the ray angles, not by reducing hemispheres.
- We can experiment with multipliers and get decent faked indirect lighting if nothing else.

Table 5.10: Final Comparisons.

Method	FPS	% Similar	Rank Value	Figure
Default Method	55	100.000	–	5.1
Faked Indirect Lighting (1.3x)	681	91.847	76.7816	5.6
Indirect Lighting w/out indirect shadows	161	98.563	73.7648	5.5a
Indirect Lighting w/ 15 indirect shadows	87	99.475	60.9524	5.5b
Indirect Lighting w/ 10 indirect shadows	106	98.853	44.4638	5.5c
Direct Lighting ONLY	681	84.426	40.1952	5.3
30 Degree Angle Between VPL Rays	84	97.956	14.1879	5.7a
Reduction down to 1 VPL Per Ray	78	91.591	2.7352	5.7b
Resolution Reduction 640x360	133	99.483	150.8704	N/A

5.2 Integrated Shadows Approach

5.2.1 Goals

After having gathered results from the accurate shadows technique, it was found that although the technique was scalable in that we could get increased performance by adjusting parameters such as decreasing VPL numbers by increasing the angle between rays, decreasing resolution, or decreasing or removing indirect shadows altogether, computers with lower-end GPU's would likely not be able to display smooth indirect shadows. This was due in large part to GPU memory limitations on the maximum supported number of shadow maps more so than performance capabilities. Therefore, this integrated shadows approach takes a different stance on calculating the indirect shadows. Instead of only doing accurate visibility indirect shadows which would require one shadow map per indirect shadow, this approach minimizes GPU memory by limiting the number of indirect shadow maps to 5 instead of 20 and then integrates between them in order to get many more indirect shadows while using less GPU memory.

5.2.2 Memory Analysis

As mentioned earlier, our shadow maps were going to be three times larger than our screen size. This number was chosen due to the introduction of jaggies as seen in figure 5.8. So for rendering a 1280x720 screen, we would be using shadow maps of size 3840x2160. Knowing this, we can calculate the total amount of GPU memory required when rendering a specific number of shadow maps. Assuming that we were going to be using 21 shadow maps (1 for direct and 20 for indirect shadows) and knowing that each value in the shadow maps were going to be 4 byte floats, we can calculate the total amount of GPU memory required:

$$(3840 * 2160 * 21 * 4)/(1024 * 1024) = 664.45MB \tag{5.4}$$

OR

$$(3840 * 2160 * 4)/(1024 * 1024) = 31.64MB \tag{5.5}$$

per shadow map used. The GPU that was used during all of the results gathering had 1GB of GPU memory, so this factor was not a problem. However, some cards have less memory than this. Older cards may still have 256MB or 512MB of GPU memory. With this in mind, this integrated shadows approach attempts to allow these older cards to also be able to render indirect shadows. Therefore, this second approach uses only 5 indirect shadow maps meaning 6 total shadow maps requiring:

$$(3840 * 2160 * 6 * 4)/(1024 * 1024) = 189.84MB \tag{5.6}$$

which would be low enough to run on these older cards. However, 5 indirect shadows by themselves would not look very realistic and would look segregated as seen in table 5.8 and in figure 5.9. Therefore, we integrate between each of these 5 shadow maps as discussed in section 4.3 in order to get more indirect shadows than the previous technique while using less memory.

5.2.3 Results

Keeping table 5.10 in mind, we gathered results from using this shadowing technique by making the most efficient adjustments such as decreasing the resolution and increasing the VPL ray angles as seen in tables 5.11 through 5.13. Also, when using this technique rather than adjusting the number of indirect shadow maps, we now just adjust the number of integration steps between shadow maps.

By looking at table 5.11, we see that by using this technique and using parameters such as 1280x720 and 6485 VPL's, we can render 60 indirect shadows using 4 steps in between each of our 5 indirect shadow maps and get 60 FPS while in our first technique we get 20 indirect shadows using 20 shadow maps and get 55FPS. By using these integrated shadows, we get smoother shadows, better performance, and use less GPU memory. See figure 5.10 for a comparison of the two techniques. The main difference is that the majority of the 60 indirect shadows are using adjusted and possibly inaccurate visibility. Naturally, tables 5.12 and 5.13 show improved performance using reduced resolution and less VPL's. This integrated shadows technique shows better scalability so far, but a true test is to see how both shadowing techniques perform when scene complexity is increased.

5.3 Scene Complexity

So far all of the scenes have been rendered using 36 triangles and 24 vertices, hardly a complicated scene. This section aims at examining the effect of scene complexity on performance. In order to do this, we use a scene with 6565 triangles

Table 5.11: 1280x720, 6485 VPL's, Resulting FPS and number of shadows from adjusting the number of steps.

NumSteps	FPS	Number of Shadows
2	81	40
4	60	60
6	48	80
10	34	120

Table 5.12: 640x360, 6485 VPL's, Resulting FPS from reduced resolution and adjusting number of steps.

NumSteps	FPS
2	137
6	107
10	89

and 3249 vertices as well as a scene with 65542 triangles and 32418 vertices instead to determine the performance impact. Using the parameter choices and alternatives from table 5.10 for the accurate shadows technique as well as our parameter choices from tables 5.11 through 5.13 for the integrated shadows technique, we apply them to our more complicated scenes resulting in tables 5.14 and 5.15.

Table 5.14 shows that for the accurate shadows technique the default method FPS dropped to 15 and 1 from 55 with the increased scene complexity making it not scale well with increased scene complexity. The best method appears to be using a 30 degree VPL ray angle to reduce the VPL count. With this method, the performance dropped from 84 to 70 and 26 with the increased scene complexity meaning that it achieved acceptable FPS with a scene consisting of 65542 triangles while maintaining full indirect shadows. This good performance is due to the fact

Table 5.13: 1280x720, 185 VPL's, 30 degree angle, Resulting FPS from reduced VPL count and adjusting number of steps.

NumSteps	FPS
2	150
6	65
10	42

that with those 65542 triangles come 32418 vertices. By reducing our VPL count, it means that for each of the 32418 vertices we perform 185 VPL indirect lighting calculations rather than 6485 calculations reducing the overall calculations performed in the vertex shader by a significant amount.

Table 5.15 shows overall better numbers while maintaining indirect shadows than table 5.14. Once again, as the scene complexity rises to 65542 triangles, reducing the VPL count by increasing the ray angle maintains great performance. With this technique, we can render 40, 80, and 120 indirect shadows at 58, 39, and 30 FPS respectively with 65542 triangles as opposed to 20 indirect shadows at 26 FPS. Reducing the VPL count through increasing the VPL ray angle allows us to keep good coverage of the scene while quickly reducing the VPL count and thus maintaining good performance when the scene complexity rises.

5.4 Conclusion

With the goal of scalability in mind in terms of portability to different kinds of machines with drastically different computing capabilities, Light Wave was born to allow the user to increase performance with minimal impact to quality using adjustable parameters. However, using accurate shadows required a GPU with

Table 5.14: Scene Complexity Performance - Accurate Shadows.

Scene	Method	FPS
6565/3249	Default Method	15
6565/3249	Indirect Lighting w/out indirect shadows	43
6565/3249	Indirect Lighting w/ 15 indirect shadows	20
6565/3249	Indirect Lighting w/ 10 indirect shadows	25
6565/3249	Direct Lighting ONLY	502
6565/3249	30 Degree Angle Between VPL Rays	70
6565/3249	Reduction down to 1 VPL Per Ray	44
6565/3249	Resolution Reduction 640x360	18
65542/32418	Default Method	1
65542/32418	Indirect Lighting w/out indirect shadows	5
65542/32418	Indirect Lighting w/ 15 indirect shadows	2
65542/32418	Indirect Lighting w/ 10 indirect shadows	3
65542/32418	Direct Lighting ONLY	173
65542/32418	30 Degree Angle Between VPL Rays	26
65542/32418	Reduction down to 1 VPL Per Ray	8
65542/32418	Resolution Reduction 640x360	1

memory of at least 664.45MB to run full indirect shadows. Even using half the indirect shadows would require 348MB of memory. Therefore, integrated shadows were used with GPU memory in mind and limited it to 189.84MB, which would run on any computer worth running it on. By doing this and including integrated shadows as opposed to accurate visibility shadows, this technique can render many more shadows at comparable or even better FPS. It also maintains better performance when scene complexity rises getting a respectable 30 FPS with 120 indirect shadows for a scene with 65542 triangles.

Table 5.15: Scene Complexity Performance - Integrated Shadows.

Scene	NumSteps	Resolution	NumVPL's	FPS
6565/3249	2	1280x720	6485	28
6565/3249	6	1280x720	6485	23
6565/3249	10	1280x720	6485	20
6565/3249	2	1280x720	185	137
6565/3249	6	1280x720	185	63
6565/3249	10	1280x720	185	42
6565/3249	2	640x360	6485	31
6565/3249	6	640x360	6485	30
6565/3249	10	640x360	6485	29
65542/32418	2	1280x720	6485	3
65542/32418	6	1280x720	6485	3
65542/32418	10	1280x720	6485	3
65542/32418	2	1280x720	185	58
65542/32418	6	1280x720	185	39
65542/32418	10	1280x720	185	30
65542/32418	2	640x360	6485	3
65542/32418	6	640x360	6485	3
65542/32418	10	640x360	6485	3

5.5 Images

This section will be used to show images captured from the real-time rendering produced using this method and the accompanying parameters chosen. Figures 5.1 through 5.7 are rendered using accurate shadows, the rest are rendered using integrated shadows.

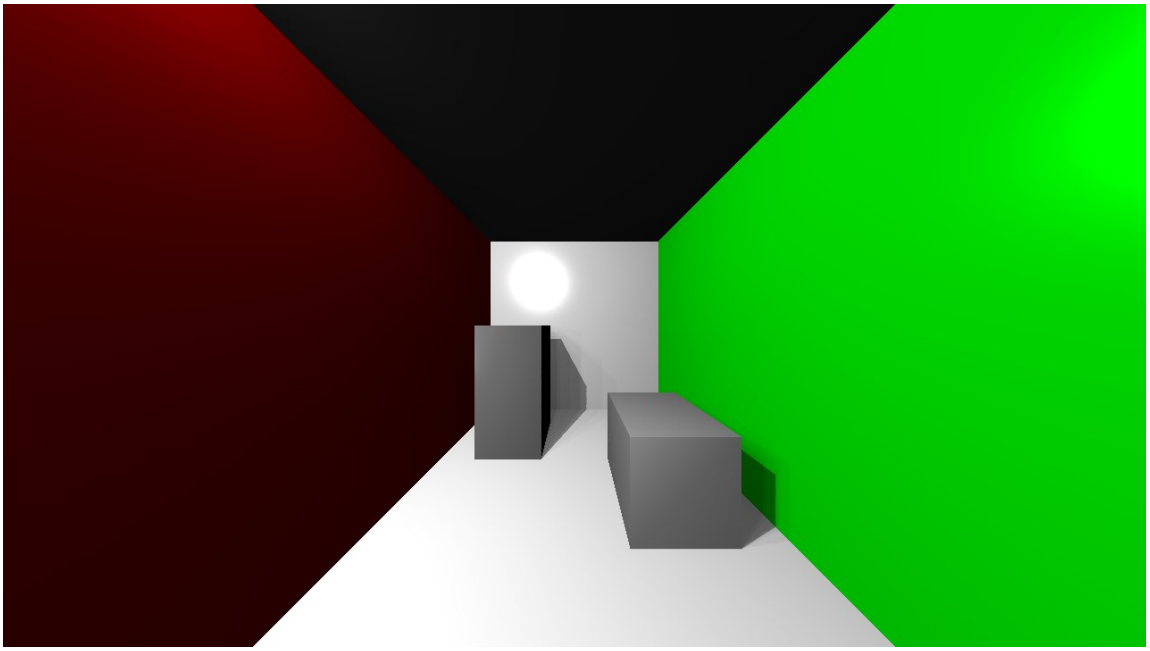
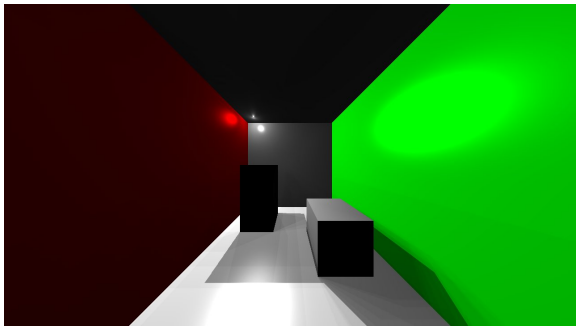
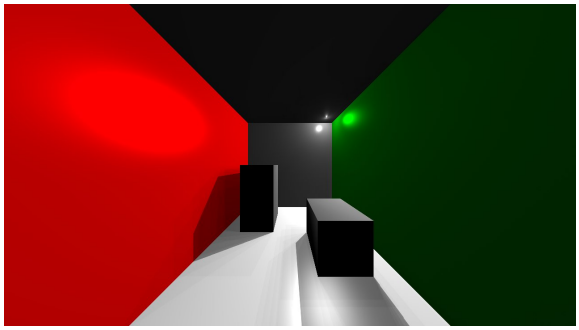


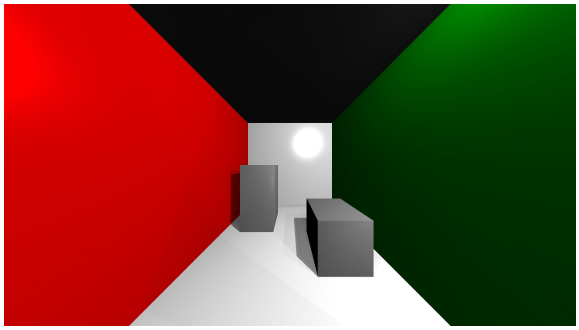
Figure 5.1: Light is at the near upper left corner of the scene. This image uses the default parameters and is used as the reference in the similarity calculations.



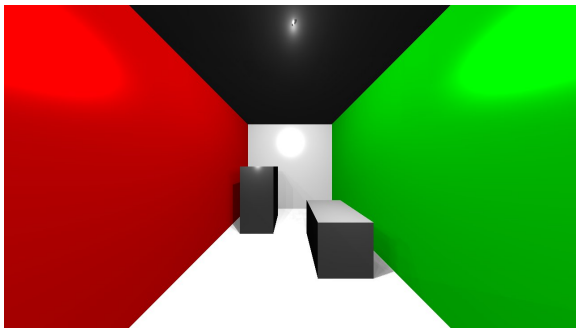
(a) Light is at the far upper left corner of the scene.



(b) Light is at the far upper right corner of the scene.



(c) Light is at the near upper right corner of the scene.



(d) Light is in the upper center of the scene.

Figure 5.2: Additional Rendered Images Using the Default Parameters.

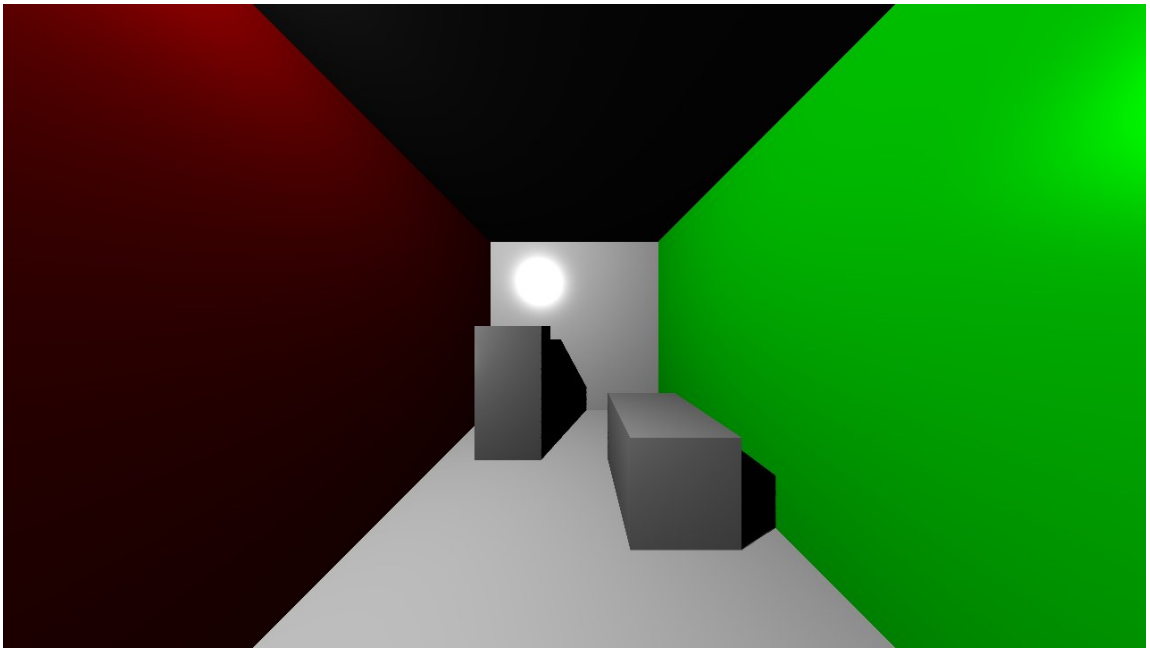
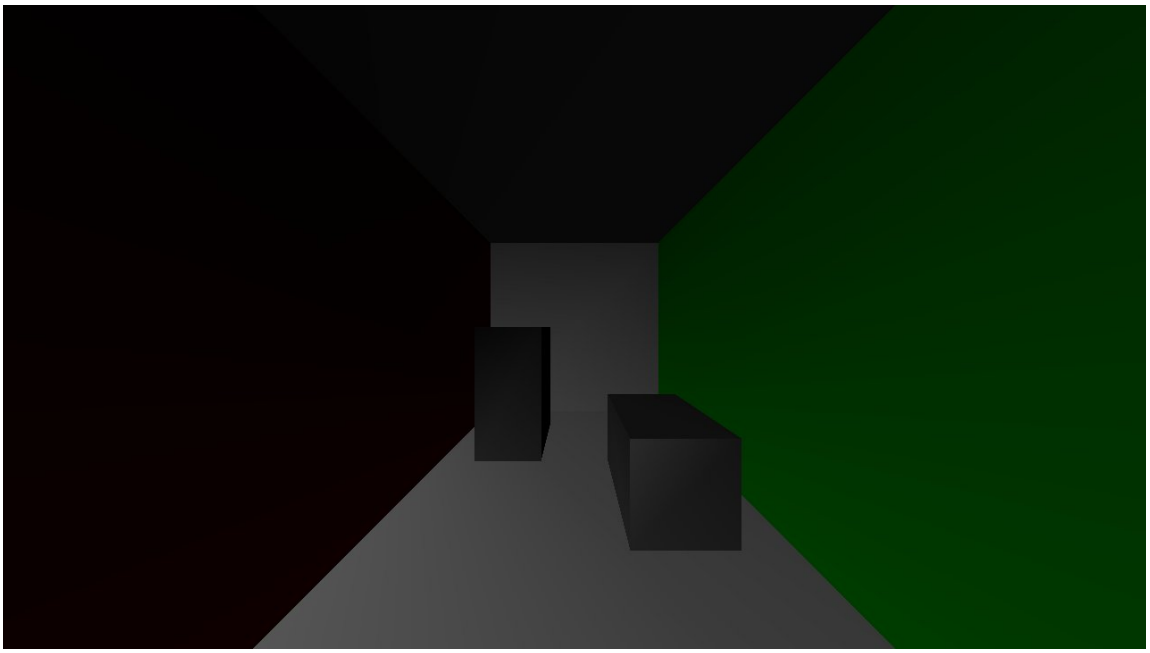
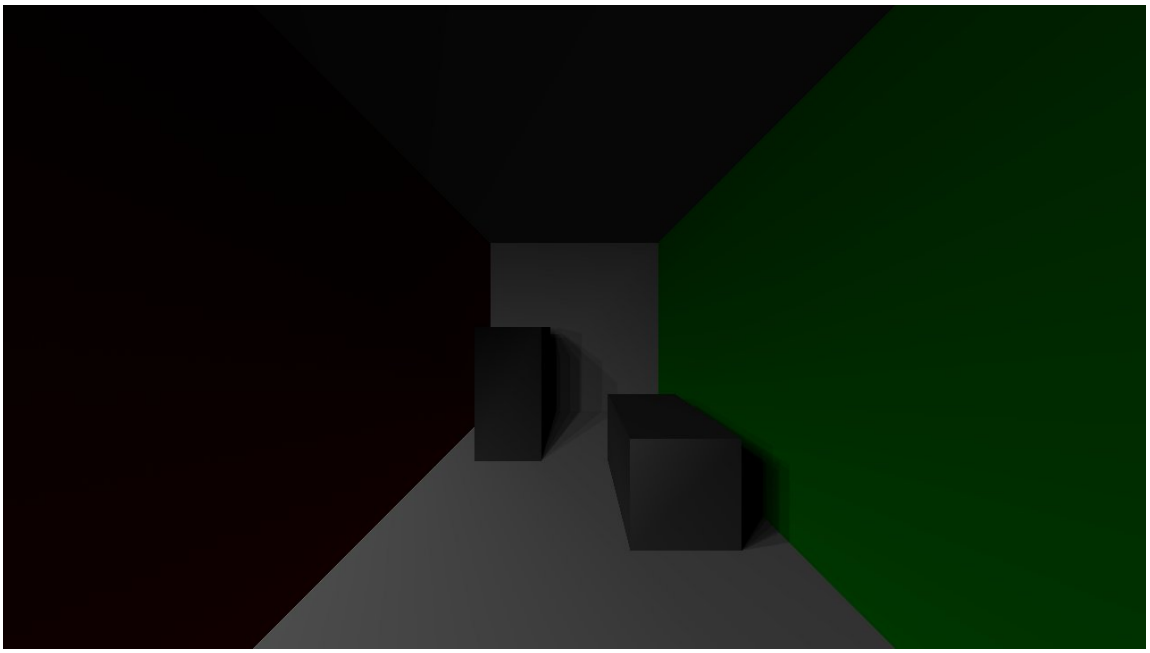


Figure 5.3: Scene rendered with only direct lighting.

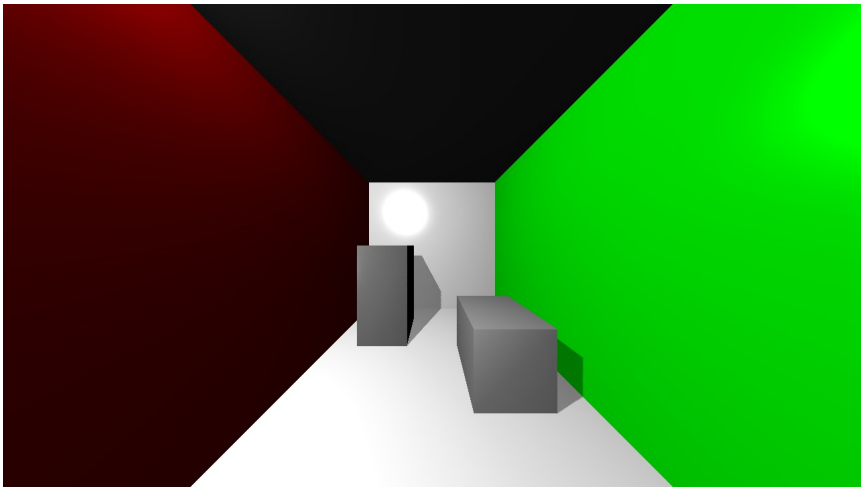


(a) Indirect Lighting ONLY w/out Indirect Shadows

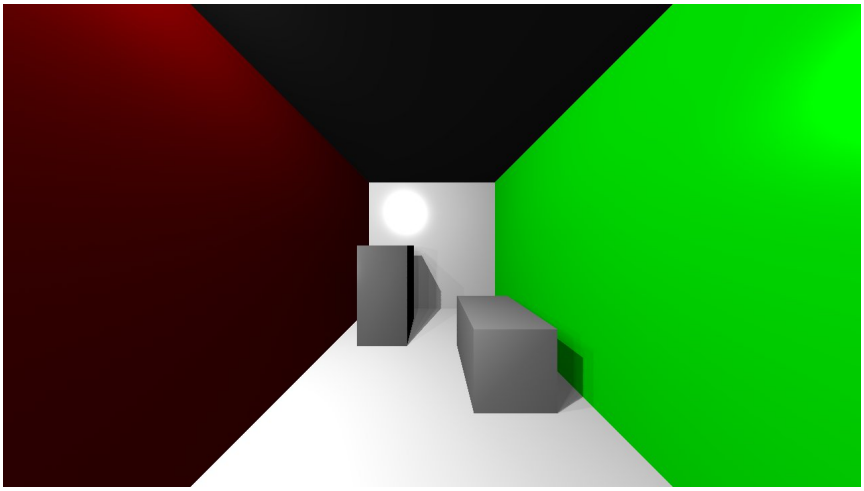


(b) Indirect Lighting ONLY w/ Indirect Shadows

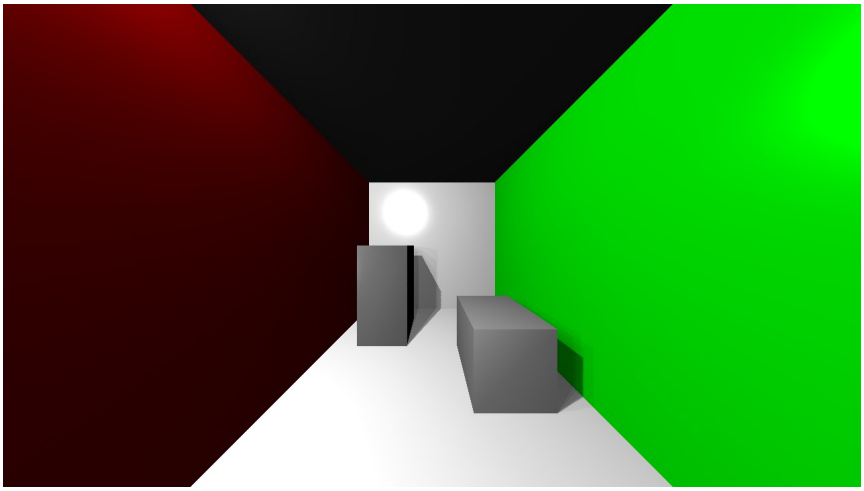
Figure 5.4: Images showing the contributions from the VPL's to indirect lighting (direct lighting is ignored).



(a) Scene Rendered with Indirect Lighting w/out Indirect Shadows.



(b) Scene Rendered with Indirect Lighting w/ 15 Indirect Shadow Maps.



(c) Scene Rendered with Indirect Lighting w/ 10 Indirect Shadow Maps.

Figure 5.5: Varying Indirect Lighting Adjustments.

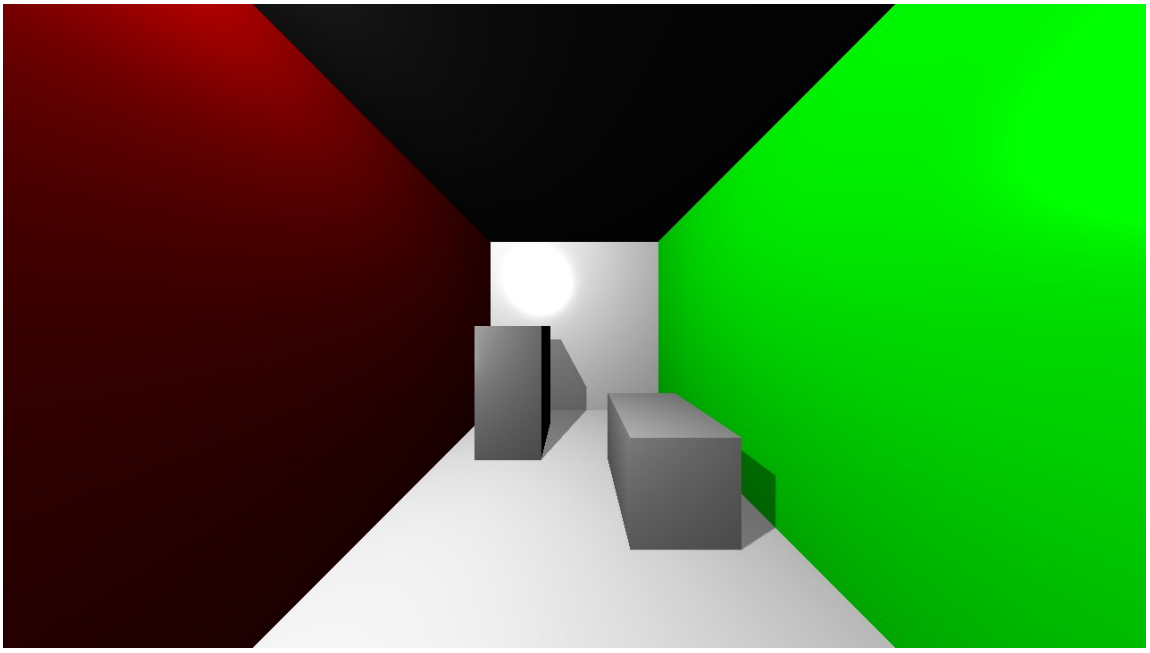
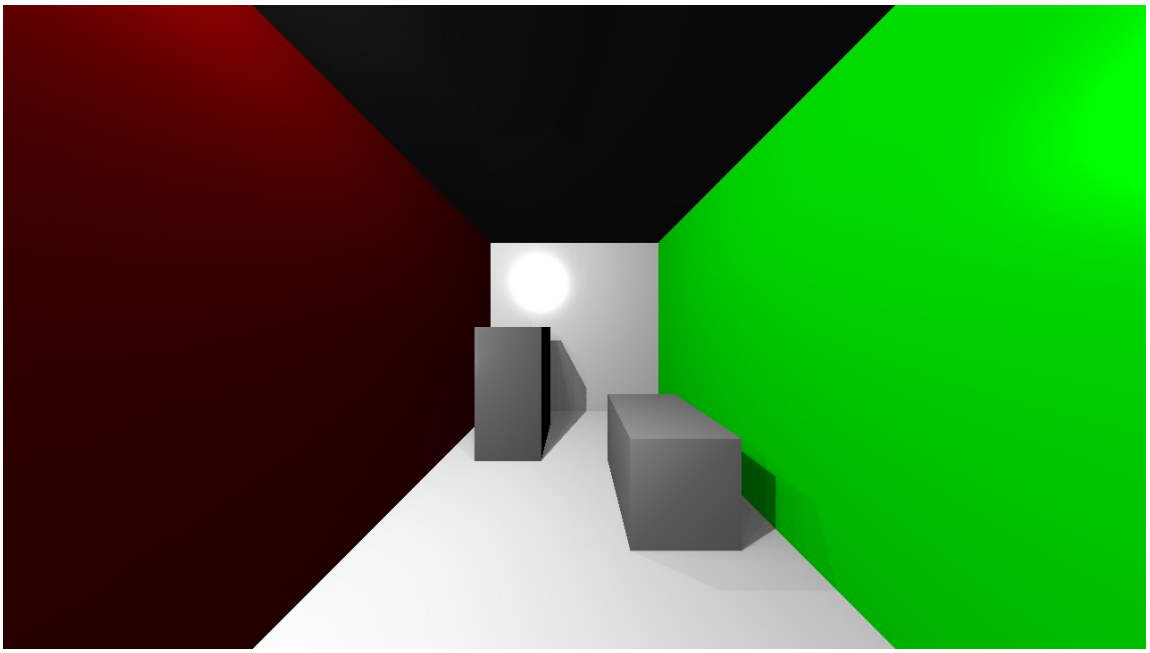
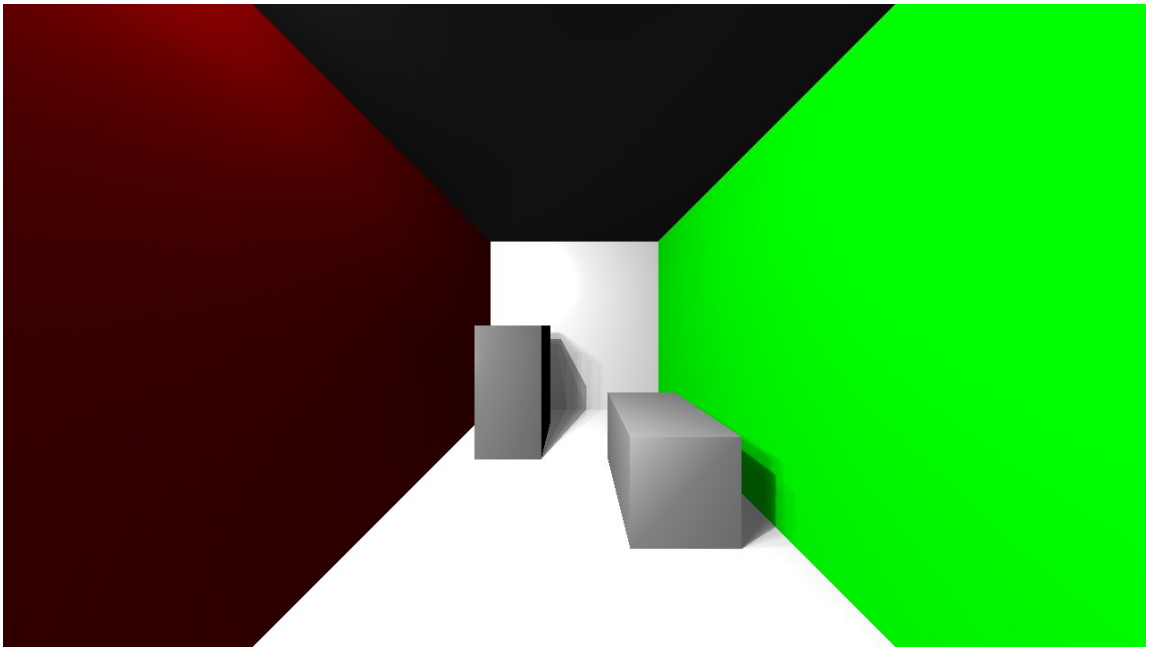


Figure 5.6: Faked Indirect Lighting with a 1.3x Multiplier.

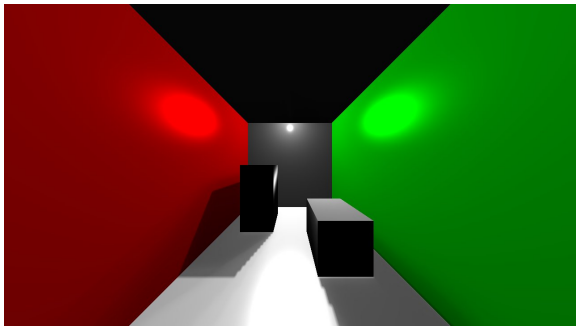


(a) 30 Degree Angle Between VPL Rays.

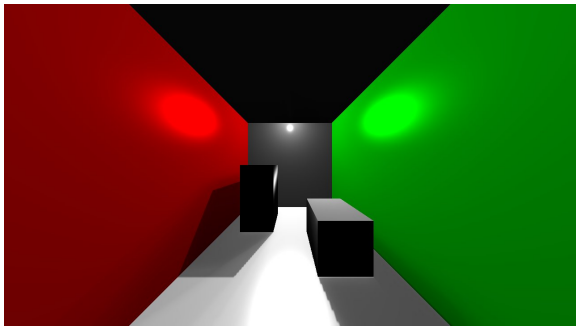


(b) Reduction down to 1 VPL Per Ray (1 Hemisphere)

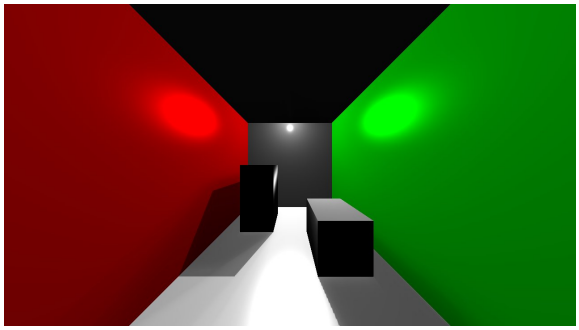
Figure 5.7: Adjusting VPL parameters.



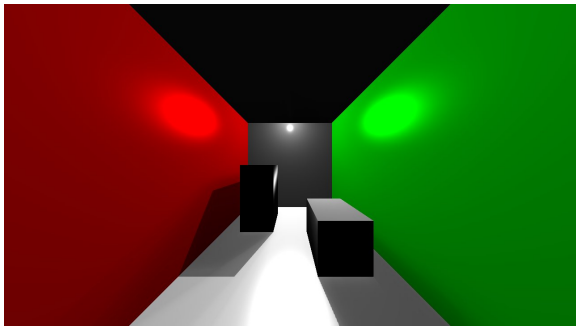
(a) Shadow map using half the resolution of the screen size.



(b) Shadow map using screen size resolution.

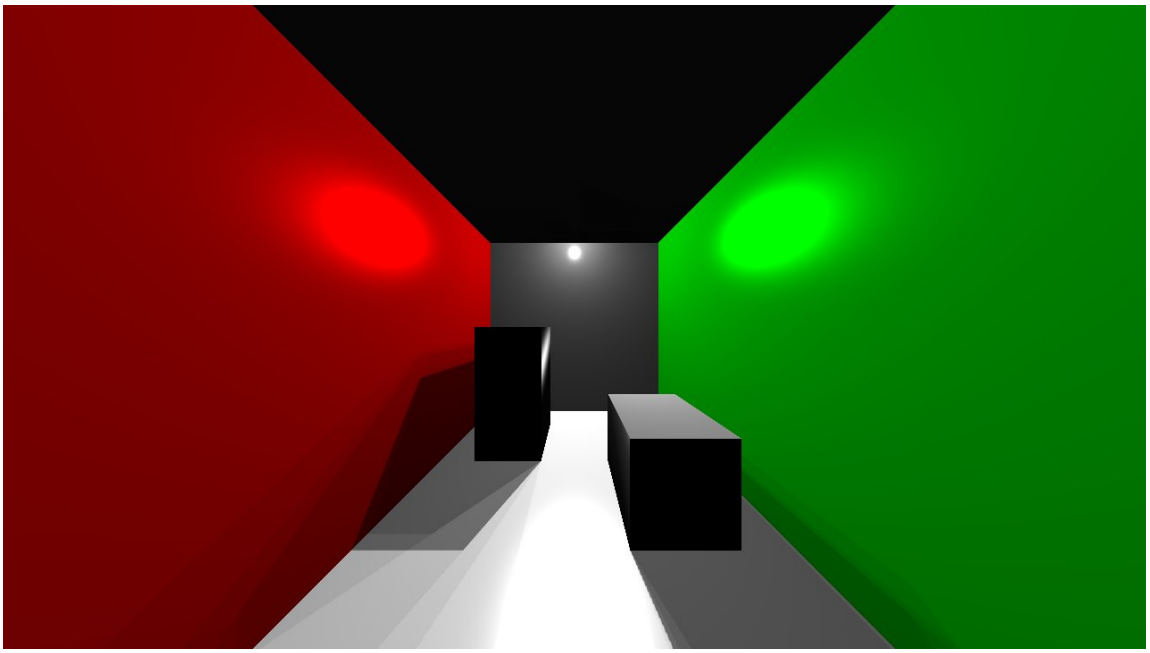


(c) Shadow map using twice the screen size resolution.

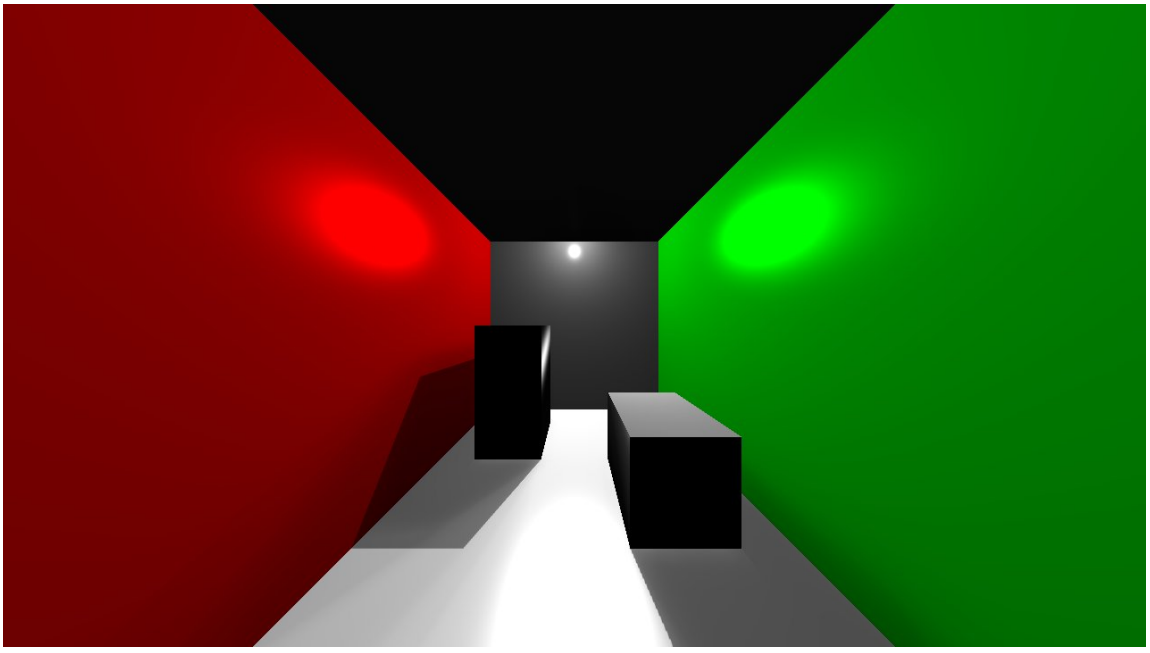


(d) Shadow map using 3X the screen resolution. (Chosen size)

Figure 5.8: Comparison of shadow map resolutions.

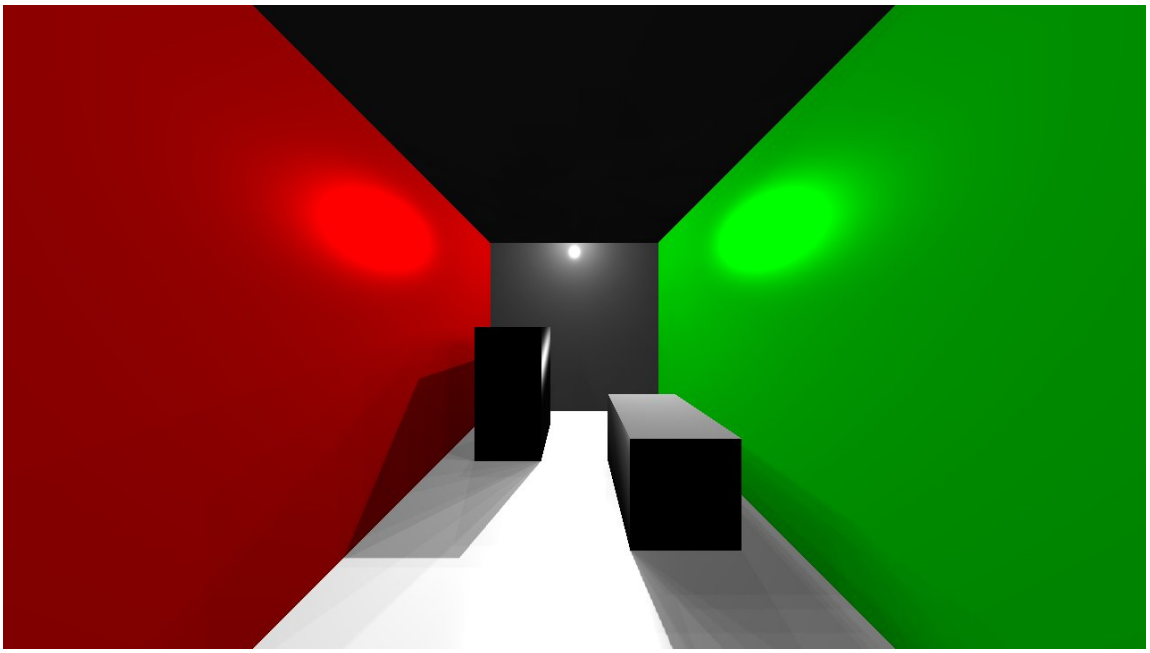


(a) No integration of shadows.

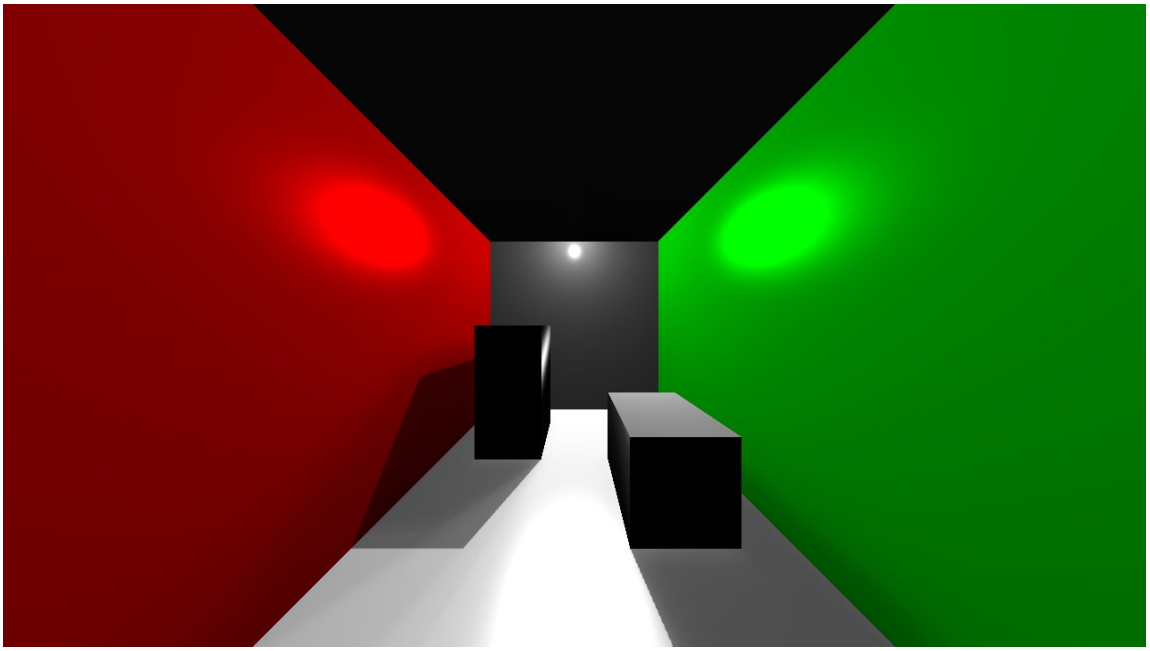


(b) Integration of shadows.

Figure 5.9: Comparison of using 5 shadow maps with integration and no integration.

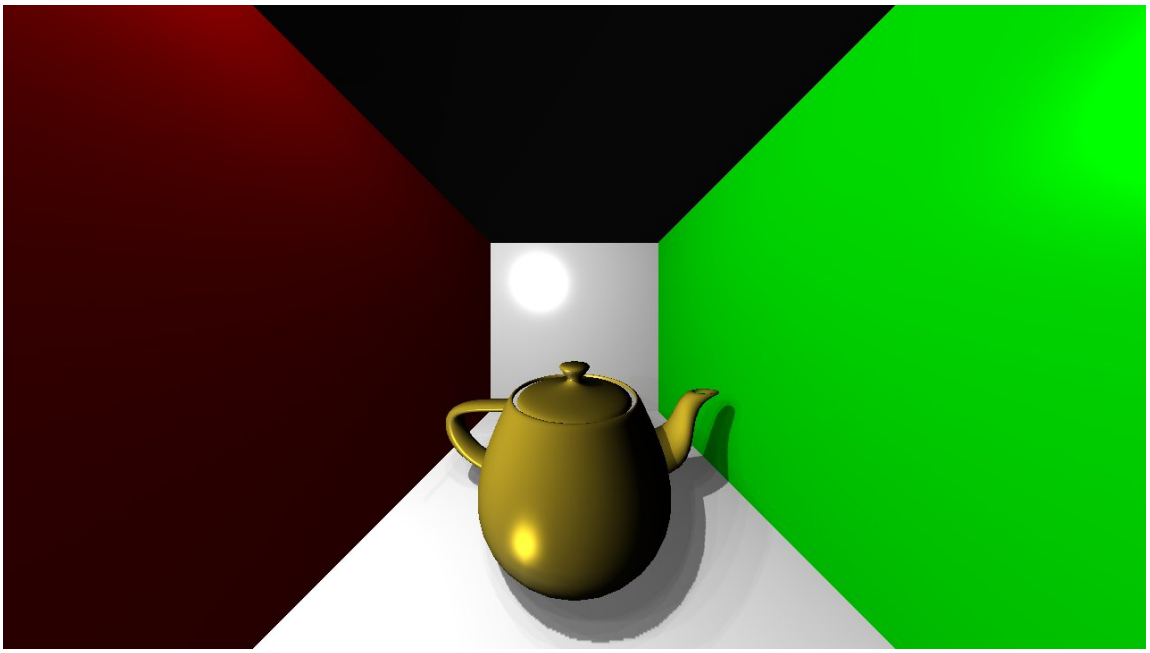


(a) Accurate Shadows - 52FPS

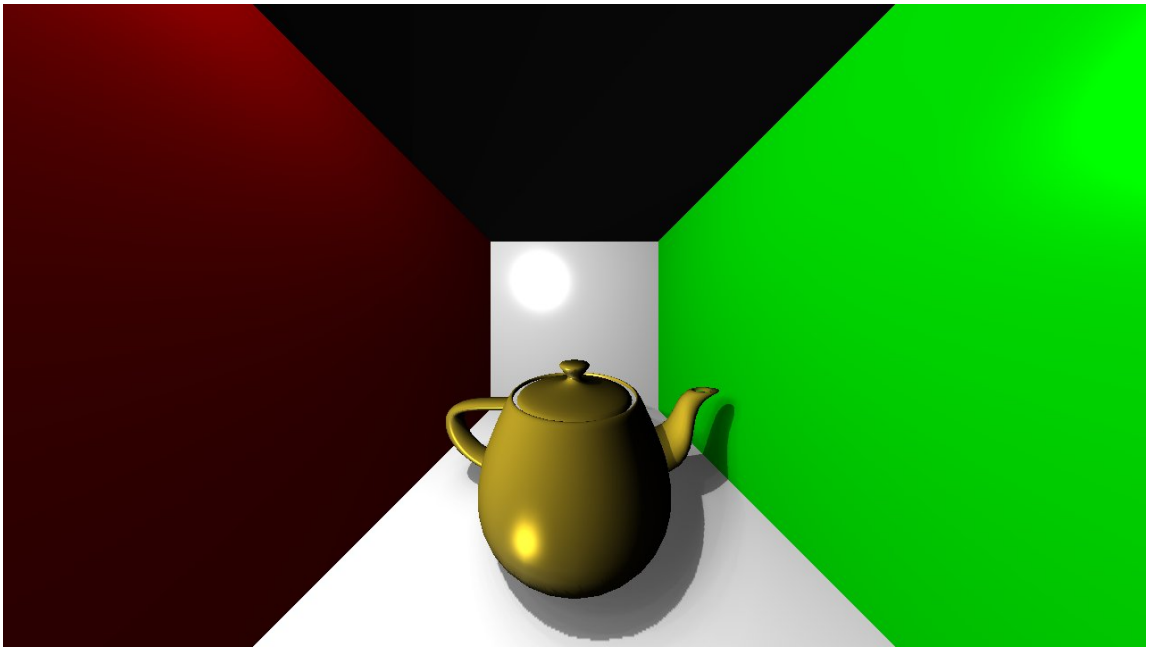


(b) Integrated Shadows - Smoother Shadows - 47FPS (10numSteps,30DegreeAngle)

Figure 5.10: Comparison of the two different shadowing techniques.

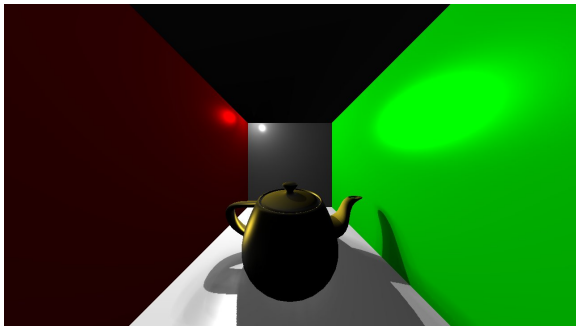


(a) Accurate Shadows

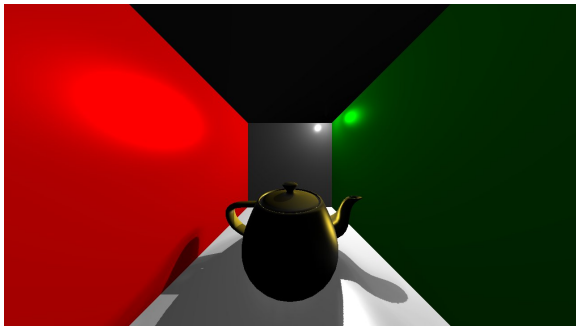


(b) Integrated Shadows - Smoother Shadows

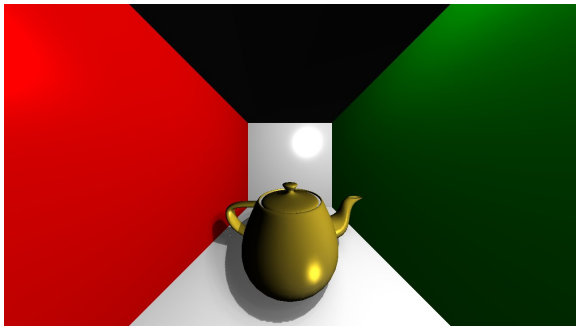
Figure 5.11: 1 Teapot. Scene totals: 6565 triangles and 3249 vertices.



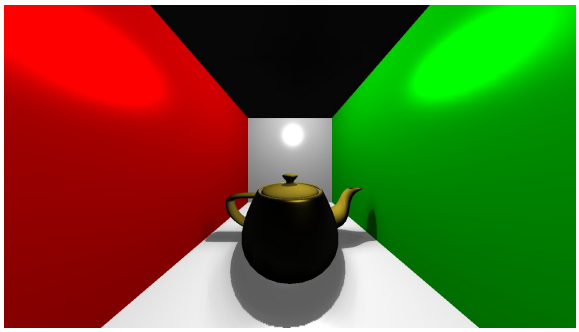
(a) Light is at the far upper left corner of the scene.



(b) Light is at the far upper right corner of the scene.



(c) Light is at the near upper right corner of the scene.



(d) Light is in the upper center of the scene.

Figure 5.12: Teapots rendered using integrated shadows.

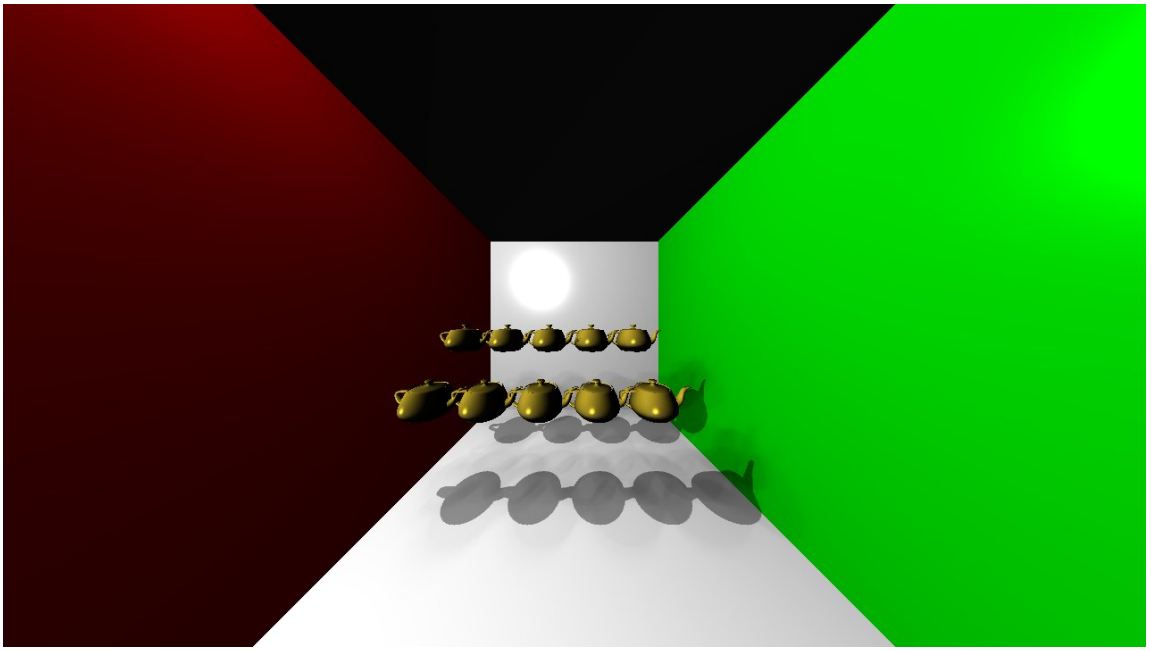


Figure 5.13: 10 Teapots rendered using integrated shadows. Scene totals: 65542 triangles and 32418 vertices.

CHAPTER 6

CONCLUSION

6.1 General Observations

Most importantly, indirect shadows are an expensive luxury. They can require a lot of GPU memory and a lot of computing power depending on the approach. If accurate indirect shadows are a must, a lot of GPU memory must be available in order to make the shadows look good. This is due in part to the need to have the shadow maps be large. They must be large enough compared to the screen size in order to minimize the amount of jaggies on edges of the shadows. For this implementation, three times larger than the screen size appeared to be best, but just like all of the other parameters of this method, this factor can be changed. As computing technology advances further, more GPU's will have more memory available and will then be able to support more shadow maps of larger size, but in the meantime, in order to get more realistic looking images, it is best to approximate indirect shadows.

As mentioned in section 3.2, due to the low frequency nature of indirect shadows, accurate visibility is not required. Also in that study, imperfect shadows were considered the most realistic of the approximated methods. Therefore, the use of integrated shadows as discussed in section 5.2 performed better than accurate shadows and due to the shadow limitations of using accurate shadows looked more realistic. These shadows appeared more realistic due to being smoother and less segregated than the accurate visibility shadows reinforcing the notion that

smoother inaccurate shadows look more realistic than segregated accurate shadows. By using these shadows, the Light Wave method is more scalable in terms of portability to different types of machines with different computing capabilities as well as more scalable when it comes to increased scene complexity.

6.2 Implementation Specific Final Thoughts

Light Wave is a technique of estimating the indirect lighting in a scene in real-time. It is able to do this through the use of virtual point lights or VPL's structured outward from the primary light source in such a way so that the direct light can wrap around objects and hit surfaces that may not be directly visible from the primary light source. These VPL's are structured in hemispheres around the primary light source with extended viewing capabilities. These attributes make the overall flow of the light in the scene to resemble a wave flowing outward from the primary light source in all directions and wrapping around obscuring objects. Therefore, we approximate indirect lighting using only direct lighting calculations from upwards of thousands of light sources. We are only concerned with where each light ray terminates and not what happens afterwards thus simplifying the calculation. By doing this we avoid the infinite number of iterations from using a Neumann series to calculate the infinite bouncing of light throughout a scene.

An aim of Light Wave is that it can be scalable. The implementation allows for the adjustment of parameters to allow the user to increase/decrease performance and thus decrease/increase quality depending on the computing power available and required quality. The implementation will also allow for more accurate renderings provided computing advancements. The default parameter

implementation using accurate shadows as detailed in prior sections leads to 55 frames per second with a simple scene and after reducing VPL count runs a complex scene at 26 FPS on the current machine (see section 3.3 for specifics). It is assumed that some machines will perform slower than ours, therefore, we provided some suggestions on how to modify the parameters in order to maximize performance with limited quality impact in section 5.1.4. For machines that can handle the defaults and crave more realistic renderings, the first thing to increase would be the number of indirect shadow maps to smooth out the indirect shadows.

When using integrated shadows, we were able to render many more shadows than when using accurate shadows at similar FPS. For example, we could render 60 integrated shadows at 60FPS while we could render 20 accurate shadows at 55FPS. Integrated shadows also scaled better with an increase in scene complexity. After similarly reducing the VPL count as with the accurate shadows, while using integrated shadows we were able to render 120 indirect shadows on the most complicated scene at 30FPS as opposed to 20 indirect shadows at 26FPS when using accurate shadows. See tables 5.14 and 5.15 for details. For machines that can handle more, the first thing to increase could be either the number of indirect shadow maps or the number of steps for the integrated shadows to render even more shadows.

Therefore, this implementation was successful in it's goal of approximating indirect illumination in real-time using the GPU. It achieves fairly realistic rendered scenes that are fully dynamic allowing the user to move objects and the light in real-time. It also proves portable with adjustable parameters allowing it to scale to the computing capabilities of the current machine as well as the complexity of the

scene. It also provides two different types of shadowing techniques using either accurate or integrated shadows. Future advances in computing power would allow for specific changes in parameters to allow for more realistic renderings. These include the use of additional VPL's, use of higher resolutions, use of additional indirect shadow maps for either shadowing technique, or additional steps for the integrated shadows technique in order to render ultra-realistic indirect shadows.

6.3 Limitations

Light Wave's limitations include some of the same problems encountered in other VPL-driven techniques. This primarily includes singularities due to the VPL contributions coming from a single discrete location. A limitation when using accurate shadows is that the number of indirect shadows drastically impacts performance due to the use of accurate visibility and the maximum number of shadow maps is limited by the amount of GPU memory available. Integrated shadows can be used instead in order to limit GPU memory required and increase the realism of the indirect shadows. Lastly, in order to keep the scene fully dynamic with full indirect shadows, VPL count needs to be lowered when scene complexity rises, but this can be done by increasing the VPL ray angle with limited impact on the resulting quality.

6.4 Improvements

Apart from the technological advances that would improve Light Wave's results, an interesting improvement that could be made to the technique would be to incorporate the idea of virtual ray lights Novák et al. (2012) to remove the VPL singularities. However, keeping with the spirit of waves, we could have each VPL

be a virtual semicircle light or arc on the VPL hemisphere. So instead of integrating each VPL on a straight line, each VPL would be integrated along this semicircle or arc removing singularities and likely reducing the number of VPL's needed to achieve adequate coverage of the scene as well as expanding the use of light waves. Shadow map memory could be minimized using some texture compression approaches since neighboring entries are likely to be similar and approximations have been shown to be sufficient. The shadow map resolution can be varied across the scene with increased resolution where depths vary greatly and reduced resolution where depths are similar. Additional improvements could include the matrix sampling techniques of Hašan et al. (2007), Ou and Pellacini (2011), or Walter et al. (2005) that would allow higher VPL numbers as well as using neighborhood optimizations found in Dachsbacher and Stamminger (2006) to increase performance.

BIBLIOGRAPHY

Cook, R. L. (1986, January). Stochastic sampling in computer graphics. *Association for Computing Machinery Transactions on Graphics 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, 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. Association for Computing Machinery.

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. Association for Computing Machinery.

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. Association for Computing Machinery.

Hašan, M., F. Pellacini, and K. Bala (2007, July). Matrix row-column sampling for the many-light problem. *Association for Computing Machinery Transactions on Graphics 26*(3), 26:1–26:10.

Immel, D. S., M. F. Cohen, and D. P. Greenberg (1986, August). A radiosity method for non-diffuse environments. *SIGGRAPH Computer Graphics 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, 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. Association for Computing Machinery.

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

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. Association for Computing Machinery 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. Association for Computing Machinery.

Nichols, G., R. Penmatsa, and C. Wyman (2010). Direct illumination from dynamic area lights with visibility. In *Proceedings of the Association for Computing Machinery SIGGRAPH Symposium on Interactive 3D Graphics*

and Games: Posters, I3D '10, New York, NY, USA, pp. 21:1–21:1. Association for Computing Machinery.

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. Association for Computing Machinery.

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. Association for Computing Machinery.

Novák, J., D. Nowrouzezahrai, C. Dachsbacher, and W. Jarosz (2012, July). Virtual ray lights for rendering scenes with participating media. *Association for Computing Machinery Transactions on Graphics 31*(4), 60:1–60:11.

Ou, J. and F. Pellacini (2011, December). Lightslice: matrix slice sampling for the many-lights problem. *Association for Computing Machinery Transactions on Graphics 30*(6), 179:1–179:8.

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

Reeves, W. T., D. H. Salesin, and R. L. Cook (1987, August). Rendering antialiased shadows with depth maps. *SIGGRAPH Computer Graphics 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. *Association for Computing Machinery Transactions on Graphics 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. Association for Computing Machinery.

Serway, R. A. and J. W. Jewett (2004). *Physics for Scientists and Engineers*, Volume 2. Salt Lake City, UT 84109-3250: 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. Institute of Electrical and Electronics Engineers 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 *Association for Computing Machinery SIGGRAPH 2004 Papers*, SIGGRAPH '04, New York, NY, USA, pp. 742–749. Association for Computing Machinery.

Szirmay-kalos, L., B. Aszodi, I. Lazanyi, and M. Premecz (2005). Approximate ray-tracing on the gpu with distance impostors. In *Proceedings of the 16th Eurographics Workshop on Rendering*, EGRW '05, Aire-la-Ville, Switzerland,

pp. 1–10. Eurographics Association.

Wald, I., C. Benthin, and P. Slusallek (2003). Distributed interactive ray tracing of dynamic scenes. In *Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, PVG '03, Washington, DC, USA, pp. 11–. Institute of Electrical and Electronics Engineers Computer Society.

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, pp. 15–24. Eurographics Association.

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

Walter, B., S. Fernandez, A. Arbree, K. Bala, M. Donikian, and D. P. Greenberg (2005, July). Lightcuts: a scalable approach to illumination. *Association for Computing Machinery Transactions on Graphics 24*(3), 1098–1107.

Walter, B., P. Khungurn, and K. Bala (2012, July). Bidirectional lightcuts. *Association for Computing Machinery Transactions on Graphics 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 Computer Graphics 22*(4), 85–92.

Whitted, T. (1980, June). An improved illumination model for shaded display. *Communications of the Association for Computing Machinery 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. Association for Computing Machinery.

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. *Association for Computing Machinery Transactions on Applied Perception 6*(4), 24:1–24:14.

Zhukov, S., A. Iones, and G. Kronin (1998). An ambient light illumination model. In *Proceedings of the 9th Eurographics Workshop on Rendering*, EGRW '98, Vienna, Austria, pp. 45–56. Eurographics Association.