UNIVERSITY OF TWENTE.

Master Thesis

GPU-Based Photon Mapping for Approximate Real-Time Indirect Diffuse Illumination

Author:
Daniël Jimenez Kwast

Supervisors:
Dr. Job Zwiers
Dr. Mannes Poel

August, 2016

Abstract

This research investigates the feasibility of a fully GPU-based photon mapping approach for real-time indirect diffuse illumination. Having designed and implemented a prototype system that uses photon mapping to compute indirect diffuse illumination, we evaluate it by measuring its performance under various configurations and in different scenes. Visual image quality is assessed by comparing produced images with high-quality reference images, using mean squared error and structural similarity measures. Computational costs are assessed by measuring average frame times. In addition to this isolated evaluation, we also compare the prototype system to other systems currently available in a number of popular game engines (relying on the same performance metrics). We conclude that even our unoptimised prototype is capable of computing indirect diffuse illumination in real-time, and that it is competitive with similar systems currently available in the commercial market.

Contents

Chapter 1

Introduction

In the domain of three-dimensional computer graphics, global illumination refers to a group of algorithms that take into account light that is emitted directly from light sources (direct illumination) as well as light that arrives at a surface point via other surfaces in the scene (indirect illumination). It is an important element of realistic computer based image synthesis, because it models physical phenomena that we all observe in daily life. Since nearly all surfaces reflect light, each surface therefore generates indirect illumination for nearly all other surfaces. This interdependence between surfaces is what makes the computation of global illumination so expensive.

Many algorithms have been developed that are able to approximate realistic light transport closely, but these methods often take minutes or hours to render a single image and thus are not suitable for real-time applications (for comparison, a fair upper bound on acceptable frame times for real-time applications is $33\frac{1}{3}$ milliseconds, which would result in 30 frames per second). In recent years, developments in both hardware and software have made dynamic global illumination more feasible for real-time applications. The existing algorithms that can compute global illumination at real-time frame rates make coarse approximations in order to achieve this. Because of this, there is currently a substantial gap in terms of visual quality between the classic global illumination algorithms that are used in off-line rendering and the algorithms that operate in real-time. The current challenge for researchers in the area of real-time global illumination is to further improve the ratio between quality and computational costs.

This research focuses on a small but crucial section of the global illumination landscape, which is the simulation of diffuse inter-reflections between surfaces in a scene. We ignore most other aspects that could be considered a part of global illumination (e.g. indirect specular reflections, refractions and shadows). One aspect that we do include in our rendering system is direct illumination (only diffuse and specular reflections). However, direct illumination is not the focus

of this research, we only include it because our indirect illumination algorithm reuses some of the information that is generated for the direct illumination, and because it allows us to better compare our results with the work of others. The main objective of this research is to investigate the feasibility of a fully GPU-based photon mapping approach for indirect diffuse illumination in real-time applications. To achieve this objective, a rendering system has been designed and implemented that is capable of computing the aforementioned components. Our evaluation of this system concludes that even in its unoptimised form it is capable of rendering real-time indirect diffuse illumination for simple scenes. Moreover, in its current form its performance already seems competitive with other dynamic global illumination systems available on the commercial market.

1.1 Outline

While the table of contents should already provide the reader with a fair idea of how this document is structured, we provide some additional information on the contents of the main chapters below.

- Chapter 2 (Background) lays a theoretical basis in radiometry and briefly discusses how light transport can be modeled. This chapter also introduces the key terms and notations that are used for radiometric quantities throughout the document.
- Chapter 3 (Related Work) discusses some of the related work published in the real-time global illumination field. This chapter will also motivate our choice for a photon mapping approach and clarify how our system differs from similar systems.
- Chapter 4 (Methodology) provides information on the methods that are used in order to complete the research objectives.
- Chapter 5 (Design and Implementation) details the design and implementation of our GPU-based photon mapping prototype.
- Chapter 6 (Evaluation) sets out the methodology and results of the performance evaluation of our prototype system.
- Chapter 7 (Discussion) discusses the results reported in chapter 6.
- Chapter 8 (Conclusion) concludes the thesis and functions as a summary of the work that was performed for this project.

Chapter 2

Background

This chapter is intended to serve as an introduction to radiometry and light transport for readers who have little or no background in these areas. We introduce the problems that are addressed in this research as well as notations and terms that are used throughout the document. Finally, we also briefly discuss how some of the more well-established algorithms tackle light transport simulation.

2.1 Radiometry

Radiometry deals with the measurement of electromagnetic radiation, which includes visible light and consists of a flow of photons. It provides a set of tools that can be used to describe light propagation and reflection. This section will provide a very brief introduction into the field of radiometry and will serve as a foundation for the rest of the study.

Photons exhibit properties of both particles and waves, depending on circumstances. In computer-based rendering, the wave-like properties are mostly ignored. This means that not all physical phenomena can be successfully modelled (e.g. the polarisation and diffraction of photons). One of the wave-like properties that cannot be fully ignored is that each photon has a specific wavelength (or frequency), which influences the interaction with sensors such as the cones and rods of the human eye. The fields of photometry and colorimetry focus on the physical interactions between photons and the human sensors, and the perception of colour. However, we will not delve further into these fields. Instead, we will focus on a few basic radiometric quantities, which will allow us to formally describe distributions of light.

2.1.1 Basic Radiometric Quantities

Radiant flux (sometimes also called *radiant power*, and is denoted as Φ) is equal to the total amount of energy passing through a surface over a period of time. Radiant flux is measured in watts, which is defined as joules per second. The radiant flux of a light source could be measured by summating the energy of all emitted photons in a second.

Irradiance (denoted as E) is the density of *incoming* radiant flux with respect to an area. The irradiance for a surface with area A is expressed as

$$E = \frac{d\Phi}{dA}. \tag{2.1}$$

The notion of measuring radiant flux over a surface area can also be extended to *outgoing* radiant flux. This quantity is called *radiant exitance* and is often denoted as M.

Intensity (denoted as I) describes the directional distribution of energy. Its definition includes the concept of a *solid angle*. A solid angle can perhaps most easily be visualised as the extension of an angle to a three-dimensional sphere. Solid angles are measured in steradians. An entire sphere subtends a solid angle of 4π and a hemisphere subtends a solid angle of 2π. Intensity can be defined as the density of radiant flux per solid angle $\boldsymbol{\omega}$

$$I = \frac{d\Phi}{d\boldsymbol{\omega}}. \tag{2.2}$$

Radiance (denoted as L) combines the ideas of the previously defined quantities and is defined as the radiant flux density with respect to both area and solid angle

$$L = \frac{d^2\Phi}{dA_{\text{proj}} d\boldsymbol{\omega}}, \tag{2.3}$$

where dA_{proj} represents the projection of dA onto a plane perpendicular to the solid angle $\boldsymbol{\omega}$ (i.e. $dA_{\text{proj}} = dA|\cos\theta|$). We can think of radiance as a measure of energy along a single ray.

2.2 Rendering Equation and Light Transport

The rendering equation describes the distribution of radiance in a scene under the assumption that the light has reached a state of equilibrium. The equation was first seen in Kajiya's study, which also introduced the path tracing algorithm [12]. The rendering equation yields the total outgoing radiance at a point on a surface in terms of its emission, the distribution of incident radiance arriving at the given point, and the Bidirectional Scattering Distribution Function (BSDF) of the surface. For now, it suffices to say that a BSDF is a function that

describes how light is scattered from a surface. More precisely, it describes the ratio of incident radiance that is scattered from a given direction toward another specified direction (hence, it is called bidirectional).

We present the rendering equation below and will briefly describe each of its terms. Keep in mind that different forms have been used for this equation. This is only one of many. Luckily, the general idea and structure remains the same. We write the equation in the form

$$\begin{aligned} L_o(x, \boldsymbol{\omega}_o) = & L_e(x, \boldsymbol{\omega}_o) \\ & + \int_{S^2} f(x, \boldsymbol{\omega}_i, \boldsymbol{\omega}_o) L_i(x, \boldsymbol{\omega}_i) \cos\theta_i d\boldsymbol{\omega}_i, \end{aligned} \tag{2.4}$$

where $L_o(x, \boldsymbol{\omega}_o)$ is the radiance leaving the surface point x along direction $\boldsymbol{\omega}_o$. The first term on the right hand side (i.e. $L_e(x, \boldsymbol{\omega}_o)$) is the radiance emitted from point x along direction $\boldsymbol{\omega}_o$. At a high level, we can view the second term as the radiance scattered at point x in the direction of $\boldsymbol{\omega}_o$. This second term consists of an integral over the sphere of possible incoming directions (S^2). For each of these incoming directions, a product of three factors is calculated. The first of these terms, $f(x, \boldsymbol{\omega}_i, \boldsymbol{\omega}_o)$, is the BSDF which we have briefly touched upon. $L_i(x, \boldsymbol{\omega}_i)$ denotes the radiance that is incident along direction $\boldsymbol{\omega}_i$. The final term is a weakening factor, which attenuates the incoming radiance at point x based on the angle between the incoming direction $\boldsymbol{\omega}_i$ and the surface normal $\boldsymbol{n}$ (this angle is denoted as θ_i, the attenuation term is simply the cosine of θ_i). Now that we have clarified the notation used for the rendering equation, we can provide a clearer definition of the BSDF. As mentioned before, the BSDF describes the ratio of incident radiance that is scattered at point x from a given direction $\boldsymbol{\omega}_i$ toward another specified direction $\boldsymbol{\omega}_o$. Formally, this ratio can be described as

$$f(x, \boldsymbol{\omega}_i, \boldsymbol{\omega}_o) = \frac{dL_o(x, \boldsymbol{\omega}_o)}{L_i(x, \boldsymbol{\omega}_i) \cos\theta_i d\boldsymbol{\omega}_i}. \tag{2.5}$$

Note that the rendering equation is recursive; the radiance function appears on both sides of the equation. This means that in order to evaluate the outgoing radiance at some point x, we require the incident radiance at x from all possible directions. Yet, the radiance incident on point x is equal to the outgoing radiance of all other surfaces in the direction of x. Essentially, this means that the incident radiance at a certain point can potentially be affected by the geometry and material properties of any object in the scene.

Now, we introduce the reflectance equation, which is different from the rendering equation in the sense that it only concerns photon reflections (i.e. the transmittance of energy through objects is ignored). We do not show it here because the difference can simply be expressed by replacing the BSDF with a Bidirectional Reflectance Distribution Function (BRDF). In this study, BRDFs are denoted by $f_r(x, \boldsymbol{\omega}_i, \boldsymbol{\omega}_o)$, where the subscript signifies that it only accounts for reflections. Additionally, we use $f_{r,d}$ and $f_{r,s}$ to denote diffuse and specular

BRDFs respectively. The change from BSDF to BRDF also means that the domain changes from a sphere of incoming directions to a hemisphere of directions around surface normal $\boldsymbol{n}$ (this is denoted as $H^2(\boldsymbol{n})$ instead of S^2). The integral in the reflectance equation can be split up into several components as shown by Jensen in [10]:

$$\begin{aligned} L_{o,r}(x, \boldsymbol{\omega}_o) = &\int_{H^2(\boldsymbol{n})} f_r(x, \boldsymbol{\omega}_i, \boldsymbol{\omega}_o) L_{i,l}(x, \boldsymbol{\omega}_i) \cos\theta_i d\boldsymbol{\omega}_i \\ &+ \int_{H^2(\boldsymbol{n})} f_{r,s}(x, \boldsymbol{\omega}_i, \boldsymbol{\omega}_o) \Big(L_{i,c}(x, \boldsymbol{\omega}_i) + L_{i,d}(x, \boldsymbol{\omega}_i) \Big) \cos\theta_i d\boldsymbol{\omega}_i \\ &+ \int_{H^2(\boldsymbol{n})} f_{r,d}(x, \boldsymbol{\omega}_i, \boldsymbol{\omega}_o) L_{i,c}(x, \boldsymbol{\omega}_i) \cos\theta_i d\boldsymbol{\omega}_i \\ &+ \int_{H^2(\boldsymbol{n})} f_{r,d}(x, \boldsymbol{\omega}_i, \boldsymbol{\omega}_o) L_{i,d}(x, \boldsymbol{\omega}_i) \cos\theta_i d\boldsymbol{\omega}_i. \end{aligned} \tag{2.6}$$

We do not delve into the details of equation 2.6, as it is only included to show that there are different components of light transport simulation and that our research only focuses on certain parts. Each of the integrals in equation 2.6 represent these different components; they describe direct illumination, indirect specular and glossy reflections, caustics, and indirect diffuse illumination respectively. Our research focuses only on indirect diffuse illumination. Although we also implement direct illumination for aforementioned reasons, our system will not consider indirect specular reflections or caustics.

Analytically solving equations 2.4 or 2.6 is impossible. However, the integrals can be evaluated numerically. There are two main approaches to the problem. The first is a group of methods known as Monte Carlo integration, which is widely used in ray tracing algorithms. The second group contains methods that are based on the finite element method (e.g. radiosity). Since the latter is not relevant to this research, our analysis only discusses the first group.

Monte Carlo methods are a group of algorithms that rely heavily on the usage of sampling. The main idea is that a large number of random samples can be taken to approximate the distribution of some unknown entity. In the context of light transport, Monte Carlo integration can be used to approximate the distribution of light in a scene by computing a weighted sum over a large number of random samples. With sufficient samples, Monte Carlo estimation will converge to the correct result; too few samples will introduce variance, which manifests itself as noise in the produced image. Variance reduction techniques such as importance sampling and Metropolis light transport can be used to attempt to make the best use of a limited number of samples. Monte Carlo integration (as well as ray tracing in general, including the aforementioned variance reduction methods) is covered in great detail in [20]. In the following paragraphs we will briefly describe the main approaches that are taken in some of the well known ray tracing algorithms in order to help the reader understand the concepts behind these various methods.

In distribution ray tracing [4] we trace multiple rays from each surface point to sample direct lighting from light sources, as well as the contribution of light reflecting between surfaces along with other desired effects. This process of tracing rays is recursive, meaning that if a ray is traced from one surface point to another, multiple rays are generated and traced again from the second surface point. This means that the number of rays will increase dramatically in just a few reflection bounces. In order to cope with the increasing number of rays as reflection level depth increases, the number of rays can be reduced after a few initial levels as these rays are unlikely to have large effect sizes anyway (both due to their diminishing intensity caused by energy absorption, and the already high number of rays).

Path tracing – an algorithm first introduced in Kajiya's rendering equation paper [12] – is a variation of distribution ray tracing. The core concept is that instead of tracing rays and generating multiple rays at each surface intersection, a sample can be computed by evaluating the contribution of a single path along which light may travel, starting from a pixel and ending at a light source (with an arbitrary number of reflections in between). The result is a flattened search space which turns the tree-like search space from distribution ray tracing into a single path. This removes the explosiveness in terms of the number of rays and reduces the computational costs of a single sample. However, a much larger number of samples is needed per pixel and ensuring a good distribution of reflection rays is considerably more difficult.

Finally, photon mapping [11] is an algorithm that is quite different from both distribution ray tracing and path tracing. One of the larger differences is that photon mapping uses forward ray tracing, that is, rays are traced starting from the scene's light sources (as opposed to more conventional ray tracing, in which the tracing of rays originates from the camera). Whenever these rays intersect a surface, a photon is created that represents the incident illumination at that surface point (from the ray direction). After being created, the photons are stored in a photon map, which is often implemented as a structure that allows for efficient sampling of nearby photons (e.g. a kd-tree). The process of tracing rays and creating photons is repeated until absorption (or another termination criterion has been met), after which the photon map can be used to compute indirect illumination. To shade a given surface point, the nearest photons can be retrieved from the photon map, which can then be used to calculate the incident indirect illumination.

Chapter 3

Related Work

A number of varied global illumination algorithms have been examined for this research. An approach to real-time global illumination that is not covered in the following sections is GPU-based path tracing. The Brigade renderer [2] showcases results from such an approach. While these results are impressive, and path tracing seems very attractive with an eye on the more distant future, we think that reducing variance to acceptable levels within real-time constraints is currently not yet possible without very powerful hardware.

In this chapter we briefly go over some of the research that is most relevant to this project. The area of real-time indirect and global illumination is still one that is actively being researched and there is a vast body of work already published. We have restricted our selection to publications that have gained traction and have seen adaptations of their methods implemented in some of the current game engines (video games are currently the largest application domain of real-time global illumination). We also briefly cover Image Space Photon Mapping, which is the publication that has inspired the work that is documented in this thesis.

3.1 Reflective Shadow Maps

Reflective Shadow Maps (RSM) [7] are based on traditional shadow mapping and are computationally the least prohibitive of the algorithms that are considered in this research. It uses the rasterisation pipeline to render the scene from the viewpoint of each of the direct light sources with the goal to generate a set of virtual point lights (VPLs). These VPLs can then be used to approximate single-bounce indirect illumination in the scene. The algorithm features an importance-sampling scheme to select only the most important indirect light sources, and a screen-space interpolation scheme to further reduce

computational costs.

RSM depends on the assumption that light sources have a single centre of projection. It is this assumption that allows the indirect lighting to be computed using the rasterisation pipeline, but this also means that area light sources cannot be supported. RSM accounts for one indirect light bounce and does not handle self-shadowing. However, the algorithm can be extended to handle self shadowing by treating the VPLs as shadow casting lights. But, evaluating a full shadow map for each VPL is much too expensive. Imperfect Shadow Maps can be used instead to greatly reduce costs [21]. Another route would be to simulate self shadowing by using an ambient occlusion technique.

3.2 Light Propagation Volumes

Light Propagation Volumes (LPV) [13][14] is an algorithm that uses 3D grids and spherical harmonics to represent the spatial and angular distribution of indirect light in a scene. RSMs are used to generate a set of VPLs that are projected onto spherical harmonic coefficients, which are in turn injected into the LPV. The distribution of indirect light is propagated through the grid in a cell by cell manner, and is later sampled to approximate indirect illumination.

This is a proven technique that has been used for indirect diffuse illumination in many commercial games. However, the 3D grids that are used are a rather coarse approximation, which introduces accuracy issues (e.g. light may appear to leak through geometry). The algorithm accounts for one indirect light bounce, but can possibly be extended to handle multiple bounces. Ambient occlusion is integrated into the algorithm, and the LPV can also be used to compute volumetric lighting.

One way to attempt to deal with the mentioned accuracy issues is to use nested 3D grids (commonly referred to as Cascaded LPVs). Doing so allows an increase in resolution for nearby objects without increasing the resolution for objects that are farther away. Indirect illumination may also appear to be smeared out due to the light propagation scheme not accurately modelling diagonal light propagation. Finally, the spherical harmonics representation and propagation scheme does not handle glossy surfaces properly. However, the algorithm could be extended to better handle glossy surfaces by manually marching through the LPV along the reflection direction and correcting for the smearing caused by the propagation scheme.

3.3 Voxel Cone Tracing

Voxel Cone Tracing [6][5] uses a sparse voxel octree to store the direct illumination in the scene. Indirect illumination can be sampled from this data structure

by tracing cones in a fashion that is similar to final gathering. The algorithm supports indirect diffuse and specular lighting. Similar to Light Propagation Volumes, Voxel Cone Tracing suffers from light leaking due to the coarse approximation of the scene geometry. The authors that initially proposed the algorithm have reported that it performs better than Light Propagation Volumes, both in terms of quality and performance [6].

Voxel Cone Tracing natively supports single bounce indirect diffuse and specular illumination. While it could theoretically support multiple light bounces by voxelising the scene numerous times, it seems unlikely that this adaptation will be used in real-time applications. The algorithm's complexity and high video memory consumption (roughly 1024 MB for the Sponza Atrium scene) appear to be its most limiting factors.

3.4 Image Space Photon Mapping

Image Space Photon Mapping (ISPM) [17][16] is very similar to traditional photon mapping, but uses the assumptions of point lights and a pinhole camera to execute parts of the photon mapping algorithm in the rasterisation pipeline of GPUs. It supports diffuse and specular lighting, but is not well suited to handle refractive surfaces.

The photon mapping algorithm is a good candidate for the computation of indirect illumination. The algorithm introduces bias by reusing previously computed results for multiple exitant radiance computations. This bias, however, also reduces computational time and high frequency noise, the latter being common in other unbiased Monte Carlo methods. Photon mapping converges to a correct solution as the computational budget increases, while being able to degrade gracefully by trading off variance for blurring in situations where the computational budget is restricted.

ISPM's basis thus lies in a consistent light transport model that is able to handle an arbitrary number of light bounces and arbitrary BSDFs. However, since ISPM is based on shadow mapping and deferred shading, it also inherits their limiting assumptions. It only supports a pinhole camera model, and more importantly, only point light sources. Another limitation is that, while photons can be traced through translucent and refractive surfaces, the radiance cannot be properly estimated for multiple points per pixel without prohibitively expensive depth peeling.

3.5 Contribution

The ISPM papers show that a photon mapping approach can produce convincing results in real-time. The method proposed in this research is similar to the

work in [17][16] in the sense that deferred shading is used to compute direct illumination, and photon mapping is used to compute indirect diffuse illumination. However, our method runs entirely on the GPU, whereas ISPM performs parts of the photon tracing process on the CPU. Therefore, our adaptation would potentially result in an increase in performance since it should better utilise the available graphics hardware. Performing all computational steps on the GPU has an added benefit of requiring less data transfers between main memory and GPU memory, which can again further increase performance. In addition, we use compute shaders instead of the rasterisation pipeline to perform the photon emission step, which allows for a higher degree of flexibility (e.g. photons can be emitted from any type of light source).

Chapter 4

Methodology

As stated earlier, the main objective of this research is to investigate the feasibility of a fully GPU-based photon mapping approach for the computation of indirect diffuse illumination in real-time applications. This chapter describes the methods that are used to complete that objective.

In the first part of this research, we design and implement a prototype system that uses GPU-based photon mapping to compute indirect illumination. The design and implementation of this system is covered in chapter 5. The second part of this research entails the feasibility investigation of the prototype system. This is done by measuring how the system performs, both in terms of visual quality and computational speed, under varying circumstances. In addition, we perform a brief comparative study between our implementation and other methods currently available in a number of game engines.

4.1 Performance Assessment

For applications whose images are intended to be viewed by humans, the best approach for quantifying visual image quality is likely via subjective evaluation. However, a perceptual user study is something that lies outside of the scope of this project. Instead, we make use of quantitative measures that can – to varying degrees of success – predict perceived image quality. The computation of these metrics requires a reference image, which we compute using an offline renderer that is configured to produce high quality images. One of the metrics that we use is the mean squared error (MSE), which can be defined as

$$MSE = \frac{1}{mn} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} \big(I(i,j) - K(i,j)\big)^2, \tag{4.1}$$

where I is an $m \times n$ monochrome image that is used as a reference, and K is its approximation. Since we are dealing with colour images, we compute the MSE for each of the three colour channels and compute the average value. Alongside the raw MSE values, we also report normalised values which are computed by dividing the results of equation 4.1 by the range of values exhibited by the reference image:

$$NRMSE = \frac{MSE}{I_{\max} - I_{\min}}. \tag{4.2}$$

In addition to the MSE metrics, we also use structural similarity as a measure of image quality. This is an approach to image quality assessment proposed by Wang et al. and is reported to better predict perceived image quality [23]. Their research provides details on how a structural similarity index (SSIM) can be computed, which can also be averaged over the image space to produce a mean structural similarity index (MSSIM). In the evaluation of our prototype system, we report measured MSSIM values alongside the MSE metrics. In some cases we also provide images that show the spatial distribution of structural similarity by directly visualising the SSIM.

Now that we have clarified how image quality is assessed, we can turn to performance assessment in terms of computational speed. This is something that is relatively straightforward and does not warrant much clarification; we simply measure the time necessary to render a single frame over a sample set of a thousand frames and report the mean and standard deviation of the given sample set. These time measurements only include the rendering of direct illumination (which includes a render queue), the emission and tracing of photons, and the rendering of indirect illumination. Matters such as handling user input, updating mesh transformations, swapping the front and back buffers, or matters such as window handling are thus *not* included.

4.2 Platform

The results that are reported in this thesis have all been computed on a machine with an NVIDIA GeForce GTX 760 GPU and an Intel Core i7 860 CPU. The operating system used is Microsoft Windows 7 64-bit. NVIDIA Optix 3.9.0 and CUDA 7.5 is what was used to perform the photon emission and tracing steps. OpenGL was used to perform the direct illumination and the radiance estimation step. Compute shaders and shader storage buffer objects are the newest OpenGL technologies that are used, which are core components since version 4.3. We compare our implementation to the following systems: CRYENGINE 5.1.0, Unity 5.3.4f1 Personal, and Unreal 4.11.2. We use the Mental Ray plugin for Autodesk Maya 2016 to produce reference images with which the results of our system, and those of the aforementioned systems, are compared.

4.3 Shading Model

Since our system only deals with reflections, we can use a BRDF instead of a BSDF. The focus of this research is not on BRDF models, but we do include the one that is used in this research here for completeness. The photon mapping parts of our system only model diffuse reflections, for which we use a simple constant Lambertian BRDF

$$f_{r,d} = \frac{c}{\pi}, \tag{4.3}$$

where c is the surface albedo. For specular reflections we use a BRDF similar to one that is used in Unreal 4 according to [15]. The general form is the one used in the GGX microfacet model, proposed by Walter et al. [22]

$$f_{r,s}(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o) = \frac{F(\boldsymbol{\omega}_o, \boldsymbol{h})D(\boldsymbol{h})G(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o, \boldsymbol{h})}{4(\boldsymbol{n} \cdot \boldsymbol{\omega}_i)(\boldsymbol{n} \cdot \boldsymbol{\omega}_o)}, \tag{4.4}$$

where $\boldsymbol{h}$ is the halfway vector between $\boldsymbol{\omega}_i$ and $\boldsymbol{\omega}_o$, which is formally defined as

$$\boldsymbol{h} = \frac{\boldsymbol{\omega}_i + \boldsymbol{\omega}_o}{|\boldsymbol{\omega}_i + \boldsymbol{\omega}_o|}. \tag{4.5}$$

In equation 4.4, F is the Fresnel term that describes how light is reflected from each microfacet, G is the geometrical attenuation factor that accounts for the shadowing and masking of microfacets, and D is the normal distribution function that represents the fraction of facets that are oriented in $\boldsymbol{h}$. Each of these functions is further defined below.

$$F(\boldsymbol{\omega}_o, \boldsymbol{h}) = F_0 + (1 - F_0)(1 - \cos\theta)^5 \tag{4.6}$$

Here, F_0 is the specular reflectance at normal incidence, and θ is the angle of incidence (angle between $\boldsymbol{\omega}_o$ and $\boldsymbol{h}$). The normal distribution function that we use is defined as

$$D(\boldsymbol{h}) = \frac{\alpha^2}{\pi((\boldsymbol{n} \cdot \boldsymbol{h})^2(\alpha^2 - 1) + 1)^2}, \tag{4.7}$$

where α is simply the surface roughness parameter squared (roughness $\in [0, 1]$). Finally, the geometric attenuation factor is defined as

$$\begin{aligned} k &= \frac{(\text{roughness} + 1)^2}{8} \\ G_1(\boldsymbol{v}) &= \frac{\boldsymbol{n} \cdot \boldsymbol{v}}{(\boldsymbol{n} \cdot \boldsymbol{v})(1 - k) + k} \\ G(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o, \boldsymbol{h}) &= G_1(\boldsymbol{\omega}_i)G_1(\boldsymbol{\omega}_o). \end{aligned} \tag{4.8}$$

Chapter 5

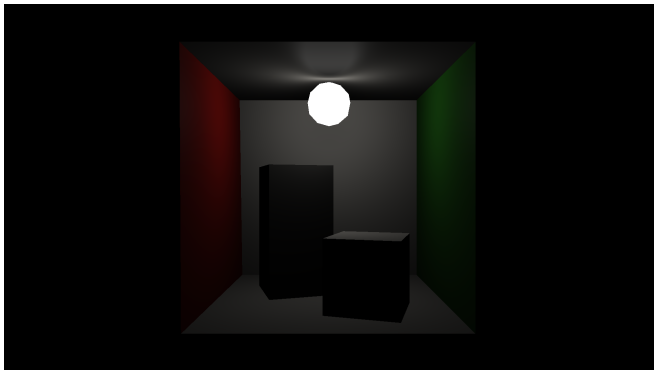
Design and Implementation

This chapter describes the design and implementation of our prototype system. First, we provide an overview of the rendering system as a whole, and afterwards we delve more deeply into its individual components. Since the focus of this research is on indirect diffuse illumination, that is the component on which we provide the most details.

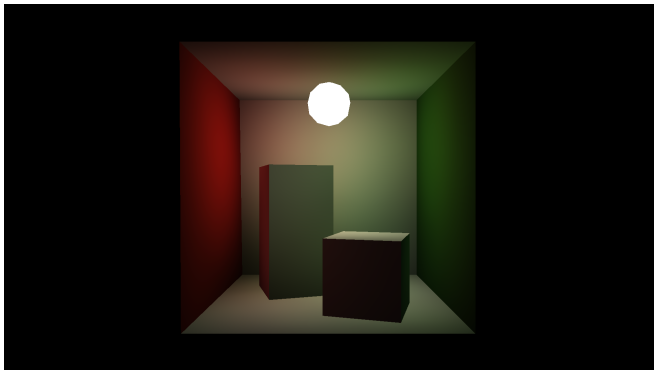
5.1 Rendering Overview

The hybrid rendering system that was developed during this research calculates direct illumination and indirect diffuse illumination separately. In order to compute the radiance at a given point due to direct illumination, we only need to consider the material parameters of its surface and the properties of the light sources that illuminate it (if shadows are taken into account, things become more complicated; our research ignores direct shadows). In contrast, if we want to compute the radiance at a point due to indirect illumination, we need to consider the entire scene (or at least large parts of it) so that we can account for light bouncing around and interacting with multiple surfaces. It is because of this difference that the computation of direct illumination is much less costly than that of indirect illumination. Figure 5.1 shows direct (diffuse and specular) and indirect (diffuse) illumination visualised separately, as well as a final composited render in which both can be observed.

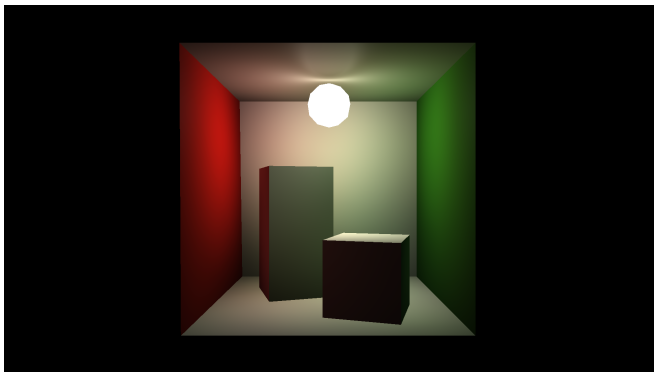
We have already established that our system uses photon mapping to compute indirect diffuse illumination. Direct illumination is computed separately via deferred shading. This is a technique that decouples rendering of the scene from the lighting process in order to avoid having to re-render the scene for



(a) Direct Illumination



(b) Indirect Diffuse Illumination



(c) Composite Image

Figure 5.1: Images showing direct and indirect illumination separately as well as a composite of the two. Note the specular reflections that are present in figure 5.1a (most prominent on the ceiling) and absent in figure 5.1b. Figure 5.1b shows colour bleeding from the red and green walls onto the other surfaces. Figure 5.1c shows a composite of the two components, where both of these effects can be observed.

each light pass. It does so by rendering the scene geometry once; but instead of directly rendering it to a shaded output image, it renders certain information to a special purpose buffer so that it can be accessed in later screen-space lighting passes. This buffer is called a geometry-buffer (often abbreviated as G-Buffer) and typically consists of multiple textures. The contents of the G-Buffer, as well as its layout, varies between applications and lighting models.

Deferred shading allows us to render simple bounding geometry for light sources (the intent here is to have the geometry bound the surfaces to be shaded, *not* just the light source), which means that the scene geometry does not have to be re-rasterised for each light pass. It is of particular importance to our application that having a G-Buffer also allows us to easily fetch geometry and material properties of surfaces during the final shading step of our photon mapping implementation.

Our system's rendering process can be outlined as follows:

1. Render the scene accounting only for direct illumination
   (a) Render scene geometry to G-Buffer
   (b) Scatter light geometry for each light source and accumulate shading results in an output texture
2. Fill photon map with indirect photons
   (a) Start photon paths from light sources
   (b) Trace paths throughout the scene. Create photons at the closest intersections and store them in the photon map (exclude the first bounce, since we are only interested in indirect illumination during this stage)
3. Assign photons from the photon map to 2D screen-space tiles based on their position
4. Compute indirect illumination for each pixel in the output image by sampling nearby photons from the tile in which the pixel is located
5. Composite direct and indirect illumination into a final image

This overview glosses over quite a few intricacies and ignores a number of steps. We provide more details as we delve deeper into different parts of the rendering process in their respective sections. The computation of direct illumination (step 1) is described in more detail in the next section. How the system computes indirect illumination (steps 2-5) is covered in section 5.3.

5.2 Direct Illumination

As mentioned in the previous section, deferred rendering is used to compute the direct lighting in the scene. In this section, we provide a slightly more detailed description of how our deferred renderer implementation is designed.

In order to be able to compute shading due to direct illumination, our deferred renderer first starts a geometry pass. In this pass, information that is required to perform shading is written to the geometry buffer. In subsequent per-light passes, fragment threads are only invoked for pixels that are affected by their respective light source. These threads can access the geometry properties for that given pixel from the G-Buffer.

The goal of the geometry pass is to store any information needed to shade a given pixel in the G-Buffer. The contents of the G-Buffer will vary between applications. The layout that is used in our system is visualised in figure 5.2. Storing this information is similar to how conventional forward rendering is done, with the exception that instead of the shader program performing shading calculations and writing to a single output buffer, it directly writes the relevant information to the different render targets of the G-Buffer.

In order to perform shading, the rendering system performs two passes per light source: a stencil pass, and a light pass. The interplay between stencil and light passes ensure that fragment threads are only invoked for fragments that are subtended by geometry within the bounding volume of the light source. The actual shading is performed in the light passes, which write their results into the accumulation part of the G-Buffer (additive blending is used during the light passes for cases where a surface is lit by multiple light sources).

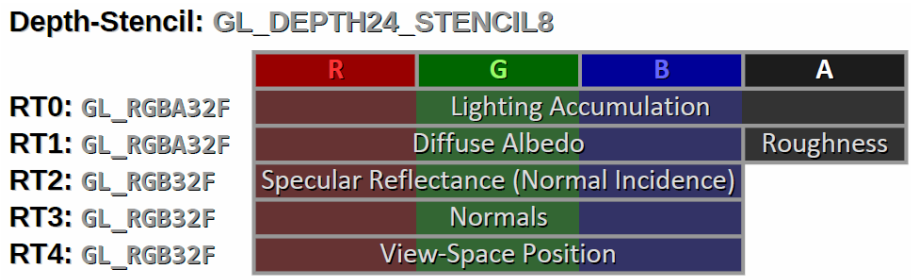


Figure 5.2: Visualisation of the layout that is used for the G-Buffer. It consists of six components; a depth-stencil buffer and five additional render targets. The labels of the different buffers are shown in black lettering, followed by their internal OpenGL format in grey. The graphic shows for what purposes RT0-RT4 are used and how they are structured. Note that the light accumulation component spans the entirety of RT0, whereas diffuse albedo and roughness only make up parts of the RT1 buffer (RGB, and A channels respectively).

```
// Derive l from representative point
vec3 R = normalize(reflect(-V, N));

vec3 centerToR = dot(L, R) * R - L;
vec3 representativePoint = L + centerToR *
  clamp(LightRadius / length(centerToR), 0.0f, 1.0f);

vec3 l = normalize(representativePoint);
```

Listing 5.1: Representative Point Method

```
float a = roughness * roughness;
float aa = clamp(a + LightRadius / (2 * distance), 0.0f, 1.0f);
float sphereNormalization = pow(a / aa, 2);
```

Listing 5.2: Representative Point Normalisation

5.2.1 Area Lights

In order to achieve real-time area lighting for direct illumination, we have adopted the representative point method proposed by Karis [15]. Although this approach could be used for multiple types of area lights (i.e. spheres, disks and rectangles), our system only supports sphere lights. The main idea behind the representative point method is that the contribution of an area light is approximated by treating all of its emitted light as if it were coming from a single representative point. The result of this approach is that the lighting pipeline hardly differs from conventional point light computation. The only thing that changes is the way in which the direction vector towards the light source is calculated, along with an additional normalisation term to ensure that energy is conserved.

So how do we choose this representative point? One way to do this is to choose a point that is likely to have a large contribution to the lighting. A reasonable approximation is to choose a point on the light source that has the smallest angle to the reflection ray. Listing 5.1 shows how this can be computed in GLSL code.

By using this representative point approach, we are effectively widening the specular distribution by the sphere's subtended angle [15]. If we want to maintain energy conservation, then the light's intensity needs to be normalised for this widening. Karis suggests the normalisation term shown in listing 5.2 for the GGX specular BRDF [15].

5.3 Indirect Illumination

In section 5.1 we briefly described the steps that are taken in order to compute indirect diffuse illumination. This section provides more details and delves

deeper into the individual parts of the process.

The first part of the process is to trace paths throughout the scene and to deposit indirect photons in the photon map. We use the NVIDIA Optix framework [19] to perform ray tracing operations on the GPU. The Optix framework is essentially a flexible ray tracing engine that is built on top of CUDA and as such can be of excellent use to projects that require rapid prototyping of ray tracing applications. However, the flexibility of this framework does come at a cost. Parker et al. have reported seeing a performance penalty of around 30% when comparing Optix to a manually optimized ray tracer [19].

The way in which we compute the radiance estimate is heavily based on the work done by Mara et al. [16]. In their research, a number of different methods for performing the photon density estimation have been compared in terms of quality and performance. We use the method that seemed most promising out of the ones examined, which is a 2D-tiled approach. After the photon tracing stage is complete, the photons in the photon map are assigned to 2D screen tiles. We do this by constructing a frustum for each of the tiles, which allows us to perform intersection test between a photon's influence sphere and a tile's frustum. Using this method, we can query for nearby photons in a manner that is easily parrallelisable and translates well to GPUs. Indirect shading is then a matter of iterating over the photons that illuminate a pixel and accumulating their contributions.

Before we delve deeper into how the photon tracing and radiance estimation phases are executed, we will first go over how pseudo random numbers – which are used to sample points on light sources as well as directions in the photon scattering process – are computed on the GPU.

5.3.1 Random Number Generation

Generating samples from a probability density function is something that lies at the basis of all methods that take a Monte Carlo approach. We use the Tiny Encryption Algorithm (TEA) [25] for GPU based random number generation. The study by Zafar et al. shows that the TEA can be used as a random number generator that satisfies all requirements of a good random number generator [26]. In addition, the algorithm allows for a trade off in terms of speed and quality by specifying the number of times it should iterate.

Listing 5.3 shows our implementation of the TEA, and listing 5.4 shows a function that computes a pseudo random normalised float value from a seed (which is generated by the TEA function). Since the number of random samples will be relatively low for our application, we use sixteen rounds for the initial seed generation and eight rounds for the following random numbers.

```
template<optix::uint N>
optix::uint2 __device__ TEA(optix::uint v0, optix::uint v1)
{
  optix::uint sum = 0u;
  for(uint i = 0; i < N; ++i)
  {
    sum += 0x9E3779B9;
    v0 += ((v1 << 4) + 0xA341316C) ^ (v1 + sum) ^ ((v1 >> 5) + 0xC8013EA4);
    v1 += ((v0 << 4) + 0xAD90777D) ^ (v0 + sum) ^ ((v0 >> 5) + 0x7E95761E);
  }

  return optix::make_uint2(v0, v1);
}
```

Listing 5.3: CUDA implementation of TEA for pseudo random number generation.

```
optix::float2 __device__ Rnd(optix::uint2& prev)
{
  using namespace optix;

  uint2 newSeed = TEA<8>(prev.x, prev.y);
  prev = newSeed;

  float2 xi = make_float2(newSeed.x & 0x00FFFFFF, newSeed.y & 0x00FFFFFF);
  return xi / static_cast<float>(0x01000000);
}
```

Listing 5.4: Random number generator based on TEA.

5.3.2 Photon Tracing

As mentioned before, we use the NVIDIA Optix framework to perform ray tracing on the GPU. The first step to our photon tracing sequence is to emit a number of photons from the light sources in the scene. The photon emission phase is computed in an Optix ray generation program, which serves as an entry point for the Optix ray tracing pipeline.

5.3.2.1 Photon Emission

The photon emission kernel is responsible for initialising a payload data structure that is used throughout the path that a ray follows. This payload is accessible and modifiable in following invocations of ray intersection programs. In our case, this data structure is composed of radiant power, a seed for random number generation, and the depth of the current ray in its path (see listing 5.5). Initialising this structure is the first action that is performed by the photon emission program. The radiant power is initialised with the light source's intensity divided by a factor of the number of paths that will be generated. The seed value is initialised by calling the `TEA` function (shown in listing 5.3) with the current kernel invocation's launch index as its seed. Finally, the depth value is set to zero.

The next step is to generate a point on a light source that can be used as the

```
struct RPMH_ALIGN(32) PhotonTracingPRD
{
  optix::float4  power;
  optix::uint2   seed;
  optix::uint    depth;
  optix::uint    padding;
};
```

Listing 5.5: Definition of the payload structure used during the ray tracing process.

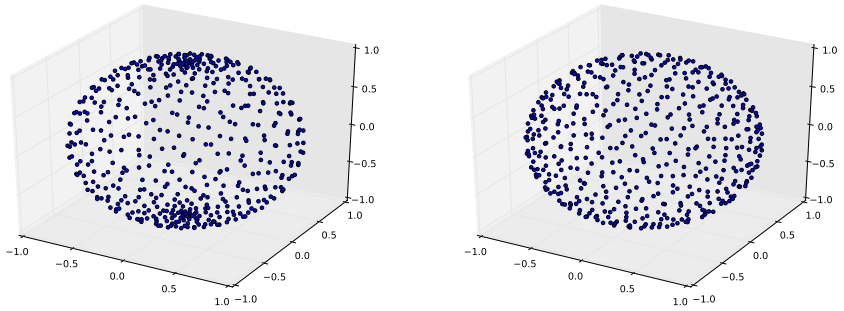
origin for the first ray in its path. Since we are dealing with spherical area lights, we need to be able to generate points on the surface of a sphere. In addition, these points should ideally be spaced out uniformly over the surface of the sphere. The Halton and Hammersley sequences are both low discrepancy sequences that can be used in situations like this [20]. We have opted for the Hammersley sequence since we always know how many samples need to be generated. The two-dimensional Hammersley sequence is based on the simpler one-dimensional van der Corput sequence, which is in turn given by the radical inverse function in base 2. This radical inverse function can be thought of as mirroring the binary representation of its input around the decimal point. The floating point implementation of the radical inverse function, along with example tables and images to make it clearer how this function works can be found in the book by Pharr and Humphreys [20]. See listing 5.6 for an implementation of the radical inverse function in base 2 that uses bitwise operators.

Now that we have described how the low discrepancy number sequences are computed, we can move on to how they are used to sample points on the surface of a sphere light. Since the numbers computed from the Hammersley sequence are two-dimensional, it seems natural to interpret these as spherical coordinates. However, doing so will cause the points to clump up near the poles of the sphere and not have a uniform distribution (this is visualised in figure 5.3a). Instead,

```
float __device__ RadicalInverseUint32(optix::uint bits)
{
  bits = (bits << 16u) | (bits >> 16u);
  bits = ((bits & 0x55555555u) << 1u) | ((bits & 0xAAAAAAAAu) >> 1u);
  bits = ((bits & 0x33333333u) << 2u) | ((bits & 0xCCCCCCCCu) >> 2u);
  bits = ((bits & 0x0F0F0F0Fu) << 4u) | ((bits & 0xF0F0F0F0u) >> 4u);
  bits = ((bits & 0x00FF00FFu) << 8u) | ((bits & 0xFF00FF00u) >> 8u);
  return ((bits >> 8) & 0xFFFFFF) / float(1 << 24);
}

optix::float2 __device__ Hammersley(optix::uint i, optix::uint N)
{
  return optix::make_float2(float(i) / float(N), RadicalInverseUint32(i));
}
```

Listing 5.6: Function implementation that computes elements of the Hammersley sequence using the radical inverse function.



(a) Uncorrected Sphere Sampling **(b)** Corrected Sphere Sampling

Figure 5.3: Images showing the difference in direct interpretation of samples as spherical coordinates (left) and the application of the correction defined in equation 5.1 (right).

we use the following correction

$$\begin{aligned} \theta &= 2\pi\xi_1 \\ \phi &= \cos^{-1}(2\xi_2 - 1), \end{aligned} \tag{5.1}$$

which gives spherical coordinates representing points that do have a uniform distribution over the unit sphere (shown in figure 5.3b)[24]. We then use the following, which yields the Cartesian coordinates of points on the unit sphere.

$$\begin{aligned} u &= \cos\phi \\ x &= \sqrt{1-u^2}\cos\theta \\ y &= \sqrt{1-u^2}\sin\theta \\ z &= u \end{aligned} \tag{5.2}$$

Since not all spherical light sources have the same dimensions as the unit sphere, we scale the Cartesian coordinates with the light source's radius. Adding the light source position to the scaled sample results in the origin of the first ray that corresponds to the sample.

After computing the ray origin for a given photon emission program invocation, all that remains is to compute the ray's direction and to start the ray tracing process. To compute the ray's direction (both in the photon emission phase and in the photon scattering events that we will describe later) we apply importance sampling. Since we only compute indirect diffuse lambertian scattering, importance sampling can be implemented relatively easily by using cosine weighted sampling. Malley's method of generating these cosine weighted points is to compute points uniformly on the unit disk and to then project these onto the unit

hemisphere [20]. Computing points uniformly on the unit disk can be done as such:

$$\begin{aligned} r &= \sqrt{\xi_1} \\ \theta &= 2\pi\xi_2. \end{aligned} \tag{5.3}$$

These coordinates can then be converted to Cartesian coordinates and be projected onto the unit hemisphere by computing

$$\begin{aligned} x &= r\cos\theta \\ y &= r\sin\theta \\ z &= \sqrt{1-\xi_1}. \end{aligned} \tag{5.4}$$

Once the payload structure has been initialised and the ray's origin and direction vectors have been computed, we are ready to continue this ray's tracing process by invoking the `rtTrace` function. The entire photon emission kernel is shown in listing 5.7.

```
#include <optix_world.h>

#include "PhotonTracingTypes.h"
#include "sampling_utils.h"

using namespace optix;

rtDeclareVariable(uint,     launchIndex, rtLaunchIndex, );
rtDeclareVariable(rtObject, topObject, , );

rtDeclareVariable(float,   sceneEpsilon, , );
rtDeclareVariable(uint,    numPaths, , );

rtDeclareVariable(float3,  lightPosition, , );
rtDeclareVariable(float3,  lightIntensity, , );
rtDeclareVariable(float,   lightRadius, , );


RT_PROGRAM void PhotonEmission()
{
  PhotonTracingPRD prd;
  prd.power    = make_float4(lightIntensity / float(0.01f * numPaths), 1.0f);
  prd.seed     = TEA<16>(launchIndex, launchIndex * 1664525u);
  prd.depth    = 0;

  float2 xi = Hammersley(launchIndex, numPaths);
  float3 sphereSample = lightRadius * UniformSampleUnitSphere(xi);
  float3 rayOrigin = lightPosition + sphereSample;
  float3 rayDirection = CosineSampleHemisphere(normalize(sphereSample), Rnd(prd.seed));

  Ray ray = make_Ray(rayOrigin, rayDirection, 0, sceneEpsilon, RT_DEFAULT_MAX);
  rtTrace(topObject, ray, prd);
}
```

Listing 5.7: Implementation of the photon emission program.

5.3.2.2 Photon Scattering

Optix supports three different types of programs that can be invoked based on how the ray intersects the scene. These are the closest hit, any hit, and miss programs. In order to implement photon scattering, we do not need the any hit or miss programs and as such only register a closest hit program that contains our photon scatting behaviour. This is thus a program that is invoked whenever the ray tracing system has found the closest surface with which the ray intersects.

Since we want photons to describe light that is incident on a surface, they need to be created and deposited in the photon map before they reflect off of the surface. At this stage, the photon map is simply an array of structures that describe the photon's position, incoming power, and incident direction (see definition of `OptixPhoton` in listing 5.8). In order to keep the size of this photon structure at 32 bytes, we've chosen to encode the direction vector into two float values here, and later decode the direction vector when the photon structures need to be read. The encoding/decoding scheme we use is the Octahedral Normal Vector (ONV) method as described by Meyer et al. and Cigolle et al. [18][3] (it is called oct instead of ONV in the work by Cigolle et al.). We show the implementation of the encoding function in listing 5.9 (the decoding function is shown in listing 5.16 on page 35).

Since we are dealing with a large number of threads running in parallel, we need a system in place that guarantees each thread is able to safely deposit photons into the photon buffer. To achieve this we use a shared counter that is atomically incremented whenever a thread wishes to deposit a photon. This

```
1  struct RPMH_ALIGN(32) OptixPhoton
2  {
3    optix::float3 position;
4    optix::float3 power;
5    optix::float2 direction;
6  };
```

Listing 5.8: Definition of photon during the photon tracing process.

```
1  float2 __device__ SignNotZero(const float2& v)
2  {
3    return make_float2((v.x >= 0.0f) ? +1.0f : -1.0f, (v.y >= 0.0f) ? +1.0f : -1.0f);
4  }
5
6  float2 __device__ ONVEncode(const float3& v)
7  {
8    float2 p = make_float2(v) * (1.0f / (abs(v.x) + abs(v.y) + abs(v.z)));
9    return (v.z <= 0.0f) ? ((1.0f - make_float2(abs(p.y), abs(p.x))) * SignNotZero(p)) : p;
10 }
```

Listing 5.9: Functions that encode `float3` direction vectors to two floats using the ONV method.

counter's old value is then used as an index for the photon buffer. When a photon has been deposited in the photon buffer – and the maximum path depth has not yet been reached – the kernel generates new random numbers, selects an outgoing direction using importance sampling, attenuates the photon power and finally updates the payload structure. After this, the ray tracing process can continue along its path, possibly resulting in additional program invocations and eventually resulting in the photon buffer being filled with indirect photons. The entirety of the photon scattering program is shown in listing 5.10.

5.3.3 Radiance Estimate

The objective of the radiance estimate step is to approximate the outgoing radiance that is directed towards the camera from a given surface. The method applied in this research is a 2D-tiled approach based on the work of Mara et al. [16]. We summarise the process here briefly and provide more details in the following sections.

The screen is first divided into a number of two dimensional tiles whose bounds are used to construct frustra. These frustra are in turn used to assign photons to certain tiles based on whether the photons and tile frustra intersect. During the indirect shading of a given pixel, nearby photons can be sampled by iterating over the photons that are stored in the tile encompassing that pixel. Using the set of nearby photons, we can approximate the indirect diffuse component of the reflectance equation (shown in equation 2.6 on page 8). This process will be described in more detail later on; first, we delve into how photons are assigned to the specific tiles.

5.3.3.1 Photon Counting and Tile Insertion

For this part of the algorithm, the objective is to create a data structure that allows us to sample photons that are spatially near to (areas of) surfaces in the scene that are represented by pixels in the G-Buffer. The use case is that while a given pixel is being shaded, we want to be able to iterate over a relatively small subset of nearby photons. The research by Mara et al. [16] compares several methods that do this in real-time, both in terms of visual quality and performance. Based on the results of their research, we have chosen to adopt a 2D-tiled approach. This method divides the viewport into a number of 2D-tiles. The bounds of these tiles are used to create frustra that are used to assign photons to the corresponding tile based on whether their influence sphere intersects the tile's frustum.

Most of the radiance estimate is performed using OpenGL compute shaders. This means that the tiled photon map will consist of several shader storage buffers. The first of these buffers, which we call the photon buffer, is the raw set of photons that is the result of the photon tracing stage (this buffer is

```
#include <optix_world.h>

#include "PhotonTracingTypes.h"
#include "sampling_utils.h"

using namespace optix;

rtBuffer<OptixPhoton>  outputBuffer;
rtBuffer<uint>         photonCountBuffer;

rtDeclareVariable(rtObject,  topObject, , );
rtDeclareVariable(float,     sceneEpsilon, , );
rtDeclareVariable(uint,      maxPathDepth, , );
rtDeclareVariable(uint,      maxNumPhotons, , );

rtDeclareVariable(float3,  Kd, , );
rtDeclareVariable(float,   lambertianProbability, , );

rtDeclareVariable(Ray, ray, rtCurrentRay, );
rtDeclareVariable(float, t_hit, rtIntersectionDistance, );
rtDeclareVariable(PhotonTracingPRD, prd, rtPayload, );

rtDeclareVariable(float3, shadingNormal, attribute shadingNormal, );
rtDeclareVariable(float3, geometricNormal, attribute geometricNormal, );

RT_PROGRAM void ClosestHit()
{
  float3 hitPosition = ray.origin + t_hit * ray.direction;

  // We are only interested in storing indirect photons
  if(prd.depth > 0)
  {
    uint photonIndex = atomicAdd(&photonCountBuffer[0], 1);

    // Break if output buffer is full
    if(photonIndex >= maxNumPhotons) return;

    // Deposit photon into output buffer
    OptixPhoton photon;
    photon.position = hitPosition;
    photon.power = make_float3(prd.power);
    photon.direction = ONVEncode(ray.direction);
    outputBuffer[photonIndex] = photon;

    // Break if maximum path depth has been reached
    if(prd.depth >= maxPathDepth) return;
  }

  float3 omega_i = ray.direction;
  float3 shadingNormalWS =
    normalize(rtTransformNormal(RT_OBJECT_TO_WORLD, shadingNormal));
  float3 geometricNormalWS =
    normalize(rtTransformNormal(RT_OBJECT_TO_WORLD, geometricNormal));
  float3 N = faceforward(shadingNormalWS, -omega_i, geometricNormalWS);

  float2 xi = Rnd(prd.seed);
  float3 omega_o = CosineSampleHemisphere(N, xi);
  float3 color = make_float3(prd.power) * Kd / M_PIf / lambertianProbability;
  prd.power = make_float4(color, 1.0f);

  prd.depth++;

  Ray newRay = optix::make_Ray(hitPosition, omega_o, 0, sceneEpsilon, RT_DEFAULT_MAX);
  rtTrace(topObject, newRay, prd);
}
```

Listing 5.10: Implementation of the photon scattering process.

currently copied from GPU memory to main memory and back to GPU memory since we are switching from a CUDA buffer to an OpenGL buffer). The second buffer contains indices to the photons in the photon buffer. The tiled structure is introduced by the third buffer, which we call the tile metadata buffer. This buffer contains elements that each represent a tile and describe where the photon indices that belong to this tile can be found (this is achieved by keeping track of the number of photons that intersect this tile as well as an offset into the photon index buffer). See listing 5.11 for the GLSL declaration of these buffers.

To avoid having to over-allocate the photon index buffer, we have split the tiling process into two phases. These two phases are very similar to one another; they construct frusta based on tile bounds and perform intersection tests between these and the photon's influence spheres. The first phase, however, only counts the number of photons that intersect a tile's frustum and stores this information in the tile metadata buffer. After all kernels have completed the photon counting process, we iterate over the elements in the tile metadata buffer on the CPU, setting the index offsets to their correct values by keeping count of the total number of indices in all preceding tiles. At this point the photon index buffer is allocated so that it can be populated during the tile insertion pass.

As mentioned previously, the photon counting and tile insertion kernels are very similar to one another. Thus, we will only go through the implementation of the tile insertion pass. Listing 5.12 partially shows the code that is used as the tile insertion kernel. Both the photon counting and tile insertion passes

```
struct Photon
{
  vec4 position;
  vec4 power;
};

struct TileMetadata
{
  uint offset;
  uint numPhotons;
};

layout (binding = 0, std430) buffer photonBuffer
{
  Photon photons[];
};

layout (binding = 1, std430) buffer photonIndexBuffer
{
  uint photonIndices[];
};

layout (binding = 2, std430) buffer tileMetadataBuffer
{
  TileMetadata tileMetadata[];
};
```

Listing 5.11: GLSL declaration of the buffers and their accompanying structs that are used during the radiance estimation phase.

```glsl
#version 430
#define USE_HIGH_RES_TILES
include(Commons.c.glsl)

layout (binding = 0) uniform sampler2D depthStencil;

uniform uvec2 gBufferDimensions;
uniform mat4 projectionMatrixInv;
uniform mat4 viewMatrix;
uniform uint numPhotons;

shared uint localIndexCounter;
shared uint localZMin;
shared uint localZMax;

include(PhotonTilingUtils.c.glsl)

void main()
{
  uint localIndex = gl_LocalInvocationIndex;
  uint tileIndex = gl_WorkGroupID.x + gl_WorkGroupID.y * gl_NumWorkGroups.x;

  // Initialize shared variables
  if(localIndex == 0)
  {
    localIndexCounter = 0;
    localZMin = 0xffffffff;
    localZMax = 0;
  }

  barrier();
  ComputeLocalDepthBounds();

  barrier();
  float zMin = uintBitsToFloat(localZMin);
  float zMax = uintBitsToFloat(localZMax);
  uint offset = tileMetadata[tileIndex].offset;

  vec4 frustumEqn[4];
  CreateTileFrustumEqn(frustumEqn);
  for(uint i = localIndex; i < numPhotons; i += NUM_THREADS_PER_TILE)
  {
    Photon photon = photons[i];
    vec4 photonPos = viewMatrix * vec4(photon.position.xyz, 1.0f);

    if(photonPos.z + zMin < photonInfluenceRadius &&
      -photonPos.z - zMax < photonInfluenceRadius)
    {
      if((SignedDistanceFromPlane(photonPos, frustumEqn[0]) < photonInfluenceRadius) &&
        (SignedDistanceFromPlane(photonPos, frustumEqn[1]) < photonInfluenceRadius) &&
        (SignedDistanceFromPlane(photonPos, frustumEqn[2]) < photonInfluenceRadius) &&
        (SignedDistanceFromPlane(photonPos, frustumEqn[3]) < photonInfluenceRadius))
      {
        uint idx = atomicAdd(localIndexCounter, 1);
        photonIndices[offset + idx] = i;
      }
    }
  }
}
```

Listing 5.12: Implementation of the tile insertion kernel.

are started by calling `glDispatchCompute` with the number of tiles in the x and y directions as the number of work groups in the respective directions. We use a tile size of 8×8, which means that each work group consists of 64 threads. Threads within a work group can communicate with each other through shared variables, which for the most part behave as global variables for all kernel invocation within the same work group. However, we do have to manually initialise these shared variables and synchronise the threads so that shared variable visibility is ensured. We synchronise the threads and ensure shared variable visibility by calling the `barrier()` function at certain locations (see lines 31 and 34). This function forces synchronisation between all kernel invocations in the work group, meaning that execution within the work group will not proceed until all other invocations have reached the barrier. Once the barrier function exits, all shared variables will be visible to the other invocations of the work group. The shared variables themselves are declared at lines 12 to 14 and are initialised by a single designated kernel invocation within the work group at lines 24 to 29.

Before we construct the frustra we first determine the minimum and maximum depth value that is present in the G-Buffer within the bounds of the tile. Each kernel invocation can correspond to a single pixel from the G-Buffer. This means that all we have to do is read the depth value for the given thread from the G-Buffer and perform atomic minimum and maximum operations on the shared variables that keep track of the lowest and highest depth values. This is done in the `ComputeLocalDepthBounds()` function, of which the implementation is shown in listing 5.13. Note that the `localZMin` and `localZMax` variables are unsigned integers despite our depth values being stored as floats. This is due to the fact that the atomic operations that can be used on shared variables only support signed and unsigned integers. Thus, we reinterpret the depth value as an unsigned integer before applying the atomic operations (see line 8 in listing 5.13) and again reinterpret the unsigned integer as a float after we have found the minimum and maximum values (see lines 35-36 in listing 5.12). The `uintBitsToFloat` and `floatBitsToUint` functions preserve the floating-point bit-level representation.

```
1  void ComputeLocalDepthBounds()
2  {
3    vec2 texCoord = vec2(gl_GlobalInvocationID.xy + 0.5f) / vec2(gBufferDimensions);
4    float z = texture(depthStencil, texCoord).r;
5
6    if(z != 0.0f)
7    {
8      uint linearZ = floatBitsToUint(LinearizeDepth(z));
9      atomicMin(localZMin, linearZ);
10     atomicMax(localZMax, linearZ);
11   }
12 }
```

Listing 5.13: Implementation of the `ComputeLocalDepthBounds()` function.

The final part of the tile insertion kernel is to construct frustra based on the tile bounds, and to perform intersection tests between them and the photon influence spheres. Each kernel invocation creates plane equations for the four sides of the frustum by calling the `CreateTileFrustumEqn()` function (see listing 5.14 for the implementation details). The construction of the frustum plane equations, as well as the methods we use for photon-tile intersection tests, are based on the light culling methods shown in the work by Harada et al. in [9] (at the time of writing AMD has also published code samples that show additional examples of how this can be done).

We divide the photons over the kernel invocations by using a for-loop that starts

```
vec4 CreatePlaneEquation(vec4 v1, vec4 v2)
{
  return vec4(normalize(cross(v1.xyz, v2.xyz)), 0.0f);
}

float SignedDistanceFromPlane(vec4 p, vec4 planeEqn)
{
  return dot(planeEqn.xyz, p.xyz);
}

void CreateTileFrustumEqn(out vec4 frustumEqn[4])
{
  uvec2 correctedWindowDimensions = gl_WorkGroupSize.xy * gl_NumWorkGroups.xy;
  vec2 corrDim = vec2(correctedWindowDimensions);

  uvec2 pMin = gl_WorkGroupSize.xy * gl_WorkGroupID.xy;
  uvec2 pMax = gl_WorkGroupSize.xy * (gl_WorkGroupID.xy + 1);

  vec4 frustum[4];
  frustum[0] = ProjectionToView(vec4(
          pMin.x / corrDim.x * 2.0f - 1.0f,
          pMin.y / corrDim.y * 2.0f - 1.0f,
          1.0f, 1.0f));

  frustum[1] = ProjectionToView(vec4(
          pMax.x / corrDim.x * 2.0f - 1.0f,
          pMin.y / corrDim.y * 2.0f - 1.0f,
          1.0f, 1.0f));

  frustum[2] = ProjectionToView(vec4(
          pMax.x / corrDim.x * 2.0f - 1.0f,
          pMax.y / corrDim.y * 2.0f - 1.0f,
          1.0f, 1.0f));

  frustum[3] = ProjectionToView(vec4(
          pMin.x / corrDim.x * 2.0f - 1.0f,
          pMax.y / corrDim.y * 2.0f - 1.0f,
          1.0f, 1.0f));

  for(uint i = 0; i < 4; ++i)
  {
    frustumEqn[i] = CreatePlaneEquation(frustum[i], frustum[(i + 1) % 4]);
  }
}
```

Listing 5.14: Implementation of functions used to create frustum plane equations and perform distance tests.

on the local thread index (this is the index of a thread within the work group) and is incremented by the number of threads per tile (see line 41 in listing 5.12). Each photon's position is then transformed into view space and a series of tests are performed to determine if the photon's influence sphere intersects the frustum (lines 43-57 in listing 5.12). If a photon's influence sphere is found to be intersecting the frustum, we increment the shared `localIndexCounter` variable and insert the photon's index into the photon index buffer (lines 54-55 in listing 5.12).

After the photon counting and tile insertion passes have finished executing, the photon map (which now consists of the photon buffer, the photon index buffer and the tile metadata buffer) will be filled with the necessary information needed to perform the indirect shading pass.

5.3.3.2 Shading Indirect Illumination

Computing the outgoing radiance at a visible surface in the scene is now relatively straightforward. For a given pixel we can now determine what the corresponding tile is, sample the nearby photons by iterating over the relevant parts of the photon map, and finally perform shading for the given pixel. Seeing as the number of photons that will be used will generally be relatively low, it is important that the radiance estimate is filtered so that edges of the photon influence spheres do not become discernible. There are of course multiple ways in which this can be done, but we have chosen to use a simple cone filter as described by Jensen [11]. This filter assigns a weight, w_p, to photons based on the distance between the photon and the surface area that is being shaded. The photon weights are computed as

$$w_p = 1 - \frac{d_p}{kr}, \tag{5.5}$$

where d_p is the distance between the photon and the surface area being shaded, k is a filter constant that characterises the filter, and r is the maximum distance allowed between the photon and the surface area. The normalisation term for this filter is $1 - \frac{2}{3k}$, which means that the radiance can be approximated by computing

$$L_r(x, \boldsymbol{w}_o) \approx \frac{\sum_{p=0}^{N} f_r(x, \boldsymbol{w}_i, \boldsymbol{w}_o)\Phi_p(x, \boldsymbol{w}_i)w_p}{(1 - \frac{2}{3k})\pi r^2}. \tag{5.6}$$

Listing 5.15 shows the GLSL implementation of the radiance estimation function. The BRDF evaluation is relatively simple since we only compute the diffuse component of indirect illumination. Most of the function revolves around iterating over the photons that are relevant for the given tile, retrieving information from the photon buffers and transforming this data so that equation 5.6 can be evaluated correctly. Note that the implementation looks slightly different from equation 5.6. This is due to the filter constant k, which we have left at 1.

```
vec3 RadianceEstimate(uint tileIndex, vec3 albedo, vec3 N, vec3 positionVS)
{
  TileMetadata tile = tileMetadata[tileIndex];
  uint indexOffset = tile.offset;
  uint numPhotonsInTile = tile.numPhotons;

  vec3 accumulation = vec3(0.0f);
  for(unsigned int i = 0; i < numPhotonsInTile; ++i)
  {
    Photon photon          = photons[photonIndices[indexOffset + i]];
    vec3 photonPosition    = photon.position.xyz;
    vec3 photonPower       = photon.power.xyz;
    vec3 photonDirection   = ONVDecode(vec2(photon.position.w, photon.power.w));

    vec3 photonPositionVS = (viewMatrix * vec4(photonPosition, 1.0f)).xyz;
    float dist = distance(photonPositionVS, positionVS);
    if(dist > photonInfluenceRadius) continue;

    vec3 L = -normalize(rotationMatrix * photonDirection);
    float NdotL = clamp(dot(N, L), 0.0f, 1.0f);
    float photonWeight = 1 - (dist / photonInfluenceRadius);
    accumulation += photonPower * photonWeight * albedo / PI * NdotL;
  }

  accumulation /= (1.f - 2.f / 3.f) * PI * photonInfluenceRadius * photonInfluenceRadius;
  return accumulation;
}
```

Listing 5.15: GLSL implementation of the radiance estimation function.

```
vec2 SignNotZero(vec2 v)
{
  return vec2((v.x >= 0.0f) ? +1.0f : -1.0f, (v.y >= 0.0f) ? +1.0f : -1.0f);
}

vec3 ONVDecode(vec2 e)
{
  vec3 v = vec3(e.xy, 1.0f - abs(e.x) - abs(e.y));
  if(v.z < 0.0f) v.xy = (1.0f - abs(v.yx)) * SignNotZero(v.xy);
  return normalize(v);
}
```

Listing 5.16: GLSL implementation of functions that decode two floats to a `vec3` using the ONV method.

Additionally, we show the implementation of the `ONVDecode` function in listing 5.16. This function decodes two float values to a three-component float vector. To see how the photon direction is encoded see listing 5.9 on page 27.

5.3.4 Further Approximations

The indirect illumination scheme we have described so far would still not achieve acceptable frame rates. In order to improve performance we introduce another approximation that exploits the low frequency nature of indirect illumination. We simply render the indirect illumination to a low resolution render target (we use a size that is one-fourth of the size of the G-Buffer) and sample this

during the final pass that combines the direct and indirect illumination. Using linear interpolation will for the most part provide good results that are nearly identical to the radiance estimate at full resolution. However, the results will deviate greatly around geometry edges which will in turn appear blurry. To combat this, we use a simple edge detection scheme and recompute the radiance estimate around these edges. This edge detection scheme is based on finding differences in normals and positions of visible surface areas. To do this we write these normals and positions to low resolution textures in addition to the estimated radiance. During the final pass that combines direct and indirect illumination, we sample the normals and positions from both the low resolution textures and the G-Buffer (full resolution). The normals are compared using a dot product, whereas the positions are checked using a distance function. If the normals and positions are similar enough, we sample the indirect illumination from the low resolution texture using linear interpolation. If either the normals or positions differ too much we recompute the radiance estimate.

We show the implementation of the pass that renders to a low resolution texture array in listing 5.17. The implementation is relatively simple; we read the necessary information from the G-Buffer (lines 26-28), perform the radiance estimate using the function that was shown in listing 5.15 (line 30), and finally store the estimated radiance, normal and position in the low resolution texture array (lines 32-34).

The final pass is shown in listing 5.18. This kernel reads the normals and positions from both the G-Buffer and the low resolution texture array (lines 28-31), evaluates whether the precomputed radiance estimate can be used (lines 33 and 36) and responds accordingly (lines 38-39, and line 43). After the indirect illumination has been computed, we read the direct illumination from the G-Buffer, combine it with the indirect illumination and finally write the output back to the light accumulation part of the G-Buffer, which will be displayed at the end of the frame.

While the approximation that was introduced in this section is generally a substantial performance increase, we should note that it does introduce a dependency on scene complexity. In complex scenes with many edges (e.g. trees, mesh fences) it could prove to be ineffective due to the radiance having to be re-estimated for larger parts of the scene. We visualise the edges found by this method in all of the scenes that are used in our performance evaluation, which also includes some scenarios that are pathological for this edge detection scheme. Finally, it should be kept in mind that our edge detection scheme is still incomplete. While it performs reasonably well in the detection of edges in geometry, it completely ignores edges that are caused by changes in surface materials. This is something that could be included in our method (by checking for similarity in material properties in addition to normals and positions), but it is possible that at that point a more general edge detection scheme (e.g. a Sobel operator) would be more effective.

```glsl
#version 430

include(Commons.c.glsl)

layout (binding = 2) uniform sampler2D RT1; // Kd + Ns
layout (binding = 3) uniform sampler2D RT2; // Ks + Ni
layout (binding = 4) uniform sampler2D RT3; // Normals
layout (binding = 5) uniform sampler2D RT4; // Position

layout (binding = 6) uniform writeonly image2DArray LowResTextureArray;

uniform mat4 viewMatrix;
uniform mat3 rotationMatrix;
uniform uvec2 indirectTextureDimensions;

const float PI = 3.14159265f;

include(RadianceEstimate.c.glsl)

void main()
{
  uint tileIndex = gl_WorkGroupID.x + gl_WorkGroupID.y * gl_NumWorkGroups.x;
  ivec2 imageCoord = ivec2(gl_GlobalInvocationID.xy);
  vec2 texCoord = vec2(imageCoord + 0.5f) / vec2(indirectTextureDimensions);

  vec3 albedo  = texture(RT1, texCoord).rgb;
  vec3 N = texture(RT3, texCoord).xyz;
  vec3 positionVS = texture(RT4, texCoord).xyz;

  vec3 indirectLighting = RadianceEstimate(tileIndex, albedo, N, positionVS);

  imageStore(LowResTextureArray, ivec3(imageCoord, 0), vec4(indirectLighting, 1.0f));
  imageStore(LowResTextureArray, ivec3(imageCoord, 1), vec4(N, 1.0f));
  imageStore(LowResTextureArray, ivec3(imageCoord, 2), vec4(positionVS, 1.0f));
}
```

Listing 5.17: GLSL implementation of kernel that renders to the low resolution texture array.

```glsl
#version 430

#define USE_HIGH_RES_TILES
include(Commons.c.glsl)

layout (binding = 1, rgba32f) uniform image2D RT0;

layout (binding = 2) uniform sampler2D RT1; // Kd + Ns
layout (binding = 3) uniform sampler2D RT2; // Ks + Ni
layout (binding = 4) uniform sampler2D RT3; // Normals
layout (binding = 5) uniform sampler2D RT4; // Position
layout (binding = 6) uniform sampler2DArray LowResTextureArray;

uniform mat4 viewMatrix;
uniform mat3 rotationMatrix;
uniform uvec2 gBufferDimensions;

const float PI = 3.14159265f;

include(RadianceEstimate.c.glsl)

void main()
{
  uint tileIndex = gl_WorkGroupID.x + gl_WorkGroupID.y * gl_NumWorkGroups.x;
  ivec2 imageCoord = ivec2(gl_GlobalInvocationID.xy);
  vec2 texCoord = vec2(imageCoord + 0.5f) / vec2(gBufferDimensions);

  vec3 N = texture(RT3, texCoord).xyz;
  vec3 positionVS = texture(RT4, texCoord).xyz;
  vec3 lowResNormal = texture(LowResTextureArray, vec3(texCoord, 1)).xyz;
  vec3 lowResPositionVS = texture(LowResTextureArray, vec3(texCoord, 2)).xyz;

  float positionDiff = distance(positionVS, lowResPositionVS);

  vec4 indirect = vec4(0.0f, 0.0f, 0.0f, 1.0f);
  if(dot(N, lowResNormal) < 0.99f || positionDiff > 0.01f)
  {
    vec3 albedo  = texture(RT1, texCoord).rgb;
    indirect.xyz = RadianceEstimate(tileIndex, albedo, N, positionVS);
  }
  else
  {
    indirect = texture(LowResTextureArray, vec3(texCoord, 0));
  }

  vec4 direct = imageLoad(RT0, imageCoord);
  imageStore(RT0, imageCoord, direct + indirect);
}
```

Listing 5.18: GLSL implementation of kernel that combines the direct and indirect illumination and writes this to the render target that will be displayed.

Chapter 6

Evaluation

The evaluation of our prototype system is performed in three parts. During this evaluation, we use a number of objective measures to assess the performance of our system in terms of visual quality and computational speed (these metrics have already been discussed in chapter 4). The first two parts of the evaluation focus on our system in isolation. First, we measure how our system performs during the rendering of a simple scene while certain system parameters are varied. Thereafter, we test how computational costs change as scene complexity is varied. In the third and final part, we perform a brief comparative study between our prototype implementation and other systems currently available in game engines.

6.1 Parameter Scaling

The rendering parameters that are most impactful on the performance of our system are the total number of photons, and the photon influence radius. The number of photons that are stored in the photon map are indirectly specified via the total number of photon *paths* that are initiated. We make this distinction because, if a photon path never intersects any scene geometry, it will not result in any stored photons. Since we only compute single-bounce indirect illumination, the number of stored photons will always be lower than the number of initiated photon paths. Whenever we report the number of photon paths, we also report the actual number of stored photons in brackets for completeness.

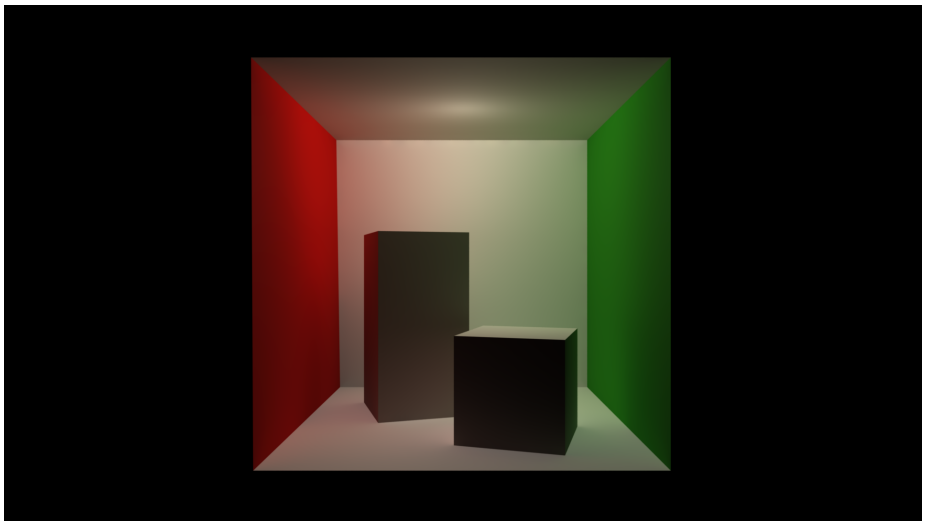
For this part of the evaluation we use a simple scene that is often used to showcase indirect illumination. The Cornell box scene consists of a box that is open on one side (front). The left and right sides are coloured red and green respectively, while the remaining sides (top, bottom and back) are a light shade of gray; this is ideal for the demonstration of colour bleeding that is caused by

indirect diffuse illumination. An additional two boxes are placed in the centre of the scene. Lighting is provided by a single light source that is positioned near the centre of the ceiling. A few renders of this scene are shown in figure 6.1. Figure 6.1a shows the reference image that is used for the quality assessment of the images produced by our system (later on in the evaluation we use the same reference image in comparing our system with others). The edge detection scheme that was mentioned in section 5.3.4 is visualised in figure 6.1c. Note how edges only make up a small portion of the scene, making this an ideal scenario for our down-sampling and interpolation scheme.

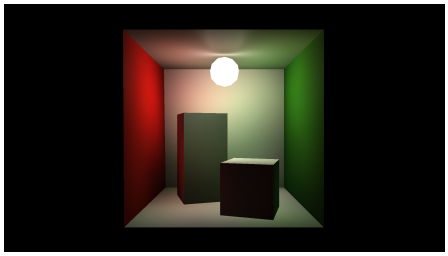
The first step in evaluating our system is to have it render the Cornell box scene in various configurations. As mentioned before, we vary the number of photon paths and the photon influence radius. For each configuration, we measure a number of metrics as described in chapter 4 (mean frame time, MSE, and MSSIM). The results of these measurements are shown in table 6.1. For the mean frame time (column 3) and MSE metrics (columns 5 and 6) a lower score is better, whereas the opposite is true for the MSSIM metric. The table is structured so that increments in the number of photons paths are grouped together. Within each of these groups, the photon influence radius is varied with each table row. We observe that within each group, the measured image quality increases as the photon influence radius is increased, up to a certain point. Increasing the radius beyond this point results in degradation of image quality despite further increases in computational costs. For all measured photon path configurations, this point seems to lie around a photon influence radius of 1.5 (for both the MSE and MSSIM metrics). This is something that is to be expected, since large photon radii will cause illumination to be smeared out across larger areas and can additionally introduce inaccuracies in the form of light leaking through geometry.

The measurements from table 6.1 also show that increasing the number of photons apparently does not always increase visual quality (according the metrics that were used). An example of this can be seen in the measurements that use a photon radius of 1.0; where using a photon count of 3588 yields images of a higher measured quality than the configuration that uses 5350 photons. These observations are further reinforced by the fact that – across all measurements – the best MSE and MSSIM scores are measured with configurations that use 703 and 3588 photons respectively (both with a photon radius of 1.5). However, it does appear that an increased photon count results in higher image quality when dealing with a relatively small photon radius. This is something that can be observed across the rows with a radius of 0.25; those with higher photon counts consistently score better in terms of measured image quality.

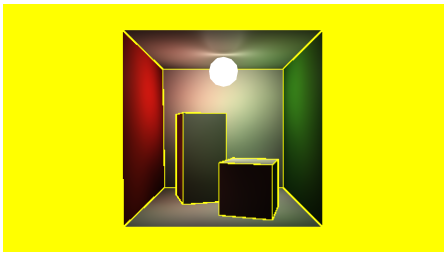
In general we can see that when the number of photons or the photon influence radius (or both) is increased, computational costs also rise. We also observe some form of diminishing returns on visual quality; initial increases in quality come cheaply in terms of computational costs, but these costs increase more rapidly as visual quality is increased further. The observations made in the



(a) High quality reference image of the Cornell box scene. This image was rendered using Mental Ray configured to use 5 million photons, allow multiple indirect bounces, and use final gathering. Render time: 18 minutes and 49 seconds.



(b) Sample render of the Cornell box scene created by our prototype system. This image was rendered using 1,000 photon paths (707 stored photons) and a photon influence radius of 1.0. Render time: 11.3 milliseconds.



(c) Visualisation of identified edges in the Conell box scene. Yellow areas are those for which the down-sampling scheme is deemed to be inappropriate and for which the radiance estimate is thus recomputed.

Figure 6.1: Renders of the Cornell box scene at 1920×1080 using different rendering methods.

		Performance Metrics		Image Quality Metrics		
Photon paths	Radius	μ (ms)	σ (ms)	MSE	NRMSE	MSSIM
500 (341)	0.25	5.4	0.08	902.23	0.1458	0.8621
500 (341)	0.50	5.9	0.05	401.95	0.0973	0.9269
500 (341)	1.00	7.9	0.51	196.09	0.0680	0.9569
500 (341)	1.50	10.9	0.51	160.79	0.0616	0.9646
500 (341)	2.00	13.4	0.38	193.16	0.0675	0.9623
1000 (703)	0.25	5.9	0.08	664.16	0.1251	0.8846
1000 (703)	0.50	6.9	0.12	330.95	0.0883	0.9386
1000 (703)	1.00	11.3	2.24	167.09	0.0627	0.9607
1000 (703)	1.50	16.9	0.34	139.55	0.0573	0.9659
1000 (703)	2.00	22.4	0.30	174.40	0.0641	0.9640
2500 (1778)	0.25	7.2	0.08	502.37	0.1088	0.9122
2500 (1778)	0.50	10.0	1.17	332.16	0.0885	0.9449
2500 (1778)	1.00	20.5	0.36	186.43	0.0663	0.9610
2500 (1778)	1.50	34.6	0.36	143.30	0.0581	0.9661
2500 (1778)	2.00	48.7	2.40	168.71	0.0631	0.9646
5000 (3588)	0.25	9.3	0.09	456.98	0.1038	0.9264
5000 (3588)	0.50	15.1	0.28	329.30	0.0881	0.9491
5000 (3588)	1.00	35.9	0.91	182.96	0.0657	0.9632
5000 (3588)	1.50	64.3	0.28	140.13	0.0575	0.9668
5000 (3588)	2.00	92.7	0.48	165.68	0.0625	0.9649
7500 (5350)	0.25	11.6	0.10	453.26	0.1033	0.9341
7500 (5350)	0.50	20.2	0.69	351.10	0.0910	0.9479
7500 (5350)	1.00	51.6	0.45	200.58	0.0688	0.9619
7500 (5350)	1.50	94.4	0.51	147.34	0.0589	0.9660
7500 (5350)	2.00	136.8	0.61	166.60	0.0627	0.9645

Table 6.1: Measurement results of the first parameter scaling test. Rendering configurations are listed in the first two columns, with the number of initiated photon paths shown in the first column (total number of stored photons is shown in brackets) and the photon influence radius shown in the second. Table rows are grouped based on the first column, and sorted within those groups based on the photon influence radius. Measurements are shown in columns 3 to 7.

previous paragraphs also indicate that the impact that the two parameters have on system performance differs considerably between the two. This is something we explore further in the following part of this evaluation.

6.1.1 Fixed Computational Budget

The measurements in this part of the evaluation are performed similarly to those in the preceding part. The only change is that the system parameters are chosen so that images are produced at similar speeds. The objective of this test is to measure how different distributions of the two parameters compare in terms of visual quality when the computational budget is fixed. To test this, we perform measurements using photon influence radii that range from 0.25 to 2.0 in increments of 0.25 and adjust the number of photon paths so that an arbitrary targeted mean frame time is achieved. We used a photon path count of 1000 with a photon radius of 1.0 as the baseline configuration, which was measured to have a mean frame time of 11.4 milliseconds. Each of the other configurations is selected to have similar computational costs.

The results of this test are shown in table 6.2. Here, we observe that quality consistently improves as the photon radius is increased (and thus, the number of photons is decreased), up to a certain point. Where this point lies is different for the two image quality metrics; the lowest measured MSE is produced by

		Performance Metrics		Image Quality Metrics		
Photon paths	Radius	μ (ms)	σ (ms)	MSE	NRMSE	MSSIM
7000 (4980)	0.25	11.5	0.18	446.30	0.1026	0.9345
2900 (2083)	0.50	11.3	0.52	364.15	0.0926	0.9428
1600 (1125)	0.75	11.4	0.60	266.94	0.0793	0.9515
1000 (703)	1.00	11.4	0.35	167.09	0.0627	0.9607
700 (491)	1.25	11.4	0.56	167.58	0.0628	0.9631
525 (357)	1.50	11.3	0.50	168.22	0.0630	0.9631
425 (286)	1.75	11.4	0.30	177.14	0.0646	0.9641
350 (241)	2.00	11.3	0.82	169.59	0.0632	0.9629

Table 6.2: Measurement results of the parameter scaling test with a soft cap on computational budget. Rendering configurations are listed in the first two columns, with the number of photon paths shown in the first column (total number of stored photons is shown in brackets) and the photon influence radius shown in the second. Rows are sorted by the photon radius parameter in an ascending order. Measurements are shown in columns 3 to 7.

the configuration with a radius of 1.0, while the highest MSSIM value is measured with the configuration that uses a radius of 1.75. The fact that there is a drop off in quality when the radius is increased too much is not surprising, since we have made similar observations in the previous test. The explanation for this observation is the same; excessively large photon radii will cause illumination to be spread out across larger areas and result in degradation of visual quality.

The images that accompany the measurements in table 6.2 are shown in figure A.1 in the appendix on page 60. These images visually demonstrate the effects of varying the photon radius setting (and proportional changes in the total number of photons). In the images with a relatively small photon radius, individual photons can be distinguished. This gives the images a splotchy appearance. On the other extreme, using a very high photon radius causes illumination to be smeared out further over larger areas, which can result in less pronounced colour bleeding. This can be observed by comparing figure A.1d with figure A.1h; the indirect illumination on the ceiling is hardly noticeable in the latter, while it is much more pronounced in the former. Additionally, we show absolute differences between the images of figure A.1 and the reference image (figure 6.1a) in figure A.2. This can be thought of as a visualisation of the spatial distribution of errors in the images that were produced by our system. We observe that increasing the photon radius spreads out the distribution of errors, which is consistent with our earlier observations on different photon radii. This same effect can be observed in the direct visualisations of the measured structural similarity indices, which are shown in figure A.3.

Table 6.2 suggests that there is a relation between the photon radius, the number of stored photons, and the associated computational costs (which is in this case kept relatively constant). To explore this relation further, we have fit a number of estimators to the measurements of table 6.2 using the Levenberg-Marquardt algorithm. This investigation includes a linear estimator $N = \frac{c}{r}$, and two quadratic estimators: $N = \frac{c}{r^2}$, and $N = \frac{c_1}{r} + \frac{c_2}{r^2}$. For all of these equations, N represents the estimated number of photons, r the photon radius, and c some constants that are optimised to minimise the errors between the measured data and the estimators. The estimator functions are plotted against the measurements from table 6.2 in figure 6.2 (the optimal values found for c are also shown in this chart). We had expected the number of photons to behave quadratically with respect to the photon radius, since the area of a sphere (or disk) is also quadratic with respect to its radius. While the estimator that most closely fits the measured data *is* a quadratic equation, it does have a different form than the definition of the area of a sphere. We are currently lacking an explanation for why this is the case. We have also not performed any additional measurements to test how well these estimators can predict newly obtained data.

In order to gain more insight into what parts of the system are computationally the most expensive, we have performed another series of measurements where

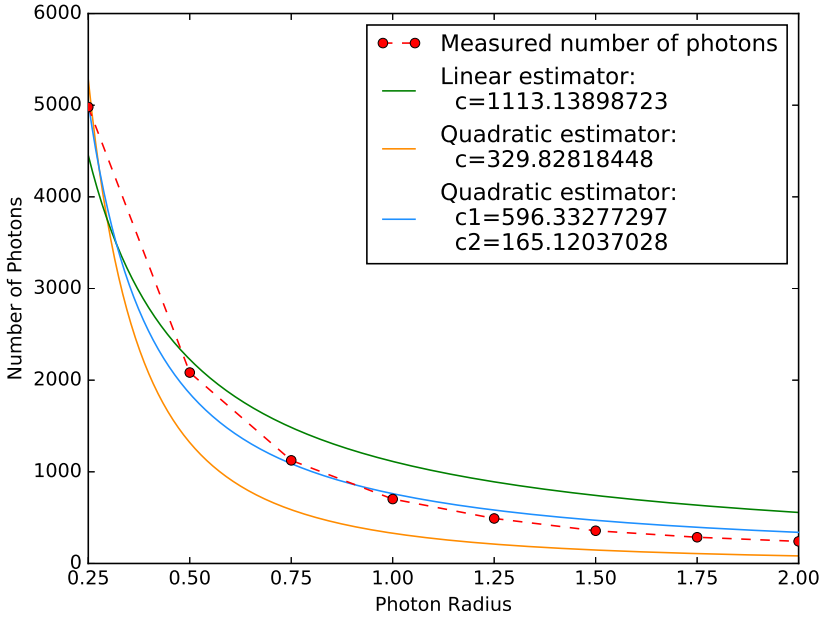


Figure 6.2: Plots of estimator equations compared to the measurements from table 6.2. The definition of the linear estimator is $N = \frac{c}{r}$. The two quadratic estimators are defined as $N = \frac{c}{r^2}$ and $N = \frac{c_1}{r} + \frac{c_2}{r^2}$. N represents the estimated number of photons, r the associated photon radius, and c represents the constants that are shown in the legend above.

the total frame times are further divided into three categories: direct illumination, photon tracing and radiance estimate. Direct illumination entails filling the G-Buffer with geometry data, as well as performing shading due to direct illumination. Photon tracing is performed by the Optix framework. However, since the radiance estimate is performed in OpenGL compute shaders, we include the transfer of the photon buffer to main memory into the photon tracing category. Transferring the photon buffer back to GPU memory so that it can be used by the OpenGL compute shaders is included in the radiance estimate category. Everything that was covered in sections 5.3.3 (Radiance Estimate) and 5.3.4 (Further Approximations) is also included in the radiance estimate category.

Figure 6.3 shows the measured distributions of computational costs in a stacked bar graph. This graph shows that the largest amount of time is spent in the photon tracing stage, followed by the radiance estimate stage. Direct illumination is computationally the least expensive category by a rather large margin. It can be observed that the two leftmost configurations deviate more strongly from

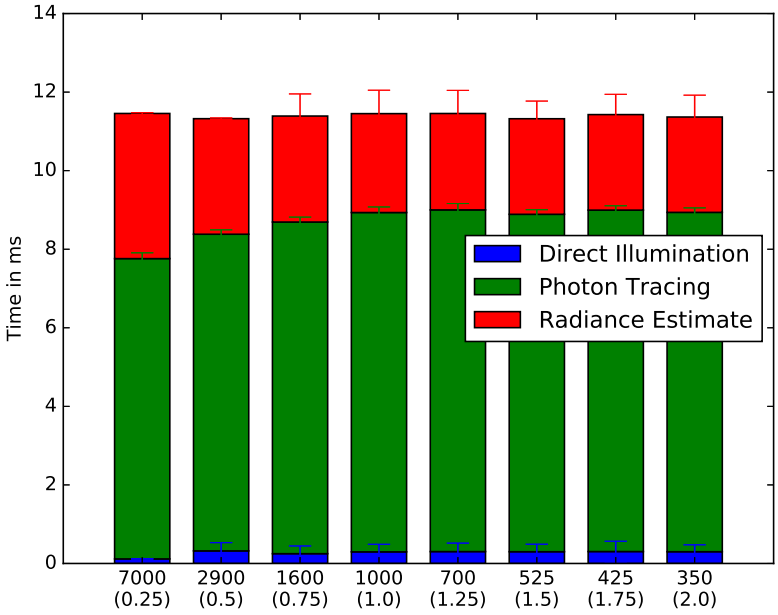


Figure 6.3: Stacked bar graph showing the distribution of computational costs for various rendering parameter configurations. The x-axis shows each configuration separately, labelled by the number of photon paths (photon influence radius is shown in brackets).

the rest in terms of their distribution (note the considerable jump in direct illumination costs between the first and second configuration, and the higher costs in terms of the radiance estimate step). We currently lack an explanation for this. However, since the standard deviations for the two leftmost configurations are also considerably different from the rest, it is possible that these differences are a result of inconsistent measuring environments. This is something that has not been further investigated in this research. Additionally, the fact that photon tracing costs remain relatively constant between the tested configurations seems to indicate that the variable costs with regard to the number of photons is lower than we initially expected. We expected that the configurations with a high photon count would spend significantly more time in the photon tracing step, but this is not observable in these measurements. It is possible that the photon tracing step is dominated by some form of fixed costs. However, this is also something we do not investigate further, since the Optix framework does not offer suitable options for further profiling.

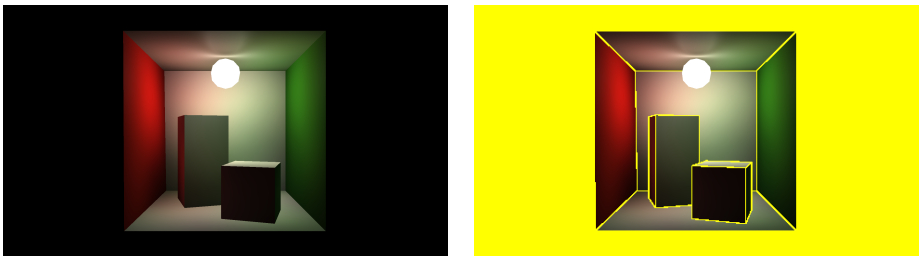
6.2 Scene Complexity

The second part in the evaluation of our system is to investigate how its performance is affected by scene complexity. The following tests are set up similarly to the preceding tests. However, the measurements are only performed on computational costs. We measure average frame times for a number of configurations, and simply repeat this process for a variety of scenes.

The first few scenes are variations of the Cornell box scene that was also used during the previous tests. The first variation replaces the two central boxes with a model of a happy Buddha statue. This models consists of 1,087,451 triangles, which is a significant increase from the 20 triangles that were used to represent the two boxes in the first scene. The second scene variation contains a complex hairball mesh that consists of 2,880,000 triangles and features high-frequency detail. This object subtends a larger portion of the output image and is a pathological scenario for our down-sampling and edge detection scheme. The Cornell box scenes are ideal because of their simplicity and how well-suited they are for demonstrating indirect illumination. However, since the images that were rendered with this scene contain large black borders where nothing is computed (due the rectangular shape not matching particularly well with modern screen resolutions), we include a different scene that does fully subtend the output images. This scene models a conference room scene and consists of 331,187 triangles.

Sample renders of the aforementioned scenes are depicted in figure 6.4, along with the corresponding visualisations of our edge detection method. The Cornell scenes are each illuminated by a single light source at the top of the scene, near the middle of the ceiling (light sources are visualised as white spheres in these images). The conference room scene is illuminated by four light sources that are specifically placed to dramatise the effects of colour bleeding. One of these light sources is located in one of the red chairs, causing primarily red indirect illumination that is most notable on the central part of the ceiling. Another light is placed in front of the green board and two more are placed next to the yellowish structure on the far side of the room (causing mostly green and yellowish indirect illumination respectively).

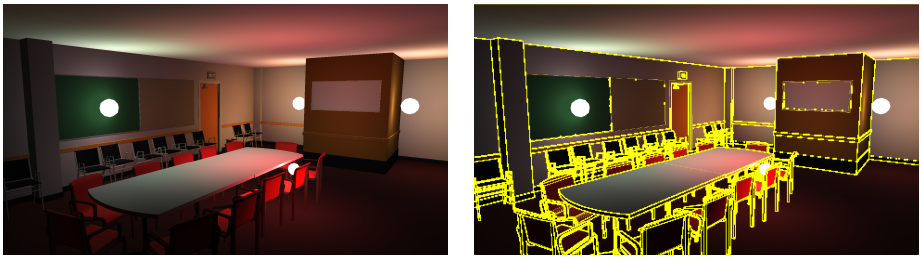
In terms of our down-sampling scheme, we can observe that the scene in figure 6.4a is close to a best-case scenario, the scene in figure 6.4c is closer to a worst-case scenario, and the scene shown in figure 6.4b lies somewhere in between. The conference room scene (figure 6.4d) exposes a limitation in our current edge detection scheme. In section 5.3.4 we noted that our edge detection scheme is incomplete since it only considers positions and normals, and thus ignores changes in material properties. The effects of this can be observed in figure 6.4d; the top border of the green board is not fully recognised as an edge, resulting in a blurry border in the final output image (this may be difficult to see in these relatively small images).



(a) Cornell box scene: two boxes

(b) Cornell box scene: happy Buddha

(c) Cornell box scene: AO hairball



(d) Conference room scene

Figure 6.4: Renders of various scenes (left) alongside their corresponding edge detection visualisations (right). The images of the Cornell box scenes use a 1,000 photon paths with a radius of 1.0. For the images of the conference room scene, we use the same number of photon paths, but the radius is set to 2.0.

The measured average rendering times and associated standard deviations are shown in tables B.1 to B.4, which can be found on pages 63 to 64 in the appendix. These tables are structured in a matrix-like fashion, where each cell represents time measurements made using the corresponding parameters (the number of photon paths and the photon radii are shown along the left-hand side and the top respectively). Within each of the tables we see familiar results. Computational costs increase as either of the two parameters are increased. More interesting are the results we can extract by comparing the measurements across the different scenes.

Comparing the measurements for the Cornell box scenes shows us that computational costs increase across all configurations as scene complexity is increased. Furthermore, it seems that the ratios between the measured average frame times of different configurations remain similar between the different Cornell box scenes. We also measured the distribution of costs for the different Cornell box scenes using the same parameter configuration for each variation. This is visualised in figure B.1 on page 65. In this graph we observe that costs increase proportionally between the photon tracing and radiance estimate stages as scene complexity is increased.

In the measured frame times for the conference room scene, we observe that computational cost are considerably higher at the lowest settings, but increase much less rapidly compared to the Cornell box scenes. The first part is explained by the higher "effective" resolution used in the conference room scene. While image resolution is factually the same for every configuration tested in this research, the Cornell box scenes do not fully subtend the output images, and as a result contain areas for which no computations need to be performed. The conference room scene *does* fully subtend the output image and is thus a more realistic indication of practical performance. The fact that increasing the rendering settings has a smaller effect on performance than we observed in the Cornell box scenes, is a result of the scene simply being larger. Since the distance between the camera and most of the scene geometry is greater in the conference room scene, a larger increase in photon radius is needed to trigger a similar increase in costs. The main conclusion from table B.4 is that our system is able to produce images of slightly more complex scenes, whilst still achieving real-time frame rates with an image resolution of 1920×1080. The configuration with a 1,000 photon paths and a radius of 2.0 has a mean frame time of 25.2 milliseconds, which can be translated to 39.7 frames per second (the image corresponding to this configuration is shown in figure 6.4d).

6.3 System Comparison

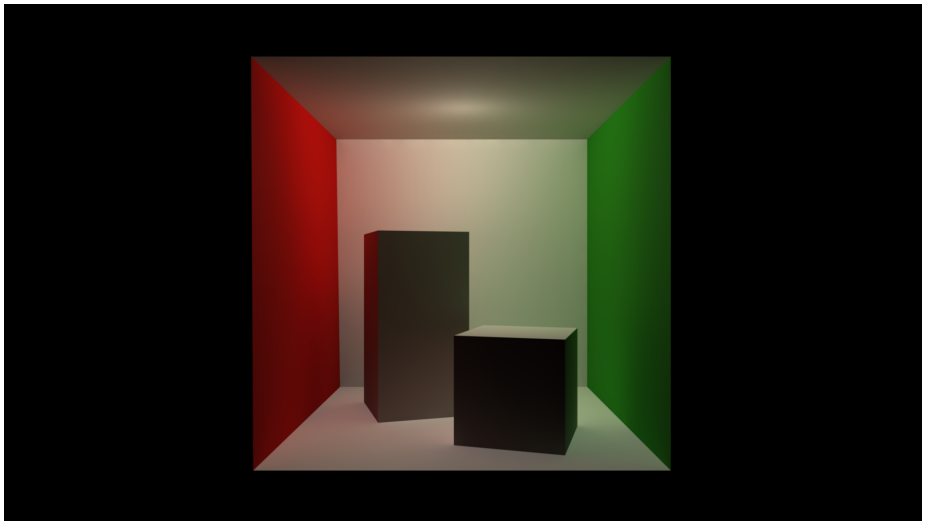
The final part of the evaluation is a brief comparative study between our implemented prototype system, and other systems currently available in some of the most popular game engines. First, we perform a qualitative analysis of images

produced by the different rendering systems. Next, we make use of quantitative measures to compare the systems more objectively. Similar to the previous tests, we make use of a high quality reference image to perform the comparisons. This reference image is the same one that was used for the parameter scaling tests described in section 6.1. Our comparison only considers a single image for each of the rendering systems. In an attempt to make this comparison as fair as possible, we have attempted to configure each system so that their output image is as similar as possible to the reference image. We acknowledge that the manual configuration (and selection of scenes) of the different rendering systems introduces human bias to this comparison, but we are unaware of any feasible alternatives. The images produced by the rendering systems, along with the reference image, are shown in figure 6.5. The absolute differences and structural similarity indices between the reference image and the output images are shown in figures C.1 and C.2 respectively (pages 66 and 67).

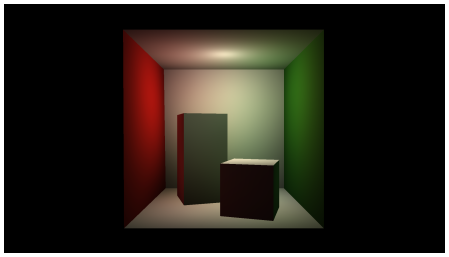
6.3.1 Qualitative analysis

Comparing the image made with our prototype system to the reference image, we observe that our system makes a relatively coarse approximation. The image produced by our system seems fairly similar to the reference image, but some of the more subtle nuances and finer details are lacking. The soft indirect shadows (cast by the two central boxes) that can be seen in the reference image are completely absent. The red *and* green shading on the front of the tall box is also something that is not observable in the image produced by our system. However, it is unsurprising that these types of fine detail are lost in the approximations that are made. We are, after all, using a large photon radius, and only compute single-bounce indirect illumination. Illumination seems more evenly distributed in the reference image, while we observe (circular) areas with higher intensity in the middle of each of the four surrounding surfaces in the image produced by our system. This could be the result of a suboptimal photon distribution, or it could also be caused by differences in shading models (or by a combination of both). Overall, our system seems to approximate the reference image fairly successfully. Our system is unable to capture fine details in the configuration that was used, but this is only natural; substantial concessions need to be made in order to produce these images at real-time frame rates.

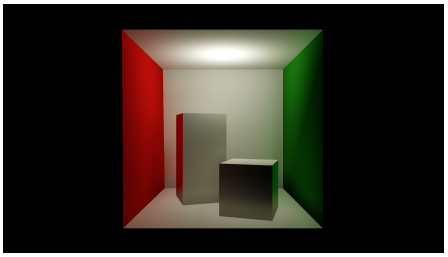
Figure 6.5c shows the Cornell box scene illuminated using CryEngine's Sparse Voxel Octree Total Illumination (SVOTI). This experimental solution seems to build upon the work of Crassin et al. [6]. SVOTI seems to be more flexible than our system in the sense that it also supports indirect specular illumination (note that this is *not* shown in this comparison). In addition, this method supports dynamic objects. However, this does require rebuilding the SVO representation on frames where geometry is updated (similar to how the acceleration structure that our system uses would also need to be updated to reflect changes in geometry). Nevertheless, since this comparison (or any other tests performed in



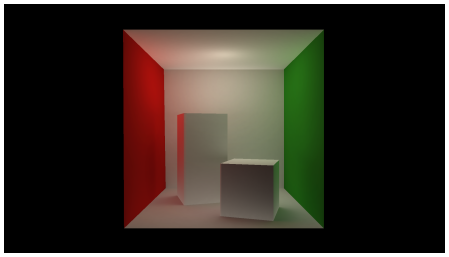
(a) High quality reference image of the Cornell box scene. This image was rendered using Mental Ray configured to use 5 million photons, allow multiple indirect bounces, and use final gathering. Render time: 18 minutes and 49 seconds.



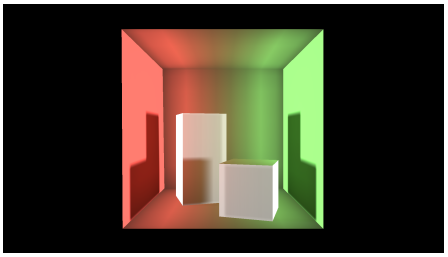
(b) Our prototype implementation. Render time: 13.2 ms



(c) CryEngine (SVOTI). Render time: 17.2 ms



(d) Unity (Enlighten). Render time: 0.2 ms



(e) Unreal (LPV). Render time: 8.5 ms

Figure 6.5: Renders of the Cornell box scene at 1920×1080 using different rendering methods.

this research) does not make use of dynamic geometry, this is not something we have investigated. The illumination computed via SVOTI differs from both figure 6.5a and figure 6.5b. While the red wall and the colour bleeding seen on the floor is more similar to the reference image than the image produced by our system, there are some other areas where it differs more than our system. The colour bleeding on the ceiling in figure 6.5c is most intense in the corners, which is not the case in the reference image. In addition, the boxes exhibit stronger colour bleeding on their sides, and the sides that face toward the viewer show light-grey shading that is not present in the reference image. Finally, the green wall shows a dark area along the bottom edge, as well as a darker triangle (split along the wall's diagonal). We are unsure what is causing the green wall to be illuminated this way. Perhaps it is an issue caused by malformed geometry (even though the same geometry does not produce similar artefacts in the other systems).

Unity does not offer a truly dynamic global illumination solution, instead they use middle-ware (called Enlighten) that offers something between static and dynamic global illumination. Implementation details on Enlighten have not been made public. However, it is presumable that their solution is based on radiosity. Enlighten pre-computes a number of things for static geometry, this then allows the static geometry to enjoy global illumination with dynamic light sources. Dynamic geometry is however not included in the global illumination; it can *receive* indirect illumination from static geometry via light probes, but it can never contribute to the global illumination. The pre-computation step, along with the fact that Enlighten runs on the CPU, allows it to enjoy a very small footprint in terms of graphical computation costs. However, since it runs on the CPU, it does take up resources that are normally used for other matters, which can potentially result in update delays. Additionally, running parts of the graphics pipeline on the CPU inherently introduces overhead and latency. Figure 6.5d shows how Enlighten performs in this comparison. In terms of visual quality, the results are also fairly similar to the reference image. There are still some observable differences however. The largest difference seems to be the shading of the two central boxes; the colour bleeding on the sides seems much brighter than the reference image, and the front is also illuminated more strongly. Enlighten does manage to capture the indirect shadows cast by the two central boxes (none of the other systems do), but its effect is made much more noticeable and intense.

The image produced by the Unreal implementation of LPVs is shown in figure 6.5e. Unreal also offers static light baking as well as Enlighten, but LPVs is the only dynamic global illumination solution that is offered. This is a somewhat older method that has been applied in a fair number of games. The image shown in figure 6.5e differs greatly from any of the preceding images. However, it is not clear to what degree this can be attributed to the algorithm itself. The scene that was used for this part of the comparison is significantly different from the others; instead of having a single point light near the ceiling, the scene features two directional shadow-casting lights (one pointing precisely to

the left, the other to the right), simply because those are the only light types that are supported. Visually, there is not much that can be said about these results, except that there is some form of colour bleeding and that shadows are included.

6.3.2 Quantitative Analysis

In this part of the comparison, we examine both computational costs and objective measures of visual image quality. In terms of computational costs, we no longer measure averages across a fixed number of frames, since we rely on the internal profiling tools provided by the various game engines. The same visual quality metrics that have been used throughout the evaluation are computed for the images shown in figure 6.5 and are shown in table 6.3 (along with the measured frame times).

These results show that – in the configurations that were used – our system produces images around 30% faster than CryEngine's SVOTI implementation, but approximately 55% slower than Unreal's LPVs implementation. The runtime footprint of Enlighten is much lower than the other methods, but only has limited support for dynamic scenes. The pre-computation step for Unity's configuration took about 5 minutes and the data takes up around 180 megabytes (both of these can be reduced or increased, but doing so will also affect image quality). In terms of computational costs, LPVs appears to be the least expensive of the *dynamic* global illumination methods that we investigated. It is likely that this method was actually faster than the measured 8.5 milliseconds, since the frame times hardly changed when we disabled global illumination via LPVs.

The computed image quality metrics are shown on the right-hand side of table 6.3, and the absolute differences and structural similarity indices are also visualised in figures C.1 and C.2. According to these measurements, our system produces images that are most similar to the reference image, followed

		Image Quality Metrics		
System	Frame Time (ms)	MSE	NRMSE	MSSIM
Prototype Implementation	13.2	140.40	0.0575	0.9648
CryEngine (SVOTI)	17.2	579.38	0.1168	0.8990
Unity (Enlighten)	0.2	406.25	0.0978	0.9398
Unreal (LPV)	8.5	3589.49	0.2908	0.8461

Table 6.3: Measured frame time and image quality metrics for the images shown in figures 6.5b-6.5e.

by Unity's usage of Enlighten, CryEngine's SVOTI, and Unreal's LPVs, in descending order of similarity. It is unclear whether this ordering would remain the same if each system was given a fixed computational budget, as opposed to the quality driven set-up that is used in this comparison.

Finally, we do want to note that this comparison has a number of issues, and we consider it to be incomplete. The first issue is that a simple scene such as the Cornell box scene is very favourable for our down-sampling scheme described in section 5.3.4. We have measured the effects of increasing scene complexity on the performance of our system in section 6.2, but to what degree the other systems behave similarly is unclear at this point. Secondly, even though we have examined systems that are intended for dynamic global illumination, the comparison and tests that have been executed feature no dynamic lighting or geometry at all. It is likely that this will prove to be a hurdle for our system, since moving light sources or geometry will cause photons to move around and possibly result in flickering. This aspect is not investigated in this research and thus remains something to be explored in the future.

Chapter 7

Discussion

Now that the design, implementation and evaluation of our prototype system has been covered, we can proceed with the discussion of our research results. The performed analyses indicate that our prototype can fairly successfully approximate indirect illumination, for restricted scenes, at real-time frame rates (using hardware that could be classified as low to mid-range). Subtle nuances that can be observed in high-quality reference images are lost due to approximation, but it is possible that these can be reintroduced to some degree as the implementation is optimised or if more computational power is otherwise made available. The performed comparative study indicates that our system is competitive with other methods currently available in some of the more popular game engines (higher visual quality for similar computational costs).

Moreover, our current implementation is far from optimised. NVIDIA has reported Optix (which we use to perform photon tracing) performing 25-35% slower than a hand-optimised domain-specific ray tracer [19]. It seems reasonable that we can expect a similar increase in performance in the photon tracing stage should we implement a manually optimised ray tracer. Doing so would also likely reduce frame times by an additional 1-2 milliseconds, since our system now performs unnecessary data transfers between main memory and GPU memory, which can be eliminated if it would adhere to a single technology. The fact that performance can still be improved considerably, and that our system already seems competitive with the other compared methods, further indicates that a GPU-based photon mapping approach is not only feasible for real-time indirect diffuse illumination but also promising and worthy of further research and development.

More modern graphics APIs such as Vulkan and Direct3D 12 are a hot topic in the computer graphics field. Since these technologies allow developers to take more direct control of graphics hardware with less overhead, it is likely this can result in some form of performance increase for applications that transition to

these newer APIs. To what degree our system could benefit from these newer technologies is currently unclear since the majority of work is already performed in compute-like shaders.

A photon mapping approach inherently exhibits a number of problems. First of all, it requires human input in order to render images at acceptable quality and performance. In the simple scenes that were shown in this research, this was a matter of tweaking the number of photons and their influence radius. In larger scenes, getting an acceptable distribution of photons will become a larger problem; certain areas will be more important for a given image, likely requiring a higher photon density than areas of lesser importance. While there are methods for importance driven photon map generation, it is unclear to us at this point if these can be employed within real-time constraints.

Another problem that merits further attention is that flickering can become an issue in dynamic scenes where the photon count and radius is relatively low. Increasing these parameters allows for a reasonable reduction in flickering, but of course this further increases computational costs. Additionally, as we have shown in the evaluation of our system, increasing the photon radius to high levels can also lead to loss of precision and introduce additional inaccuracies. Optimising the system for efficiency will allow for better image and animation quality. However, there might also be other more worthwhile changes that could be implemented to greater effect. Our system is a relatively straightforward and bare-bones implementation of the photon mapping algorithm. There is a vast amount of literature focused on photon mapping, which could be used to improve and expand our system in a wide variety of ways.

Since we use deferred shading, we inherit its limitations as well. The largest one being that it is ill-suited for dealing with translucent surfaces. Photons can be traced through surfaces without problems. However, the G-Buffer only contains geometry data for a single depth layer per pixel, meaning that our current prototype can also only perform the radiance estimate for that same depth layer. Shading indirect illumination for translucent surfaces could be introduced by performing this in some form of separate forward rendering step. Whether or not something like this is preferential to the entire replacement of deferred shading in our system remains to be seen.

Aside from the limitations of our prototype implementation, it is also necessary to consider the shortcomings of this research itself. First and foremost is the methodology used in the evaluation of our system, particularly the comparative study between our prototype and other systems. The qualitative comparison is performed by ourselves, which raises obvious objections. Our quantitative comparison relies on the usage of automatic image quality metrics that *approximate* perceived image quality (to varying degrees of success). Despite this, the quantitative comparison is not wholly objective; human bias is introduced via the configuration of rendering systems and the selection of scenes. We therefore ask readers to be extra critical in this regard.

Another shortcoming is that while this research is focused on dynamic indirect illumination, we have not included dynamic elements in the scenes that were used during the evaluation of our system. The only aspect of our system that is directly influenced by dynamic elements is the acceleration structure that is constructed and used by Optix to trace photon paths. This structure needs to be updated to reflect any changes in geometry. The degree in which the introduction of dynamic elements affects computational costs will depend on scene structure and the nature of the dynamics. Anecdotally, it may be noted that introducing rotation of the two boxes around the middle of the Cornell box scene (using the same configuration that was used to produce the image in figure 6.5b) resulted in an increase in the average render time of around one millisecond. However, since we do not have implementation details on the acceleration structure that is used, nor have the tools needed to properly profile the Optix framework, this is not something we can investigate rigorously or fairly at this point.

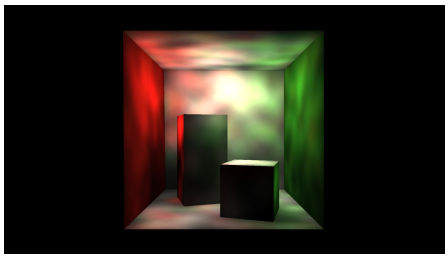
Chapter 8

Conclusion

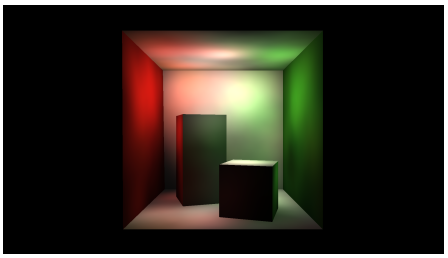
The main objective of this research was to investigate the feasibility of a fully GPU-based photon mapping approach for indirect diffuse illumination in real-time applications. We have designed and implemented a relatively bare-bones implementation of the photon mapping algorithm that is computed entirely on the GPU. The evaluation of our prototype indicates that it is indeed capable of computing approximate real-time indirect diffuse illumination for restricted scenes. There is a considerable difference in visual image quality when comparing our prototype to rendering systems that do not have a restricted computational budget, but this is to be expected. The comparison of our system with other dynamic real-time indirect illumination systems, concludes that our currently unoptimised prototype already produces images of higher measured image quality at similar computational costs. Given the fact that there is still much room for improvement, our approach seems promising and worthy of further research. Since our system is based on the photon mapping algorithm, it will be able to converge to correct solutions as more computational power is made available. However, the implemented method of photon mapping does exhibit a number of limitations that need to be taken into consideration in any future research or applications.

Appendix A

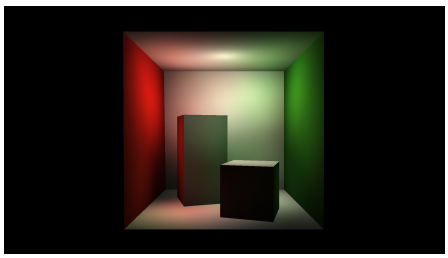
Parameter Scaling Images



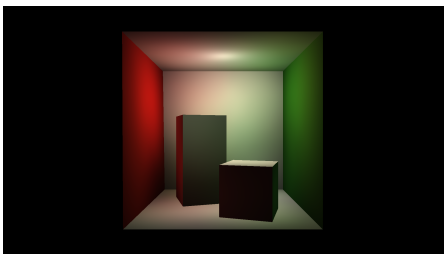
(a) Photon paths: 7000, Radius: 0.25



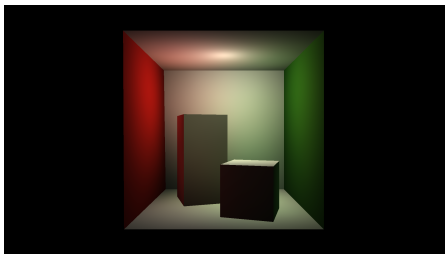
(b) Photon paths: 2900, Radius: 0.5



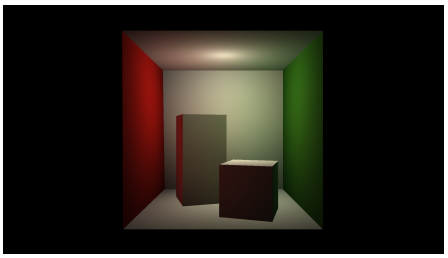
(c) Photon paths: 1600, Radius: 0.75



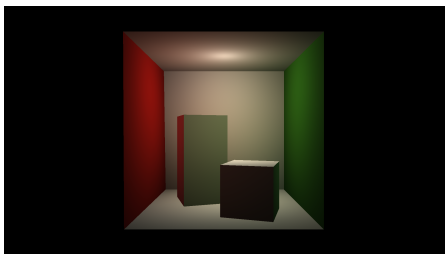
(d) Photon paths: 1000, Radius: 1.00



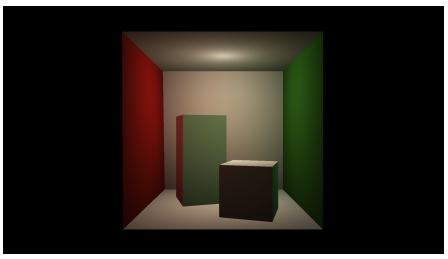
(e) Photon paths: 700, Radius: 1.25



(f) Photon paths: 525, Radius: 1.50

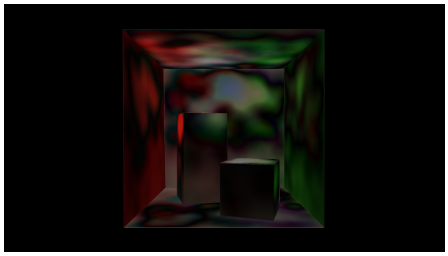


(g) Photon paths: 425, Radius: 1.75

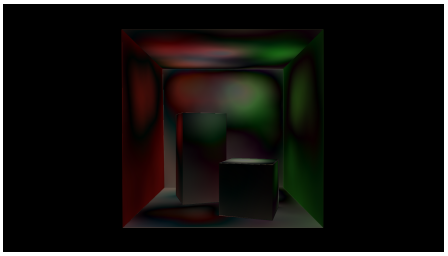


(h) Photon paths: 350, Radius: 2.00

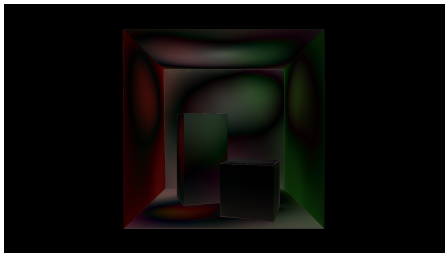
Figure A.1: Rendered images accompanying the measurements shown in table 6.2 on page 43.



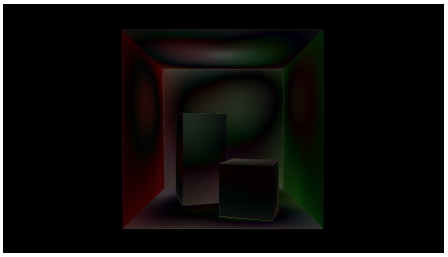
(a) Photon paths: 7000, Radius: 0.25



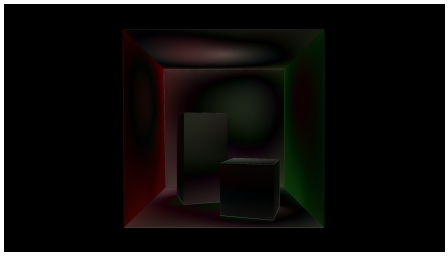
(b) Photon paths: 2900, Radius: 0.5



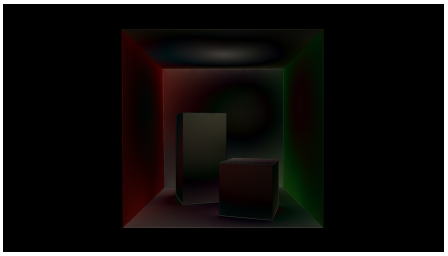
(c) Photon paths: 1600, Radius: 0.75



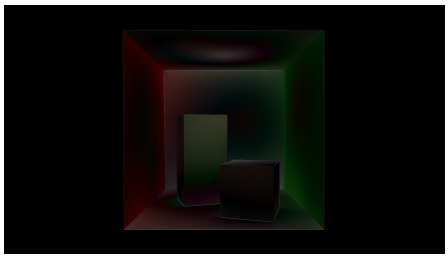
(d) Photon paths: 1000, Radius: 1.00



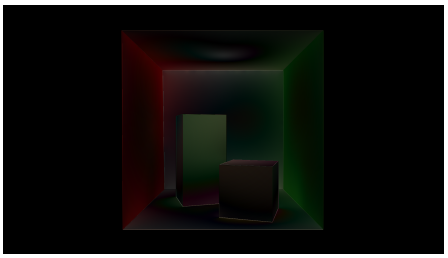
(e) Photon paths: 700, Radius: 1.25



(f) Photon paths: 525, Radius: 1.50

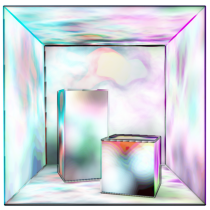


(g) Photon paths: 425, Radius: 1.75

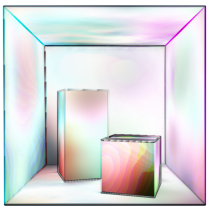


(h) Photon paths: 350, Radius: 2.00

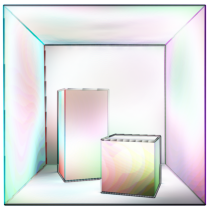
Figure A.2: Absolute differences between the images shown in figure A.1 and the reference image shown in figure 6.1a (shown on page 41). Darker areas are closer to the reference image.



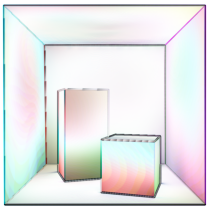
(a) Photon paths: 7000, Radius: 0.25



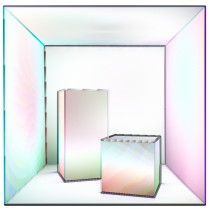
(b) Photon paths: 2900, Radius: 0.5



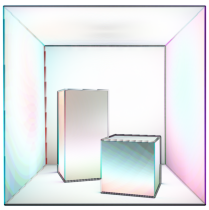
(c) Photon paths: 1600, Radius: 0.75



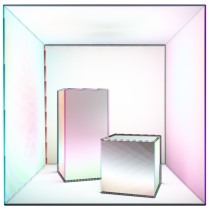
(d) Photon paths: 1000, Radius: 1.00



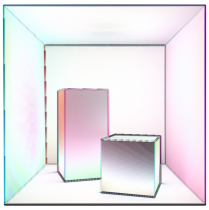
(e) Photon paths: 700, Radius: 1.25



(f) Photon paths: 525, Radius: 1.50



(g) Photon paths: 425, Radius: 1.75



(h) Photon paths: 350, Radius: 2.00

Figure A.3: Visualisation of structural similarity between the images shown in figure A.1 and the reference image shown in figure 6.1a (shown on page 41). Lighter areas have a higher structural similarity index.

Appendix B

Scene Complexity Measurements

	Photon Influence Radius				
Photon paths	0.25	0.5	1.0	1.5	2.0
500 (341)	5.4 (0.08)	5.9 (0.05)	7.9 (0.51)	10.9 (0.51)	13.4 (0.38)
1000 (707)	5.9 (0.08)	6.9 (0.12)	11.3 (2.24)	16.9 (0.34)	22.4 (0.30)
2500 (1776)	7.2 (0.08)	10.0 (1.17)	20.5 (0.36)	34.6 (0.36)	48.7 (2.40)
5000 (3566)	9.3 (0.09)	15.1 (0.28)	35.9 (0.91)	64.3 (0.28)	92.7 (0.48)
7500 (5339)	11.6 (0.10)	20.2 (0.69)	51.6 (0.45)	94.4 (0.51)	136.8 (0.61)

Table B.1: Average rendering times in milliseconds, measured over 1,000 frames, for the Cornell box scene with two boxes in the centre (standard deviation is shown in brackets).

	Photon Influence Radius				
Photon paths	0.25	0.5	1.0	1.5	2.0
500 (335)	6.3 (0.07)	6.8 (0.08)	9.1 (0.08)	12.6 (0.40)	14.9 (0.29)
1000 (705)	6.8 (0.07)	7.9 (0.09)	13.1 (0.37)	20.0 (0.29)	25.1 (0.94)
2500 (1759)	8.2 (0.12)	11.5 (0.35)	23.7 (0.26)	41.2 (1.40)	53.9 (0.83)
5000 (3548)	10.7 (0.10)	16.9 (0.28)	41.8 (0.22)	77.1 (0.41)	103.3 (0.37)
7500 (5313)	13.2 (0.10)	22.5 (0.15)	60.0 (0.25)	113.5 (4.39)	152.1 (0.37)

Table B.2: Average rendering times in milliseconds, measured over 1,000 frames, for the Cornell box scene with the happy Buddha mesh in the centre (standard deviation is shown in brackets).

	Photon Influence Radius				
Photon paths	0.25	0.5	1.0	1.5	2.0
500 (360)	8.7 (0.07)	9.9 (0.07)	14.5 (1.20)	19.3 (0.31)	22.1 (1.21)
1000 (737)	9.1 (0.08)	11.4 (0.08)	20.5 (0.17)	30.5 (0.08)	36.3 (0.34)
2500 (1867)	11.7 (0.09)	17.7 (0.26)	40.4 (0.30)	65.9 (0.27)	80.8 (1.83)
5000 (3754)	15.5 (0.14)	27.6 (0.39)	73.5 (1.44)	125.1 (1.34)	155.2 (0.43)
7500 (5614)	19.5 (0.21)	37.2 (0.30)	105.9 (0.89)	183.3 (0.40)	228.6 (2.48)

Table B.3: Average rendering times in milliseconds, measured over 1,000 frames, for the Cornell box scene with the hairball mesh in the centre (standard deviation is shown in brackets).

	Photon Influence Radius				
Photon paths	0.25	0.5	1.0	1.5	2.0
500 (497)	14.0 (0.23)	14.5 (0.28)	15.4 (0.28)	16.9 (0.23)	19.5 (0.52)
1000 (996)	14.6 (0.28)	15.0 (0.25)	17.1 (0.24)	20.2 (0.23)	25.2 (0.25)
2500 (2496)	16.2 (0.24)	17.7 (0.18)	23.2 (0.24)	31.6 (0.42)	43.9 (0.30)
5000 (4996)	18.3 (0.24)	21.6 (0.27)	32.7 (0.58)	48.6 (0.30)	73.7 (0.41)
7500 (7496)	20.3 (0.17)	25.4 (0.30)	42.7 (0.32)	67.8 (1.91)	105.8 (1.13)

Table B.4: Average rendering times in milliseconds, measured over 1,000 frames, for the conference room scene (standard deviation is shown in brackets).

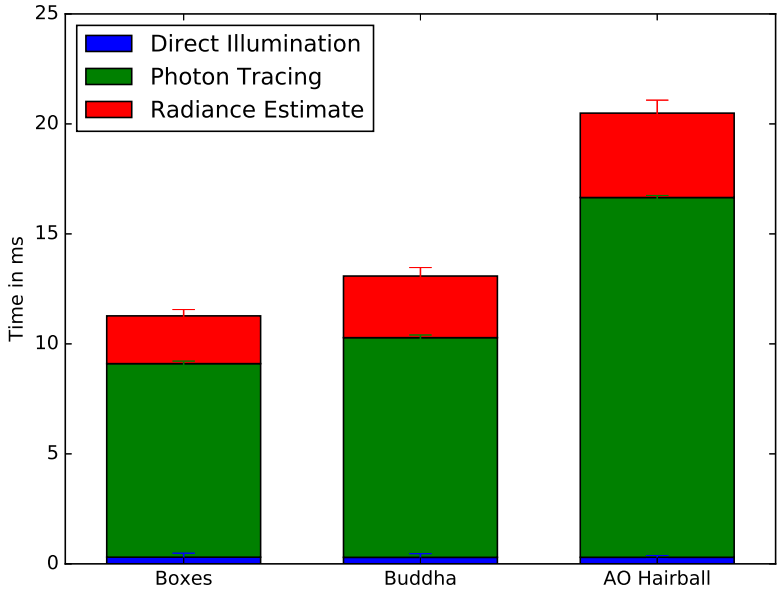
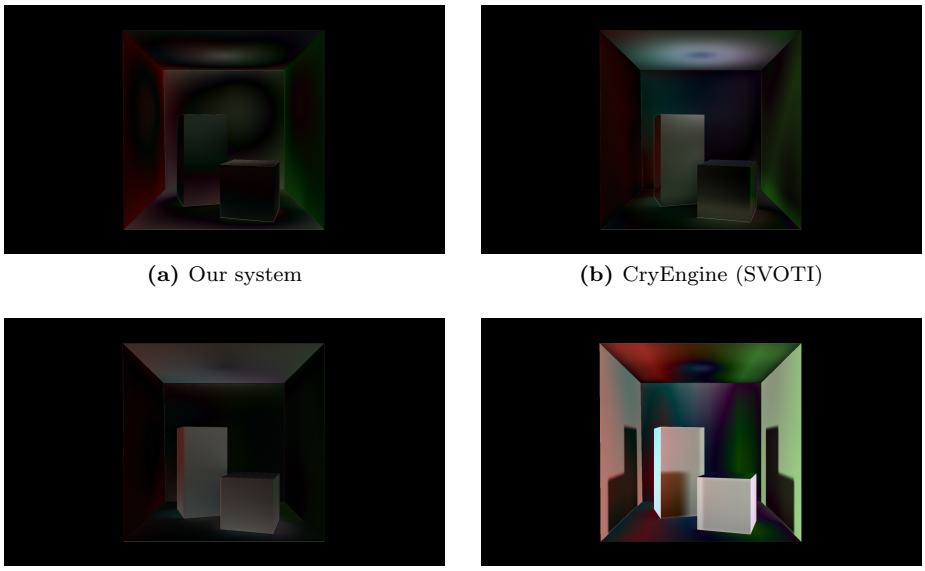


Figure B.1: Stacked bar graph showing the distribution of computational costs for the Cornell box scenes using 1,000 photon paths and a photon influence radius of 1.0.

Appendix C

System Comparison Images



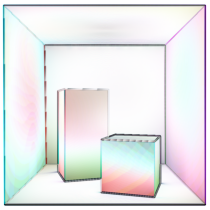
(a) Our system

(b) CryEngine (SVOTI)

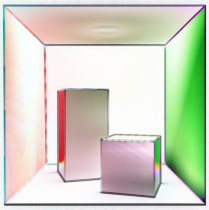
(c) Unity (Enlighten)

(d) Unreal (LPV)

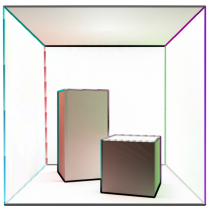
Figure C.1: Absolute differences between the images shown in figures 6.5b-6.5e and the reference image shown in figure 6.5a on page 51. Darker areas are closer to the reference image.



(a) Our system



(b) CryEngine (SVOTI)



(c) Unity (Enlighten)

(d) Unreal (LPV)

Figure C.2: Visualisation of structural similarity between the images shown in figures 6.5b-6.5e and the reference image shown in figure 6.5a on page 51. Lighter areas have a higher structural similarity index.

Bibliography

[1] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008.

[2] Jacco Bikker and Jeroen van Schijndel. The brigade renderer: A path tracer for real-time games. *International Journal of Computer Games Technology*, 2013:1–14, 2013.

[3] Zina H Cigolle, Sam Donow, and Daniel Evangelakos. A survey of efficient representations for independent unit vectors. *Journal of Computer Graphics Techniques Vol*, 3(2), 2014.

[4] Robert L Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. In *ACM SIGGRAPH Computer Graphics*, volume 18, pages 137–145. ACM, 1984.

[5] Cyril Crassin and Simon Green. Octree-based sparse voxelization using the gpu hardware rasterizer. *OpenGL Insights*, pages 303–318, 2012.

[6] Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, and Elmar Eisemann. Interactive indirect illumination using voxel cone tracing. In *Computer Graphics Forum*, volume 30, pages 1921–1930. Wiley Online Library, 2011.

[7] Carsten Dachsbacher and Marc Stamminger. Reflective shadow maps. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 203–231. ACM, 2005.

[8] Michal Drobot. Physically based area lights. *GPU Pro 5: Advanced Rendering Techniques*, page 67, 2014.

[9] Takahiro Harada, Jay McKee, and Jason C Yang. Forward+: A step toward film-style shading in real time. *GPU Pro 4: Advanced Rendering Techniques*, 4:115, 2013.

[10] Henrik Wann Jensen. Global illumination using photon maps. In *Rendering Techniques 96*, pages 21–30. Springer, 1996.

[11] Henrik Wann Jensen, Per H Christensen, Toshiaki Kato, and Frank Suykens. A practical guide to global illumination using photon mapping. *SIGGRAPH 2002 Course Notes CD-ROM*, 2002.

[12] James T Kajiya. The rendering equation. In *ACM SIGGRAPH Computer Graphics*, volume 20, pages 143–150. ACM, 1986.

[13] Anton Kaplanyan. Light propagation volumes in cryengine 3. *ACM SIGGRAPH Courses*, 7:2, 2009.

[14] Anton Kaplanyan and Carsten Dachsbacher. Cascaded light propagation volumes for real-time indirect illumination. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pages 99–107. ACM, 2010.

[15] Brian Karis and Epic Games. Real shading in unreal engine 4. *part of "Physically Based Shading in Theory and Practice," SIGGRAPH*, 2013.

[16] Michael Mara, Morgan McGuire, and David Luebke. Toward practical real-time photon mapping: Efficient gpu density estimation. In *Interactive 3D Graphics and Games 2013*, March 2013.

[17] Morgan McGuire and David Luebke. Hardware-accelerated global illumination by image space photon mapping. In *Proceedings of the 2009 ACM SIGGRAPH/EuroGraphics conference on High Performance Graphics*, New York, NY, USA, August 2009. ACM.

[18] Quirin Meyer, Jochen Süßmuth, Gerd Sußner, Marc Stamminger, and Günther Greiner. On floating-point normal vectors. In *Computer Graphics Forum*, volume 29, pages 1405–1409. Wiley Online Library, 2010.

[19] Steven G Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, et al. Optix: a general purpose ray tracing engine. In *ACM Transactions on Graphics (TOG)*, volume 29, page 66. ACM, 2010.

[20] Matt Pharr and Greg Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2010.

[21] Tobias Ritschel, Thorsten Grosch, Min H Kim, H-P Seidel, Carsten Dachsbacher, and Jan Kautz. Imperfect shadow maps for efficient computation of indirect illumination. In *ACM Transactions on Graphics (TOG)*, volume 27, page 129. ACM, 2008.

[22] Bruce Walter, Stephen R Marschner, Hongsong Li, and Kenneth E Torrance. Microfacet models for refraction through rough surfaces. In *Proceedings of the 18th Eurographics conference on Rendering Techniques*, pages 195–206. Eurographics Association, 2007.

[23] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.

[24] Eric W Weisstein. Sphere point picking. 2002.

[25] David J Wheeler and Roger M Needham. Tea, a tiny encryption algorithm. In *Fast Software Encryption*, pages 363–366. Springer, 1995.

[26] Fahad Zafar, Marc Olano, and Aaron Curtis. Gpu random numbers via the tiny encryption algorithm. In *Proceedings of the Conference on High Performance Graphics*, pages 133–141. Eurographics Association, 2010.