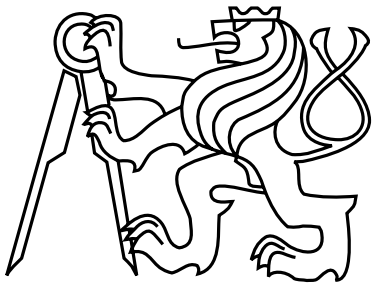
Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Graphics and Interaction



Master Thesis

Progressive Consistent Computation of Global Illumination

Bc. Michal Lukáč

Supervisor: Ing. Vlastimil Havran, Ph.D.

Study Programme: Electrical Engineering and Information Technology

Field of Study: Computer Science and Engineering

May 13, 2011

Aknowledgements

At this point, I wish to express my gratitude to
Dr. Havran, for his guidance and advice, which made this work a lot better than it would otherwise be,
Ondra, for peer review and other friendly help,
Tomáš, for a great deal of impromptu advice,
and Ludvík “Rawalanche” Koutný, for a scene that breaks pretty much everything.

Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used.
I have no objection to usage of this work in compliance with the act §60 Zákon č. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

Prague, 13. 5. 2011 .....

Abstract

Progressiveness in a global illumination algorithm is a desirable, yet seldom considered property. We have decided to explore this class of algorithms with additional focus on estimate consistency and suitability for GPU implementation. Of these, several were chosen for implementation, including path tracing and various filtering techniques on one hand and several progressive photon tracing algorithms on the other.

In total, seven algorithms were implemented. The process of parallelizing these for a *Single Program Multiple Data* (SPMD) parallel computer is described, the resulting implementations were then tested on several scenes with varying complexity and illumination features. Results were compared, with main focus on progression of estimation error in time.

Abstrakt

Progresivita v algoritmech pro globální osvětlení je žádoucí, ale zřídka uvažovaná vlastnost. Rozhodli jsme se prozkoumat tuto třídu algoritmů s dodatečným důrazem na konzistenci odhadu a vhodnost pro implementaci na GPU. Několik z těchto algoritmů bylo zvoleno k implementaci, včetně sledování cest a různých filtrovacích technik, jakož i několik progresivních algoritmů založených na sledování fotonů.

Dohromady bylo implementováno sedm algoritmů. Je popsán proces jejich paralelizace pro *Single Program Multiple Data* (SPMD) paralelní počítač, výsledné implementace pak byly testovány na několika scénách různé složitosti a skladby osvětlení. Výsledky pak byly porovnány s hlavním důrazem na průběh chyby odhadu v čase.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

In August 1986, James Kajiya published his seminal paper on the *Rendering Equation* [Kaj86], laying down the theoretical foundations for the field of realistic image synthesis, which he formulated as a recursive integration problem. In the 25 years since, this field has found applications in the entertainment industry, industrial design, architecture, scientific visualization and many other areas.

In the same paper, path tracing was proposed as an unbiased algorithm to solve said equation, but for over a decade, *radiosity* [GTGB84] was instead accepted as the optimal solution for the problem of realistic image synthesis, due to, mostly, restrictions on available computational power. Towards the end of the millennium, evolution of computer hardware brought sufficient computational power for more universal methods, such as path tracing and recently developed photon mapping [Jen96], to regain prominence.

However, these rendering methods rely quite heavily on the ray tracing operator, which is rather computationally intensive. This has been the limiting factor in the drive for higher quality images. In response, the research community focused on making ray tracing faster with accelerating data structures, while at the same time attempting to get better convergence with the same number of rays through more sophisticated sampling schemes and biased techniques.

Outside of realistic image synthesis, GPUs have been used to accelerate real-time rendering for computer games, virtual realities and real-time visualisation. Starting out as simple hardware rasterizing units, more and more parts of the *Rendering Pipeline* were being moved to the GPU. This prompted programmers, mainly game developers, to demand greater freedom in configuring GPU behaviour, which led to the development of programmable shaders at the beginning of the millennium. Since then, the programmability of the GPUs was being expanded and circa 2005 GPUs effectively became specialized SIMD computers. Rising interest in harnessing the power of the GPU for purposes other than game graphics, such as scientific computations, led to the development of several GPGPU programming platforms, most notable of which was NVIDIA CUDA first presented in 2007 [NVI07].

This new technology allowed massive parallelisation of many types of computation, potentially providing substantial speed-ups and, in the case of realistic image synthesis, completely upsetting conventional wisdoms on which operations are to be considered “cheap”. In comparing GPU implementations, we may no longer rely on such measures as convergence per number of samples, because the relationship between the computational cost of drawing a sample and of any additional calculation is potentially very

different from the same relationship in a CPU implementation. To address this issue, we will have to be very careful in analysing convergence rates and computational costs.

As of writing of this thesis, fully programmable SIMD computers with a performance of over 1000 GFLOPS are available as consumer-grade computer hardware. Software developers have yet to utilize this, but in the research community, GPUs have become quite popular. Even in realistic image synthesis, new algorithms are already being developed with GPU utilization in mind. Therefore, one of the key points of this thesis is to attempt to evaluate the impact of GPU utilization on existing algorithms.

At the same time, the focus of this thesis are *progressive algorithms*. In common implementations, realistic image synthesis algorithms have to be configured prior to rendering, by providing them with a set of inputs (besides the scene information), characteristic for each algorithm. These inputs then determine both rendering time and quality of the resultant image. In practice, however, finding optimal values for these input settings is non-trivial and users, who are usually not privy to the theory behind their renderers, end up configuring them wrong. If the result is unsatisfactory, they are often forced to either touch it up manually in an image editor, or discard the result and start the computation from scratch with different settings.

To avoid this, a class of *progressive* algorithms have been developed. While these also often require input parameters, an ideal progressive algorithm is designed so that no matter what the parameters are, the result will eventually converge to the correct solution. The rendering can then be stopped at any point, and if the intermediate result is satisfactory, it may be used; or if not, rendering may be resumed to improve quality without discarding previous results.

Out of these algorithms, several that are known to be *consistent* – that is, theoretically guaranteed to converge to the correct solution – will be selected, implemented on the GPU via the NVIDIA CUDA platform, and their rates of convergence will be compared on different scenes. Such a comparison is particular to this thesis, because these algorithms are progressive, so the rate of convergence of a single run can be analysed, rather than rates of convergence of different runs configured to run for a given time, and they are all implemented on the GPU, which impacts the computation cost of even the most elementary operations and upsets the traditional measures, such as convergence per paths traced, or similar.

Thesis Structure

This thesis shall be organized as follows: *Chapter* 2 will present the theoretical groundwork for the rest of the thesis. Radiometric quantities and their relations as well as the *Rendering Equation* will be briefly presented, along with a short introduction into *Monte Carlo estimators* and on how they may be used to solve the aforementioned equation. *Chapter* 3 will present each of the GI solving methods we have chosen to implement in greater detail, from the standpoint of their mathematical formulation. In *Chapter* 4, we will describe how can the methods from the previous chapter be paralelized and implemented on a SIMD architecture, or rather CUDA in particular. *Chapter* 5 will present the methodology and results of the testing of our implementations, as well as a discussion on the meaning and reliability of these measurements. *Chapter* 6 will conclude the thesis with a discussion of the outcome of the testing and with proposals for future research.

Chapter 2

Fundamental Theory

In this chapter, we are going to present some of the fundamentals of the theory of realistic image synthesis. First, mainly for reference, we will present an overview of basic radiometric quantities and the relations between them. After that, we will present the *Rendering Equation* and related concepts and finally, we will describe some of the theory of *Monte Carlo estimators*, and how it pertains to the problem at hand.

2.1 Radiometric Quantities

In attempting to model behaviour of light within a system of interest in a physically correct manner, we first need to be able to quantitatively describe aspects of that system. To this end, we will be using *Radiometric quantities* – these come from a field of physics concerning itself with description and modelling of propagation of radiation, and are commonly used as the basic theory of realistic image synthesis. A more thorough introduction to radiometry is presented in [DBB06].

Alternatively, we could have used *Photometric quantities*, which are used in practice to describe and predict illumination of real areas in the visible spectrum. The distinction is that while photometry measures radiation in terms of how it is perceived by human vision (and thus is restricted to the visible spectrum), radiometry measures it in terms of physical energy. It is important to note that any radiometric unit is always used in relation to a particular frequency or band of frequencies of electromagnetic radiation, while photometric units are always understood to represent total visual brightness. It is possible to use photometric rather than radiometric quantities to describe image synthesis, as any radiometric quantity has a direct photometric equivalent. The relationship between them may be derived from the base relationship of radiant flux, expressed in watts, and luminous flux, expressed in lumens. The conversion is done by considering the spectral distribution of radiant flux and weighting it with a *Luminosity function*.

As has been mentioned, the basis of radiometry is **Radiant energy** Q $[J]$. This quantity represents total electromagnetic energy within the area of interest at a point in time[1]. However, this quantity is usually of little interest, as it cannot easily be assigned a visual equivalent.

[1]According to quantum physics, we may calculate this energy as a sum of all photons, or $E = \sum_{photons} hf_p$, where h is the *Planck constant* and f_p is the frequency of each particular photon.

A more tangible quantity is the **Radiant flux**:

$$\Phi = \frac{\mathrm{d}Q}{\mathrm{d}t} \ [W]$$

In everyday use, it is commonly associated with light sources as the total energy a light source emits per unit of time (that is, a second) and represents total energy incident to a particular surface, regardless of direction, per unit of time. In realistic image synthesis, it is commonly quantized to *photons*, where each photon carries a quantum of radiant flux and has an associated position and direction at a point in time[2].

Differentiating radiant flux emitted per unit area, regardless of direction, yields **Radiosity** B (also called **Radiant exitance**). A photometric counterpart of this particular quantity is called *Luminance* L_V $\left[\frac{Cd}{m^2}\right]$ and is used in everyday practice to measure brightness of computer screens, for instance. Radiosity may be expressed in a differential form thus:

$$B = \frac{\mathrm{d}\Phi}{\mathrm{d}A} \left[\frac{W}{m^2}\right] \tag{2.1}$$

If we consider incoming, rather then outgoing radiant flux, we get **Irradiance** E, which is otherwise identical to Radiosity.

If we choose to take direction into account as well, we get radiant flux incident to a point on surface per solid angle, a quantity called **Radiance** L. Radiance may be either incoming or exitant and is a quantity associated with a ray, as (barring participating media) radiance along a ray is constant. This is the most important quantity in realistic image synthesis, because it is the quantity that elicits sensor response (that is, we may actually see or photograph it). Because radiance is usually measured with respect to a point of surface, it is projected and may be expressed thus:

$$L = \frac{\mathrm{d}E}{\mathrm{d}\vec{\omega}\cos\theta} \left[\frac{W}{m^2 srad}\right]$$

As it will be used prolifically, radiance warrants some further attention. In the rest of this thesis, we will be using subscripts to indicate what kind of radiance is meant, using L_i for incoming radiance, L_o for outgoing radiance, and L_e for emitted radiance. $L_i(\mathbf{x}, \vec{\omega})$ shall denote incoming radiance at a particular point $\mathbf{x}$ from direction $\vec{\omega}$. Analogically, $L_o(\mathbf{x}, \vec{\omega})$ and $L_e(\mathbf{x}, \vec{\omega})$ shall denote outgoing and emitted radiance from a point $\mathbf{x}$ in direction $\vec{\omega}$, respectively.

This concludes our brief introduction into radiometric quantities.

2.2 The Rendering Equation

Rendering in realistic image synthesis may be formulated in radiometric terms as follows: For a given image plane, incoming radiance is integrated over the area of each individual pixel. This is done by the

[2]These photons are, however, distinct from photons in the physical sense. For one, physical photons carry energy rather than flux, and in practice a single "graphics" photon represents many orders of magnitude more of physical photons. This makes attempting to simulate waveform properties of light impractical at best, and quite impossible in practice.

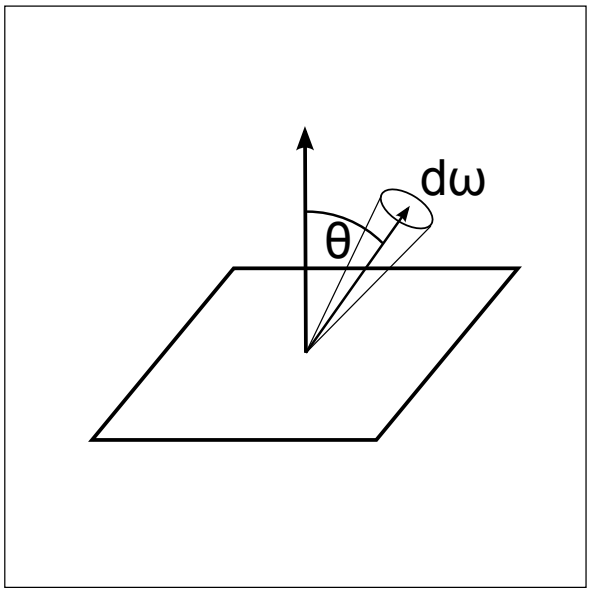


Figure 2.1: How radiance is differentiated at a point of surface

way of drawing samples from the image plane and using a camera model to project them into the scene. As each of the samples has a position (as determined by its position within the image plane and the image plane's position within the scene) and orientation (which is necessary because radiance is always associated with a direction), projecting it into the scene, we get a ray[3].

Barring participating media, radiance along this ray may be determined by the way of *Ray Tracing*. That is, we trace the ray to find its intersection with an object in the scene and then attempt to determine exitant radiance at that given point of scene in that particular outgoing direction. The question of just how exactly do we go about doing that is answered by the *Rendering equation*.

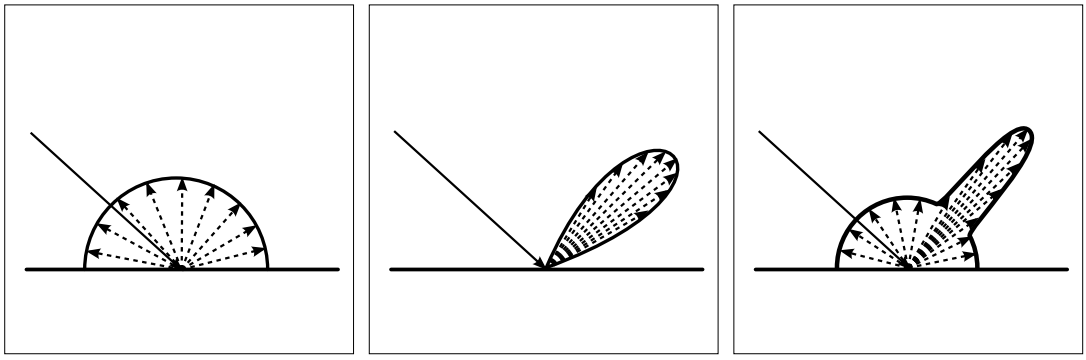
The *Rendering Equation*, first published by James Kajiya [Kaj86], describes the relation between material properties, incoming radiance and exitant radiance. Exitant radiance is calculated as a sum of emitted radiance and reflected radiance. Both of these terms depend on material properties – the first one is directly a material property, as we have to require description of light sources be part of scene information, while the other is dependent on both the incoming radiance and the material's reflective properties.

The reflective properties may be quantitatively described by a *Bidirectional Reflectance Distribution Function*. In radiometric terms, it describes the relationship between incoming radiance $L_i(\mathbf{x}, \vec{\omega}_i)$ and reflected radiance $L_r(\mathbf{x}, \vec{\omega}_o)$ at a given point thus:

$$f(\vec{\omega}_i, \vec{\omega}_o, \mathbf{x}) = \frac{\mathrm{d}L_r(\mathbf{x}, \vec{\omega}_o)}{L_i(\mathbf{x}, \vec{\omega}_i) \cos\theta_i \ \mathrm{d}\vec{\omega}_i}$$

In addition, laws of physics (or more precisely, radiometry) dictate that the BRDF of a realistic material has to satisfy several constraints:

[3]This is a necessary consequence of the fact mentioned in the previous section, that is, radiance is quantity associated with a ray and constant along a ray.



(a) Diffuse (Lambertian) BRDF (b) Specular (Glossy) BRDF (c) Generic BRDF

Figure 2.2: Visualisations of different possible BRDFs. A solid arrow represents incoming radiance, while dashed arrows represent exitant radiance. *Figures* 2.2a and 2.2b represent classic model BRDFs, but in practice, a BRDF will often look as something in between, as seen in *Figure* 2.2c.

Non-negativity $\forall \vec{\omega}_i, \vec{\omega}_o : f(\vec{\omega}_i, \vec{\omega}_o, \mathbf{x}) \geq 0$

Helmholtz reciprocity $\forall \vec{\omega}_i, \vec{\omega}_o : f(\vec{\omega}_i, \vec{\omega}_o, \mathbf{x}) = f(\vec{\omega}_o, \vec{\omega}_i, \mathbf{x})$

Conservation of energy $\forall \vec{\omega}_i : \int_\Omega f(\vec{\omega}_i, \vec{\omega}_o, \mathbf{x}) \cos\theta_o \ \mathrm{d}\vec{\omega}_o \leq 1$

Such BRDF is then called *physically plausible*. The integral $\int_\Omega f(\vec{\omega}_i, \vec{\omega}_o, \mathbf{x}) \cos\theta_o \ \mathrm{d}\vec{\omega}_o$ is called **Albedo** ρ, a dimensionless quantity describing how much of incoming light is reflected by the material, the rest being absorbed.

Since the BRDF is characteristic for the material being modelled, it has to be part of the scene description as well. In practice, BRDF values are either experimentally measured, using a specialized device called *gonioreflectometer*, or modelled by an analytical model (also called *Empirical BRDF*). While the measured BRDFs have the advantage of being more faithful to the original and thus more realistic, empirical models have other desirable properties. The two most important of these are that empirical models may be analytically inverted and directly used as a PDF for importance sampling, and that most of the commonly used BRDFs may be separated into their *diffuse* and *specular* components.

For this reason, we will be using the *Extended Phong BRDF Model* in this thesis. This is an empirical BRDF derived from the original Phong model [Pho75] as extended by Lafortune [LW94], as the original does not necessarily conserve energy and thus was not physically plausible. Lafortune's Phong model [LW94] may be analytically inverted and allows the separation of diffuse and specular components.

As a side note, the BRDF is not the only possible way of modelling interaction of a material with light. If we also consider transparent (refractive) materials, we get the *Bidirectional Transfer Function*, which is analogous to the BRDF in all respects save that it permits incoming and outgoing directions below the normal plane, and allowing for sub-surface scattering we get the *Bidirectional Surface Scattering Reflectance Distribution Function*, which also has to be integrated over an area. The extension of all the methods we will be using to cover these more complicated material descriptions is quite straightforward, but will not be explored in this thesis.

Having formulated our material description convention, we may now formulate the rendering equation [Kaj86] as follows:

$$L_o(\mathbf{x}, \vec{\omega}_o) = L_e(\mathbf{x}, \vec{\omega}_o) + \int_\Omega L_i(\mathbf{x}, \vec{\omega}_i)\, f(\mathbf{x},\ \vec{\omega}_i,\ \vec{\omega}_o) \cos\theta_i \, \mathrm{d}\vec{\omega}_i$$

As is immediately obvious, we need to be able to integrate the product of incoming radiance, BRDF and the dot product of the incoming direction and shading normal, if we are to arrive at a solution of this equation. However, given that the incoming radiance can almost never be analytically integrated (save for the most trivial of scenes), the only solution we may attempt is a numerical one. Furthermore, the equation is recursive and in order to get $L_i(\mathbf{x}, \vec{\omega}_i)$, we will have to resort to using the ray tracing operator once more and solving the same equation at a different point of scene.

Before we propose a method to solve this equation, we should note that there are multiple other ways in which this equation can be formulated, most important for our purposes being the *area formulation*:

$$L(\mathbf{x}, \vec{\omega}_o) \ = \ L_e(\mathbf{x}, \vec{\omega}_o) + \int_A L(\mathbf{y},\ \mathbf{y} \to \mathbf{x}) f_r(\mathbf{x},\ \mathbf{x} \to \mathbf{y},\ \vec{\omega}_o)\, V(\mathbf{x}, \mathbf{y})\, \frac{\cos\theta_x \cos\theta_y}{\|\mathbf{x} - \mathbf{y}\|}\, \mathrm{d}A_{\mathbf{y}}$$

Where $\mathbf{y} \to \mathbf{x}$ is shorthand for the direction from $\mathbf{y}$ to $\mathbf{x}$ from the point of view of $\mathbf{y}$ and $V(\mathbf{x}, \mathbf{y})$ is the *Visibility term*, which evaluates to 1 if and only if there is unoccluded line of sight from $\mathbf{y}$ to $\mathbf{x}$ and to 0 otherwise. This formulation integrates over the entire surface area of the scene rather than over a hemisphere at a point, which has repercussions on possible sampling schemes, but in the limit, yields the same result. This particular formulation is in fact the basis of *Radiosity* [GTGB84] method.

2.3 Monte Carlo Methods

Monte Carlo is a method of stochastically estimating the result of a deterministic calculation that is either very complex or impossible to solve analytically. While the term was coined by Stanislaw Ulam and John von Neumann [MU49, Met87] as a part of their work on simulating neutron behaviour in a nuclear device, this methods date back to Comte de Buffon and his famous needle throwing experiment, which Pierre Laplace noted could be used to experimentally estimate the value of π [Bad94].

In Monte Carlo, samples drawn from a random variable are used as an input for a deterministic calculation. The results of this function of a random variable are then aggregated and this aggregation is taken to be the result of the calculation in question. This may be applied to many problems, the most significant of which (for our purposes, at least) is *Monte Carlo Integration*.

2.3.1 Monte Carlo Integration

While the principles and use of Monte Carlo integration are well documented in literature [DBB06] (as is the fact that deterministic methods are much more efficient at estimating one-dimensional integrals), we shall nonetheless provide a brief introduction, so that we may build on it later.

Suppose we have a function $f(x)$ defined over $x \in [a,b]$. If we wish to evaluate the integral

$$I = \int_a^b f(x)\,\mathrm{d}x,$$

we may do so by drawing samples from a uniform random variable G that distributes samples $g_1 \dots g_n$ evenly over $[a,b]$ and aggregating the results into an estimator

$$\langle I \rangle = \frac{1}{N}\sum_{i=1}^{N}\frac{f(g_i)}{p(g_i)},$$

where N is the total number of samples drawn so far and $p(g_i)$ is the *probability density function* (PDF) corresponding to the distribution of variable G, which in this case evaluates to a constant $\frac{1}{b-a}$. Thus we may rewrite the estimator like this:

$$\langle I \rangle = \frac{1}{N(b-a)}\sum_{i=1}^{N} f(g_i)$$

Using the original form, we may analytically calculate the expected value of $\langle I \rangle$:

$$E\left[\langle I \rangle\right] = E\left[\frac{1}{N}\sum_{i=1}^{N}\frac{f(g_i)}{p(g_i)}\right] = \frac{1}{N}\sum_{i=1}^{N}E\left[\frac{f(g_i)}{p(g_i)}\right] = \frac{1}{N}N\int\frac{f(g)}{p(g)}\,p(g)\,\mathrm{d}g = \int f(g)\,\mathrm{d}g = I$$

The variance of this estimator can be calculated as follows:

$$\sigma^2 = \frac{1}{N}\int\left(\frac{f(g)}{p(g)} - I^2\right)p(g)\,\mathrm{d}g,$$

which goes to show that the standard deviation of the estimate will be on the order of $O\left(\frac{1}{\sqrt{N}}\right)$. Note that this is only asymptotically true – the actual standard deviation may differ significantly due to hidden constants caused by the ratio $\frac{f(g)}{p(g)}$ and multiplication by $p(g)$ itself. While this asymptotic rate of convergence is not great, it is one of the strengths of the Monte Carlo method. Using multivariate distributions, Monte Carlo integration can easily be extended into arbitrary dimension while maintaining this rate of convergence. In comparison, in d dimensions, the classic quadrature integration has a rate of convergence of $O\left(\frac{1}{\sqrt[d]{N}}\right)$. Also note that this is a stochastic bound – the result will be a random variable with an unknown distribution, but with the expected value of I and the aforementioned standard deviation. As the number of samples increases, the standard deviation decreases and value of the estimator is more likely to fall close to the expected value.

2.3.2 Improving Convergence

In the previous section, we have demonstrated how we can estimate the value of a definite integral by uniformly randomly sampling the integration domain and averaging the function results at these samples.

While this approach is robust and versatile, there are strategies that allow us to improve convergence with the same number of samples, using additional information about the integral in question.

The first of these is *Stratified Sampling*, also called *Jittering* in the context of computer graphics. This simply means that rather than attempting to calculate a single definite integral, we split the integration domain into several disjoint subdomains (or *strata*), estimate each of them separately and sum the results, in effect calculating several definite integrals. This reduces variance caused by "clustering", where randomly drawn samples are concentrated in a particular subarea of the domain, rather than distributed evenly[4]. Using stratified sampling, we are more likely to explore greater part of the integration domain (or in layman's terms, at most 2^d samples may "clump" together in d dimensions) and reduce variance more quickly. However, this becomes difficult as the dimensionality grows, and stratifying becomes more complex. Also, we may not draw an arbitrary number of samples, but have to instead draw a multiple of the number of strata.

Another method is using *Importance Sampling*. This means that rather than a uniform distribution, we use some other distribution that we believe to be more "similar" to the function we are integrating. In fact, if $p(g) \propto f(g)$, we get the correct value upon drawing the very first sample, and have zero variance altogether. The PDF does not need to be identical to the function we are integrating to get a correct estimator. The only limitation is that this PDF has to be non-zero wherever $f(x)$ is non-zero. However, a poor choice of $p(g)$ may actually *increase* variance rather than decrease it, because we will spend more time sampling parts of the domain that do not significantly contribute to the result.

If the integrand is a product of known functions and an unknown one, we may pick a PDF that is proportional to the known factors. In computer graphics, and solving the rendering equation in particular, we commonly use a PDF proportional to $f(\vec{\omega}_i, \vec{\omega}_o, \mathbf{x}) \cos\theta_i$ for sampling. Furthermore, there are several methods of randomly sampling a known PDF using a source of uniform random samples. If we can analytically calculate the *Quantile Function* (an inverse of the *Cumulative Distribution Function*), we may directly use it to draw samples from a given distribution. Otherwise, if we only know the PDF an its maximum, we may use *Rejection Sampling*. One of the advantages of empirical BRDFs is that their quantile functions are known, and we can thus sample them rather efficiently.

Also of note is, that if we know the quantile function, we may combine importance sampling and stratified sampling for an additional improvement in convergence – rather than using a uniform distribution as an input for the quantile function, we use a stratified uniform distribution.

Further methods, such as *Metropolis Sampling* or *Quasi Monte Carlo* exist, and are dealt with in introductory works such as [AFH+01], but are beyond the scope of this thesis.

2.3.3 Unbiased, Biased and Consistent Methods

For the purposes of Monte Carlo Integration, we have introduced an estimator $\langle I_N \rangle$, that draws N samples and uses them to estimate value of a definite integral I. We have further demonstrated that the expected

[4]This is possible because we assume the samples are *independent, identically distributed*. Independence implies that any new sample is drawn completely regardless to positions of those already drawn, so there is nothing to stop us from drawing them in close proximity to each other. Note that this only affects variance, not the final outcome – in the limit, the result will be correct.

value of this estimator is equal to the value of the integral, regardless to the number of samples drawn:

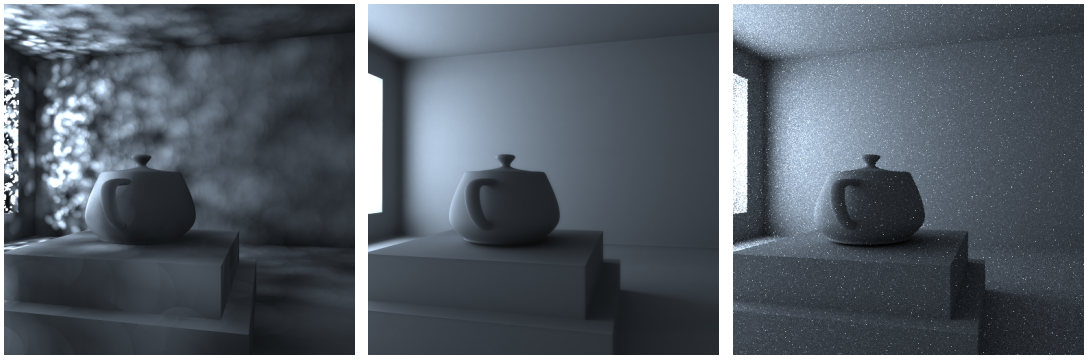
$$\forall N \geq 1, E\left[\langle I_N \rangle\right] = I$$

Now, there is a property of a Monte Carlo estimator commonly called *Bias*. This is defined, for N samples drawn, as follows[5]:

$$B_N = \left|E\left[\langle I_N \rangle\right] - I\right|$$

The estimator $\langle I_N \rangle$ we have introduced earlier has a bias of 0 for any positive N. We may thus call it *unbiased*. If this were not so, and the expected value would differ from the final result, such an estimator would be called *biased*. Apart from this, there is a special class of *consistent* estimators, for which the following holds:

$$\lim_{N \to \infty} B_N = 0$$



(a) Biased estimate (b) Reference (c) Estimate with variance

Figure 2.3: Estimation Error in Realistic Image Synthesis – the correctly converged result (2.3b) compared to an unconverged biased estimate (2.3a) and an unconverged unbiased one (2.3c).

The advantage of using consistent estimators is that while they still converge to the correct result (which is guaranteed by their bias being zero in the limit), they may exhibit lower variance in particular situations, leading to a lower overall error[6]. Exploring exactly how the overall error behaves is the principal purpose of this thesis.

2.3.4 Progressiveness

In addition to properties already defined, we may judge an estimator based on whether or not it is *progressive*. An estimator is called *progressive* if it produces a useful estimate for any number of samples N, and is capable of incrementally drawing more samples to improve the estimate. Any such estimator is effectively

[5]Note that bias is commonly introduced without the lower index. We have used it here merely as a convenient shorthand to help understand this particular equation. As it is at best non-trivial to actually calculate bias of a given biased estimator, we will not be using it much more.

[6]On the other hand, unbiased estimators have an arguably advantageous property that any weighted average of any number of unbiased estimates is in itself an unbiased estimate.

formulated recursively, which may be used to prove progressiveness. For instance, our estimator $\langle I \rangle$ may be formulated as follows:

$$I_{N+1} = \frac{1}{N+1} \sum_{i=1}^{N+1} \frac{f(g_i)}{p(g_i)}$$

$$I_{N+1} = \frac{1}{N+1} \left(\left(\sum_{i=1}^{N} \frac{f(g_i)}{p(g_i)} \right) + \frac{f(g_{N+1})}{p(g_{N+1})} \right)$$

$$I_{N+1} = \left(\frac{1}{N+1} \sum_{i=1}^{N} \frac{f(g_i)}{p(g_i)} \right) + \frac{1}{N+1} \frac{f(g_{N+1})}{p(g_{N+1})}$$

$$I_{N+1} = \left(\frac{1}{N+1} \frac{N}{N} \sum_{i=1}^{N} \frac{f(g_i)}{p(g_i)} \right) + \frac{1}{N+1} \frac{f(g_{N+1})}{p(g_{N+1})}$$

$$I_{N+1} = \left(\frac{N}{N+1} \frac{1}{N} \sum_{i=1}^{N} \frac{f(g_i)}{p(g_i)} \right) + \frac{1}{N+1} \frac{f(g_{N+1})}{p(g_{N+1})}$$

$$I_{N+1} = \left(\frac{N}{N+1} \frac{1}{N} \sum_{i=1}^{N} \frac{f(g_i)}{p(g_i)} \right) + \frac{1}{N+1} \frac{f(g_{N+1})}{p(g_{N+1})}$$

$$I_{N+1} = \frac{N}{N+1} I_N + \frac{1}{N+1} \frac{f(g_{N+1})}{p(g_{N+1})},$$

with a postulation that $I_0 = 0$. We have thus demonstrated that the basic Monte Carlo estimator is not only unbiased, but also progressive. This holds for any such estimator based on importance sampling, but does not hold for estimators based on stratified sampling. This is because we have stratified the integration domain for a given number of expected samples, and if we were to re-stratify it, we could no longer use any of the previous results. We can, however, add samples in each of the strata, but that is a different formulation that sacrifices some of the advantages that convinced us to use stratified sampling in the first place.

However, it does not necessarily hold that:

$$\forall i > j, |\langle I_i \rangle - I| < |\langle I_j \rangle - I|$$

This is because the estimator is still effectively a random variable, and while asymptotically the error is guaranteed to decrease, error with number of samples drawn is not necessarily a monotonous function even for unbiased estimators.

Note that bias and progressiveness are independent properties. We have shown we have a progressive unbiased estimator, as well as a non-progressive unbiased one – we might as well have defined a progressive biased estimator, or a non-progressive biased one. In fact, as the title of this thesis suggests, the group of biased (consistent) progressive estimators is rather significant and in our opinion, merits research.

Chapter 3

Selected Algorithms

This chapter shall present the algorithms we have chosen to implement. These will include *Path Tracing*, three different algorithms of *Filtered Path Tracing*, *Progressive Photon Mapping*, *Stochastic Progressive Photon Mapping* and *Progressive Photon Ray Splatting*. The theoretical basis of each of these techniques shall be explained in this chapter, while their parallel implementation on the CUDA platform shall be described in detail in the following chapter.

3.1 Path Tracing

Path Tracing, introduced by Kajiya in the same paper as the Rendering Equation [Kaj86], is a straightforward combination of the equation (in its directional formulation) with a Monte Carlo estimator. Upon tracing the *primary ray*, the rendering equation at the point of impact is estimated by casting a single ray, and so on recursively. This sequence of rays, illustrated in *Figure* 3.1, is called a *path*, hence *Path Tracing*.

In addition, if we are using simple model materials, like the first two in *Figure* 2.2, or a BRDF that is a simple composition of these, we may view each bounce of the path as either a specular or a diffuse interaction. It is common practice to describe paths by their interactions, using a combination of letters D (for diffuse) and S (for specular). A path may then be described as a string of these letters starting from a certain direction of interest (eg. "DD path", "SD path", etc.), or as a substring describing only the start of the path. More complex descriptive grammars exist, but will not be used in this thesis.

Usually, the rays are terminated either after a given number of bounces, or when their contribution (that is, the product of the BRDF and cosine terms accumulated along the path) gets lower than a preset threshold. This approach, however, introduces bias, as it may inadvertently eliminate paths with significant contribution. To avoid both bias and infinitely long paths that would appear in closed scenes, we instead terminate paths by the way of *Russian Roulette*, proposed in [DLW93]. At each bounce a termination probability is determined. A sample is randomly drawn from a uniform $[0, 1]$ distribution, and if it is less than the determined probability, the path is terminated. Otherwise, the path continues with a contribution divided by the termination probability. For convenience, surface albedo at bounce point is used as termination probability, which corresponds to the physical behaviour of light.

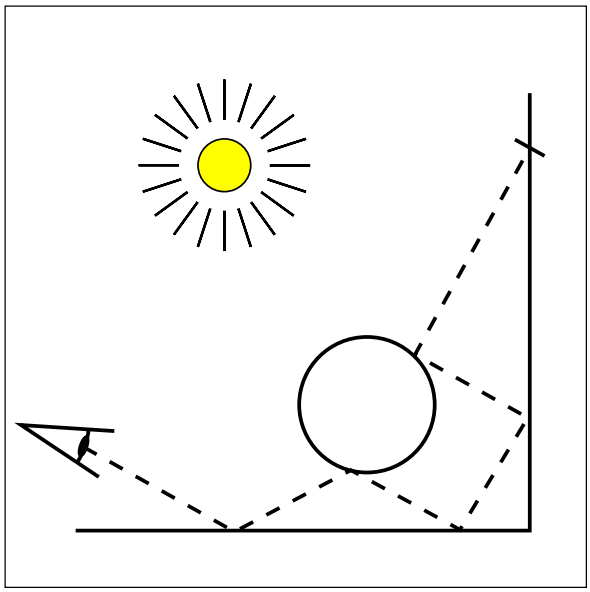


Figure 3.1: An example path tracing path. Notice that it starts at the eye and never hits a light source.

Based on the idea of importance sampling, we may attempt to improve the convergence of path tracing by selecting a good PDF for sampling. Since we know the contribution of a sample is proportional to the expression $f(\vec{\omega}_i,\ \vec{\omega}_o,\ \mathbf{x})\cos\theta_i$, we will choose a PDF to be proportional to this expression as well. This is a common approach in path tracing that helps improve convergence especially on glossy surfaces. Using an empirical BRDF with a known inversion allows efficient sampling directly via the quantile function.

To get a proper PDF out of the term $f(\vec{\omega}_i,\ \vec{\omega}_o,\ \mathbf{x})\cos\theta_i$, we need to normalize it so that it integrates to 1 on a hemisphere. As we know that for a fixed $\vec{\omega}_i$ this expression integrates to albedo ρ, we can get a PDF by simply dividing by albedo:

$$p(\mathbf{x},\ \vec{\omega}_o) = \frac{f(\vec{\omega}_i,\ \vec{\omega}_o,\ \mathbf{x})\cos\theta_i}{\rho}$$

Using this PDF to draw a sample, we get the following Monte Carlo estimator for reflected radiance:

$$\frac{L_i(\mathbf{x},\vec{\omega}_i)\, f(\mathbf{x},\ \vec{\omega}_i,\ \vec{\omega}_o)\cos\theta_i}{\frac{f(\mathbf{x},\ \vec{\omega}_i,\ \vec{\omega}_o)\cos\theta_i}{\rho}\,\rho},$$

which can be simplified to:

$$L_i(\mathbf{x},\vec{\omega}_i)$$

Taking these modifications into account, we get the following as a radiance estimate:

$$\langle L_o(\mathbf{x},\vec{\omega}_o)\rangle = \begin{cases} L_e(\mathbf{x},\vec{\omega}_o) + L_i(\mathbf{x},\vec{\omega}_i) & \text{if } t \geq \rho \\ L_e(\mathbf{x},\vec{\omega}_o) & \text{if } t < \rho \end{cases},$$

where t is a random variable drawn from a $[0,1]$ uniform distribution.

This is, of course, contingent on $\vec{\omega}_i$ being chosen from the aforementioned PDF, and on only a single sample being drawn. If we were to draw more samples, the second term of the estimator would be expressed as an arithmetic mean of the samples:

$$\langle L_o(\mathbf{x},\vec{\omega}_o)\rangle = L_e(\mathbf{x},\vec{\omega}_o) + \frac{1}{N}\sum_{n=1}^{N} L_i(\mathbf{x},\vec{\omega}_{i_n})$$

Assuming that most of the scene surfaces are non-emissive, using the directional formulation, it is rather unlikely that a path hits a light source (point light sources, which are impossible to hit with a ray, notwithstanding). Thus most vertices, and indeed most paths would have zero total contribution. To counter this, path tracing is usually further refined by separating the computation of emission and reflection terms. This is called *Next Event Estimation* (illustrated in *Figure* 3.2) and uses the area formulation of the rendering equation to modify the estimator thus:

$$\langle L_o(\mathbf{x},\vec{\omega}_o)\rangle = \frac{L_e(\mathbf{y},\mathbf{y}\to\mathbf{x})\, f(\mathbf{x},\ \vec{\omega}_o,\ \mathbf{y}\to\mathbf{x})}{p(y)}\ \cos\theta_x\ \cos\theta_y\, V(\mathbf{y},\mathbf{x})\ \|\mathbf{y}-\mathbf{x}\|\ +\ L_i(\mathbf{x},\vec{\omega}_i)$$

In this equation, $\mathbf{y}$ is a randomly selected point in scene drawn from distribution Y with a PDF $p(y)$. Importance sampling is commonly used here as well, usually with a PDF proportional to emitted radiosity B_e (recall *Equation* 2.1) at a given point. Assuming only area light sources with uniform emission, we get:

$$p(y) \propto B_e(y)$$

or more precisely

$$p(y) = \frac{B_e(y)}{\Phi_{TOTAL}},$$

where Φ_{TOTAL} is the total emitted radiant flux of all lights in the scene. This is easily calculated by summing up the emitted radiant flux of each of the lights, which in turn is calculated as area multiplied by emitted radiosity. If we also assume only diffuse light sources, it holds that

$$L_e(\mathbf{y},\vec{\omega}_o) = \frac{B(y)}{\pi}, \forall\vec{\omega}_o$$

And we may express the estimator as:

$$\langle L_o(\mathbf{x},\vec{\omega}_o)\rangle = \frac{L_e(\mathbf{y},\mathbf{y}\to\mathbf{x})\, f(\mathbf{x},\ \vec{\omega}_o,\ \mathbf{y}\to\mathbf{x})}{\frac{B_e(y)}{\Phi_{TOTAL}}}\ \cos\theta_x\ \cos\theta_y\, V(\mathbf{y},\mathbf{x})\ \|\mathbf{y}-\mathbf{x}\|\ +\ L_i(\mathbf{x},\vec{\omega}_i)$$

$$\langle L_o(\mathbf{x},\vec{\omega}_o)\rangle = \frac{L_e(\mathbf{y},\mathbf{y}\to\mathbf{x})\, f(\mathbf{x},\ \vec{\omega}_o,\ \mathbf{y}\to\mathbf{x})\,\Phi_{TOTAL}}{L_e(\mathbf{y},\mathbf{y}\to\mathbf{x})\,\pi}\ \cos\theta_x\ \cos\theta_y\, V(\mathbf{y},\mathbf{x})\ \|\mathbf{y}-\mathbf{x}\|\ +\ L_i(\mathbf{x},\vec{\omega}_i)$$

$$\langle L_o(\mathbf{x},\vec{\omega}_o)\rangle = f(\mathbf{x},\ \vec{\omega}_o,\ \mathbf{y}\to\mathbf{x})\frac{\Phi_{TOTAL}}{\pi}\ \cos\theta_x\ \cos\theta_y\, V(\mathbf{y},\mathbf{x})\ \|\mathbf{y}-\mathbf{x}\|\ +\ L_i(\mathbf{x},\vec{\omega}_i),$$

with $V(\mathbf{y},\mathbf{x})$ evaluated by casting an additional *shadow ray*.

As has been demonstrated in the previous chapter, such an estimator is both progressive and unbiased. Additionally, with a path tracer, we may start each path at a different position within the pixel (or in other words, with a different primary ray) and we may thus get progressive anti-aliasing, as well.

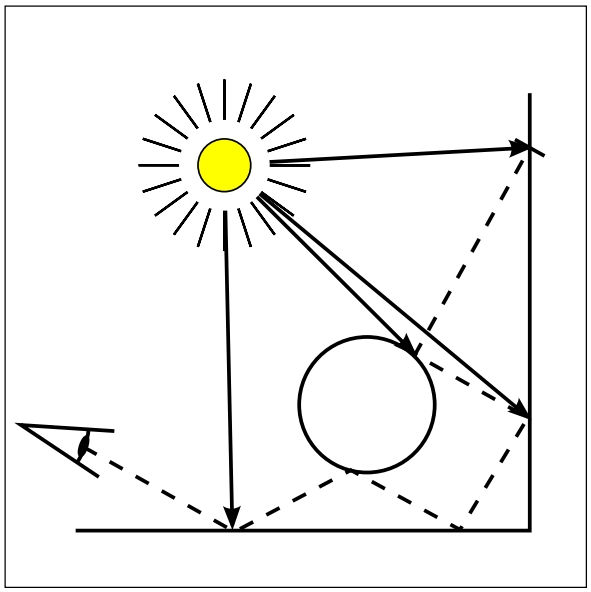


Figure 3.2: How a light source contributes to a path when using *Next Event Estimation*.

3.2 Filtered Path Tracing

One proposed way to deal with noise exhibited by path tracing has always been through the use of post-processing filters. These filters define a matrix of coefficients, whereby each pixel in the resultant image is calculated as a weighted sum of its neighbouring pixels in the original image, with matrix members determining the weight. If the sum of these weights is equal to 1, we call such a filter *Energy Preserving*. A correctly implemented energy preserving filter does not alter the total luminance of the image[1]. These energy preserving filters tend to be based on statistical kernels, as they share some of their properties.

The purpose of these filters is usually to spread out the radiance of the noisy (too bright) pixels in their neighbourhood. This trades variance for bias, exhibited as blur. While blur tends to be more visually pleasing on flat surfaces than high-frequency white noise (as seen in *Figure* 2.3c), it becomes visually disruptive on object boundaries, as these filters usually blur the image indiscriminately. The challenge, then, is to find the right balance by selecting such filter shape and bandwidth, that variance is reduced as much as possible, while introducing the minimum amount of blur. The approaches we will describe here achieve this either by only filtering certain components of the image, or by using locally-varying kernel footprint based on local image characteristics.

The filters we will be implementing are Jensen's and Christensen's component separating filter [JC94], Willems' and Suykens' adaptive kernel filter [SW00] and Xu's and Pattanaik's bilateral filter [XP05].

[1]"Correct implementation" in this case means that pixels along the border should be handled correctly, because in their case, part of the filter falls outside of the image. There are multiple ways to handle this – one is to render the original image slightly bigger (which is correct, but not in fact energy conserving), the other is to only consider pixels in the image, but to normalize the filter matrix in such a way that the used weights sum up to 1.

3.2.1 Component Separating Kernel Filter

This filtering technique was proposed by Jensen in Christensen in 1994 [JC94]. It is based on the assumption that most of the variance is introduced through diffuse inter-reflections, or more precisely, paths that start with a double diffuse bounce from the eye. The proposed solution is to store these paths in a separate image buffer and, prior to visualization, filter them and then add them to the rest of the image.

The paper proposed the use of either a kernel filter or a median filter. These would eliminate noise while not disrupting high-frequency features, such as direct illumination changes at object boundaries. Furthermore, if an energy preserving kernel filter is used[2] and the filtering only occurs directly before visualization, we retain the option of presenting the original, unbiased image.

The size and shape of the matrix filter are important parameters. Shaped kernels (such as the *Epanechnikov kernel*) are thought to be better for this purpose than box (or flat) kernels, because they are better at preserving high-frequency features and do not introduce as much blur. The size of the matrix filter is then a trade-off between noise and blur. In the original paper [JC94], a 3×3 pixel shaped kernel filter was determined to be best at eliminating variance.

3.2.2 Adaptive Kernel Filter

Adaptive Kernel Filter proposed by Suykens and Willems [SW00] operates on individual paths, rather than pixels. Image synthesis is first formulated as kernel density estimation in the image plane, where value at each pixel $p(t)$ is estimated thus[3]:

$$\langle p(t) \rangle = \frac{1}{N h^2} \sum_{i=1}^{N} \mathscr{K} \left(\frac{t - x_i}{h} \right),$$

where h is the kernel width, x_i is a sample (path) value, and $\mathscr{K}$ is a kernel function. However, this estimator necessarily introduces bias proportional to the size of kernel footprint h. Again, there is a trade-off here between reducing variance effectively (with a larger h) and introducing minimum bias (with a smaller h). Suykens mentions that one of the ways to optimize this trade-off is to derive the kernel width from the properties of a sample in relation to its surroundings (called *Variable Kernel Density Estimation*).

The concept of determining the kernel width from the value of a previous estimate is then introduced. Based on different previous estimates of optimum heuristic, they note that kernel bandwidth should be proportional to

$$h(x_i) \propto \sqrt{\frac{1}{f(x_i)}}$$

[2]Note that a median filter can never be energy preserving.

[3]It should be mentioned that their method was developed for *Bidirectional Path Tracing* [LW93], which is more complicated and less well suited for GPU implementation than regular path tracing. However, the filtering is trivially adapted, since we can simply disregard the weights $G_{k,i}$ that come with BDPT samples and go on from there.

Furthermore, they note that to obtain a progressive estimate that converges to the correct solution, bandwidth should converge to 0 as the number of samples N increases. Therefore, they propose the following heuristic:

$$h(x_i) = C \cdot \left(\frac{1}{N}\right)^{\alpha} \cdot \sqrt{\frac{1}{f(x_i)}}, \tag{3.1}$$

where α is a user-determined exponent between $1/2$ and $1/6$ and C is a reference base kernel width.

Based on this heuristic, they propose an algorithm that progressively refines a filtered estimate:

1. Trace N_0 paths per pixel and store the screen hits (x_i)
2. Compute a reference image f'_0 using a fixed width kernel density estimation (with width $h = C' \cdot N_0^{-1/2}$)
3. Compute the image f_0 with a variable width kernel density estimation, using f'_0 as reference
4. Dispose of stored hits
5. $N_{total} = N_0$
6. Each iteration:
   (a) Choose a number of samples N_j
   (b) $N_{total} += N_j$
   (c) $f_j = \frac{N_{total} - N_j}{N_{total}} \cdot f_{j-1}$ (scale down)
   (d) Trace paths (N_j samples per pixel) and spread them through variable width kernel density estimation, using f_{j-1} as reference.

As is apparent from the algorithm description, this filtering method is progressive in nature. Adaptive kernel width ensures that high frequency features are preserved (as long as they are present in the reference image), while variance is reduced with brighter paths being spread out. Eventually, as kernel width goes to 0, paths are simply added instead of being spread out, and the algorithm becomes a regular path tracer. This guarantees that in the limit, there is no bias, and the algorithm is thus consistent.

3.2.3 Bilateral Kernel Filter

The de-noising operator put forward by Xu and Pattanaik [XP05] is based on earlier bilateral filters. An archetypical bilateral filter might look like this:

$$\langle p(x) \rangle = \frac{\int_A f(\xi)\, c(\xi, x)\, s(f(\xi), f(x))\ \mathrm{d}\xi}{\int_A c(\xi, x)\, s(f(\xi), f(x))\ \mathrm{d}\xi}$$

Where $\langle p(x) \rangle$ is the estimator of a pixel at x, $f(x)$ is the unfiltered value of the same pixel, ξ represents the coordinates of a pixel in the neighbourhood and $c(\xi, x)$ and $s(f(\xi), f(x))$ are the domain and range filter

kernels, respectively. Note that the denominator guarantees that the resulting filter is energy preserving by normalizing the result with the sum of weights used.

The filters are modelled by a Gaussian function with parameters σ_d and σ_r, respectively:

$$c(\xi,x) = \exp\left(-\frac{1}{2}\left(\frac{|\xi - x|}{\sigma_d}\right)^2\right) \tag{3.2}$$

$$s(f(\xi),f(x)) = \exp\left(-\frac{1}{2}\left(\frac{|f(\xi) - f(x)|}{\sigma_r}\right)^2\right) \tag{3.3}$$

The domain filter $c(\xi,x)$ reduces the contribution of incoherent neighbours (preserving, for instance, object boundaries), while the range filter $s(f(\xi),f(x))$ restricts contribution to a local neighbourhood. Xu, however, points out that this simple bilateral filter cannot be relied upon to suppress outliers. This is because a radically different pixel value will only assign minute weight to the neighbours that might moderate that value. On the other hand, it keeps the neighbourhood of an outlier from being "contaminated" with high variance.

To counteract this, Xu proposes to instead use a *near-true estimator* $\tilde{f}(x)$ as a domain reference:

$$\langle p(x)\rangle = \frac{\int_A f(\xi)\, c(\xi,x)\, s\left(f(\xi),\tilde{f}(x)\right)\, \mathrm{d}\xi}{\int_A c(\xi,x)\, s\left(f(\xi),\tilde{f}(x)\right)\, \mathrm{d}\xi}$$

At the same time, a Gaussian filtered input is proposed as $\tilde{f}(x)$:

$$\tilde{f}(x) = \frac{\int_A f(\xi)\, c(\xi,x)\, \mathrm{d}\xi}{\int_A c(\xi,x)\, \mathrm{d}\xi}$$

In a numerical formulation, where the integral is instead expressed as a discreet sum, we may restrict the filter window to a range of $6\sigma_d \times 6\sigma_d$ around the pixel of interest, as the value of the Gaussian beyond this range is negligible. That allows us to rewrite the expressions thus:

$$\langle p(i,j)\rangle = \frac{\sum_{u=-3\sigma_d}^{3\sigma_d}\sum_{v=-3\sigma_d}^{3\sigma_d} f(i+u,j+v)\, c(u,v)\, s\left(f(i+u,j+v),\tilde{f}(i,j)\right)}{\sum_{u=-3\sigma_d}^{3\sigma_d}\sum_{v=-3\sigma_d}^{3\sigma_d} c(u,v)\, s\left(f(i+u,j+v),\tilde{f}(i,j)\right)}$$

$$\tilde{f}(i,j) = \frac{\sum_{u=-3\sigma_d}^{3\sigma_d}\sum_{v=-3\sigma_d}^{3\sigma_d} f(i+u,j+v)\, c(u,v)}{\sum_{u=-3\sigma_d}^{3\sigma_d}\sum_{v=-3\sigma_d}^{3\sigma_d} c(u,v)}$$

For a pixel at coordinates (i,j).

Note that as the filter width is in and of itself independent of the total sample count, the filter is not by itself progressive and introduces bias. On the other hand, this filter is self-normalizing and may be restricted on image borders and remain energy preserving.

In addition, Xu proposes to use this algorithm to filter only on separated DD paths, much like Jensen and Christensen. Also, he recommends that range filter be filtered according to luminance in the logarithmic domain.

3.3 Photon Mapping Family

In this section, *Photon Mapping* [Jen96] shall be introduced as a precursor to *Progressive Photon Mapping* [HOJ08] and *Stochastic Progressive Photon Mapping* [HJ09] to demonstrate how these are related.

3.3.1 Photon Mapping

In 1996, Jensen presented a novel approach to solving the rendering equation [Jen96]. Rather than recursively sample the incoming radiance, he proposed to calculate it from a local flux density estimate. This estimate would the be calculated prior to primary ray shooting by emitting a number of "photons" from light sources, tracing them through the scene and storing their interactions with the scene.

Thus the reflected radiance at a point in scene would be calculated as follows:

$$L_r(\mathbf{x}, \vec{\omega}_i) = \int_\Omega f(\mathbf{x}, \vec{\omega}_o, \vec{\omega}_i) \frac{\mathrm{d}^2\Phi_i(\mathbf{x}, \vec{\omega}_i)}{\mathrm{d}A \ \mathrm{d}\vec{\omega}_i} \mathrm{d}\vec{\omega}_i,$$

where $\Phi_i(\mathbf{x}, \vec{\omega}_i)$ is the flux estimate at $\mathbf{x}$ from direction $\vec{\omega}_i$. Note the absence of the cosine term from the equation – this is unnecessary, because incoming flux from a direction is proportional to the $\cos\theta$ of that direction. A caveat is that this cosine is with respect to geometry normal rather than the shading normal, which causes artefacts where these two are noticeably different.

Using a discreet estimate we get:

$$L_r(\mathbf{x}, \vec{\omega}_i) = \sum_{p=1}^{N} f(\mathbf{x}, \vec{\omega}_o, \vec{\omega}_{i,p}) \frac{\Delta\Phi_i(\mathbf{x}, \vec{\omega}_{i,p})}{\pi r^2}$$

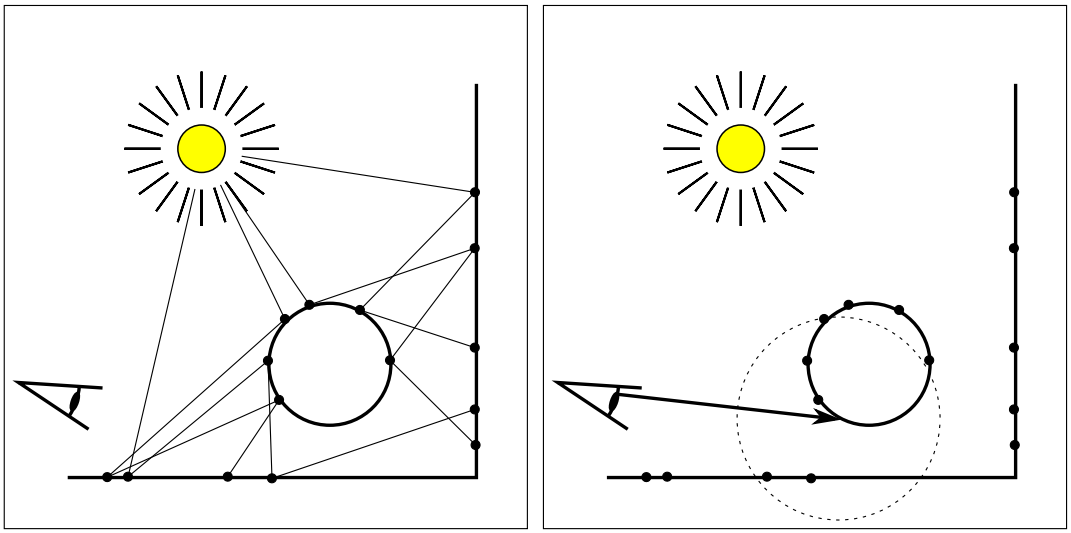
This in fact represents searching an area for photons, summing all their contributions and then dividing the result by surface area. In the original paper, the search radius was to be set to the distance of Nth nearest photon, thus using the same number of photons for the estimate at any point of scene. In addition, the photon's contributions were to be weighted by a conical filter, resulting in a slightly different estimate:

$$L_r(\mathbf{x}, \vec{\omega}_i) = \frac{\sum_{p=1}^{N} f(\mathbf{x}, \vec{\omega}_o, \vec{\omega}_{i,p}) \Delta\Phi_i(\mathbf{x}, \vec{\omega}_{i,p}) w_p}{\left(1 - \frac{2}{3k}\right) \pi r^2}$$

With w_p a function of the distance d from photon's position to $\mathbf{x}$:

$$w_p = \max\left(0, 1 - \frac{d}{kr}\right)$$

However, this also works for any 2D-normalized kernel function. An additional improvement of the algorithm, not presented in the original paper, was in the photon tracing step. We may use the same PDF



(a) Photon pass (b) Eye pass

Figure 3.3: The two phases of photon mapping. First, many *Photon passes* (3.3a) generate photon records throughout the scene, and later in an *Eye pass* (3.3b) they are used to estimate local flux density. The dashed circle in 3.3b shows the area considered in a 5-NN search, the size of which is the source of *proximity bias*.

as in *Next Event Estimation* in path tracing to select the emission point, and use a cosine-weighted PDF (that is, a cosine lobe for a diffuse light) to sample emission direction, we get a flux of

$$\Phi_p = \frac{\Phi_{TOTAL}}{N}$$

For each photon, where N is the total number of photons emitted. This is convenient, because we do not need to store the flux of photons. Furthermore, using *Russian Roulette*, we also guarantee that their flux remains the same regardless of the number of reflections they have gone through.

The resulting algorithm is a robust one, able to calculate difficult illumination features (such as caustics) quite efficiently, at the cost of introducing bias. Several different kinds of bias are identified for photon mapping, most significant being *Proximity Bias*. This is caused by the fact that area used to estimate flux is greater than a differential area, causing problems at illumination feature boundaries (shadow boundaries, for instance), and creating the characteristic "splotchy" appearance.

The bias disappears as the number of photons emitted approaches infinity, but emitting very large quantities of photons is often infeasible due to memory constraints, as all the photons have to be stored in memory. This makes the algorithm consistent in theory, but biased in practice as the lack of memory is always a limit.

To overcome bias, a technique called *Final Gathering* is often used. In this technique, the primary rays (ie. those that originate in the camera) are not stopped at first bounce, and the hemisphere around the hitpoint is sampled using traditional Monte Carlo. Only the secondary hitpoints are then solved using photon mapping. To preserve crisp caustics, two photon maps are constructed in the photon tracing

phase – a caustic map that stores photons that have been reflected specularly (once or several times) and then diffusely, and a regular photon map that stores the rest. The caustics map is then used at primary hitpoints and both are used at secondary hitpoints.

While this eliminates noticeable bias, many secondary samples (often thousands) have to be taken at each hitpoint, which in turn generates billions of photon map queries on typical modern image resolution. Even with very efficient accelerating structures (such as k-D trees [SHAP01]) for photon queries, this rapidly becomes prohibitive, due to computation time required. Photon mapping is therefore usually relegated to the role of a secondary solver for techniques such as *Irradiance Caching* [WRC88].

3.3.2 Progressive Photon Mapping

In 2008, Hachisuka, Ogaki and Jensen presented a novel technique called *Progressive Photon Mapping* [HOJ08]. In this variant, the photon tracing and gathering steps are reversed. Primary rays are cast, their hitpoints are stored and assigned *gather radii*. Then, fixed-size batches of photons are emitted from light sources and traced, and are only recorded if they land within a hitpoint's gather radius. This leads to the following density estimate:

$$d(x) = \frac{n}{\pi r^2},$$

for a fixed radius r, and n photons fallen within this radius. Tracing another batch of photons and recording their intersections with hitpoint radii would yield a different estimate. These two could be averaged, but that is not very helpful, as both of them are irredeemably biased. To achieve consistency, we have to progressively decrease the radius while increasing the photon density, until it becomes infinite in the limit. To that end, we have to reformulate the estimate. Taking $R(x)$ to be the radius at point x, $N(x)$ to be the number of previously accumulated photons and $M(x)$ to be the photons accumulated in this batch, we get:

$$\hat{d}(x) = \frac{N(x) + M(x)}{\pi R(x)^2}$$

Since we have to reduce the radius to achieve consistency, we will do so by subtracting $dR(x)$, or $\hat{R}(x) = R(x) - dR(x)$. Assuming constant local density, we can then recalculate the number of photons within that radius as:

$$\hat{N}(x) = \pi \hat{R}(x)^2 \hat{d}(x) = \pi (\hat{R}(x) = R(x) - dR(x))^2 \hat{d}(x)$$

The exact amount of photons to add is controlled by a parameter $\alpha \in [0,1]$:

$$\hat{N}(x) = N(x) + \alpha M(x) \tag{3.4}$$

The radius reduction $dR(x)$ may then be expressed as follows:

$$\pi \left(R(x) - dR(x)\right)^2 \hat{d}(x) = \hat{N}(x)$$

$$\pi \left(R(x) - dR(x)\right)^2 \frac{N(x) + M(x)}{\pi R(x)^2} = N(x) + \alpha M(x)$$

$$dR(x) = R(x) - R(x)\sqrt{\frac{N(x) + \alpha M(x)}{N(x) + M(x)}}$$

And the reduced radius $\hat{R}(x)$ as:

$$\hat{R}(x) = R(x) - dR(x) = R(x)\sqrt{\frac{N(x)+\alpha M(x)}{N(x)+M(x)}} \tag{3.5}$$

Each hitpoint then stores an unnormalized (not divided by the number of photons) flux estimate $\tau_N(x,\vec{\omega}_o)$ calculated as:

$$\tau_N(x,\vec{\omega}_o) = \sum_{p=1}^{N} f(x,\vec{\omega}_o,\vec{\omega}_{i,p})\Phi_p(x_p,\vec{\omega}_{i,p})$$

Note that this flux is pre-multiplied by the BRDF so we no longer need to take photon's incoming direction into account. Similarly, flux contribution of the last batch is calculated as:

$$\tau_M(x,\vec{\omega}_o) = \sum_{p=1}^{M} f(x,\vec{\omega}_o,\vec{\omega}_{i,p})\Phi_p(x_p,\vec{\omega}_{i,p})$$

However, as we decrease the radius, we need to reduce the flux to account for the photons that are now outside the gather radius. To avoid the need to store all the photons, we may exploit the assumption that local density is constant and express the reduction like this:

$$\tau_{\hat{N}}(x,\vec{\omega}_o) = (\tau_N(x,\vec{\omega}_o) + \tau_M(x,\vec{\omega}_o))\frac{\pi\hat{R}(x)^2}{\pi R(x)^2}$$

$$\tau_{\hat{N}}(x,\vec{\omega}_o) = \tau_{N+M}(x,\vec{\omega}_o)\frac{\pi\left(R(x)\sqrt{\frac{N(x)+\alpha M(x)}{N(x)+M(x)}}\right)^2}{\pi R(x)^2}$$

$$\tau_{\hat{N}}(x,\vec{\omega}_o) = \tau_{N+M}(x,\vec{\omega}_o)\frac{N(x)+\alpha M(x)}{N(x)+M(x)} \tag{3.6}$$

Where $\tau_{N+M}(x,\vec{\omega}_o) = \tau_N(x,\vec{\omega}_o) + \tau_M(x,\vec{\omega}_o)$ and $\tau_{\hat{N}}(x,\vec{\omega}_o)$ is the new, reduced value corresponding to $\hat{N}(x)$. Note that the above assumption may not hold initially, causing effects such as light leaking, but as the radius is reduced, it becomes very nearly true.

Finally, the photon count is updated simply as:

$$N(x+1) = N(x) + M(x) \tag{3.7}$$

Using the above, progressive photon mapping produces a progressive and consistent flux density estimate. As the flux is pre-multiplied by the BRDF, we may directly use it to estimate outgoing radiance by normalizing it with $N_{emitted}$ – the total number of emitted photons:

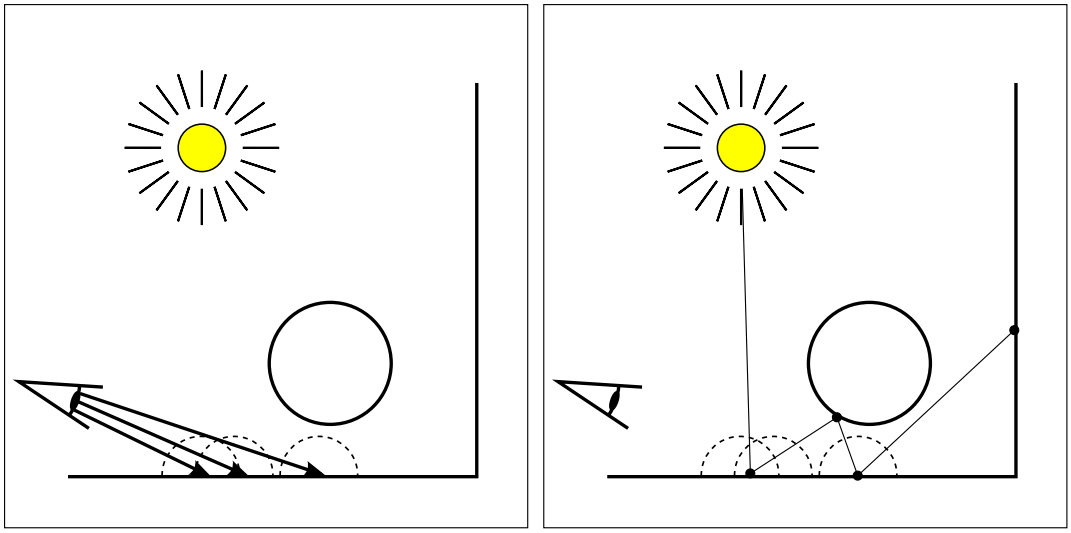
$$L_r(x,\vec{\omega}_r) \approx \frac{1}{\Delta A}\sum_{p=1}^{n} f(x,\vec{\omega}_o,\vec{\omega}_p)\Delta\Phi_p(x_p,\vec{\omega}_p) = \frac{1}{\pi R(x)^2}\frac{\tau(x,\vec{\omega}_o)}{N_{emitted}} \tag{3.8}$$

This estimator is also both progressive and consistent. The major advantage of *Progressive Photon Mapping* over regular photon mapping is that the photons do not need to be stored anywhere, so the algorithm is

not memory-bounded by total number of photons emitted. Thus, in theory, arbitrarily many photons may be emitted only at expense of rendering time, with the guarantee that bias will disappear in the limit.

The key parameter in this algorithm is α. Through controlling radius reduction, this parameter directly influences the trade-off between bias (high radius) and variance (low radius). There is not yet any formal analysis of how exactly this parameter influences convergence, but setting it appropriately is crucial.

In a more recent paper [HJJ10], Hachisuka presented an error estimation framework for progressive photon mapping and re-derived some of the equations above, relaxing the assumption that flux needs to be locally constant. The equations remain unchanged, save for one detail – it is proven that the estimate is correct even if the photons are weighted by a radial-symmetric kernel function.



(a) Eye pass (b) Photon pass

Figure 3.4: The two phases of progressive photon mapping. First, primary rays are cast in an *eye pass* (3.4a), establishing primary hitpoints within the scene. Then, photons are traced in an arbitrary number of *photon passes* (3.4b), contributing to any primary hitpoints to the *gather radius* of which they fall. As photons are contributed to hitpoints, their gather radii are reduced. In *Stochastic Progressive Photon Mapping* (*Section* 3.3.3), positions of primary hitpoints are changed every so often by recasting eye rays while retaining the hitpoint statistics.

3.3.3 Stochastic Progressive Photon Mapping

Stochastic Progressive Photon Mapping is a technique proposed by Hachisuka and Jensen [HJ09]. It builds on the original progressive photon mapping to allow simulating of effects usually simulated by distributed ray tracing, such as anti-aliasing, depth of field, motion blur, and so on.

The principle of this method is that the estimation is reformulated in terms of estimating radiance over pixel area (in potentially much more than two dimensions) by re-casting the primary rays after a certain number of photons has been emitted. However, while averaging several images rendered by progressive

photon mapping with distinct primary rays would yield a valid estimate, this estimate would be biased (as each of the inputs is biased) and would never converge to the correct solution.

Instead, a radiance estimator over area $L(S,\vec{\omega})$ is defined and formulated as:

$$L(S,\vec{\omega}) = \lim_{i\to\infty} \frac{\tau_i(S,\vec{\omega})}{N_e(i)\pi R_i(S)^2},$$

where i is the number of photons emitted, and $\tau_i(S,\vec{\omega})$, $N_e(i)$ and $R_i(S)$ are shared flux, photon count and radius respectively.

The key difference is that estimator variables are stored for the entire pixel rather than for individual primary samples, and are retained between primary ray recasts, maintaining consistency of the estimate. Radius and flux reduction are performed in the same manner as in regular photon mapping, and there is no implementation difference. Photon count, however, is updated differently. For N accumulated photons and M new photons in the last round, the updated photon count $\hat{N}$ is given as:

$$\hat{N} = N + \alpha M, \tag{3.9}$$

where α is the same parameter used in computing radius reduction in *Equation* 3.5.

The result is a consistent progressive estimator capable of simulating a variety of global illumination effects, such as anti-aliasing, motion blur, depth of field, and more.

3.4 Photon Ray Splatting

Photon Ray Splatting is a global illumination technique developed in 2007 by Herzog et al. [HHK+07]. It is similar to progressive photon mapping in that eye rays are cast first, and then energy is distributed among them from the scene lights.

The basic idea is that the flux of a photon ray is splatted to hitpoints within a conical frustum of the ray, as shown in *Figure* 3.5. The width of this frustum is a key parameter, and has to be set so that all points within the frustum are very likely visible from the point of origin, or light leaking might occur.

In effect, photons are traced normally, but a conical frustum around each of the photon rays is searched for hitpoints and part of the photon's energy is then splatted to each hitpoint, weighted by a kernel function perpendicular to ray direction:

$$E(x) = \sum_{i=1}^{N_{total}} \mathscr{K}_h(x, x_i, \vec{\omega}_i)\Delta\Phi_i(x_i,\vec{\omega}_i)\cos\theta_i,$$

where $\mathscr{K}_h(x,x_i,\vec{\omega}_i)$ is a 2D kernel function with a domain oriented perpendicular to $\vec{\omega}_i$ and centred on x_i and bandwidth h. Note that while irradiance is measured per projected differential area, if it is guaranteed that $\int_S \mathscr{K}_h(x,x_i,\vec{\omega}_i)\,\mathrm{d}x = 1$ we need not explicitly divide by footprint area.

Reformulating this, we may calculate outgoing radiance due to a photon ray:

$$L_r(x,\vec{\omega}_o) = \sum_{i=1}^{N_{total}} f(x,\vec{\omega}_o,\vec{\omega}_i)\ \mathscr{K}_h(x,x_i,\vec{\omega}_i)\ \Delta\Phi_i(x_i,\vec{\omega}_i)\ \cos\theta_i \tag{3.10}$$

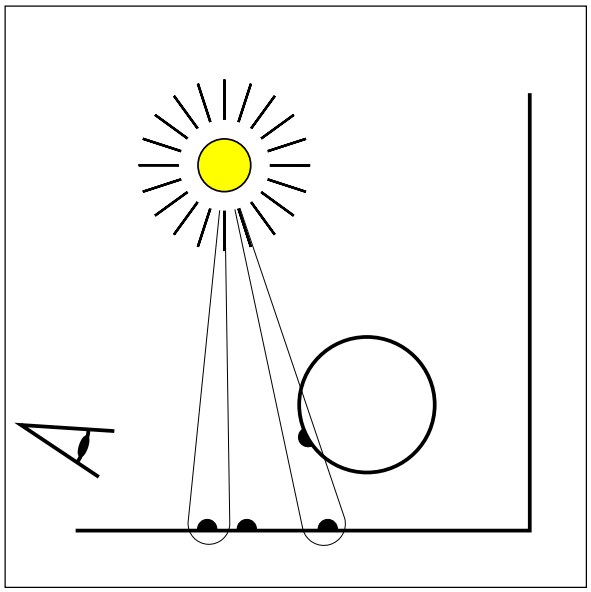


Figure 3.5: A photon pass in *Photon Ray Splatting*. Notice that while hitpoints are treated as points, a photon ray contributes to any hitpoint it passes, rather than just to those that are near path vertices (compare *Figure* 3.4b).

With the caveat that photon rays coming from a direction below the normal plane should be discarded.

The most important theoretical issue, then, is to find a way to correctly determine the splatting footprint bandwidth, or equivalently the width of the conical frustum. The original paper presents a very sophisticated heuristic based on probability density of each photon:

$$\tilde{p}(x_{i+1}|x_i) = \begin{cases} \dfrac{p_e(x_0,x_1)}{D(x_0,x_1)} & i=0 \\ \tilde{p}(x_i,x_{i-1}) \cdot \dfrac{p_s^{\perp}(x_{i-1},x_i,x_{i+1})}{D(x_i,x_{i+1})} & \forall i>0 \end{cases},$$

where p_e is the probability density of a photon being emitted in that particular direction, while $p_s^{\perp}$ is the projected probability density of a photon reflection. $D(x,y)$ is then the distance between two points. The original proposal called for these values to be clamped with scene-dependent thresholds.

Based on this probability density, a heuristic for determining bandwidth at bounce point was defined:

$$h(x_i) = \begin{cases} 0 & i=0 \\ \dfrac{C}{\sqrt[6]{M}} \dfrac{w}{\sqrt{\tilde{p}(x_i|x_{i-1})^S}} & \forall i>0 \end{cases}, \tag{3.11}$$

where M is the total number of photons emitted, S and C are user-defined parameters, and w is a constant estimated during the photon tracing phase as:

$$w = m_0 \frac{a \cdot E[D(x,y)]}{\frac{1}{E[\sqrt{\tilde{p}(x_i|x_{i-1})^S}]}},$$

where m_0 and a are more user-defined parameters. Original proposal also called for value of $h(x)$ to be clamped.

While the sixth root and dependency on mean values of parameters correspond to what is proven to make an optimum heuristic, our early efforts have found it entirely unsuited for progressive rendering. The problem is that many of the user-defined parameters have a significantly non-linear response, causing most of the computed values to be clamped anyway, as the heuristic only works reasonably for a constant photon count.

Another problem was found with the sixth root – this method introduces bias directly related to the width of the splatting footprint, most notably, discontinuity bias and light leaking beyond corners in certain cases. This bias eventually goes away as the splatting footprint decreases[4], but using a sixth root effectively gives the error function an upper bound of $O(\sqrt[6]{n})$. Until a theoretical study proposes a heuristic better suited for progressive rendering, we have decided to replace the heuristic from *Equation* 3.11 with the following formula:

$$dh(x) = \frac{C}{\sqrt[\alpha]{N}} \tag{3.12}$$

Where dh is the differential of footprint per unit of ray length, N is the number of photons emitted so far and C and α are user-defined parameters.

This new heuristic guarantees that the splatting footprint decreases with the number of photons emitted, and while most likely not optimal, ensures consistency.

Note that in a progressive rendering algorithm, photon flux is unnormalized, much like it was in progressive photon mapping, and the resulting radiance has to be divided by number of photons emitted prior to visualization. Because we do not discard prior rendering results, this in fact means that the bias introduced by the first photon rays remains in the image for good. On the other hand, in the final result, we could express a bias function $B(n)$ that computes bias for a certain number of photons traced:

$$B(n) = \sum_{i=1}^{n} \frac{1}{n} b(i)$$

As the contribution, as well as bias, of each photon ray is weighted by the inverse of the photon count. Thus, assuming a photon-wise bias function $b(i)$ that decreases with i (as we know that in the limit with a zero footprint and infinite density, we get a perfect flux density estimate, and the method is unbiased), the bias introduced by each individual photon decreases with n, ensuring consistency.

[4]This is because bias comes from width of the splatting kernel, as well as low photon ray density in the initial phases. We could simply set the footprint to zero, but then we would have a zero probability of connecting any particular photon ray to a hitpoint.

Chapter 4

Parallelization and Implementation

In this chapter, we will describe the implementation specifics of the algorithms presented in the previous chapter. At the beginning, we present the software framework we were using and the platform it is based. Then, we will describe the process of parallelizing originally serial algorithms for our specific paradigm of parallel computation. And finally, the implementation particulars of each algorithm will be pointed out.

All of the algorithms were implemented in the Uskglass renderer, based on NVIDIA CUDA [NVI07] platform. This platform provides a comprehensive API for programming graphics cards from nVidia corporation. As the programming model used by this platform is directly based on the hardware architecture, the exposed elements of parallelism are directly analogous to their hardware implementation. Programs can thus be easily tailored to the hardware even in the design phase, without having to sacrifice abstraction.

Another advantage of this particular architecture is that it provides a C/C++ language extension CUDA C and a compiler (called NVCC) that allows easy integration of CUDA code into an existing C/C++ project. This is a major advantage over other platforms, such as *OpenCL* [Khr08], because a CUDA project may use C++ classes on the GPU, or share routines between CPU and GPU.

4.1 CUDA C

In CUDA, the GPU is introduced as a co-processor for the CPU. It has its own memory space and the CPU (called *host*) can allocate memory on the GPU (called *device*), transfer data in both directions and run massively parallel computation routines called *kernels*. A kernel is executed by multiple *threads* in parallel, with these arranged in a *grid*, making this a SPMD architecture. This *grid* is in fact a two-dimensional array of *blocks*, each block itself a three-dimensional array of threads.

Threads are executed concurrently in groups called *warps* of 32 (usually, this may differ between architectures). As memory access is by far the most expensive operation, this architecture relies on rapid context switching and executing many warps in parallel to hide memory latency.

As this is a SPMD architecture, jump instructions are possible and widely used. However, as the architecture is SIMD with respect to warps, if individual threads in a warp diverge in program flow, both

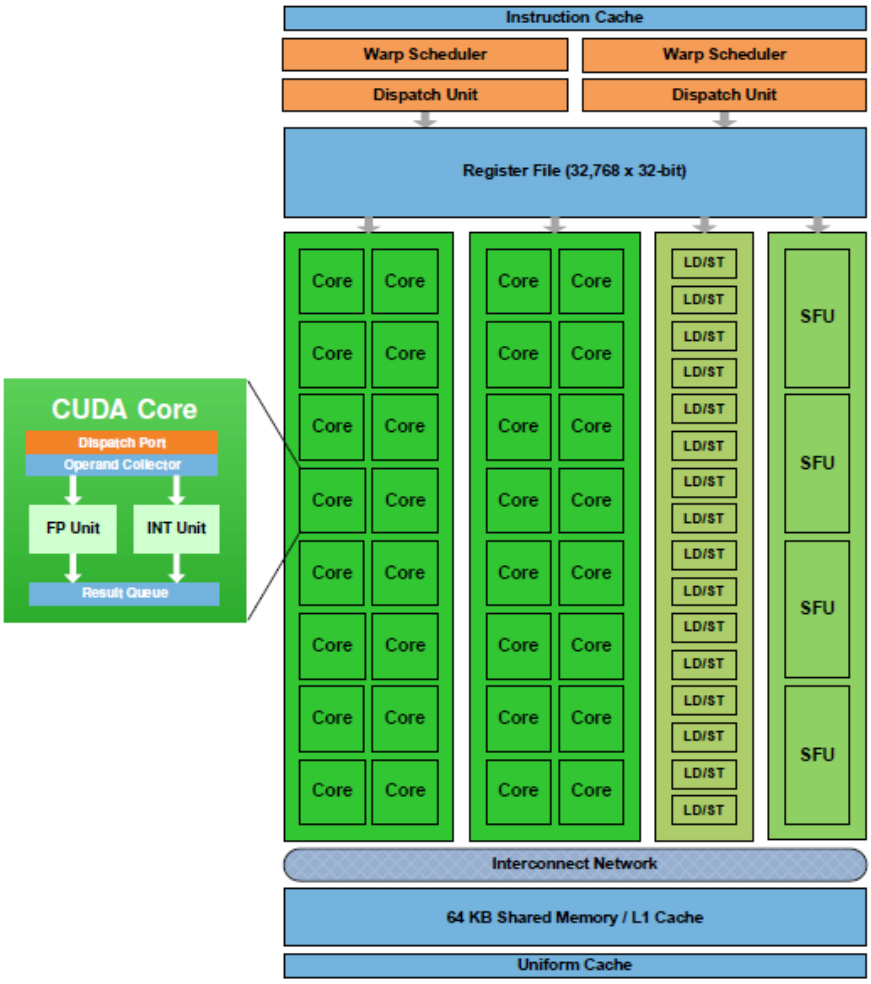


Figure 4.1: A CUDA *Streaming Multiprocessor* with its constituent parts. Each of these can execute two warps at any given point in time, with yet more performing memory operations. Taken from [NVI09].

paths need to be executed consecutively, with some of the threads deactivated in each path. This incurs performance penalties for code that has threads take wildly varying program paths, but does not incur any penalties where the entire warp jumps coherently, regardless of the program flow complexity. This has important implications that will be explored later.

Grid configuration is specified by user on each kernel launch and may be varying, though there are architectural constraints on possible dimensions of blocks and the grid, as well as constraints on the maximum number of threads/blocks in a block/grid. During execution, each thread has access to its index within a block, as well as its block's index within a grid and overall grid and block dimensions. Blocks in execution are of importance, because threads in a block may access a common on-chip memory space called *shared memory*, as well as utilize some synchronization primitives.

When a kernel is executed, grid blocks are assigned to *Streaming Multiprocessors* (see *Figure* 4.1) on the GPU. Each multiprocessor may run several blocks concurrently, but a block is never divided among multiprocessors. When blocks are assigned to a multiprocessor, its shared resources are divided among

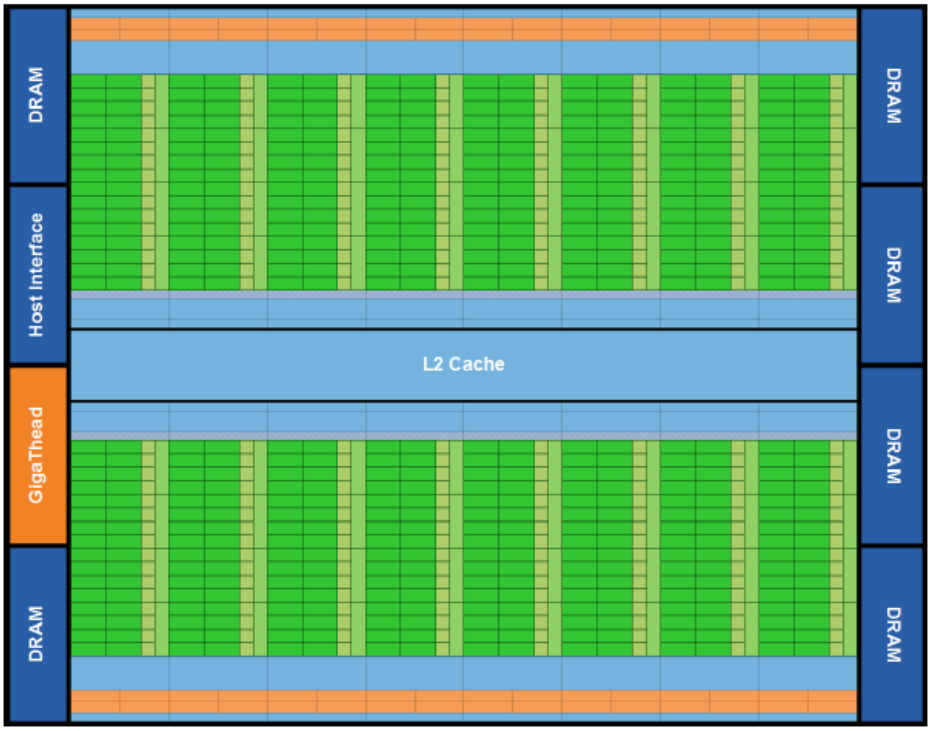


Figure 4.2: The CUDA *Fermi* architecture, pictured with 16 constituent *Streaming multiprocessors* (see *Figure* 4.1), the newly introduced L2 cache and other supplementary features. Taken from [NVI09].

them, and block size may thus impact performance. Each streaming multiprocessor contains a number of *CUDA Cores* for most common operations, as well as memory access handlers and special function units that implement some of common mathematical functions, like square root, goniometric functions, and such. Multiple warps may execute on a streaming multiprocessor in parallel (with others waiting their turn) provided they utilize different units of the streaming multiprocessor[1].

In CUDA, there are several different segments of memory. Though their address space has been unified in the Fermi architecture, there still are differences. Aside from the Shared memory already mentioned, there is the *Linear Memory* that contains both thread-local *Local Memory*[2] and application-local *Global Memory*. Furthermore, the GPU has *Texture Memory* and *Constant Memory*, both of which have their dedicated cache. In addition, Texture Memory supports hardware-accelerated interpolation and multi-dimensional indexing, albeit for only specific data formats. The downside of texture memory is that it can only be accessed via *Texture References*, which cannot be dynamically created at runtime. In addition, Fermi architecture has added a new layer of caching for Linear Memory, which especially benefits applications with many non-local memory accesses, as ours is guaranteed to be.

[1]Note that this is a very general description that does not cover many of the architecture particulars. It has been included mainly to establish a frame of reference and introduce some of the terminology that will be used later. For a more thorough description of Fermi architecture, that this description was based on and that was the target architecture for our implementation, a reader may consult the Fermi Architecture White Paper [NVI09].

[2]Local memory contains any thread local variables that are not stored in a register. In practice, the rule of thumb is that any structured data types are stored in linear memory (with slower access), while primitive variables may be stored in registers (for rapid access). There are, however, many exceptions to this rule particular to each architecture.

CUDA exposes two APIs to the programmer – a very lightweight *Runtime API*, as well as the more hands-on *Device API*. These facilitate memory operations, as well as kernel launches and general handling of device-host cooperation. Both of these may be used in conjunction with *CUDA C*, which is a C/C++ language extension accepted by the NVCC (or CUDACC) compiler. This allows kernel code to be written directly beside host code, compiled together with the application and linked with it immediately. Kernels are annotated as functions with the `__global__` qualifier. They accept arguments either as value, or as a pointer into global memory, as shown in *Listing* 4.1.

```
1  static void __global__ permutationKernel(unsigned int * indices,
       unsigned int * flags, DataType * input, DataType * output,
       unsigned int length) {
2      //get linear index of this thread
3      unsigned int threadId=blockIdx.x*blockDim.x+threadIdx.x;
4
5      //if we have spawned more threads than was needed,
           superfluous threads are terminated here
6      if (threadId>=length)
7          return;
8
9      //if flag for this object is set, perform the permutation
10     if (flags[threadId]){
11         output[indices[threadId]]=input[threadId];
12     }
13 }
```

Listing 4.1: A sample kernel that performs a permutation operation. Note that it is assumed that both grid and block configurations are one-dimensional in the X dimension.

Such a kernel may then be called from application code by a special call syntax that provides the grid configuration, shown in *Listing* 4.2.

```
1 unsigned int * indices, * flags;
2 DataType * input, *output;
3 unsigned int length;
4 //arrays are allocated and initialized using a combination of
      kernels and API calls
5 permutationKernel<<<dim3(64,1,1), //set block dimensions
6     dim3((length/64)+1,1,1)>>> //set grid dimensions. Note that a
          variable is used
7     (indices, flags, input, output, length); //provide arguments
          to the call
```

Listing 4.2: A sample kernel call. Note the grid configuration specified in angled braces, passed as a special data type.

Kernels may be executed asynchronously or synchronously. On the Fermi architecture, asynchronously

launched kernels may execute concurrently on different streaming multiprocessors, providing a speed-up if "small" kernels that are embarrassingly parallel need to be executed. This is, however, not usually the case in our context.

In addition to the `__global__` keyword, two more important qualifiers are introduced. Any function or method can be annotated by the `__device__` keyword, meaning that it should be compiled to run on the GPU. Such a function (or method) may then be called by a kernel (also called "*called from device code*"). The `__host__` keyword annotates functions and methods that should be compiled for the CPU, but is not usually used, since that is the default behaviour. However, when used together, a function or method annotated as `__host__device__` may be called both from device code and from host code. There are, however, restrictions. A `__device__` function may only in turn call other `__device__` functions (or `__host__device__` functions), and `__host__device__` functions may only call other `__host__device__` functions. In addition, a function or method that is to be run on the device may not be recursive, since the device does not implement a full-fledged call stack.

This provides a brief introduction into CUDA programming. Before we go on to describe the framework used for rendering, it should be noted that there are several libraries that perform some of the basic parallel operations (as well as freely available libraries that perform much more complicated routines, one of the strengths of CUDA). Among these routines are *reduction* (using an associative operator on all the elements of an array and reporting the result), *parallel prefix sum* or *parallel scan* (using an associative operator on all elements of an array and reporting for each element the "sum" of all preceding elements) and *parallel sort* (self-explanatory). Since an efficient implementation of these often-used operations is crucial, we have used the *CUDA Data Parallel Primitives* (CUDPP) library [HSO07] to carry them out.

4.2 Uskglass Renderer

The Uskglass renderer[3] is a CUDA-based progressive interactive renderer originally developed as a GPU-based offshot of Corona renderer to test the feasibility of *Volumetric Radiance Caching* for interactive applications (such as scene walkthroughs). Rather than being a plug-in for a modelling program, the original aim in development of this renderer was to allow the user to explore the scene, supplying him with photo-realistic imagery at an interactive rate, while at the same time having the ability to progressively improve render quality if the user stops moving through the scene, in order to allow the user to explore the more interesting parts of the scene in detail and at a higher visual quality.

To this end, the renderer provides a framework of numerous utility classes, as well as a ray tracing framework. Furthermore, it defines progressive work flow that a rendering algorithm must implement. This is embodied in the interface of `ProgressiveSolver` class, from which the actual implementations of all our algorithms are derived. The progressive work flow is described in *Algorithm* 1.

For displaying results, Uskglass supports either copying a framebuffer to the CPU and then displaying it as an SDL software buffer, or by copying the framebuffer into an OpenGL pixel unpack buffer, which is then displayed directly. The latter variant is more convenient for GPU-based renderers, as CUDA provides direct interoperability with OpenGL, where OpenGL buffers may be mapped into device memory space and then written into or read from.

[3]Named after John Uskglass in tribute to Susanna Clarke's *Jonathan Strange and Mr. Norrell*.

Algorithm 1 Progressive work flow

```
1: ProgressiveSolver solver
2: solver.initialize //Allocate and initialize internal structures
3: cameraChanged = true
4: repeat
5:    process user input and set the cameraChanged flag if appropriate
6:    if cameraChanged then
7:       //User has moved the camera
8:       solver.clear //Clear the framebuffer and other internal structures
9:    end if
10:   if solver.canProgress then
11:      //Some solvers may eventually be incapable of progressing any further, due to numerical con-
         straints, running out of data or due to self-termination
12:      solver.progress //Next step in calculation
13:   end if
14:   display result
15: until program terminated
```

The basic design paradigm in Uskglass is that as much computation as possible should be performed on the GPU. Because copying data between host and device is a comparatively expensive operation (in fact this is one of the recognized bottlenecks), vector data should only be copied between host and device in the initialization phase of the rendering. A renderer should not need to move vector data to host in order to progress in computation[4].

For this reason, all vector operations are executed on the device. Essentially, any **do in parallel** code block is implemented as a kernel launch, with each thread processing an item as specified by the annotation. For ray tracing, several accelerating structures, implemented as classes derived from the `ParallelIntersector` class, are available, built during the scene loading phase and then moved to device[5]. Primary rays are generated on the device as well, using the `ParallelCamera` class. An instance of this class is filled with data on the host, based on user input, and then moved to device, where it generates primary rays on demand (the solver only specifies how should a pixel be sampled).

Also, large quantities of pseudo-random numbers are needed for Monte Carlo algorithms. To perform well and to not cause any structured noise, these numbers should be independent both in time in a thread, as well as between threads. On the CPU, high-quality PRNGs such as *Mersenne Twister* [MN98] may be used, but these are difficult to implement on the GPU, as they tend to have a large internal state and would be impractical to retain persistently, as the thread counts as well as their configurations may vary wildly during execution.

To face this issue, Zafar et al. [ZOC10] have tested feasibility of using the *Tiny Encryption Algorithm* as a pseudo-random number generator on the GPU. They have concluded that for at least six rounds, TEA passes most of NIST randomness tests, and quality may be traded for speed with less rounds. As

[4]This terminology gets slightly confusing in computer graphics, when we consider that the "scalar" data that may be copied are in fact often vectors. In this chapter, a vector is understood to be an array of data elements not limited in size, while the vectors mentioned further will almost invariably be three-dimensional.

[5]The one used in these implementations is a SAH BVH.

the algorithm is very simple and the generator itself only has an internal state of 8 bytes, it is well-suited for being implemented on the GPU. Uskglass provides a TEA PRNG as the `TEA` class template, where the number of rounds may be set as a template argument. In practical application, six or eight rounds are used. To ensure independence between threads, 4 bytes of the seed are the linearised thread index (occasionally multiplied by a prime constant) and the other 4 bytes, uniform among all the threads, are a seed value generated host-side by a Mersenne Twister, to ensure independence between launches.

This concludes the description of important general features of Uskglass. More of them will be mentioned in individual implementations, where appropriate.

4.3 Path Tracing

The canonical implementation of path tracing is a recursive one. In practice, this would often be rewritten to use an explicit stack or otherwise optimized, but the gist of the algorithm is the same as in *Algorithm* 2.

Algorithm 2 A serial recursive path tracing implementation

```
1:  procedure recursivePathtracer ( in Ray r)
2:      Hitpoint h = traceRay(r)
3:      if h.invalid then
4:          //Ray went out of scene
5:          return Color.BLACK
6:      end if
7:      //Performs the next event estimation as described in Chapter 3.1
8:      Color emitted = nextEventEstimate(h)
9:      Ray next = h.BRDF.sample //Samples the BRDF as described in the same section
10:     if h.BRDF.shouldTerminate then
11:         //plays the Russian Roulete
12:         return emitted
13:     else
14:         return emitted + h.BRDF.valueNormalized(r,next) * recursivePathtracer(next)
15:         //The value of the BRDF is normalized (with an intensity of 1) and only modifies colour
16:     end if
17: end procedure
```

Even though individual paths are embarrassingly parallel, this implementation is highly inconvenient for GPU use. There are two main reasons for this. First, most GPU programming platforms (and CUDA among them) do not support recursion. Second, if we were to implement this procedure with an explicit stack, the entire kernel execution would have to wait for the longest path to finish. To counteract this, Coulthurst et al. [CDD+10] have proposed an implementation well-suited to SIMD coprocessors. In their method, called *Incoherent Path-Atom Binning*, path tracing is reformulated as an iterative process. It is broken down to *path tracing primitives*, described in *Algorithm* 3, and these are than assigned to individual processors in order to keep them all occupied. The ability to de-couple paths from processors mid-path allows us to assign new primitives to processors that have become idle, bringing an originally work-optimal algorithm much closer to cost-optimality.

For an array of a mix of active and inactive paths, we may use the parallel scan operation to compact the active paths to the start of the array and assign new paths contiguously into unused positions at the end. Additionally, unlike in the original paper, in CUDA, we may want to assign multiple paths to each processor to improve *occupancy*. This results in *Algorithm* 4.

Algorithm 3 The path tracing primitive and how it can implement serial path tracing.

```
1:  procedure pathtracingPrimitive (in out Path p)
2:      Hitpoint h = traceRay(path.ray)
3:      if h.invalid then
4:          p.setInactive
5:          return
6:      end if
7:      //Perform the next event estimation and add its result multiplied by path's accumulated BRDF to
        the pixel
8:      p.addContribution(p.accumulatedBRDF ∗ nextEventEstimate(h))
9:      Ray next = h.BRDF.sample
10:     if h.BRDF.shouldTerminate then
11:         p.setInactive
12:         return
13:     end if
14:     //BRDF value is accumulated over path lifetime by multiplication
15:     p.accumulatedBRDF ∗ = h.BRDF.valueNormalized(r,next)
16:     return
17: end procedure
18: procedure iterativePathtracer ( in Sample screenSample, in Camera c)
19:     Path p = c.makePath(screenSample)
20:     repeat
21:         pathtracingPrimitive(p)
22:     until p.isInactive
23: end procedure
```

Also, unlike the recursive implementation, we have to explicitly keep track of accumulated BRDF of the entire path. As this also presupposes importance sampling by the $\frac{f(\vec{\omega}_i, \vec{\omega}_o, \mathbf{x}) \cos\theta_i}{\rho}$ function, we could disregard BRDF completely if we were only interested in radiance. In practice, however, radiance is defined in terms of RGB components and we use per-component multiplication by a normalized BRDF value to keep track of colour.

The key point here is exactly how the method *Path.addContribution* is implemented. In our implementation, each path is explicitly associated with a pixel. As the value of a pixel may be expressed as an arithmetic mean of the contributions of all the paths that pass through it, we may simply add whatever contribution the path generates, provided we keep track of the number of paths per pixel. As this number is actually uniform among pixels, there is no need to store it explicitly. We only have to divide all the pixel values by the number of paths per pixel during visualisation as we copy the buffer contents into the framebuffer.

However, as more paths may compete contribute to a single pixel in parallel, errors known as *race condi-*

Algorithm 4 The SIMD parallel path tracing algorithm

```
1: procedure parallelPathtracer (in Sample[] screenSamples, in Camera c, in uint sampleCount)
2:   uint parallelPaths = PROCESSOR_COUNT * OCCUPANCY_CONSTANT
3:   Path paths[parallelPaths]
4:   uint activePaths = 0
5:   uint processedPaths = 0
6:   repeat
7:     //Fills the path array with freshly generated paths from the sample array.
8:     c.fillPaths(paths, activePaths, parallelPaths, processedPaths)
9:     do in parallel per path
10:      pathtracingPrimitive(paths[index])
11:    end parallel
12:    compact(paths, activePaths) //Compacts active paths to the start of the array.
13:  until processedPaths = sampleCount && activePaths = 0
14: end procedure
```

tions may arise. To prevent these, we make use of CUDA's *atomic functions*, namely floating point atomic addition. This ensures that multiple threads writing to a single float each add their value properly. As a downside, however, this restricts our application to GPUs with a Compute Model version 2.0 or higher.

While this all is sufficient for a pure path tracer, we have implemented additional features. In order to support path separation necessary for component separated filtering, we have added a variable that tracks the first two path interactions. On demand, DD paths may be separated and stored in a separate buffer (with a separate counting buffer keeping track of how many paths in a pixel were separated).

In addition, our path tracing implementation is a *distributed path tracer*, that is, paths start at the primary hitpoint and direct illumination is sampled separately (and stored in a separate buffer). On the other hand, the primary rays are re-cast each iteration from a random point in pixel, so progressive anti-aliasing is kept. The main advantage of this approach is that the user may specify how many direct illumination samples should be taken, as well as how many paths should be traced. This may be advantageous in scenes where direct illumination contributes to variance significantly more, or vice versa[6].

4.3.1 Component Separating Kernel Filter

Having implemented parallel path tracing with path separation, the component separating kernel filter is relatively straightforward to implement. In the visualization phase, we separately calculate estimates by dividing the pixel values by the number of paths that contributed to them. In the second buffer for DD paths, we then apply kernel filtering over a square grid area, using a radially symmetric epanechnikov

[6]In addition, we use a compaction scheme on our primary hitpoints to gather the valid ones (those that did have a valid intersection with the scene) into a contiguous array and avoid processing the invalid ones. That is the only implementation difference, as once the gathered radiance has been stored in the frame buffer, all hitpoint information may be discarded and new ones may be created. This holds even for path separation, as we have a global paths per pixel counter and invalid hitpoints are considered to contribute black.

kernel. Using $\mathfrak{K}(x,h)$ for the epanechnikov kernel, the formula used in implementation is as follows:

$$\hat{f}(u,v) = \frac{\sum_{i=-h}^{h}\sum_{j=-h}^{h} f(u+i,v+j)\mathfrak{K}(\sqrt{i^2+j^2},h)}{\sum_{i=-h}^{h}\sum_{j=-h}^{h}\mathfrak{K}(\sqrt{i^2+j^2},h)}$$

The iterator values of i and j are clamped so as not to reference invalid indices. Note that the result is normalized by the sum of weights, so that this filter remains valid and energy-conserving even along image boundaries.

Having filtered the DD component, the result for visualization is obtained by simply adding the direct illumination component, DD component and the residual component. These operations are embarrassingly parallel per result pixel, so the filter as well as addition are easily parallelized and all of these operations are performed on the GPU in the `ParallelProgressiveKernelFilteredPT` class, defined in file `ParallelProgressiveKernelFilteredPathtracer.h`.

4.3.2 Adaptive Kernel Filter

Our adaptive kernel filtered pathtracing implementation is built on top of the regular pathtracer (in the `ParallelProgressiveAdaptiveFilteredPT` class derived from the original `ParallelPRogressivePathtracer`). However, only one path per pixel is traced before they are splatted as per *Section* 3.2.2. The splatting operation is parallelized per path (that is, per source pixel), so that each thread only iterates through the neighbourhood determined by kernel width h. If this were parallelized per result pixel, we would have to first determine maximum splatting radius and have all threads search this radius, significantly increasing runtime.

Note that this filter is not applied as a post-processing operation, but the newly traced paths are splatted directly into the internal persistent radiance buffer.

4.3.3 Bilateral Kernel Filter

Bilateral kernel filter is applied as a post-processing filter after path tracing in the `ParallelProgressiveBilateralFilteredPT` class. It is built on top of regular path tracer and does not modify the internal radiance buffer. Bilateral kernel filter is parallelized per pixel (both source and output, as these are identical in this case). As per *Section* 3.2.3, a gaussian-filtered estimate is first calculated and then used as a basis for the bilateral filter. Kernel filtering is done the same way as in *Component Separating Kernel Filter*, with range filter used over total luminance, expressed with a luminance function similar to one mentioned by Xu and Pattanaik [XP05].

4.4 Progressive Photon Mapping

The implementation of progressive photon mapping has two principal phases – the eye ray shooting phase, and the photon tracing phase. The eye ray shooting phase is quite straightforward, and is implemented similarly as a path tracer would be. On intersection, a `PPMHitpoint` is created, that stores

additional data required for photon mapping, as per the original paper [HOJ08]. The variables stored are enumerated in *Table* 4.1.

Member name	Meaning
hit point	Index into the primary hitpoint array. Referenced entry contains the position, normal, material etc. of this hitpoint.
$R(x)$	The gather radius of this hitpoint
$N(x)$	The total photon count
$M(x)$	Photons gathered this round
$\tau(x)$	The pre-multiplied unnormalized flux at this hitpoint

Table 4.1: The `PPMHitpoint` structure and its members.

However, unlike the original proposal, we never let the primary ray bounce on a glossy surface. This is due to difficulty associated with dynamic memory allocation in CUDA, where having to save a potentially unlimited number of hitpoints per ray would require laborious workarounds that could potentially break the algorithm. This change should only manifest itself on specular surfaces, where specular reflections will take longer to converge.

The photon tracing step is much like path tracing described above, with the difference being that photons have to be gathered by hitpoints after each bounce. For gathering, we use an algorithm described by Hachisuka and Jensen [HJ10], where a hashed grid is built over photon hitpoints and these are then stochastically assigned to grid cells. This is done with an understanding that photon assignment is parallelized per photon, and each grid cell only contains a photon counter (that is incremented atomically) and a photon index (that is written to competitively). Assuming that there is an even chance for any photon to be the last, using just the last photon with a flux multiplied by N, the number of photons that fell to the same grid cell, is an unbiased choice. This is in fact a variation of the *Russian Roulette*, where we assume that the probability for each of the photons to be chosen is equal to $\frac{1}{N}$. Dividing by the probability then translates to multiplying by N as specified.

This approach, embodied by the `ParallelPointGrid` class, implemented in a file of the same name, saves many the problems associated with organizing photons into a search structure, at the cost of increasing variance. This, however, is potentially offset by an improvement in computation speed.

As in path tracing, photons that leave the scene or are absorbed are declared inactive. After each iteration, those that are left active are compacted to the start of the array, and new ones are generated. Note, however, that absorbed photons still contribute to hitpoints, and thus should be written into the grid as well.

The overall Parallel Progressive Photon Mapping algorithm is described in *Algorithm* 6 and implemented in the `ParallelProgressivePhotonMapper` class. Note that by gathering photons (parallelized per hitpoint) after each bounce, we get improved convergence as a more diverse selection of photons is retained, and as an additional advantage, we need not allocate persistent storage for our photons, because we gather them directly from the storage used for tracing. The gathering operation can be called reasonably fast as we take time to set grid cell dimensions to be equal to maximum gather radius. This guarantees that any hitpoint needs to search at most 8 grid cells.

Algorithm 5 The Parallel Photon Tracing Routine

```
1: procedure tracePhotons(in uint parallelPhotons, in uint roundPhotons, ref HashGrid grid, ref
   PPMHitpoint[] hitpoints)
2:     uint activePhotons = 0
3:     uint generatedPhotons = 0
4:     repeat
5:         generatePhotons(photons, activePhotons, generatedPhotons, roundPhotons)
6:         grid.clear()
7:         do in parallel per photon
8:             photons[index].trace
9:             if photon.isActive then
10:                //Only for photons that did not leave the scene
11:                //Competitively writes photon index into the appropriate grid cell and atomically incre-
                   ments photon count in the same cell
12:                grid.writePhoton(photons[index])
13:                photons[index].nextBounce //Chooses the next interaction
14:                //The Russian roulette based on selected interaction and either absorbs the photon, or
                   modifies its flux (and colour) appropriately
15:                photons[index].russianRoulette
16:            end if
17:        end parallel
18:        do in parallel per hitpoint
19:            grid.search(hitpoints[index])
20:        end parallel
21:        //Compact active photons to the start of the array and update the active photon count
22:        compactInactive(photons, activeCount)
23:    until activePhotons = 0 && generatedPhotons = roundPhotons
24: end procedure
```

Algorithm 6 The Parallel Progressive Photon Mapping algorithm

```
1: PPMHitpoint[] hitpoints //The primary hitpoints
2: Photon[] photons //The array for currently traced photons
3: HashGrid grid //A search grid for photon splatting
4: procedure parallelProgressivePhotonMapping ( in ScreenSample[] samples, in uint sampleCount,
   in float initialRadius, in float alpha in uint parallelPhotons, in uint roundPhotons)
5:    hitpoints = tracePrimary(samples, sampleCount, initialRadius) //Traces primary rays and ini-
      tializes their hitpoints
6:    BoundingBox hitpointBox = makeBox(hitpoints)
7:    grid.makeGrid(hitpointBox, initialRadius)
8:    loop
9:       //Using procedure from Algorithm 5
10:      tracePhotons(parallelPhotons, roundPhotons, grid, hitpoints)
11:      do in parallel per hitpoint
12:         reduceHitpoint(hitpoints[index]) //Performs radius and flux reduction, updates photon
            count
13:      end parallel
14:      grid.reGrid(parMAX(hitpoints[].radius))
15:      displayRadianceEstimate(hitpoints)
16:   end loop
17: end procedure
```

The *reGrid* operation recalculates the grid cell dimensions, to keep them equal to largest used search radius, as suggested by [HJ10]. As the grid is empty at this point, this is an $O(1)$ operation. Our implementation uses a hash grid suggested by Hachisuka, meaning that we may change cell dimensions (and cell counts) without requiring a reallocation of underlying storage. In addition, the *reGrid* operation does not need to be performed each round. In fact, as the radius reduction is done at a rate similar to exponential, after the initial few iterations, it may be omitted altogether.

On initialization, a bounding box of primary hitpoints is constructed with a series of parallel MAX reductions, and is immediately expanded by initial search radius. This allows us to eliminate from the gather step any photons that fall outside this bounding box. The grid cell dimensions are set to the initial radius, identical for all hitpoints.

A radiance estimate is calculated from unnormalized gather flux by dividing it with the total number of photons emitted.

4.5 Stochastic Progressive Photon Mapping

As has been mentioned, Progressive Photon Mapping does not allow for distributed ray tracing effects, most important for our purposes being anti-aliasing. Stochastic Progressive Photon Mapping allows us to remedy this, as with the modified update routine we are able to recast primary rays while maintaining consistency. To do this, we introduce a method that recasts the primary hitpoints, while leaving their

respective PPMHitpoints untouched. Having defined the `PPMHitpoint` as we did above, this is trivial to do[7].

Recasting the primary rays necessitates reconstruction of the photon grid. In addition to resizing grid cells, we need to again build a bounding box over primary hitpoints, as their positions might be substantially different. However, as we are using a hash grid, this operation is only as expensive as the parallel reductions over the hitpoints needed to build the box. The new box is then expanded by current, rather than initial, maximal search radius.

The other key difference is in photon count update – rather than using formula from *Equation* 3.7, we use the updated one from *Equation* 3.9.

The overall *Parallel Stochastic Progressive Photon Mapping* algorithm is described in *Algorithm* 7 and implemented in the `ParallelStochasticPPM` class, as derived from the original `ParallelProgressivePhotonMapper`.

Algorithm 7 The Parallel Stochastic Progressive Photon Mapping algorithm

```
1:  PPMHitpoint[] hitpoints //The primary hitpoints
2:  Photon[] photons //The array for currently traced photons
3:  HashGrid grid //A search grid for photon splatting
4:  procedure parallelProgressivePhotonMapping ( in ScreenSample[] samples, in uint sampleCount,
    in float initialRadius, in float alpha in uint parallelPhotons, in uint roundPhotons)
5:    hitpoints = tracePrimary(samples, sampleCount, initialRadius)
6:    BoundingBox hitpointBox = makeBox(hitpoints)
7:    grid.makeGrid(hitpointBox, initialRadius)
8:    loop
9:      //Use the routine from Algorithm 5.
10:     tracePhotons(parallelPhotons, roundPhotons, grid, hitpoints)
11:     do in parallel per hitpoint
12:       //Perform radius and flux reduction and updates photon count as per Equations 3.5,3.6, and
          3.9
13:       reduceHitpoint(hitpoints[index])
14:     end parallel
15:     recastPrimary(hitpoints) //Recasts the primary rays
16:     hitpointBox = makeBox(hitpoints)
17:     grid.reBox(hitpointBox)
18:     grid.reGrid(parMAX(hitpoints[].radius))
19:     displayRadianceEstimate(hitpoints)
20:   end loop
21: end procedure
```

[7]Actually, our implementation does retain a `PPMHitpoint` for each pixel in a persistent buffer. This is done because we use a compaction scheme on primary hitpoints and their corresponding PPMHitpoints to avoid processing arrays of mixed valid and invalid (missed the scene) hitpoints.

4.6 Progressive Photon Ray Splatting

Progressive Photon Ray Splatting is quite distinct from the previous algorithms. While photon tracing is much the same, the ray splatting is a different challenge altogether. In the original proposal, it is specified that a kD-tree is to be used to splat photon rays. For each ray, we search a conical frustum capped with a hemisphere on the wider end. In order to be able to efficiently search a point kD-tree for a conical area, we first need to define efficient intersection tests.

For the initial intersection test of a bounding box against a conical frustum, Herzog et al. [HHK+07] proposed to expand the bounding box by terminal ray radius and then perform a box-segment intersection test. This is a straightforward enough test, and a conservative one. However, crucial for a kD-tree traversal is a test against an axis-aligned plane. This can in turn be done by a simple interval test. If this were just a segment, we could construct the interval for the x dimension (and analogically for any other dimension) simply as:

$$\langle min(x_{start}, x_{end}); max(x_{start}, x_{end})\rangle$$

However, as this is a conical frustum, we need to project the cone as a whole onto the axis of interest. Using r_{start} for the initial radius and r_{end} for the terminal radius, we may express the interval as:

$$\langle min(x_{start} - r_{start}, x_{end} - r_{end}); max(x_{start} + r_{start}, x_{end} + r_{end})\rangle$$

And we may directly test the splitting plane coordinate against this. Note that as at the start, the frustum is capped by a plane rather than by a hemisphere, we could have calculated the term for the starting radius exactly as $r_{start} \cdot \sin\alpha$, where $alpha$ is the angle between the frustum directional vector and the axis, but this would likely not be worth the additional calculation and logic. Thus we satisfy ourselves with this conservative test.

For this search, we use a kD-tree build on host (the `ParallelPointKdTree` class and associated classes and kernels), utilizing the *Sliding Midpoint* method and *Box over Bounds* optimization presented by Sample et al. [SHAP01]. The tree is built on host because this is only done once in algorithm initialization, and we have not discovered any published research on efficient point kD-tree construction with this method on the GPU.

Overall, as photon rays are bound to photons, the photon tracing, as well as generating, method is exactly the same as in progressive photon mapping. Utilizing a photon tracing algorithm described as *Algorithm* 8, the entire algorithm is described in *Algorithm* 9.

The key differences are that even the photon rays that leave the scene have to have their frustums searched, as these may still contain some hitpoints. Any hitpoints that are found than have their radiance contributed to as per *Equation* 3.10 (provided they are not back-faced). The differential frustum width per unit length is derived from the heuristic described by *Equation* 3.12, with base frustum width either retained from last bounce, or 0 on ray start – this information then fully defines a photon ray frustum. Photon rays that leave the scene are capped at their intersection with the bounding box of primary hitpoints. Radiance estimate is calculated exactly the same way as in progressive photon mapping.

Note that unlike the previous method, progressive photon ray splatting does not require a special hitpoint structure. A regular hitpoint together with a radiance buffer will suffice.

Progressive photon ray splatting is implemented by the `ParallelProgressivePhotonRaySplatter` class.

Algorithm 8 Photon tracing in Progressive Photon Ray Splatting

```
1: procedure tracePhotons(in uint photonCount, in uint parallelPhotons, ref PointKdTree tree)
2:     uint activePhotons = 0
3:     uint generatedPhotons = 0
4:     photonRay[parallelPhotons]photons
5:     repeat
6:         generatePhotons(photons, activePhotons, parallelPhotons, generatedPhotons)
7:         do in parallel per photon ray
8:             photons[index].trace
9:             photons[index].setCone()
10:            tree.contribute(photons[index])
11:            if photons[index].isActive then
12:                //We can only bounce photons that are still in scene
13:                photons[index].bounce()
14:                photons[index].russianRoulette()
15:            end if
16:        end parallel
17:        compactActive(photons, activePHotons)
18:    until activePhotons = 0 && generatedPhotons = photonCount
19: end procedure
```

Algorithm 9 Parallel Progressive Photon Ray Splatting

```
1: procedure parallelProgressivePhotonRaySplatting(in ScreenSample[] samples, in
   uint sampleCount, in uint photonsPerRound, in uint parallelPhotons)
2:     Hitpoint[sampleCount] hitpoints
3:     hitpoints = tracePrimary(samples, sampleCount) //Trace the primary rays
4:     PointKDTreetree
5:     tree.build(hitpoints) //Constructs the kD-tree host-side
6:     loop
7:         tracePhotons(photonsPerRound, parallelPhotons, tree) //Perform tracing and contribution

8:         displayResults(hitpoints)
9:     end loop
10: end procedure
```

Chapter 5

Testing and Discussion of Results

In this chapter, we will present the methodology used for our testing, as well as the individual test cases. Next, the results of this test suite shall be presented. A more detailed analysis of each algorithm's performance will follow, with the aim to determine to what degree may our results be skewed by particulars of our implementation, as well as to discover where there is room for improvement in each of the algorithms.

5.1 Testing Methodology

In this thesis, the principal area of interest is the use of presented algorithms as radiance estimators, rather than any aesthetic criteria. For this reason, we will be performing measurements directly on the radiance estimate, represented as a 96-bit floating point RGB triplet, rather than on any tone mapped values.

For the purposes of testing, we have selected five scenes, varying in complexity and illumination features. In each of these scenes, we picked a view and let each of the algorithms render it for four hours. During this time, no other computationally intensive processes were run and in algorithm-dependent intervals, the renderer would dump the complete state of its radiance buffer into a file on the hard drive, with the application measuring intervals between these events.

After we were done rendering, we selected a reference image for the given scene (typically the path tracing output) and compared each frame to this reference. To aggregate estimation error over the image, we used the *Root Mean Square Error*:

$$RMSE(\vec{x},\vec{r}) = \sqrt{\frac{\sum_{i=1}^{n}(\vec{x_i}-\vec{r_i})^2}{n}},$$

where images are taken to be vectors of pixels. Each colour channel is considered separately, without any perceptive weighting, as we are interested strictly in estimator precision, rather than the visual result.

As we can associate each of the RMSE values with its rendering time, we may plot these values on a time line and thus compare the algorithms side-to-side for a single scene. Whenever an algorithms requires any configuration parameters, we will provide the values we used for that scene.

Furthermore, to provide a better idea of how the error manifest itself in each algorithms, we will provide some rendering snapshot and difference images in *Appendix* C. For each snapshot, the difference image will be calculated as the vector of absolute values of channel- and pixel-wise differences. In addition, a *mapped difference image* shall be provided for each difference image.

In these, *Relative Luminance Error*:

$$RLE(\vec{x}_i, \vec{r}_i) = \frac{L(|\vec{x}_i - \vec{r}_i|)}{L(\vec{r}_i)}$$

With $L(\vec{x}_i)$ being the *relative luminance* of a given RGB triplet, calculated as $L = 0.2126R + 0.7152G + 0.0722B$, shall be displayed in false colour mapping for visual reference.

For performance analysis, one of the scenes will be chosen and rendered with each of the algorithms run in the *Compute Visual Profiler* [NVI07]. The output of the profiler shall be presented along with a discussion and analysis of the results.

5.2 Test Cases

The following scenes were used for testing:

Cornell Box

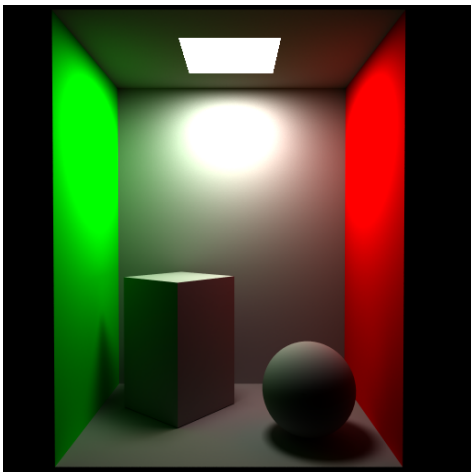


Figure 5.1: The Cornell Box

The Cornell Box is one of the classic test scenes in realistic image synthesis. In our version, it contains a single box and a sphere. Geometrically, it is a simple enough scene that should not be very hard on ray tracing, and contains diffuse materials exclusively. We should see here how our algorithms handle diffuse inter-reflections, and how well they can cope with paths or photons leaving the scene unexpectedly.

Glossy Interior

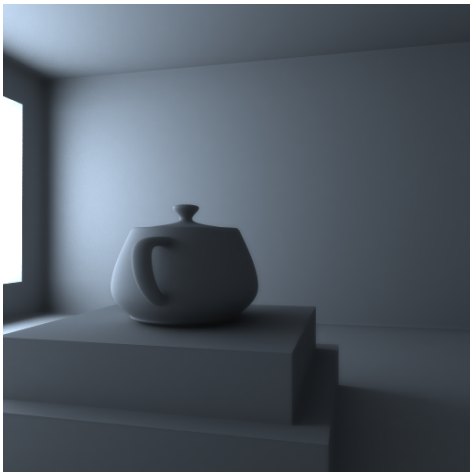


Figure 5.2: The Glossy interior scene

This scene, courtesy of its author Rawalanche, is a slightly more complex interior scene with a moderately glossy material (that is, specular, but with a very low shininess), falling into a class of scenes generally considered difficult. Geometry wise, it is a plain interior with a Utah teapot in the middle. We should be able to see many specular inter-reflections in this scene.

Ring

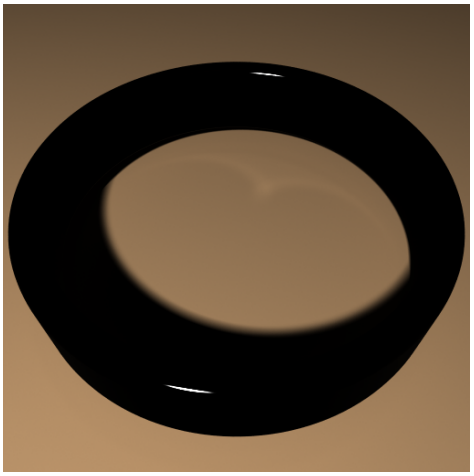


Figure 5.3: The Ring scene

This is a simple scene specifically engineered to present several difficulties. It consists of a large diffuse plane with a mirror-like purely specular ring in the middle, illuminated by an area light placed some

distance above. Since the ring is by far the most complex object in the scene, this is a sort of ”teapot in a stadium” scenario that is known to present difficulties to ray tracers. Furthermore, this is a very open scene, so there will be many photons falling outside of area of interest, or out of the scene altogether. We have chosen a view that should show off a mild cardioid caustic on the surface below the ring.

Conference

Figure 5.4: The Conference scene

Another one of the classic test scenes, Conference presents a mixture of diffuse and specular materials as well as an assortment of objects of varying geometric complexity. Of the scenes we have, this one can be said to be the closest to a real-world production scene. The mixture of materials should provide for a variety of path combinations, as well as areas dominated by indirect illumination.

Sponza

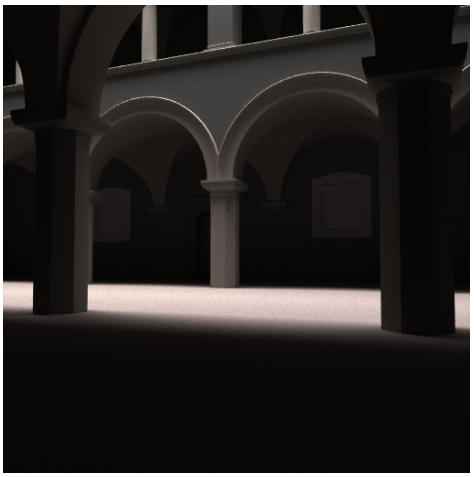


Figure 5.5: The Sponza scene

A fine model of the courtyard of the Sponza palace in Dubrovnik, this scene is a partially open one, composed exclusively of diffuse materials. The arcades of the courtyard are illuminated exclusively by indirect diffuse paths, making this a production-scale diffuse scene. This should test how our algorithms handle diffuse inter-reflections in a scene closely resembling production.

5.3 Testing Results

Having run the tests and aggregated the output data, we are able to present the resulting graphs. Before we do that, however, we will first explain some of the particulars of the testing. The testing configuration used is presented in *Table* 5.1.

CPU	Intel Pentium Dual CPU E2180 2,00GHz 2,00GHz
RAM	4,00 GB DDR3
GPU	NVIDIA GeForce GTX460 2GB GDDR
OS	Windows 7 Professional 64

Table 5.1: The testing configuration

As has been mentioned, no other computationally intensive processes were run on the testing machine during the time of the testing. To accurately measure time spent on GPU computations, we used CUDA's built-in time measurement function, which measure with a precision of a hundredth of a millisecond. What we factually measured was the time to take a rendering step, while time spent on transferring data for visualization and saving the frame buffer on the hard drive was not counted towards rendering time.

We used a base resolution of 512×512 for each of the algorithms. Those that did not support progressive antialiasing were using antialiasing by $4\times$ super-sampling with regularly spaced samples.

Furthermore, all of the algorithms require specific configuration variables to be provided. In each test case, the values used for rendering were determined in preliminary testing and will be presented in full in *Appendix* B. The values will be presented in a table form, and their specific meanings are explained in *Table* 5.2[1].

The performance graphs we will be providing will only span the first hour of rendering. This is because the results themselves show that any algorithm plateaus during the first hour and only converges very slowly after that. Since the convergence doesn't change dramatically after that point, we may easily extrapolate it from the data provided and by omitting it, gain more space to provide a detailed analysis of the initial convergence.

5.3.1 Cornell Box

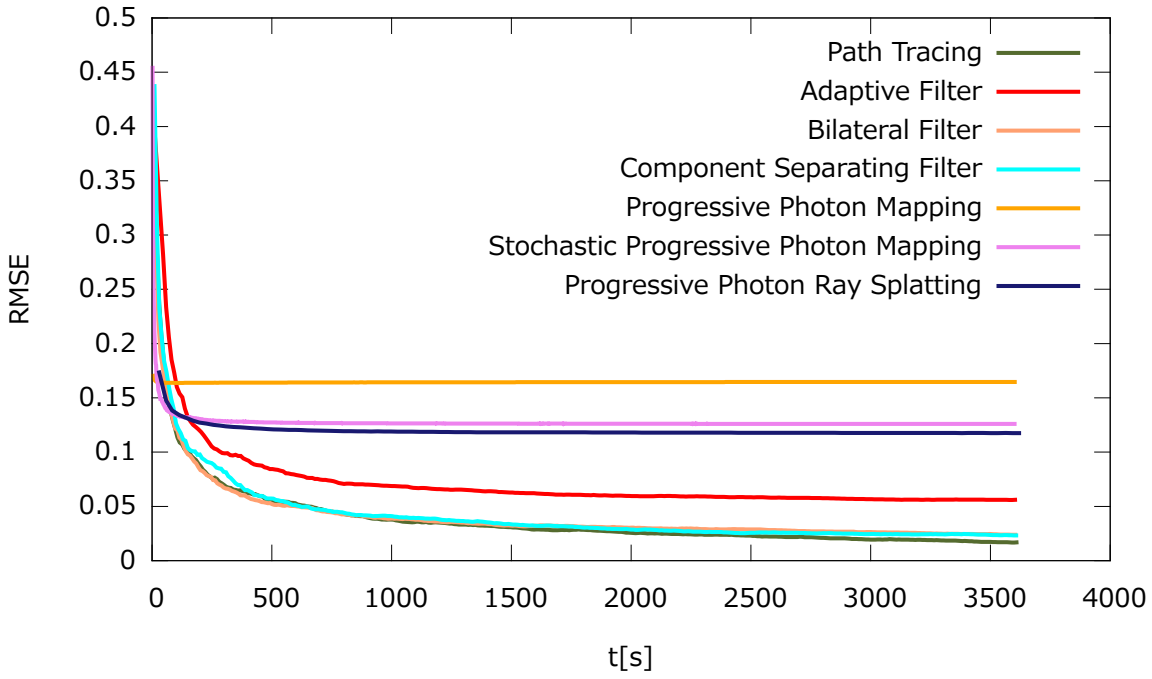


Figure 5.6: The time-error plot for the Cornell Box scene.

As Cornell Box is a very simple scene, it is hardly surprising that all of the implemented algorithms successfully converge to an acceptable error rather quickly. An interesting observation is that while the photon-based methods converge rather quickly at first, they are the first to plateau, being all overtaken in the course of approximately three minutes. As for the path tracing methods, it is clearly visible that the adaptive filter is consistently worse with regard to RMSE than any other path tracing method, while the other filters exhibit little discernible difference.

[1]Note that the filtering techniques based on path tracing necessarily inherit configuration variables from path tracing. These are not explicitly mentioned and were always set the same for both filtered and unfiltered path tracing.

Algorithm	Variable	Meaning
Path Tracing	parallelPaths	How many paths should be traced in parallel, as per *Algorithm* 4
	pathsPerRound	How many paths should be traced from a single primary hitpoint, before the results are visualized and the primary ray is recast
	directSamples	How many Next Event Estimation samples should be taken from the primary hitpoint to determine direct illumination
Component Separation Filter	kernelWidth	The filtering kernel width, in pixels
Adaptive Filter	α	The sample exponent, as per *Equation* 3.1
	C	The reference kernel width in pixels, as per *Equation* 3.1
Bilateral Filter	σ_d	The domain filter width from *Equation* 3.2
	σ_r	The range filter width from *Equation* 3.3
Progressive Photon Mapper	parallelPhotons	How many photons should be traced in parallel, as per *Algorithm* 6
	photonsPerRound	How many photons should be traced before each reduction step
	initialRadius	Initial gather radius for new hitpoints
	α	The reduction coefficient from *Equation* 3.4
Stochastic Progressive Photon Mapper	parallelPhotons	same as PPM
	photonsPerRound	same as PPM
	initialRadius	same as PPM
	α	same as PPM
Progressive Photon Ray Splatting	parallelPhotons	How many photon rays should be traced in parallel, as per *Algorithm* 9
	photonsPerRound	How many photon rays should be traced before visualization
	C	The reference differential width as per *Equation* 3.12
	α	The exponent from the same

Table 5.2: The names and meaning of configuration variables for our algorithms

As this is a scene dominated by diffuse inter-reflections, there were expectations that a path traced solution would be noisy at first, but would clear up quickly. This expectation was largely met, with the comparatively poor performance of the adaptive filter being somewhat surprising.

5.3.2 Glossy Interior

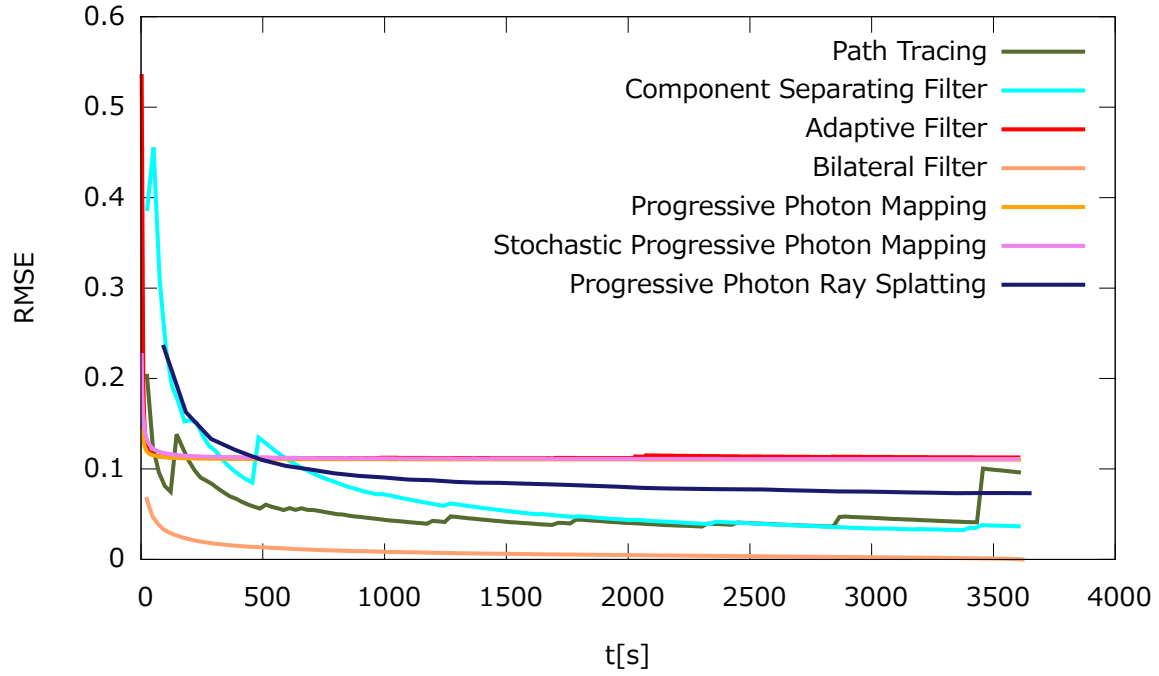


Figure 5.7: The time-error plot for the Glossy Interior scene

This one scene consists exclusively of slightly glossy materials, making this a very difficult scene. Indeed, even after a full day of rendering, a path tracer would not converge, still exhibiting significant high-frequency noise. As noise like this would irreparably skew our RMSE measurement, we were forced to use a different renderer for reference in this scene. For this we have chosen the bilateral filtered path tracer, which is energy conserving, doesn't exhibit white noise, but is otherwise visually very close to the path traced solution.

Therefore it is immediately obvious why the bilateral filter exhibits the lowest error overall, during the entire measurement. The shapes of convergence graphs of unfiltered path tracing and component separation filter show how the error may actually increase, even as variance decreases, as was mentioned in *Section* 2.3.4.

Of the photon-based methods, both photon mapping algorithms exhibit good initial convergence, but are soon overtaken first by path tracers and later by photon ray splatter, as well. A closer examination of resultant imagery shows, that while the photon ray splatter handles indirect illumination quite well, with bias spread out over the scene, direct illumination is very visibly biased. This can be ascribed to low

initial cone radius, which causes uneven illumination when the initial ray segment is comparatively short. In comparison, both photon mapping methods tend to under-illuminate the ceiling immediately next to the light.

On examining the adaptive filtered path tracer, we can see that it exhibits visibly less noise than path tracer, but still significantly more than the bilateral filter. Also, indirect illumination appears to have been suppressed by this filter – shadows are noticeably darker.

The bilateral filter, on the other hand, doesn't exhibit any visible noise and converges to the final solution very rapidly. It appears to have handled this very difficult scene quite well.

5.3.3 Ring

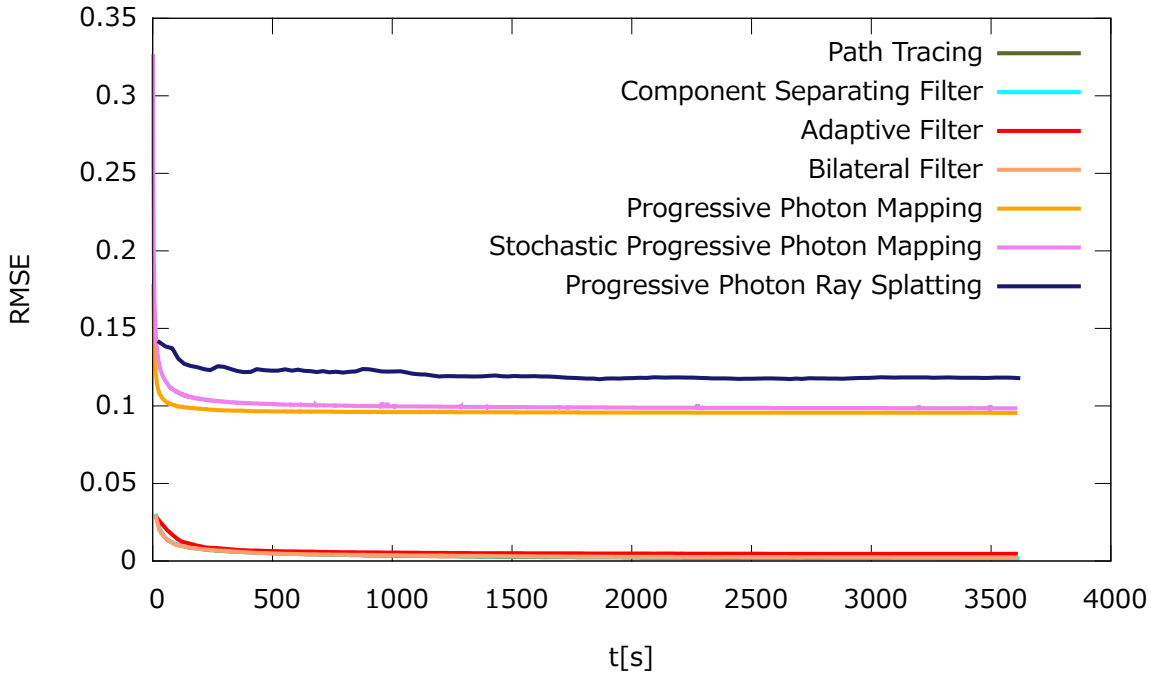


Figure 5.8: The time-error plot for the Ring scene

In this scene, two illumination components are clearly dominant. First, it is direct illumination, visible on the specular highlights, as well as the base. Second, it is the mild cardioid caustic visible inside the ring. Surprisingly, photon-based methods do not do very well in this particular scene. Not only do they miss the caustic, they miss the specular highlights, as well. Additionally, photon ray splatting exhibits significant light leaking.

The reason for this should be probably seen in the scene layout – the light is comparatively far up, causing most photons that are emitted to leave the scene without contributing. The probability of a photon successfully contributing to a hitpoint is minute, as the area of interest only takes up a small part of the scene.

In conclusion, this scene demonstrates that photon-based methods, as they are implemented, are ill-suited to exterior scenes.

5.3.4 Conference

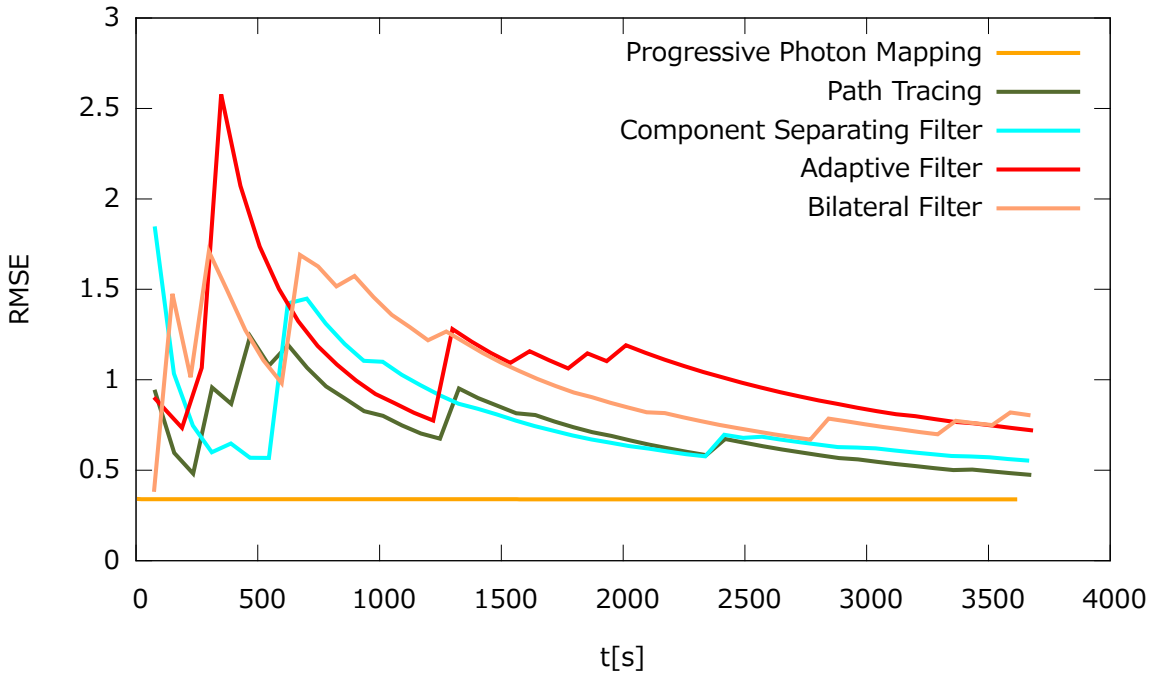


Figure 5.9: The time-error plot for the Conference scene, part 1

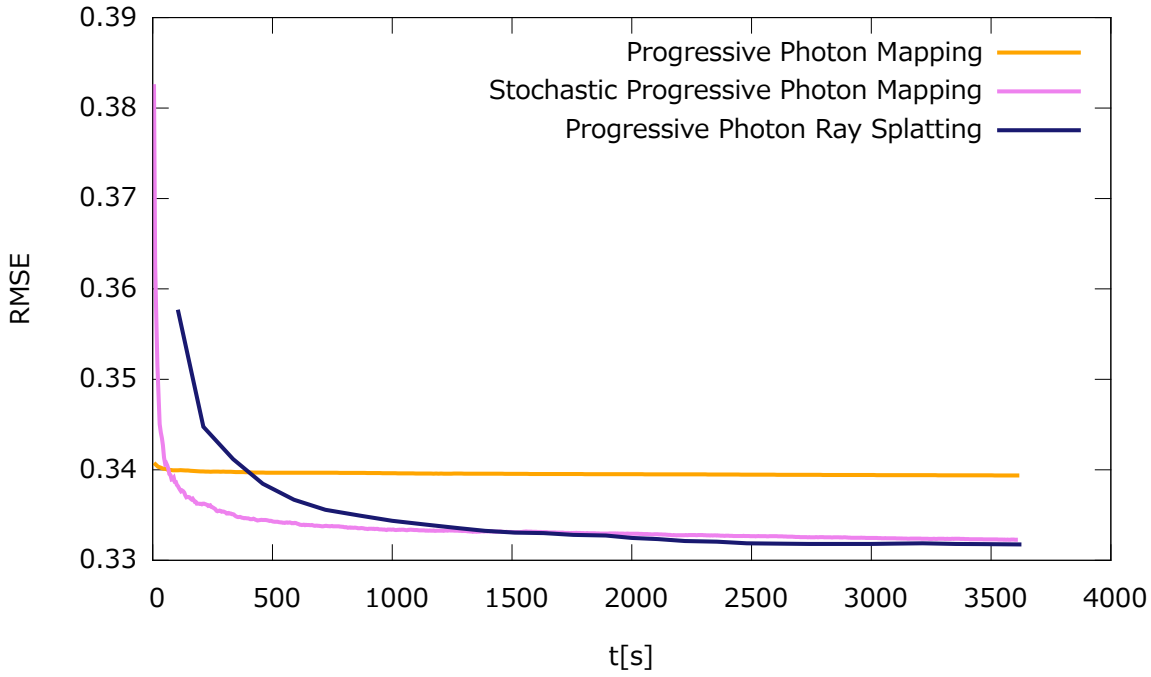


Figure 5.10: The time-error plot for the Conference scene, part 2

The time-error plot for this scene has been split into two parts, as there are performance-wise two groups of algorithms with significantly different rates of convergence.

The path tracers, even though unfiltered path tracing was eventually used for reference, initially exhibit an order of magnitude higher error, and for at least the first twenty minutes are visibly struggling with variance. Examination of images shows that regardless of filtering, there is a great deal of visible white noise.

In comparison, photon-based algorithms converge very rapidly. Stochastic progressive photon mapping shows the lowest error at first, but is eventually overtaken by photon ray splatter. On the other hand, examination of images reveals that there is very visible bias in photon ray splatting, due poor direct illumination handling. The difference images also show, that the main source of bias in progressive photon mapper is very likely aliasing – there is a significant amount of bias around the edges of light fixtures.

Also, due to differences in photon counting, stochastic progressive photon mapper does away with corner bias more quickly, which is visible especially on the conference chairs.

Overall, this scene highlights the strengths of photon mapping. There is a significant amount of indirect illumination, which is approximated handily by photon mapping, in spite of some rather complex paths.

5.3.5 Sponza

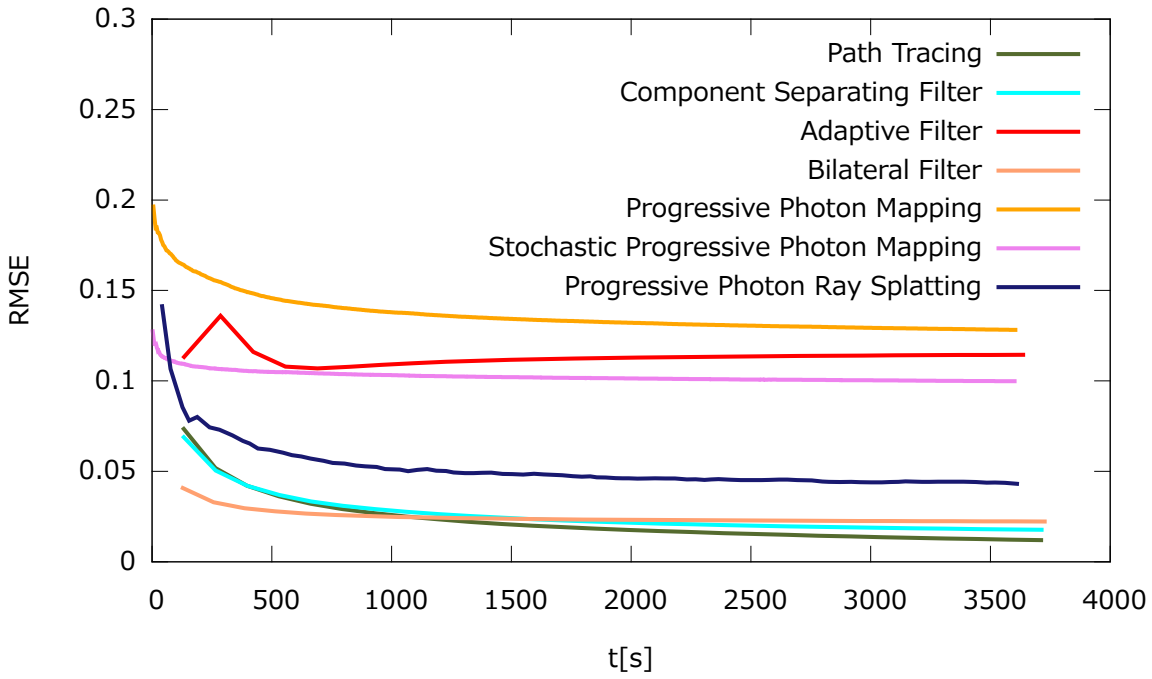


Figure 5.11: The time-error plot for the Sponza scene

Sponza is a semi-open scene with a large area light above the courtyard. Thus, not unlike the ring scene, many of generated photons will hit the roof or leave the scene entirely, handicapping photon-based algorithms. Out of these, photon ray splatting appears to be the most successful in this scene, likely due to the fact that a single photon ray can contribute to many more hitpoints than a photon in progressive photon mapping. Lack of contributing photons is clearly visible in discontinuity bias that only goes away very slowly. In comparison, photon ray splatting initially has significant bias due to light leaking, but this goes away as more photon rays are cast and the splatting footprint decreases.

Of the path tracers, bilateral filter initially exhibits the lowest error, but is later overtaken by both unfiltered path tracer, as well as the component separating filter. Adaptive filter appears to be rather ineffective in dealing with noise, as well as again leaving shadows visibly darker. Meanwhile, the component separating filter appears to introduce slightly more bias than it removes variance. Bilateral filter exhibits problems of its own, most significantly a lot of highly visible blurring that does not seem to go away as the render progresses.

Overall, the path tracers converge significantly faster on this scene, in great part due to their view-independent nature. This scene also seems to be the breaking point for several of the filters.

5.4 Performance Analysis

Performance analysis was performed on the Glossy Interior scene, using that scene's settings under the Compute Visual Profiler. This profiler takes and reports many different measurements, such as runtime of each kernel, breakdown of memory operations in each kernel, even taking into account cache utilization and cache misses. While this information is important when attempting to tune, debug or optimize an application, for our purposes the most important is the GPU Time Summary Plot, which provides aggregated information on how much time was spent in each of the kernels in the application lifetime. As our algorithms are based on the idea of path-atom binning, the tracing procedure is broken down into several kernels rather then using a single, monolithic one, so we may explore the computational complexity of each step separately.

To provide a detailed idea of the efficiency of each particular step of computation, we will provide the summary plots in full, and provide a brief explanation of the role of the most computational intensive kernels. An interested reader may explore the source code and documentation for more information.

5.4.1 Path Tracing

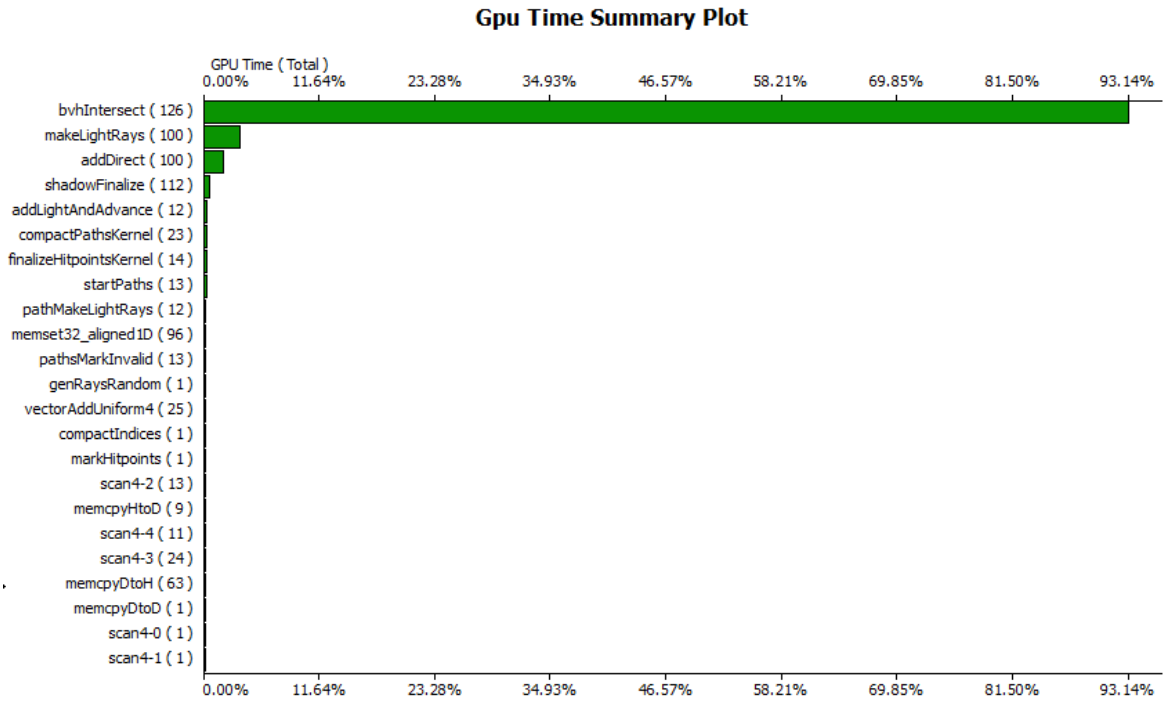


Figure 5.12: Profiling results for path tracing

As is evident from *Figures* 5.12 through 5.15, the single most computationally intensive part of path tracing is the `bvhIntersect` kernel. This kernel performs the ray tracing proper (that is, for an array of rays, finds the scene intersection for each of them, or reports that there is none). While we know from previous measurements that our ray tracing implementation is reasonably efficient, and is in fact

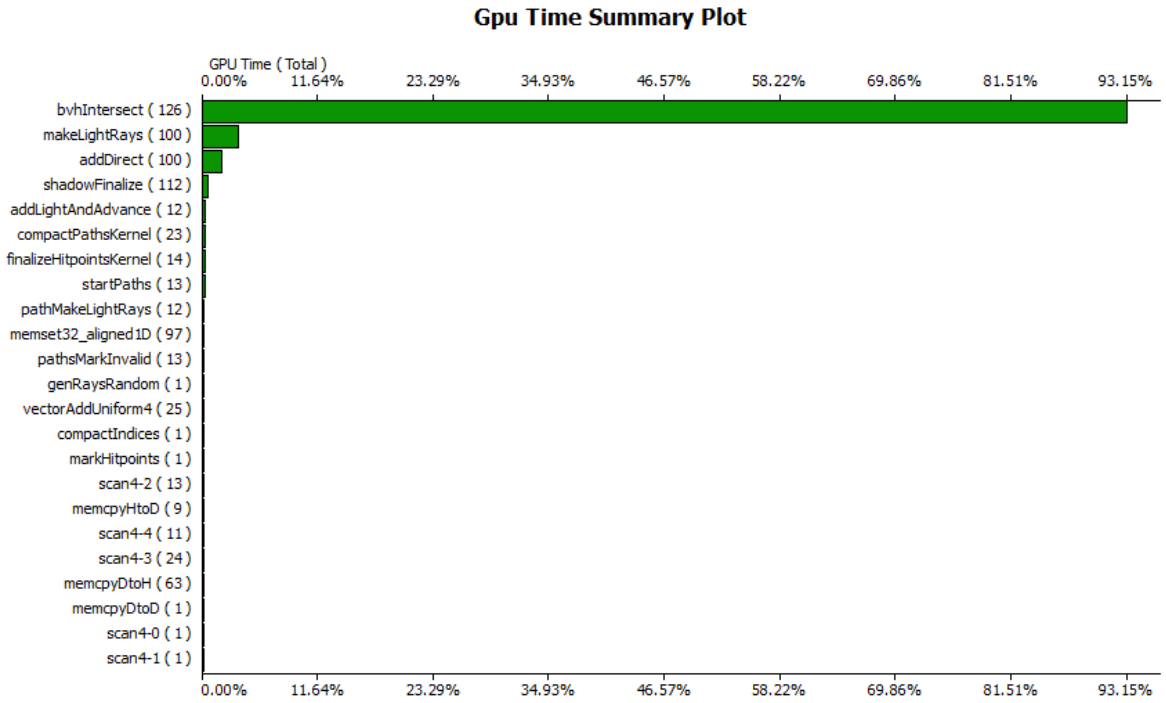


Figure 5.13: Profiling results for component separation filtered path tracing

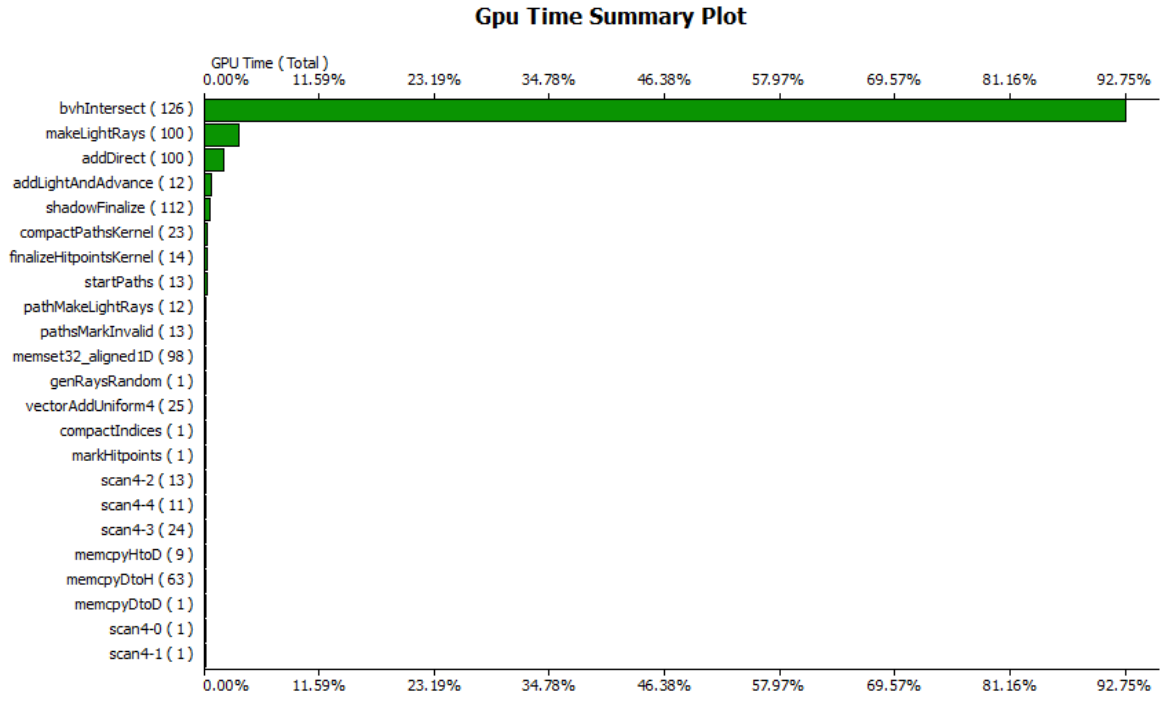


Figure 5.14: Profiling results foradaptive filtered path tracing

much faster than a comparable CPU implementation, it is very likely that other part of the path tracing primitive benefit from the parallelization significantly more.

The reason why ray tracing consumes so much computing time is that with the accelerating structure we are using, there are bound to be many memory operations that are not coherent within a warp (we call these operations uncoalesced). Such memory operations are likely *the* most expensive operations in CUDA, and in each ray tracing step, many have to be performed. Further problems stem from the fact that secondary rays in path tracing tend to be very incoherent in themselves, making any attempt at exploiting coherence a losing proposition.

This operation is so significant because many more rays are traced in path tracing than in any other method used – each path casts two rays for every bounce, and we have to cast hundreds to thousands paths per pixel, so in our test case with 100 paths per pixel per iteration, 25 million paths would be traced each round, requiring hundreds of millions rays to be traced each iteration.

The only other significant contributor in our implementation is the `makeLightRays` kernel, using up about 4% of GPU time. The purpose of this kernel is to sample lights, create shadow rays and compute radiance transfer for direct illumination. While a quick glance at this method reveals there is room for improvement (for instance, searching the light array with a binary search rather than linearly, or using only 8 TEA rounds instead of 16), there is little reward in attempting to do so.

Note that none of the filtering kernels are even in the plot – this means that the filtering operation is essentially "free" (probably due to it exhibiting good memory access coherence), there is no penalty in using filtering whenever we might think it is convenient, or rather, whenever measurements show it would lead to better image quality with less variance. Conversely, the filtering step may be omitted at will, if we find it to be unnecessary with a converged path traced image.

5.4.2 Progressive Photon Mapping

Looking at *Figures* 5.16 and 5.17, the `gatherPhotons` kernel takes up the most GPU time. This kernel in fact implements photon gathering described in [HJ10] over a hashed grid. Compared to this, the runtime of `contributeNaive` kernel that assigns photons to grid cells seems negligible. We believe the cause of this to be that while `contributeNaive` is parallelized per photon and is performed by thousands of threads at most, each only exploring a single grid cell, the `contributeNaive` kernel is parallelized per hitpoint, is executed by hundreds of thousands to millions threads in parallel, each of which accesses up to eight grid cells. These accesses are then uncoalesced both in time and among threads, due to hashing and due to there being little coherence between hitpoint positions in the grid and their placement in the hitpoint array.

There could be several ways to remedy this. At the expense of memory, we could use a full grid rather than a hashed one (though we would not be able to resize it and change cell counts at will) and impose some sort of spatial sorting on the hitpoints, thus improving coherence. Alternatively, it would make sense from an algorithmic standpoint to splat photons to hitpoints (peforming thousands of searches of an accelerating structure rather than millions) rather than gather them as we do now. This could also save us the expense of rebuilding the grid, but the cost of that operation is not high enough to substantiate that. Whether this approach would bring any improvement is a question for future research.

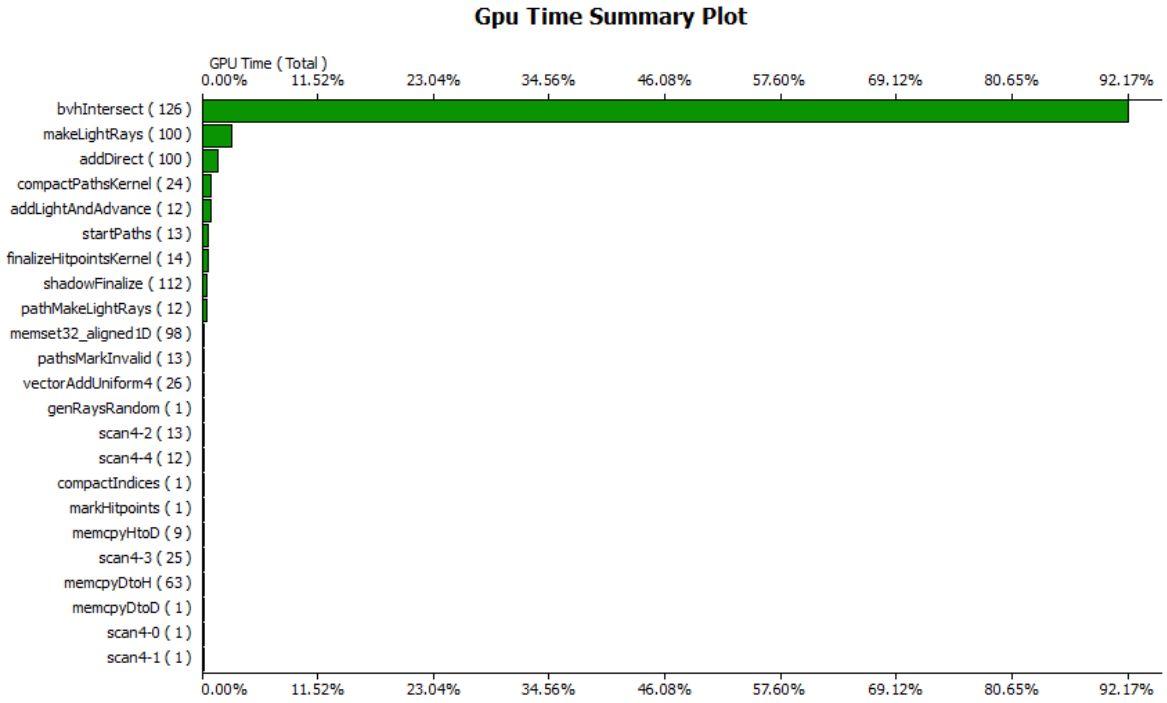


Figure 5.15: Profiling results for bilateral filtered path tracing

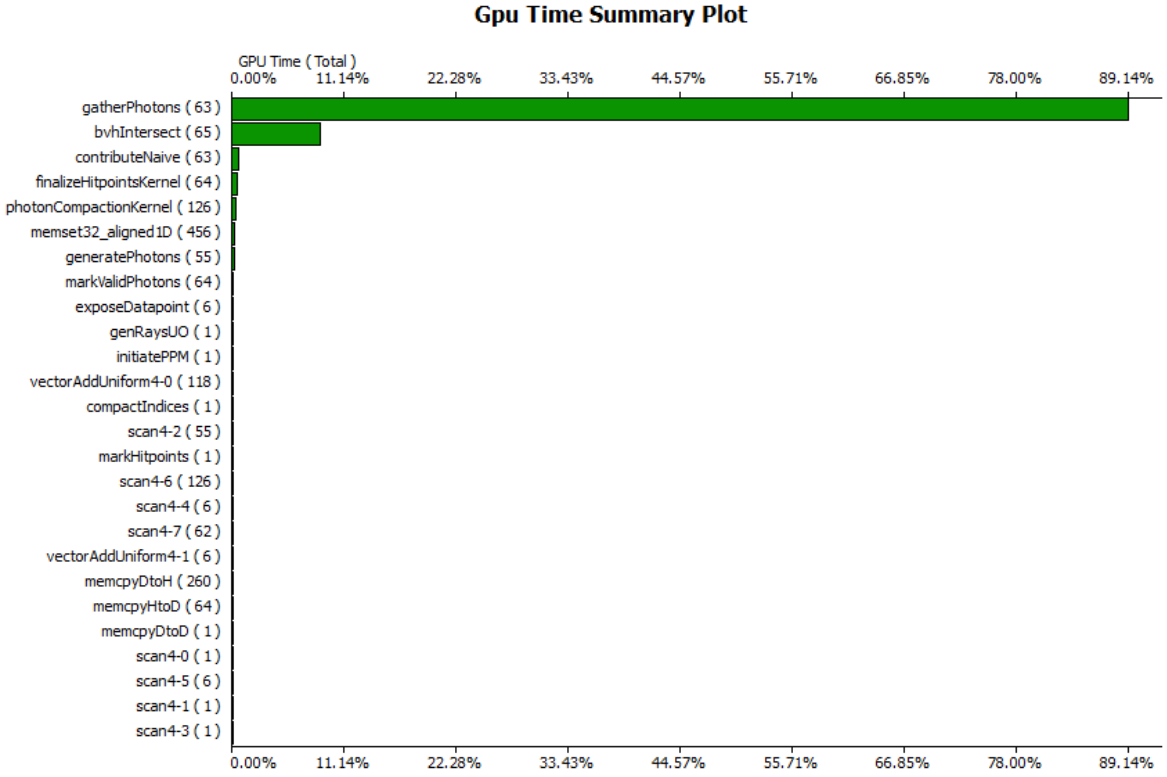


Figure 5.16: Profiling results for progressive photon mapping

One should also note that *Figure* 5.17 shows much more time being spent in the `bvhIntersect` kernel, tracing rays. While the exact same amount of photons is traced in Progressive Photon Mapping as in Stochastic Progressive Photon Mapping, the primary ray recasts appear to be significant. This is true even in spite of their being much more coherent than photon rays – there is simply significantly more of primary rays than there are photon rays.

The greatest potential for improvement thus lies in the splatting/gathering step. Since photon mapping techniques have been shown to exhibit comparable convergence to a full path tracer with significantly less rays traced, this may be a viable avenue of research.

5.4.3 Progressive Photon Ray Splatting

Figure 5.18 shows that practically the entire runtime of Photon Ray Splatting is spent in the `splatAndBounce` kernel. This kernel runs the heuristic from *Equation* 3.12 to determine ray cone dimensions and than traverses a kD-tree to find all hitpoints within that area. The photon is then bounced or terminated, as in regular photon mapping. This may indicate either that the kD-tree is in fact a very poor structure to traverse on a GPU, or that the cone dimensions that resulted from our choice of parameters are not very viable for practical computation.

Some improvement could be found if we attempted to spatially sort the photon rays to improve memory access coherence. Thought could also be given to the use of alternative accelerating structures. On the other hand, it has shown that the heuristic we have proposed is far from optimal, and a better one could improve performance both by reshaping the cones to minimize bias and by being more judicial with cone dimensions. Again, this is a topic for future research.

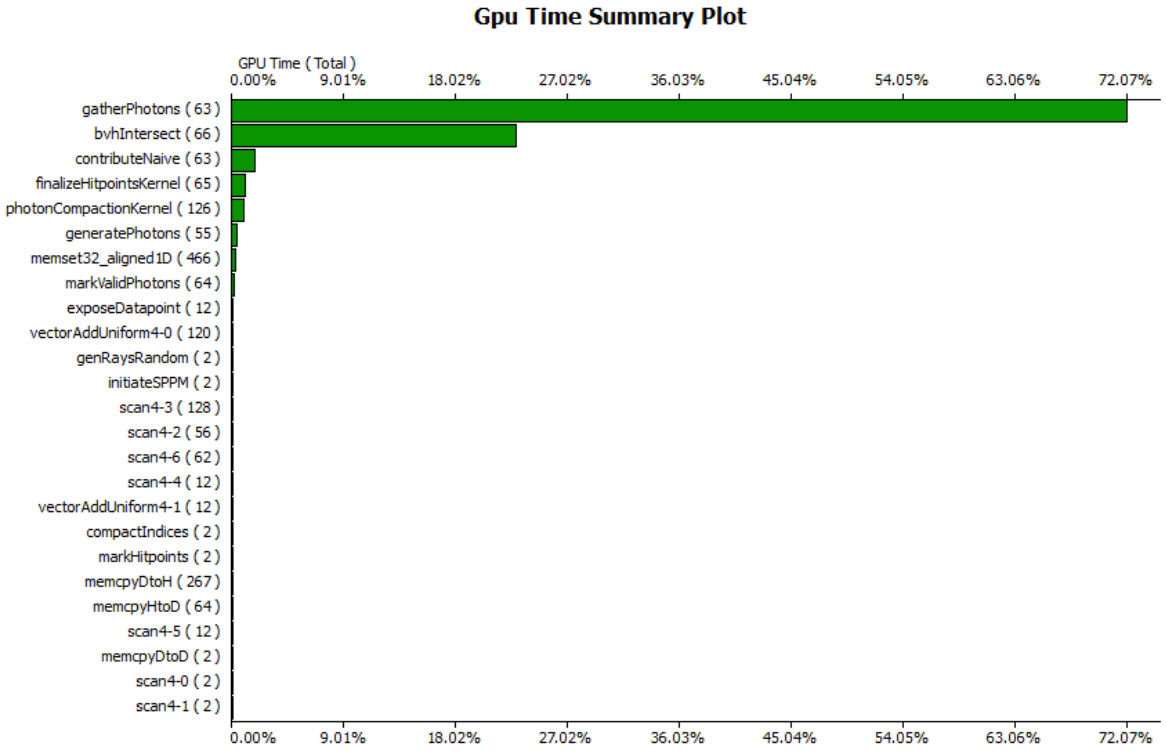


Figure 5.17: Profiling results for stochastic progressive photon mapping

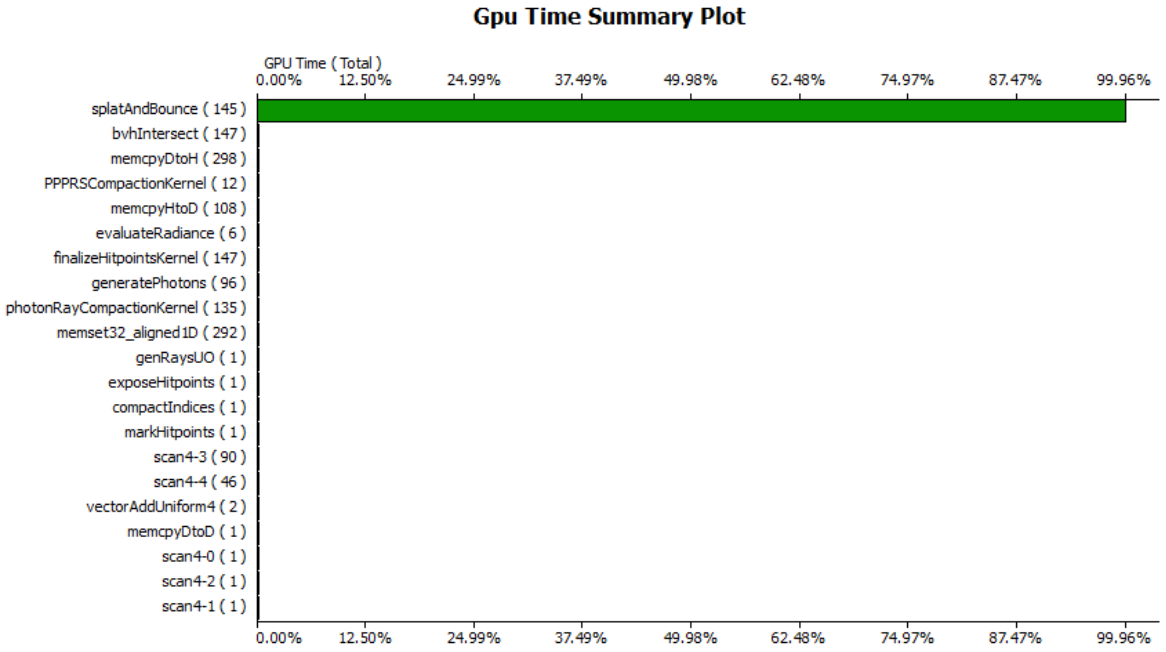


Figure 5.18: Profiling results for photon ray splatting

Chapter 6

Conclusion

Our testing has shown that each of the algorithms we tried is in itself viable for GPU use. None of them was found to be inherently superior to others, as relative performance was shown to be scene-dependent. Overall, photon-based algorithms converged faster in interior scenes, while path tracers exhibited better convergence on scenes dominated by diffuse inter-reflections or direct illumination.

Incoherent Path-atom Binning [CDD+10] has been shown to be useful in CUDA, improving thread coherence with minimum overhead. An extension of the original principle allowed an efficient implementation of photon tracing.

Measurements have shown that powerful filters for path tracing may be implemented in CUDA for a negligible computational cost, and that some of them, particularly the *Bilateral Filter* [XP05], are very efficient in removing high-frequency noise even from difficult scenes, such as scenes comprising mostly of moderately glossy materials.

Among photon mapping algorithms, *Stochastic Progressive Photon Mapping* [HJ09] has proven superior to *Progressive Photon Mapping* [HOJ08], in no small part due to its enhanced capabilities.

While *Progressive Photon Ray Splatting* [HHK+07] has proven viable, neither the original kernel footprint heuristic, nor our progressive heuristic are entirely satisfactory.

Future Work

For path tracing, a progressive version of the bilateral filtering operator could be developed, reducing the range of the near-true estimator as well as the domain range as variance is expected to decrease. By figuring in some sort of error estimation, a progressive filter could avoid introducing too much bias through blurring, as we have seen in testing.

For photon mapping, it would certainly be worth the effort to explore more efficient data structures and approaches to photon splatting or gathering, with GPU implementation specifically in mind.

Progressive photon ray splatting definitely needs a better splatting footprint heuristic, to resolve both the problems with light leaking and poor performance in direct illumination. Furthermore, the data structure

used along with a rather inconvenient shape of the search area significantly impact performance, so the new heuristic could help in this respect by selecting cone dimensions more judiciously, to maintain thread coherence.

Additionally, all of the photon-based methods suffer when used in exterior scenes, due to a significant proportion of photons leaving the scene without contributing to the image. Progressive rendering provides us with a unique opportunity to interactively keep track of the contribution of each photon, and in response, re-shape the emission probability density function so that areas and directions that contribute more to the final image are preferred.

Pertaining specifically to path tracing and stochastic progressive photon mapping, utilizing GPUs to progressively render entire animations rather than a single frame might have applications as a superior way to render an animation compared to frame-by-frame rendering.

Bibliography

[AFH+01] James Arvo, Marcos Fajardo, Pat Hanrahan, Henrik Wann Jensen, Don Mitchell, Matt Pharr, and Peter Shirley. State of the art in monte carlo ray tracing for realistic image synthesis, 2001. Siggraph 2001 course.

[Bad94] Lee Badger. Lazzarini's lucky approximation of π. *Mathematics Magazine*, 67:93–91, April 1994.

[CDD+10] David Coulthurst, Piotr Dubla, Kurt Debattista, Simon McIntosh-Smith, and Alan Chalmers. Parallel path tracing using incoherent path-atom binning. In *Proceedings of the 24th Spring Conference on Computer Graphics*, SCCG '08, pages 91–95, New York, NY, USA, 2010. ACM.

[DBB06] Phil Dutre, Kavita Bala, and Philippe Bekaert. *Advanced Global Illumination, 2nd Edition*. A K Peters, Natick, MA, 2006.

[DGNP88] Frederica Darema, David A. George, V. Alan Norton, and Gregory F. Pfister. A single-program-multiple-data computational model for epex/fortran. *Parallel Computing*, 7:11–24, 1988.

[DLW93] Phil Dutre, Eric P. Lafortune, and Yves D. Willems. Monte carlo light tracing with direct computation of pixel intensities. In *Proceedings of Third International Conference on Computational Graphics and Visualization Techniques (COMPUGRAPHICS '93)*, pages 128–137, Alvor, 1993.

[Fly72] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, C-21:948+, 1972.

[GTGB84] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modeling the interaction of light between diffuse surfaces. *SIGGRAPH Comput. Graph.*, 18:213–222, January 1984.

[HHK+07] Robert Herzog, Vlastimil Havran, Shinichi Kinuwaki, Karol Myszkowski, and Hans-Peter Seidel. Global illumination using photon ray splatting. In Daniel Cohen-Or and Pavel Slavik, editors, *Computer Graphics Forum (Proceedings of Eurographics)*, volume 26(3), pages 503–513, Prague, Czech Republic, 2007. Blackwell.

[HJ09] Toshiya Hachisuka and Henrik Wann Jensen. Stochastic progressive photon mapping. In *ACM SIGGRAPH Asia 2009 papers*, SIGGRAPH Asia '09, pages 141:1–141:8, New York, NY, USA, 2009. ACM.

[HJ10] Toshiya Hachisuka and Henrik Wann Jensen. Parallel progressive photon mapping on gpus. In *ACM SIGGRAPH ASIA 2010 Sketches*, SA '10, pages 54:1–54:1, New York, NY, USA, 2010. ACM.

[HJJ10] Toshiya Hachisuka, Wojciech Jarosz, and Henrik Wann Jensen. A progressive error estimation framework for photon density estimation. *ACM Trans. Graph.*, 29:144:1–144:12, December 2010.

[HOJ08] Toshiya Hachisuka, Shinji Ogaki, and Henrik Wann Jensen. Progressive photon mapping. In *ACM SIGGRAPH Asia 2008 papers*, SIGGRAPH Asia '08, pages 130:1–130:8, New York, NY, USA, 2008. ACM.

[HSO07] Mark Harris, Shubho Sengupta, and John Owens. Cuda data parallel primitives. `http://code.google.com/p/cudpp/`, 2007.

[JC94] Henryk Wann Jensen and Niels Jørgen Christensen. Optimizing path tracing using noise reduction filters. In *The Third International Conference in Central Europe on Computer Graphics and Visualisation 95 Proceedings Volume II*, November 1994.

[Jen96] Henrik Wann Jensen. Global illumination using photon maps. In *Proceedings of the eurographics workshop on Rendering techniques '96*, pages 21–30, London, UK, 1996. Springer-Verlag.

[Kaj86] James T. Kajiya. The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '86, pages 143–150, New York, NY, USA, 1986. ACM.

[Khr08] Khronos Group. The OpenCL homepage. `http://www.khronos.org/opencl/`, 2008.

[LW93] Eric P. Lafortune and Yves D. Willems. Bi-directional path tracing. In *Proceedings of Third International Conference on Computational Graphics and Visualization Techniques (COMPUGRAPHICS '93)*, pages 145–153, 1993.

[LW94] Eric P. Lafortune and Yves D. Willems. Using the modified phong brdf for physically based rendering. *Computing*, (CW197):19, 1994.

[Met87] Nicholas Metropolis. The beginning of the Monte Carlo method. *Los Alamos Sci.*, Special Issue, 1987.

[MN98] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8:3–30, January 1998.

[MU49] Nicholas Metropolis and Stanislaw Ulam. The Monte Carlo Method. *Journal of the American Statistical Association*, 44, September 1949.

[NVI07] NVIDIA. The CUDA homepage. `http://www.nvidia.com/object/cuda_home.html`, 2007.

[NVI09] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Fermi, 2009. `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_ Compute_Architecture_Whitepaper.pdf`.

[Pho75] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18:311–317, June 1975.

[SHAP01] Neal Sample, Matthew Haines, Mark Arnold, and Timothy Purcell. Optimizing search strategies in k-d trees. In *5th WSES/IEEE World Multiconference on Circuits, Systems, Communications & Computers (CSCC 2001)*, July 2001.

[SW00] Frank Suykens and Yves D. Willems. Adaptive filtering for progressive monte carlo image rendering. In *The 8-th International Conference in Central Europe on Computer Graphics, Visualization and Interactive Digital Media 2000 (WSCG' 2000), February 2000. Held in Plzen, Czech Republic*, 2000.

[WRC88] Gregory J. Ward, Francis M. Rubinstein, and Robert D. Clear. A ray tracing solution for diffuse interreflection. *SIGGRAPH Comput. Graph.*, 22:85–92, June 1988.

[XP05] Ruifeng Xu and Sumanta N. Pattanaik. A novel monte carlo noise reduction operator. *IEEE Comput. Graph. Appl.*, 25:31–35, March 2005.

[ZOC10] Fahad Zafar, Marc Olano, and Aaron Curtis. Gpu random numbers via the tiny encryption algorithm. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 133–141, Saarbrücken, Germany, 2010. Eurographics Association.

Appendix A

List of Acronyms

2D Two-Dimensional

BRDF Bidirectional Reflection Distribution Function

CUDA Compute Unified Device Architecture, a GPU programming platform from nVidia Corporation

FLOPS Floating Point Operations per Second

GI Global Illumination

GPU Graphics Processing Unit, also known as Graphics Card

GPGPU General Purpose GPU, a GPU utilization paradigm

MIMD Multiple Instruction Multiple Data, a class of parallel computer architectures

RMSE Root Mean Square Error, an estimator error metric

SIMD Single Instruction Multiple Data, a class of parallel computer architectures in Flynn's taxonomy [Fly72]

SPMD Single Program Multiple Data, a class of parallel computers. Introduced in [DGNP88] and is in fact a subclass of MIMD.

Appendix B

Algorithm Configuration in Testing

Algorithm	Variable	Value				
		Cornell	Rawalanche	Ring	Conference	Sponza
PT	parallelPaths	65536	65536	65536	65536	65536
	pathsPerRound	100	100	100	100	100
	directSamples	100	100	100	100	100
CSF	kernelWidth	4	4	4	4	4
AF	α	1/3	1/3	1/3	1/3	1/3
	C	512	512	512	512	512
BF	σ_d	8	8	8	8	8
	σ_r	1	1	1	1	1
PPM	parallelPhotons	65536	65536	65536	65536	65536
	photonsPerRound	2M	2M	2M	2M	2M
	initialRadius	1	1	1	1	1
	α	0.8	0.8	0.8	0.8	0.8
SPPM	parallelPhotons	65536	65536	65536	65536	65536
	photonsPerRound	2M	2M	2M	2M	2M
	initialRadius	0.5	0.5	0.5	0.5	0.5
	α	0.8	0.8	0.8	0.8	0.8
PPRS	parallelPhotons	1024	1024	1024	1024	1024
	photonsPerRound	8192	8192	8192	8192	8192
	C	0.01	0.01	0.005	0.01	0.01
	α	1/3	1/3	1/3	1/3	1/3

Table B.1: The names and meaning of configuration variables for our algorithms

Appendix C

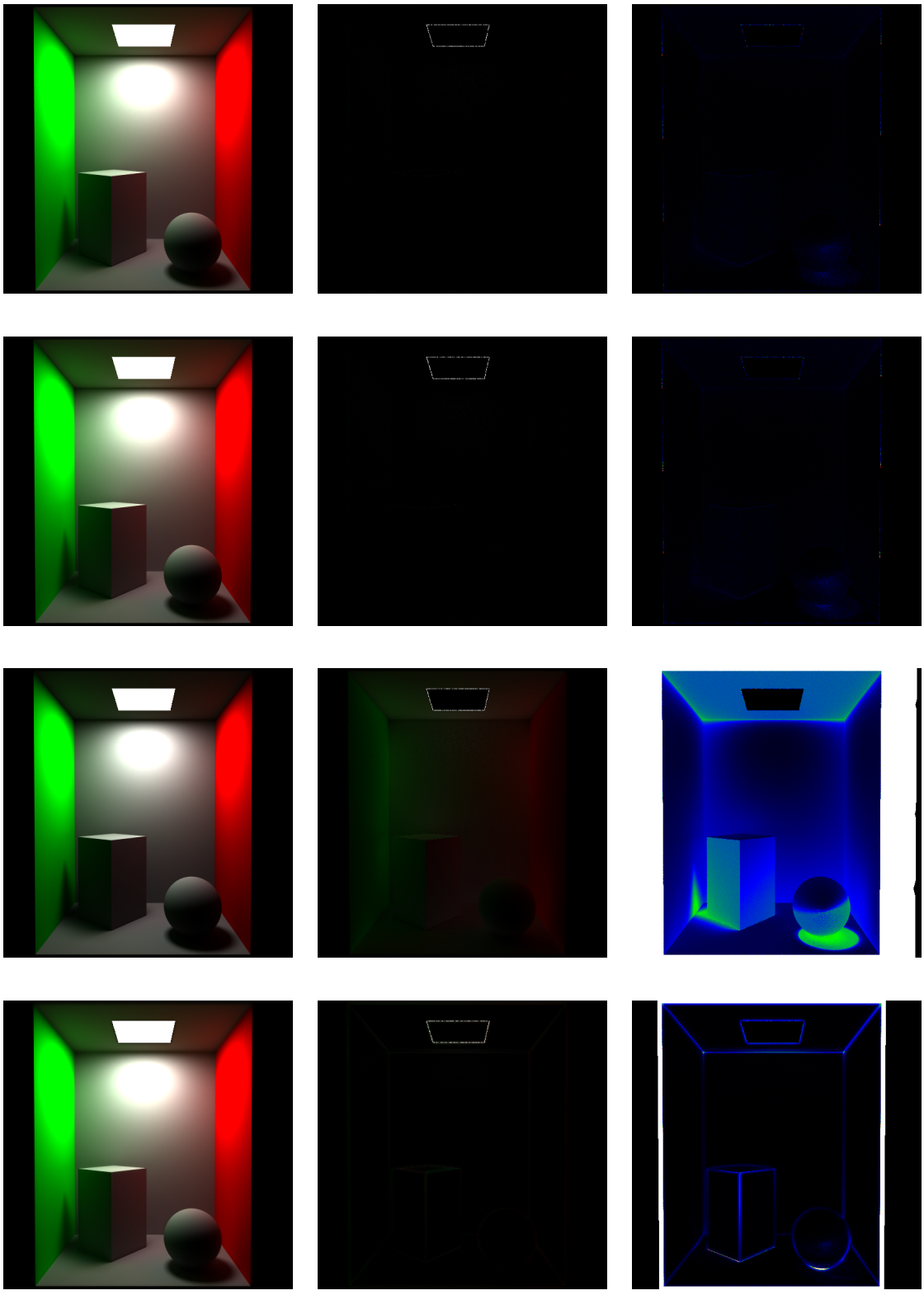
Selected Rendering Results

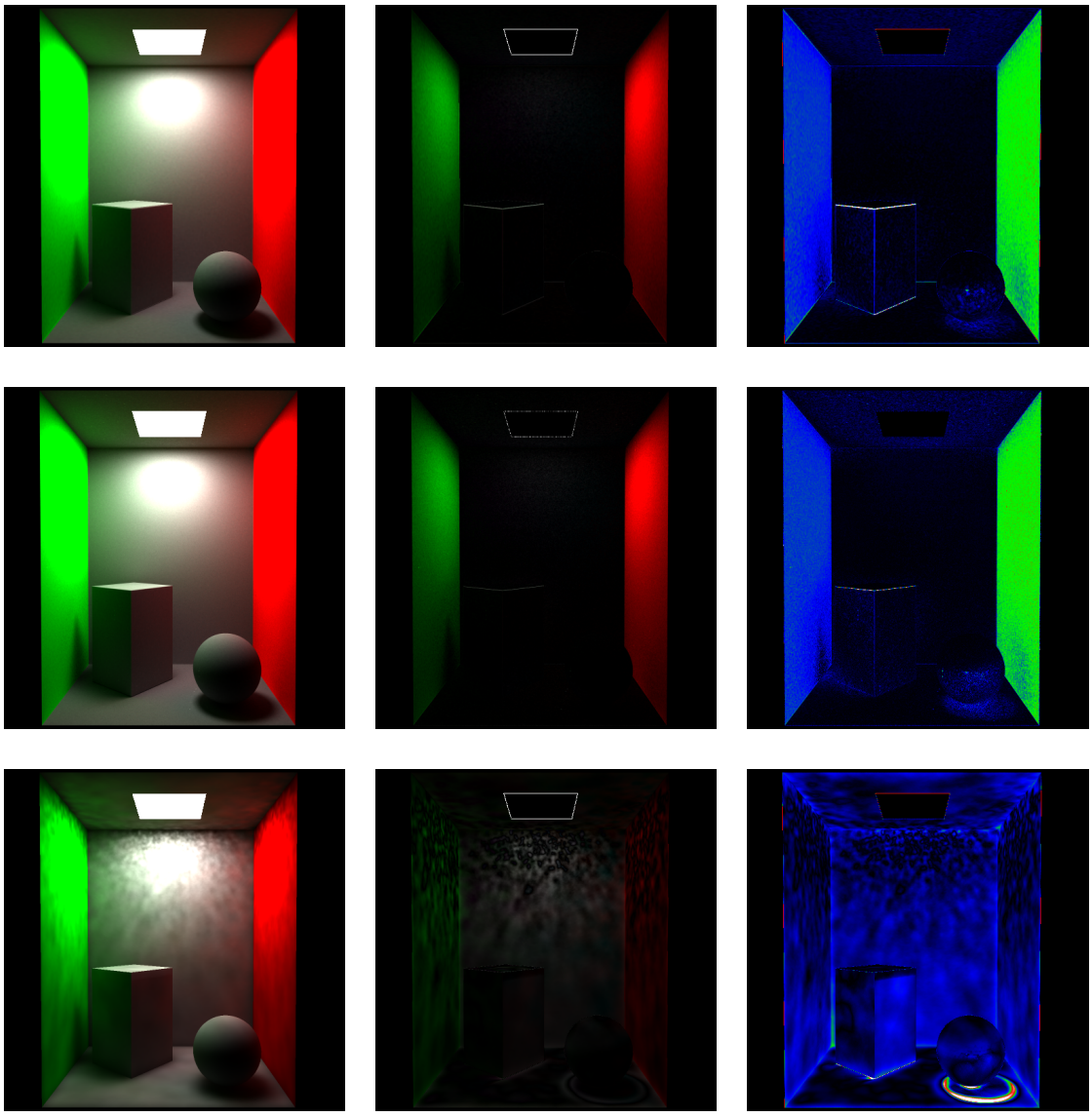
For each scene and each algorithm, this appendix contains a representative output image. A complete set of extracted images (one taken every 15 minutes) and differential images can be found on the enclosed DVD. The images presented are a rendering output, a differential image and a false colour relative difference image, mapped to a BGR gradient (for relative difference between 0 and 1, with greater difference shown as white).

The algorithm order is:

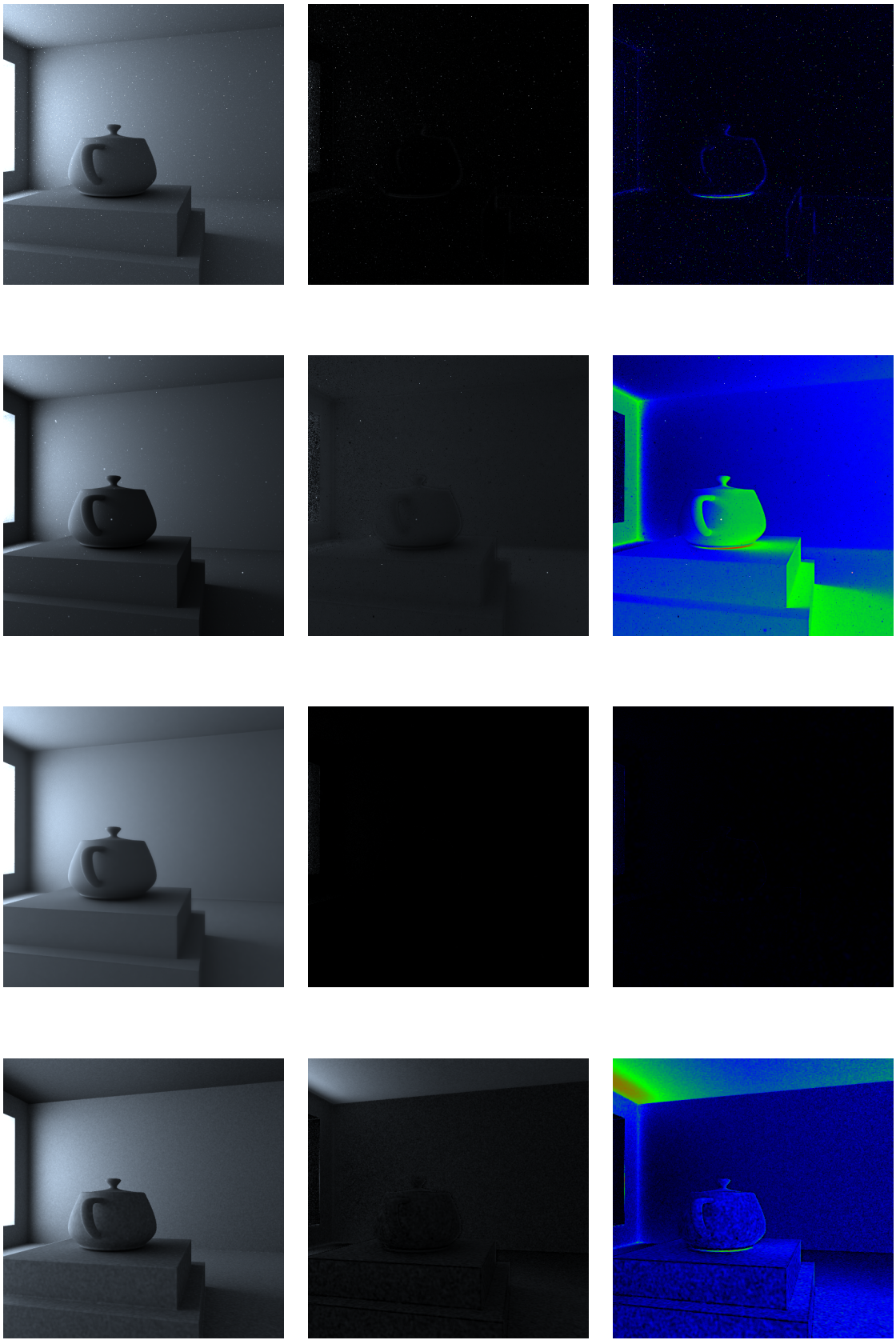
1. Path Tracing
2. Component Separation Filtered Path Tracing
3. Adaptive Filtered Path Tracing
4. Bilateral Filtered Path Tracing
5. Progressive Photon Mapping
6. Stochastic Progressive Photon Mapping
7. Progressive Photon Ray Splatting

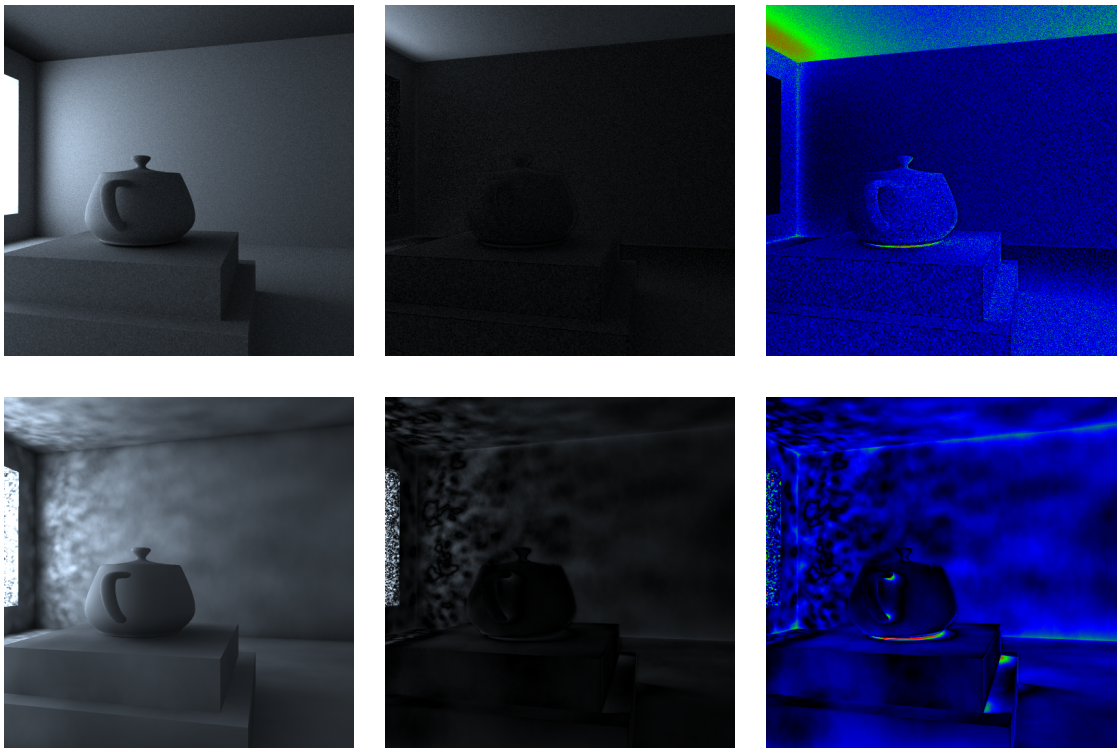
Cornell Box



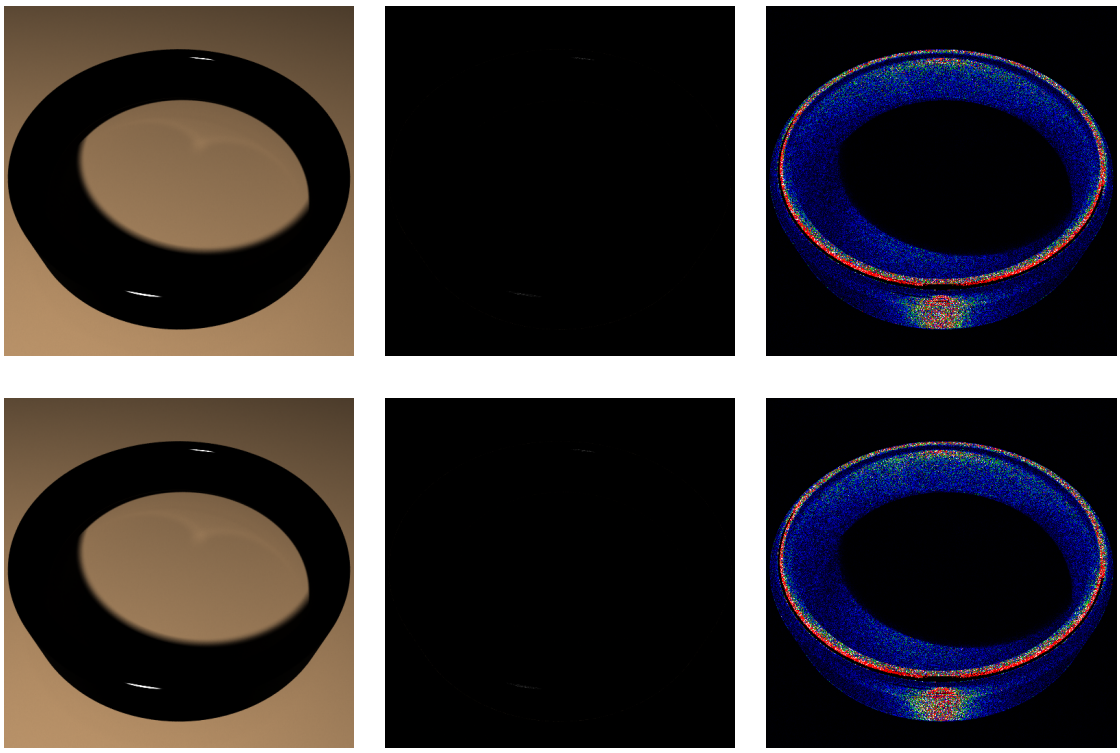


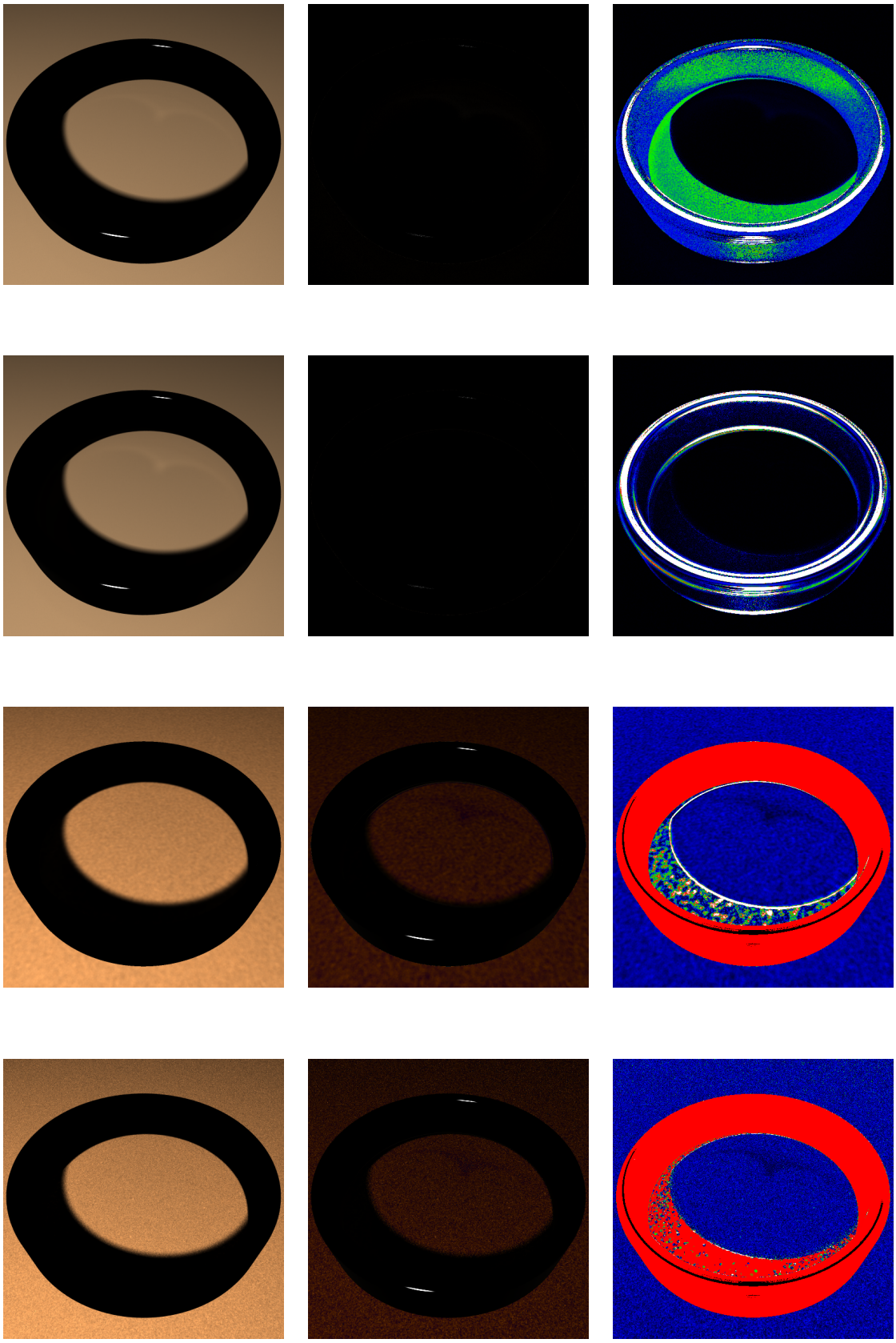
Glossy Interior



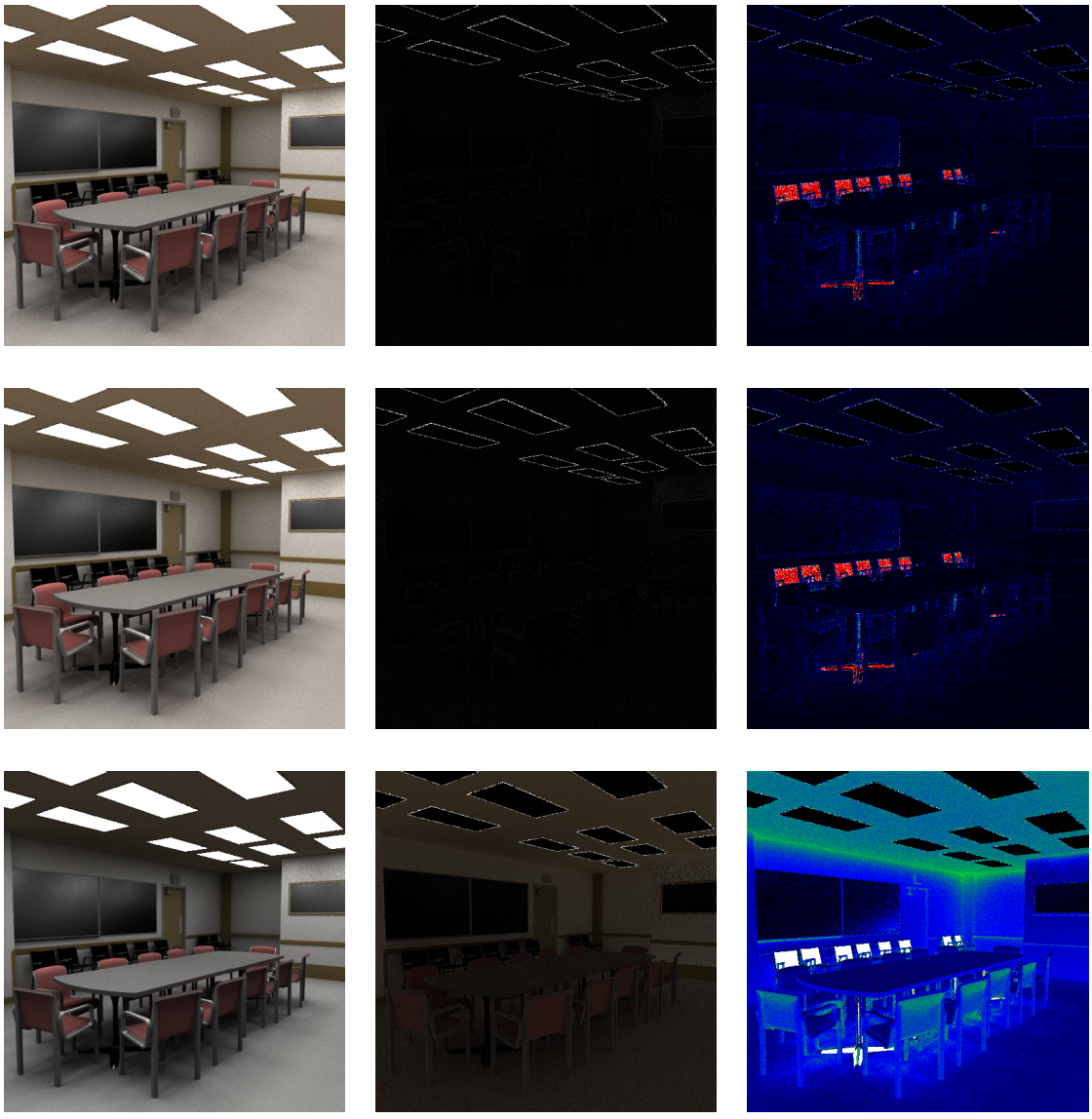


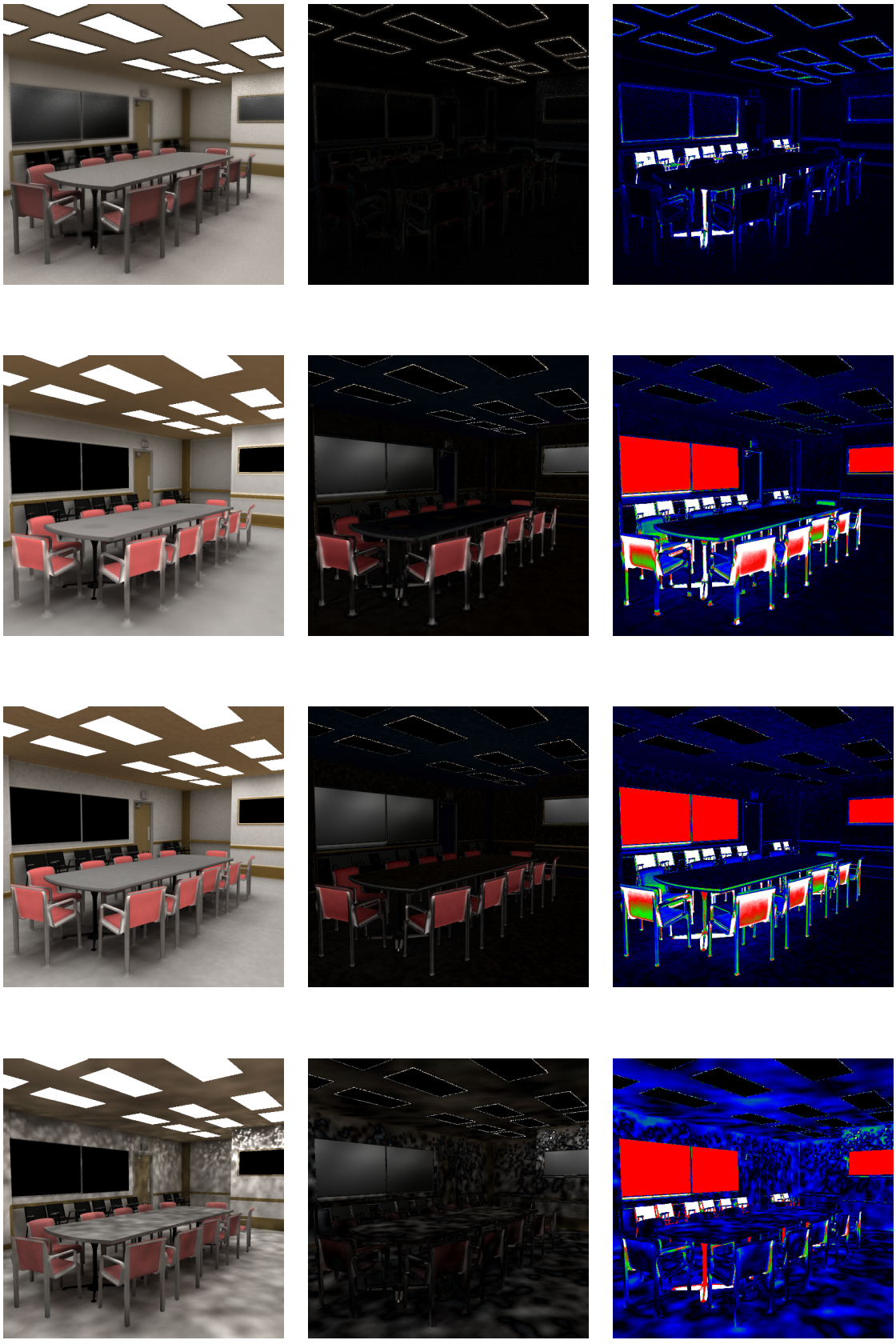
Ring



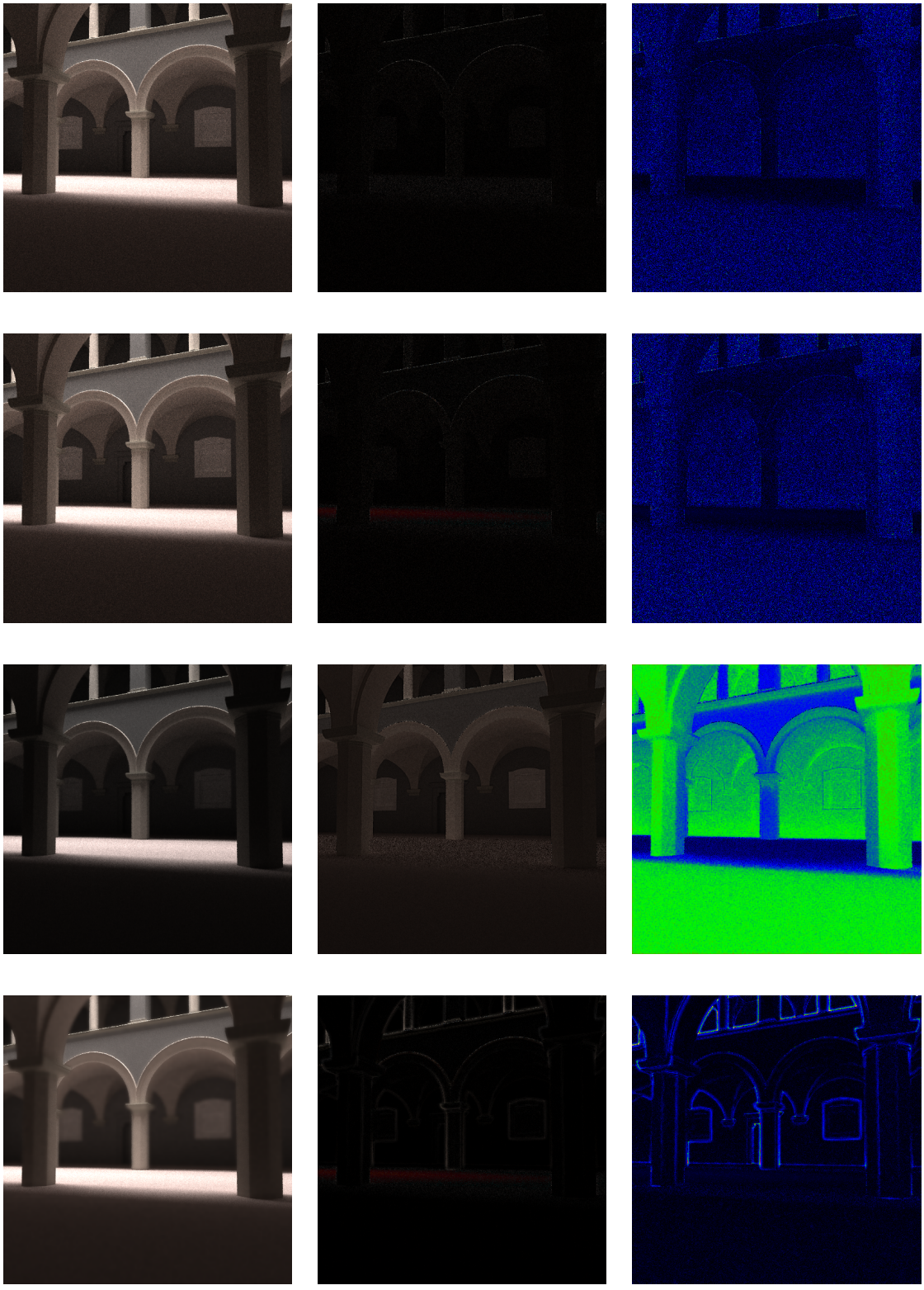


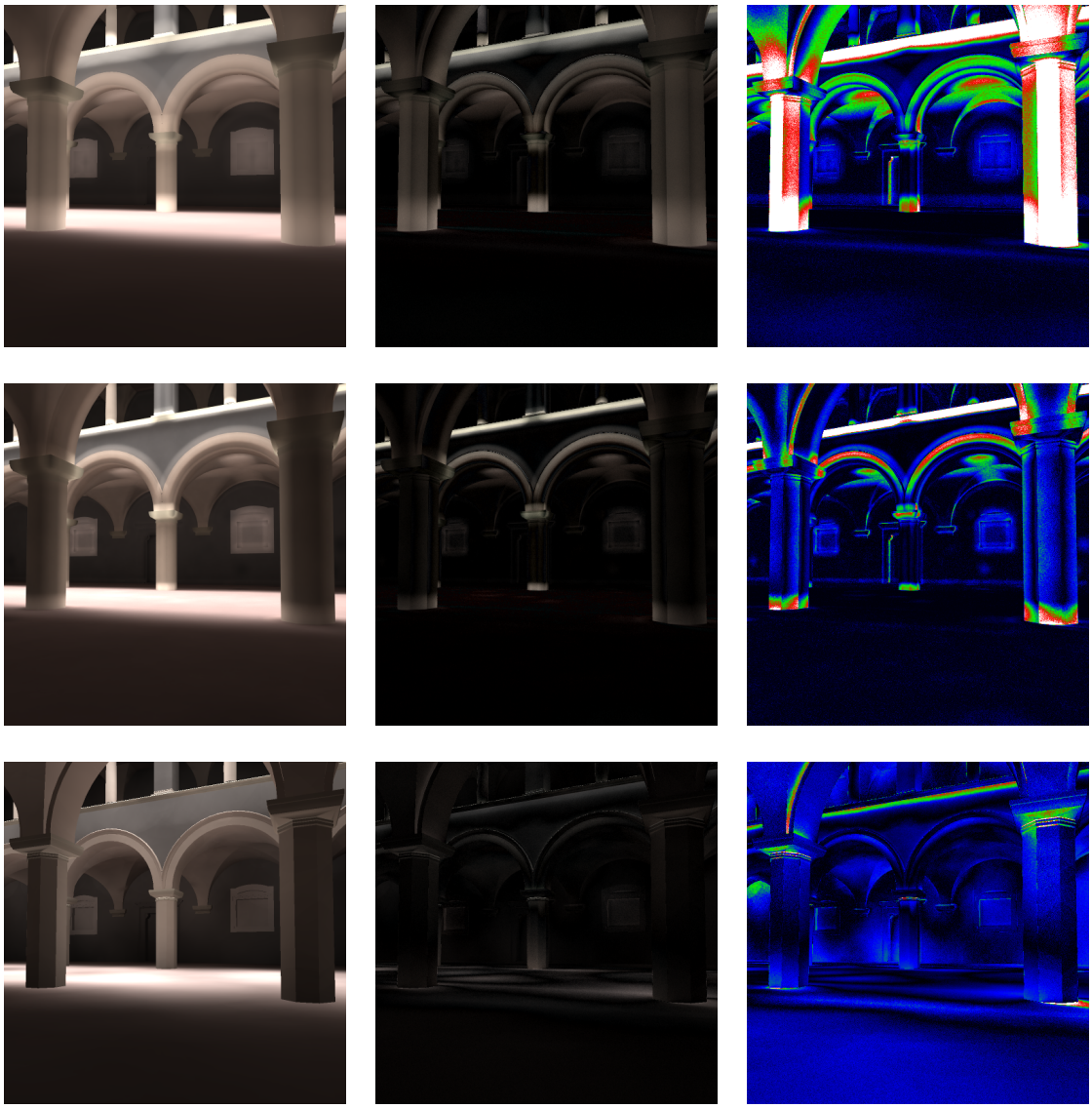
Conference





Sponza





Appendix D

Enclosed DVD Contents

The enclosed DVD contains the source code of our implementation, as well as a helper project used to process and aggregate rendering results. Also enclosed is documentation for both projects and their executable binaries.

All the data necessary to replicate our testing results is included, although the executables have certain hardware requirements.

The enclosed DVD is organized thus:

```
.
├── bin
│   └── The binaries and their associated DLLs
├── doc
│   ├── proj
│   │   └── Code documentaton
│   ├── thesis
│   │   └── This thesis
│   └── thesis-src
│       └── Source code of this thesis
├── results
│   └── Aggregated testing results
├── scenes
│   └── Testing scenes
└── src
    ├── RawViz
    │   └── The .raw format visualizer
    └── Uskglass
        └── The Uskglass renderer
```