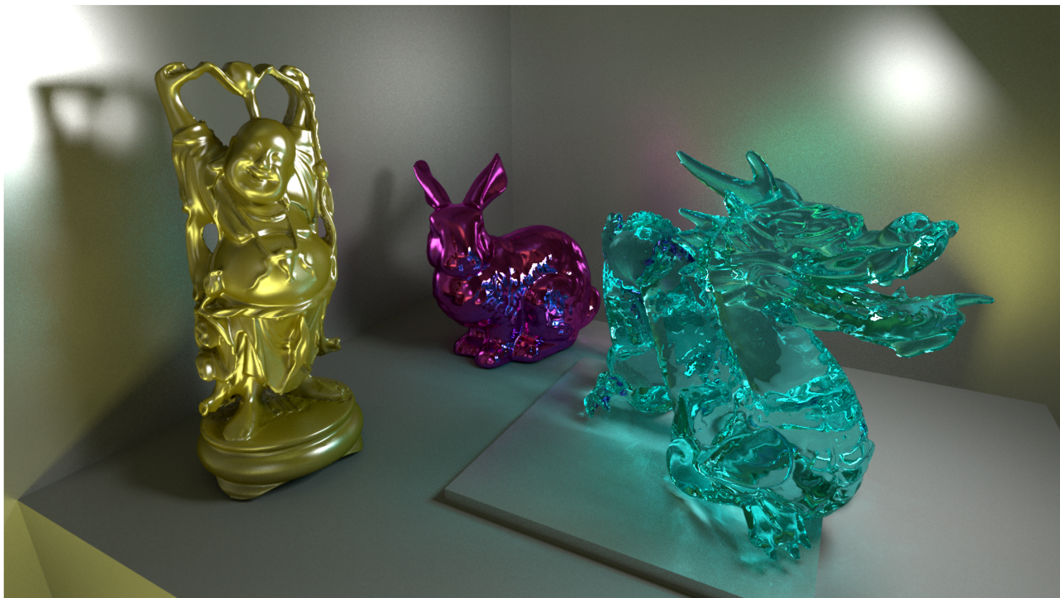
Unbiased physically based rendering on the GPU



Dietger van Antwerpen

Unbiased physically based rendering on the GPU

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Dietger van Antwerpen
born in Rotterdam, the Netherlands

Computer Graphics Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

© 2010 Dietger van Antwerpen.

Cover picture: A GPU rendering of the STANFORD scene.

Unbiased physically based rendering on the GPU

Author: Dietger van Antwerpen
Student id: 1230506
Email: Dietger@xs4all.nl

Abstract

Since the introduction of General-Purpose GPU computing, there has been a significant increase in ray traversal performance on the GPU. While CPU's perform reasonably well on coherent rays with similar origin and direction, on current hardware the GPU vastly outperforms the CPU when it comes to incoherent rays, required for unbiased rendering. A number of path tracers have been implemented on the GPU, pulling unbiased physically based rendering to the GPU. However, since the introduction of the path tracing algorithm, the field of unbiased physically based rendering has made significant advances. Notorious improvements are BiDirectional Path Tracing and the Metropolis Light Transport algorithm. As of now little effort has been made to implement these more advanced algorithms on the GPU.

The goal of this thesis is *to find efficient GPU implementations for unbiased physically based rendering methods*. The CUDA framework is used for the GPU implementation.

To justify the attempts for moving the sampling algorithm to the GPU, a hybrid architecture is investigated first, implementing the sampling algorithm on the CPU while using the GPU as a ray traversal and intersection co-processor. Results show that today's CPU's are not well suited for this architecture, making the CPU memory bandwidth a large bottleneck.

We therefore propose three streaming GPU-only rendering algorithms: a Path Tracer (PT), a BiDirectional Path Tracer (BDPT) and an Energy Redistribution Path Tracer (ERPT).

The streaming PT uses compaction to remove all terminated paths from the stream, leaving a continuous stream of active samples. New paths are regenerated in large batches at the end of the stream, thereby exploiting primary ray coherence during traversal.

A streaming BDPT is obtained by using a recursive reformulation of the Multiple Importance Sampling computations. This requires only storage for a single light and eye vertex in memory at any time during sample evaluation, making the method's

memory-footprint independent of the maximum path length and allowing high SIMT efficiency.

We propose an alternative mutation strategy for the ERPT method, further improving the convergence of the algorithm. Using this strategy, a GPU implementation of the ERPT method is presented. By sharing mutation chains between all threads within a GPU warp, efficiency and memory coherence is retained.

Finally, the performance of these methods is evaluated and it is shown that the convergence characteristics of the original methods are preserved in our GPU implementations.

Thesis Committee:

Chair:	Prof.dr.ir. F.W. JANSEN, Faculty EEMCS, TU Delft
University supervisor:	Prof.dr.ir. F.W. JANSEN, Faculty EEMCS, TU Delft
Company supervisor:	J. BIKKER, Faculty IGAD, NHTV Breda
Committee Member:	Dr. A. IOSUP , Faculty EEMCS, TU Delft
Committee Member:	Prof.dr.ir. P. DUTRÉ , Faculty CS, K.U. Leuven
Committee Member:	Prof.dr. C. WITTEVEEN , Faculty EEMCS, TU Delft

Preface

This thesis discusses the work I have done for my master project at the Delft University of Technology. The subject of this thesis has been somewhat of a lifelong obsession of mine. Triggered by animated feature films such as Pixar's *A Bug's Life* and Dreamwork's *Antz* and *Shrek*, I developed a fascination for computer graphics and photo realistic rendering in particular. My early high school attempts at building a photo realistic renderer resulted in a simplistic and terribly slow physically based renderer, taking many hours to produce a picture of a few spheres and boxes. But hé, it worked! Since then, I have come a long way (and so has the computer hardware at my disposal). In this thesis, I present my work on high performance physically based rendering on the GPU, and I am proud to say that render times have dropped from hours to seconds.

I am grateful to my supervisors, Jacco Bikker and Erik Jansen, for giving me the opportunity to work on this interesting topic and providing valuable feedback on my work. I enjoyed working with Jacco Bikker on the Brigade engine, showing me what it means to really optimize your code. I also like to thank Erik Jansen for providing me with a follow-up project on the topic of this thesis. Special thanks goes to Reinier van Antwerpen for modeling the GLASS EGG scene, used throughout this thesis. Furthermore, I thank Lianne van Antwerpen for correcting many grammatical errors in this thesis.

While studying physically based rendering, I came to appreciate the fascinating and complex light spectacles that surround us every day. This often resulted in exclamations of excitement whenever a glass of hot tea produced some interesting caustic on the kitchen table. I thank my family for patiently enduring this habit of mine. I would like to thank my parents without whom I would not have come this far. In particular, I thank my father for letting me in on the secret that is called the *c programming language*, familiarizing me with many of its intriguing semantics. My mom I thank for getting me through high school; without her persistence I would probably never have reached my full potential. Finally, I would like to thank the reader for taking an interest in my work. I hope this thesis will be an interesting read and spark new ideas and fresh thoughts by the reader.

Dietger van Antwerpen
Delft, the Netherlands
January 6, 2011

Contents

Preface iii

Contents v

List of Figures ix

List of Tables xiii

1 Introduction 1
- 1.1 Summary of Contributions . . . 3
- 1.2 Thesis Outline . . . 4

I Preliminaries 5

2 Unbiased Rendering 7
- 2.1 Rendering equation . . . 7
- 2.2 Singularities . . . 9
- 2.3 Unbiased Monte Carlo estimate . . . 10
- 2.4 Path Tracing . . . 10
- 2.5 Importance sampling . . . 12
- 2.6 Multiple Importance Sampling . . . 13
- 2.7 BiDirectional Path Tracing . . . 14
- 2.8 Metropolis sampling . . . 17
- 2.9 Metropolis Light Transport . . . 18
- 2.10 Energy Redistribution Path Tracing . . . 22

3 GPGPU 25
- 3.1 Introduction . . . 25
- 3.2 Thread Hierarchy . . . 25
- 3.3 Memory Hierarchy . . . 27

3.4 Synchronization and Communication 30
3.5 Parallel scan 30

4 Related Work 33
4.1 GPU ray tracing 33
4.2 Unbiased rendering 35

II GPU Tracers 37

5 The Problem Statement 39
5.1 Related work 40
5.2 Context 42

6 Hybrid Tracer 43
6.1 Introduction 43
6.2 Hybrid architecture 43
6.3 Results 46

7 Path Tracer (PT) 51
7.1 Introduction 51
7.2 Two-Phase PT 51
7.3 GPU PT 53
7.4 Stream compaction 54
7.5 Results 58

8 Streaming BiDirectional Path Tracer (SBDPT) 67
8.1 Introduction 67
8.2 Recursive Multiple Importance Sampling 69
8.3 SBDPT 77
8.4 GPU SBDPT 80

9 Energy Redistribution Path Tracer (ERPT) 93
9.1 Introduction 93
9.2 ERPT mutation 93
9.3 GPU ERPT 103

III Results 121

10 Comparison 123
10.1 Performance 123
10.2 Convergence 126

11 Conclusions and Future Work 129

Bibliography **131**

A Glossary **137**

B Sample probability **139**
B.1 Vertex sampling 140
B.2 Subpath sampling 141
B.3 Bidirectional sampling 141

C Camera Model **143**

D PT Algorithm **145**
D.1 Path contribution 145
D.2 MIS weights 146
D.3 Algorithm 147

E BDPT Algorithm **149**
E.1 Path contribution 149
E.2 Algorithm 151

F MLT Mutations **155**
F.1 Acceptance probability 155
F.2 Algorithm 158

List of Figures

1.1 Approximation vs physically based rendering 2

2.1 Light transport geometry 7
2.2 Measurement contribution function 9
2.3 PT sample 11
2.4 Importance Sampling 13
2.5 BDPT Sample 15
2.6 Lens mutation 20
2.7 Lai Lens mutation 21
2.8 Caustic mutation 21

3.1 CUDA thread hierarchy 26
3.2 CUDA device architecture 28
3.3 Coalesced memory access with compute capability 1.0 or 1.1 28
3.4 Coalesced memory access with compute capability 1.2 29

5.1 Test scenes 41

6.1 Sampler flowchart 44
6.2 CPU-GPU pipeline 45
6.3 CPU PT performance 46
6.4 Hybrid PT performance 47
6.5 Hybrid PT performance partition 48
6.6 Hybrid Sibenik cathedral rendering 49

7.1 Two-Phase PT flowchart 52
7.2 Rearanging rays for immediate coalesced packing 56
7.3 Sampler Stream PT Flowchart 57
7.4 SSPT streams 58
7.5 GPU PT performance 61
7.6 PT Extension ray traversal performance 62

7.7 TPPT time partition 62
7.8 SSPT time partition 63
7.9 TPPT performance vs. stream size 64
7.10 SSPT performance vs. stream size 64
7.11 GPU PT conference room rendering 65

8.1 MIS denominator 70
8.2 SBDPT sample 77
8.3 SBDPT flowchart 81
8.4 Collapsed SBDPT flowchart 82
8.5 SBDPT vertex memory access 83
8.6 SBDPT performance with stream compaction 88
8.7 SBDPT time partition 88
8.8 SSPT vs SBDPT 90
8.9 Contribution of bidirectional strategies 91

9.1 Feature shapes 97
9.2 High correlation in feature 98
9.3 Local feature optimum 99
9.4 Low energy feature 99
9.5 Improved mutation strategy 102
9.6 ERPT flowchart 105
9.7 ERPT path vertex access 108
9.8 ERPT progress estimation 112
9.9 ERPT performance 115
9.10 ERPT time partition 116
9.11 Complex caustic with ERPT and SBDPT 117
9.12 ERPT and SSPT comparison 118
9.13 Invisible date with ERPT, SBDPT and SSPT 119
9.14 Stanford scene with ERPT and SBDPT 120

10.1 Iteration performance comparison 124
10.2 Traversal performance comparison 125
10.3 Sponza with ERPT, SBDPT and SSPT 126
10.4 Glass Egg with ERPT, SBDPT and SSPT 127
10.5 Invisible Date with ERPT, SBDPT and SSPT 128

B.1 Conversion between unit projected solid angle and unit area 139
B.2 Vertex sampling 140
B.3 Bidirectional Sampled Path 142

C.1 Camera model 144

D.1 Implicit path contribution 146
D.2 Explicit path contribution 146

E.1 One eye subpath contribution 151
E.2 Bidirectional path contribition 151

F.1 Acceptance probability for partial lens mutation 157
F.2 Acceptance probability for full lens mutation 157
F.3 Acceptance probability for partial caustic mutation 157
F.4 Acceptance probability for full caustic mutation 158

List of Tables

5.1 Test scenes 42

7.1 TPPT SIMT efficiency 59
7.2 SSPT SIMT efficiency 60
7.3 GPU PT memory usage 63

8.1 SBDPT SIMT efficiency 87
8.2 SBDPT memory usage 89

9.1 Average rays per mutation 113
9.2 Average mutation acceptance probability 113
9.3 ERPT path tracing SIMT efficiency 114
9.4 ERPT energy redistribution SIMT efficiency 114
9.5 ERPT memory usage 116

List of Algorithms

1 Metropolis Light Transport 19
2 Energy Redistribution 23
3 Parallel Scan 31
4 Mutate 106
5 PT 148
6 SampleEyePath 152
7 SampleLightPath 153
8 Connect 154
9 CausticMutation 159
10 LensMutation 160

Chapter 1

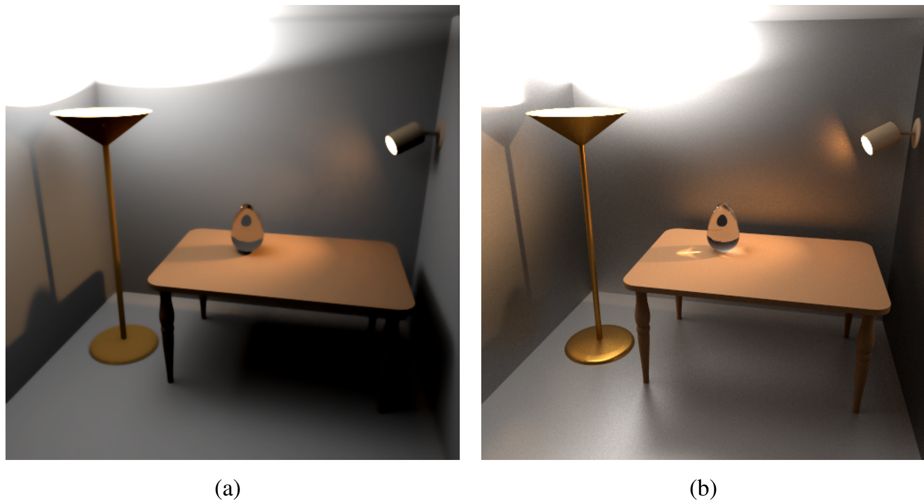
Introduction

Since the introduction of computer graphics, synthesizing computer images has found many useful applications. Of these, the best known are probably those in the entertainment industries, where image synthesis is used to create visually compelling computer games and to add special effects to movies. Besides these and many others, important applications are found in advertisement, medicine, architecture and product design. In many of these applications, image synthesis is used to give a realistic impression of the illumination in virtual 3D scenes. For example, in architecture, CG is used to answer questions about the illumination in a designed building [15]. Such information is used to evaluate the design before realizing the building. In advertising and film production, computer graphics is used to combine virtual 3D models with video recordings to visualize scenes that would otherwise be impossible or prohibitively expensive to capture on film. Both these applications require a high level of realism, giving an accurate prediction of the appearance of the scene and making the synthesized images indistinguishable from real photos of similar scenes in the real world.

Physically based rendering algorithms use mathematical models of physical light transport for image synthesis and are capable of accurately rendering such realistic images. For most practical applications, physically based rendering algorithms require a lot of computational power to produce an accurate result. Therefore, these algorithms are mostly used for off-line rendering, often on large computer clusters. This lack of immediate feedback complicates the work for the designer of virtual environments.

To get some immediate feedback, the designer usually resorts to approximate solutions [6, 32, 36, 55]. Although interactive, these approximations sacrifice accuracy for performance. Complex illumination effects such as indirect light and caustics are roughly approximated or completely absent in the approximation. Figure 1.1 shows the difference between an approximation[1] and a physically based rendering. Although the approximation looks plausible, it is far from physically accurate. The lack of accuracy reduces its usefulness to the designer. Designers would benefit greatly from interactive or near-interactive physically based rendering algorithms on a single workstation for fast and accurate feedback on their

[1]This approximation algorithm simulates a single light indirection and supports only perfect specular and perfect diffuse materials.



(a) (b)

Figure 1.1: Rendering of the GLASS EGG scene with (a) an approximation algorithm and (b) a physically based algorithm.

designs.

The speed of physically based rendering can be improved by either using more advanced algorithms, or by optimizing the implementation of these algorithms. Since the advent of physically based rendering, several advanced physically based rendering algorithms, capable of rendering very complex scenes, have been developed and implemented on the CPU. At the same time, many fast approximation algorithms have been developed for the GPU. Because these algorithms are used by most modeling software packages, modeling workstations usually contain one or more powerful GPUs. Nowadays, these GPUs often provide more raw processing power and memory bandwidth than the workstations main processor and system memory [13]. Therefore, the performance of a physically based rendering algorithm could benefit greatly from utilizing the available GPUs.

The goal of this thesis will be to find efficient GPU implementations for physically based rendering algorithms. Parts of these algorithms, such as ray-scene intersection tests, have already been efficiently implemented on the GPU, significantly outperforming similar CPU implementations. Furthermore, some simple rendering algorithms have been fully implemented on the GPU. However, little effort has been made to implement more advanced physically based rendering algorithms on the GPU. Implementing these algorithms on the GPU will allow for a wider range of scenes to be rendered accurately and at high speed on a single workstation. In the following sections, we will give a brief summary of our contributions and give a short overview of the thesis outline.

1.1 Summary of Contributions

In this thesis we will present the following contributions:

- We present a general framework for hybrid physically based rendering algorithms where ray traversal is performed on the GPU while shading and ray generation are performed on the CPU.

- We show that on modern workstations the performance of a hybrid architecture is limited by the memory bandwidth of system memory. This reduces scalability and prevents the full utilization of the GPU, justifying our further attempts to fully implement the rendering algorithms on the GPU.

- We present an alternative method for computing Multiple Importance Sampling weights for BDPT. By recursively computing two extra quantities during sample construction, we show that the weights can be computed using a fixed amount of computations and data per connection, independent of the sample path lengths. This allows for efficient data parallel weight computation on the GPU.

- We introduce the notion of mutation features and use these to investigate the noise occurring in ERPT. We show that structural noise patterns in ERPT can be understood in the context of mutation features.

- We propose an alternative mutation strategy for the ERPT method. We show that by increasing mutation feature size, our mutation strategy trades structural noise for more uniform noise. This allows for longer mutation chains, required for an efficient GPU implementation.

- We present streaming GPU implementations of three well known, physically based rendering algorithms: Path Tracing (PT), BiDirectional Path Tracing (BDPT) and Energy Redistribution Path Tracing (ERPT). We show how our implementations achieve high degrees of coarse grained and fine grained parallelism, required to fully utilize the GPU. Furthermore, we discuss how memory access patterns are adapted to allow for high effective memory bandwidth on the GPU.

  - We show how to increase the ray traversal performance of GPU samplers by immediately packing output rays in a compact output stream.
  - Furthermore, we show that immediate sampler stream compaction can be used to speed up PT and increase ray traversal performance further by exploiting primary ray coherence.
  - We present a streaming adaption of the BDPT algorithm, called SBDPT. We show how our recursive computation of MIS weights allows us to only store a single light and eye vertex in memory at any time during sample evaluation, making the methods memory footprint independent of the path length and allowing for high GPU efficiency.

  - By generating mutation chains in batches of 32, our ERPT implementation realizes high GPU efficiency and effective memory bandwidth. All mutation chains in a batch follow the same code path, resulting in high fine grained parallelism.

- We compare the three implementations in terms of performance and convergence speed. We show that the convergence characteristics of the three rendering algorithms are preserved in our adapted GPU implementations. Furthermore, we show that the performance of the algorithms are all in the same order of magnitude.

1.2 Thesis Outline

The thesis is divided in three parts. Part I contains background information on the subject of unbiased rendering and GPGPU programming. In chapter 2, we will give a short introduction to physically based rendering. In this discussion, we will emphasize the use of importance sampling to improve physically based rendering algorithms and show how this naturally leads to the three well known rendering algorithms PT, BDPT and ERPT. This chapter will introduce all related notation and terminology we will be using in the rest of this thesis. Next, in chapter 3, we give an overview of the General Purpose GPU architecture in the context of CUDA: the parallel computing architecture developed by NVIDIA [13]. In our discussion we will focus mainly on how to achieve high parallelism and memory bandwidth, which translates to high overall performance. Finally, we will discuss the parallel scan primitive. In chapter 4 we discuss related work on ray tracing and unbiased rendering using the GPU. We will further discuss some variations on the unbiased rendering algorithms not mentioned in chapter 2.

In part II we present the main contributions of this thesis. We start with a problem statement in chapter 5. In chapter 6, we investigate a hybrid architecture where the rendering algorithm is implemented on the CPU, using the GPU for ray traversal and intersection only. We will show that the CPU side easily becomes the bottleneck, limiting the performance gained by using a GPU. Following, in chapter 7, we present our GPU Path Tracer. First, we present the Two-Phase PT implementation which forms a starting point for the BDPT and EPRT implementations in later chapters. We will further show how to improve the performance of this PT through stream compaction, resulting in a highly optimized implementation. After that, we present our SBDPT algorithm in chapter 8. We first present a recursive formulation for Multiple Importance Sampling and show how this formulation is used in the SBDPT algorithm, allowing for an efficient GPU implementation. Finally, in chapter 9, we study the characteristics of the ERPT mutation strategy using the concept of mutation features. We will then propose an improved mutation strategy, which we use in our ERPT implementation. Next, we present our streaming ERPT algorithm and its GPU implementation.

Finally, in part III, we discuss our findings. We first compare the performance and convergence characteristics of the three GPU implementations in chapter 10. We conclude this thesis in chapter 11, with a discussion and some ideas for future work.

Lastly, the appendix provides a small glossary of frequently used terms and abbreviations and further details on the PT, BDPT and ERPT algorithms.

Part I

Preliminaries

Chapter 2

Unbiased Rendering

In this chapter, we will give a short introduction to physically based rendering. In this discussion, we will emphasize the use of importance sampling to improve physically based rendering algorithms and show how this naturely leads to the three well known rendering algorithms: Path Tracing, BiDirectional Path Tracing and Energy Redistribution Path Tracing. This chapter will also introduce all related notation and terminology we will be using in the rest of this thesis. For a more thorough discussion on most of the topics encountered in this chapter, we redirect the reader to [16]. For an extensive compendium of useful formulas and equations related to unbiased physically based rendering. we refer to [17, 2].

2.1 Rendering equation

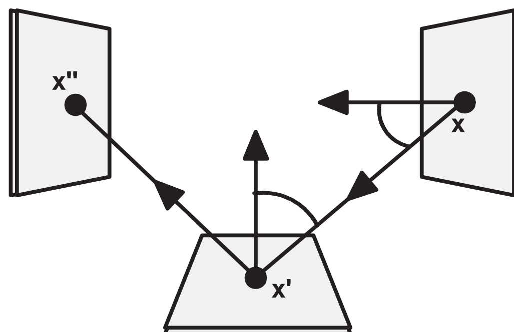


Figure 2.1: Geometry for the light transport equation.

Physical based light transport in free space (vacuum) is modeled by the light transport equation. This equation describes how radiance arriving at surface point x'' from surface point x' relates to radiance arriving at x' (figure 2.1). Because light transport is usually perceived in equilibrium, the equation describes the equilibrium state:

$$L\left(x' \to x''\right) = L_e\left(x' \to x''\right) + \int_M L\left(x \to x'\right) f_s\left(x \to x' \to x''\right) G\left(x \leftrightarrow x'\right) dA_M\left(x\right) \tag{2.1}$$

In this equation, $L(x' \to x'')$ is the radiance arriving at surface point x'' from surface point x' due to all incoming radiance at x' from all surface points M. M is the union of all scene surfaces. $L_e(x' \to x'')$ is the radiance emitted from x' arriving at x''. $f_s(x \to x' \to x'')$ is the *Bidirectional Scattering Distribution Function* (BSDF), used to model the local surface material. It describes the fraction of radiance arriving at x' from x that is scattered towards x''. Finally, $G(x \leftrightarrow x')$ is the geometric term to convert from unit projected solid angle to unit surface area.

$$G\left(x \leftrightarrow x'\right) = V\left(x \leftrightarrow x'\right) \frac{|\cos(\theta_o)\cos(\theta'_i)|}{||x - x'||^2} \tag{2.2}$$

In this term, $V(x \leftrightarrow x')$ is the visibility term, which is 1 iff the two surface points are visible from one another and 0 otherwise. θ'_i and θ_o are the angles between the local surface normals and respectively the incoming and outgoing light flow.

The light transport equation is usually expressed as an integral over unit projected solid angles instead of surface area, by omitting the geometric factor and integrating over the projected unit hemisphere. We will however only concern ourselves with the surface area formulation.

In the context of rendering, the transport equation is often called the rendering equation. When rendering an image, each image pixel j is modeled as a radiance sensor. Each sensor measures radiance over surface area and has a sensor sensitivity function $W_j(x \to x')$, describing the sensor's sensitivity to radiance arriving at x' from x. Let I be the surface area of the image plane, then the measurement equation for pixel j equals:

$$I_j = \int_{I \times M} W_j\left(x \to x'\right) L\left(x \to x'\right) G\left(x \leftrightarrow x'\right) dA_I\left(x'\right) dA_M(x) \tag{2.3}$$

In this thesis, we used a sensor sensitivity function corresponding to a simple finite aperture camera model. For more details on the finite aperture camera model, see appendix C.

Note that the expression for radiance L is recursive. Each recursion represents the scattering of light at a surface point. A sequence of scattering events represents a light transport path from a light source to the pixel sensor. The measured radiance leaving a light source and arriving at the image plane along some light transport path $x_0...x_k \in I \times M^k$, can be expressed as:

$$\begin{aligned} f_j(x_0 \cdots x_k) = & L_e(x_k \to x_{k-1})\, G(x_k \to x_{k-1}) \\ & \prod_{i=1}^{k-1} f_s(x_{i+1} \to x_i \to x_{i-1})\, G(x_i \leftrightarrow x_{i-1}) \cdot \\ & W_j(x_1 \to x_0) \end{aligned} \tag{2.4}$$

This function is called the *measurement contribution function*. The surface points $x_0 \cdots x_k$ are called the path vertices and form a light transport path $\mathbf{X}$ of length k (see figure 2.2). Note that the vertices on the path are arranged in reverse order, with x_0 on the image plane and x_k on the light source. This is because constructing a path in reverse order, from eye to light source, is often more convenient. The recursion in the measurement equation is

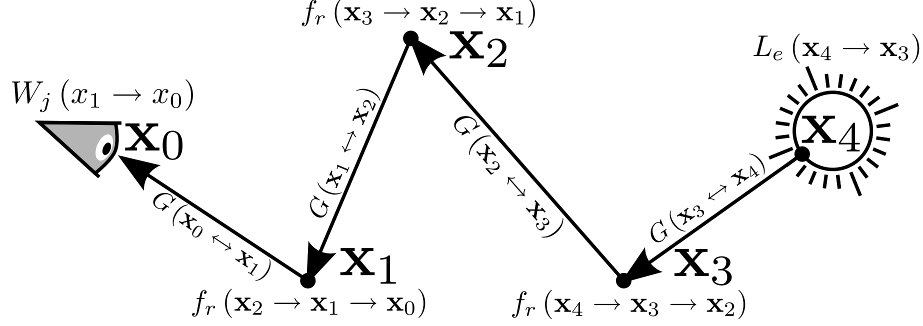


Figure 2.2: Measurement contribution function for an example path.

removed by expanding the equation to

$$I_j = \sum_{k=1}^{\infty} \int_{I \times M^k} f_j(x_0 \cdots x_k)\, dA_I(x_0) dA_M(x_1) \cdots dA_M(x_k) \tag{2.5}$$

The set of all light transport paths of length k is defined as $\Omega_k = I \times M^k$. This gives rise to a unified path space, defined as $\Omega = \bigcup_{i=1}^{\infty} \Omega_i$ and a corresponding product measure $d\Omega(\mathbf{X}_k) = dA_I(x_0) \times dA_M(x_1) \times \cdots \times dA_M(x_k)$ on path space. The measurement integral can be expressed as an integral of the measurement contribution function over this unified path space, resulting in

$$I_j = \int_{\Omega} f_j(\mathbf{X}) d\Omega(\mathbf{X}) \tag{2.6}$$

In this equation, the measurement contribution function f_j describes the amount of energy per unit path space as measured by sensor j. Physical based rendering concerns itself with estimating this equation for all image pixels.

2.2 Singularities

Many of the aforementioned functions may contain singularities. For example, the BSDF of perfect reflective and refractive materials is modeled using Dirac functions. Furthermore, L_e may exhibit singularities to model purely directional light sources. Finally, common camera models such as the pinhole camera are modeled using Dirac functions in the sensor sensitivity functions W_j. Special care must be taken to support these singularities. In this work, we assume that the camera model contains a singularity and that BSDF's may exhibit singularities. However, unless explicitly addressed, we assume that light sources do *not* exhibit singularities.

Heckbert introduced a notation to classify light transport paths using regular expressions of the form $E(S|D)^* L$ [24]. Each symbol represents a path vertex. E represents the eye and is the first path vertex. This vertex is followed by any number of path vertices, classified as specular(S) or diffuse(D) vertices. A path vertex is said to be *specular* whenever the

scattering event follows a singularity in the local BSDF, also called a specular bounce. Finally, the last vertex is classified as a light source(L). We refer to the classification of a path as its signature.

2.3 Unbiased Monte Carlo estimate

The measurement equation contains an integral of a possibly non-continuous function over the multidimensional path space. Therefore, the measurement equation is usually estimated using the Monte Carlo method. The Monte Carlo method uses random sampling to estimate an integral. The idea is to sample N points $X_1 \cdots X_N$ according to some convenient probability density function $p : \Omega \to \mathbb{R}$. Then, the integral $I = \int_\Omega f(x)\mu(x)$ is estimated as follows

$$I = E\left[F_N\right] \approx F_N = \frac{1}{N}\sum_{i=1}^{N}\frac{f(\mathbf{X}_i)}{p(\mathbf{X}_i)} \tag{2.7}$$

This estimator F_N is called unbiased because its expected value $E\left[F_N\right]$ equals the desired outcome I, independent of the number of samples N. Note that a biased estimator may still be consistent if it satisfies $\lim_{N\to\infty} F_N = I$, that is, the bias goes to zero as the number of samples goes to infinity.

In the context of rendering, the N samples used in the estimate are random light transport paths from unified path space. These paths are generated by a sampler. Possible samplers are the Path Tracing sampler(PT), BiDirectional Path Tracing sampler(BDPT) or Energy Redistribution Path Tracing(ERPT) sampler, discussed in later sections. The different samplers generate paths according to different probability distributions.

Generating a path is usually computationally expensive because it requires the tracing of multiple rays through the scene. Although a sampler may generate one independent path at a time, it is often computationally beneficial to generate collections of correlated paths. From now on, when referring to a sample generated by some sampler, we mean such a collection of paths generated by the sampler. When generating collections of paths, special care must be taken to compute the probability with which each path is generated by the sampler. A simple solution is to partition path space, allowing a sampler to generate at most one path per part within each sample; for example by using the partition $\Omega = \bigcup_{i=1}^{\infty} \Omega_i$ and allowing at most one path per collection for each path length. This often simplifies computations considerably. In the next section, this method is used in the PT sampler.

2.4 Path Tracing

In path tracing, a collection of paths is sampled backward, starting at the eye. Figure 2.3 shows a single path tracing sample. A sampler starts by tracing a path $y_0 \cdots y_k$ backwards from the eye, through the image plane into the scene. The path is repeatedly *extended* with another path vertex until it is *terminated*. At each path vertex, the path is terminated with a certain probability using *Russian roulette*. Each path vertex y_i is explicitly *connected* to a random point z on a light source to form the complete light transport path $y_0 \cdots y_i z$ from the

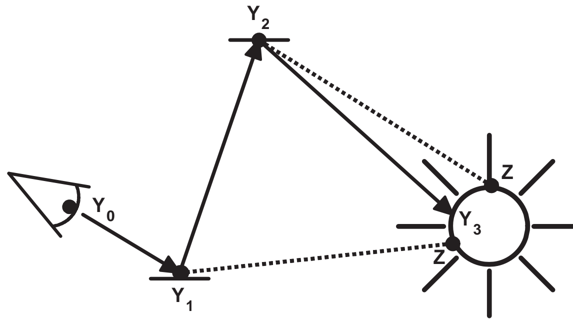


Figure 2.3: Path tracing sample.

eye to the light source. These complete paths are called *explicit* paths. However, when the path 'accidentally' hits a light source at vertex y_k, this path $y_0 \cdots y_k$ also forms a valid light transport path. Such paths are called *implicit* paths. Hence, light transport paths may be generated as either an explicit or implicit path. For simplicity, we assume that light sources do not reflect any light, so the path may be terminated whenever it hits a light source.

A path vertex must be diffuse for an explicit connection to make sense, because otherwise the BSDF will be non-zero with zero probability. Therefore, path space is partitioned in explicit and implicit paths, where explicit paths must be of the form $E\,(S|D)^*\,DL$ while implicit paths must be of the form $E\,\big(L|\,(S|D)^*\,SL\big)$. All other explicit and implicit paths are discarded. Note that a PT sample contains at most one explicit and one implicit path for each path length. Hence, by further partitioning path space according to path length, it is guaranteed that the sampler generates at most one valid light transport path per part.

When the PT sampler extends the partial path $y_0 \cdots y_i$ with another vertex y_{i+1}, an outgoing direction Ψ_i at y_i is sampled per projected solid angle and the corresponding ray is intersected with the scene to find the next path vertex y_{i+1}. Hence, path vertices are sampled per projected solid angle. To compute the sampling probability per unit area, the probability is converted using the geometric term:

$$P_A\left(y_i \rightarrow y_{i+1}\right) = P_{\sigma^\perp}\left(y_i \rightarrow y_{i+1}\right) G\left(y_i \leftrightarrow y_{i+1}\right) \tag{2.8}$$

For more details on converting probabilities per unit projected solid angle to probabilities per unit area and back, see appendix B. Note that in practice, $P_{\sigma^\perp}\left(y_i \rightarrow y_{i+1}\right)$ usually also depends on the incoming direction. To emphasize this, one could instead write $P_{\sigma^\perp}\left(y_{i-1} \rightarrow y_i \rightarrow y_{i+1}\right)$. To keep things simple, we omitted this. It is however important to keep in mind that the incoming direction is usually needed to calculate this probability. We further assume that the inverse termination probability for Russian roulette is incorporated in the probability $P_A\left(y_i \rightarrow y_{i+1}\right)$ of sampling y_{i+1}. Because the point z on the light source and y_0 on the eye are directly sampled per unit area, no further probability conversions are required for these vertices. The probability that a sampler generates some explicit,

respectively implicit, path as part of a PT sample equals

$$\begin{aligned}P(y_0 \cdots y_i z) &= P_A(y_0) \prod_{j=0}^{i-1} P_A(y_j \to y_{j+1}) P_A(z) \\ P(y_0 \cdots y_k) &= P_A(y_0) \prod_{i=0}^{k-1} P_A(y_i \to y_{i+1})\end{aligned} \tag{2.9}$$

For more details on generating and evaluating path tracing samples, see appendix D.

2.5 Importance sampling

A sampler may sample light transport paths according to any convenient probability density function p, provided that $\frac{f(\mathbf{X})}{p(\mathbf{X})}$ is finite whenever $f(\mathbf{X}) > 0$. However, the probability density function greatly influences the quality of the estimator. A good measure for quality is the variance of an estimator, where smaller variance results in a better estimator. To reduce variance in the estimator, p should resemble f as much as possible. To see why this is true, assume that p equals f up to some normalization constant c, so that $\frac{f(\mathbf{X})}{p(\mathbf{X})} = c$ for all $\mathbf{X} \in \Omega$. Then the estimator reduces to

$$F_1 = E\left[\frac{f(\mathbf{X}_1)}{p(\mathbf{X}_1)}\right] = E[c] = c \tag{2.10}$$

Hence, the estimator will have zero variance.

Sampling proportional to f is called importance sampling. In general it is not possible to sample proportional to f without knowing f beforehand [1]. It is however possible to reduce the variance of the estimator by sampling according to an approximation of f. This is usually done locally during path generation. At each path vertex x_i, the importance of the next path vertex x_{i+1} is proportional to the reflected radiance $L(x_{i+1} \to x_i) f_s(x_{i+1} \to x_i \to x_{i-1}) G(x_i \leftrightarrow x_{i+1})$. This leads to two importance sampling strategies:

1. Try to locally estimate $L(x_{i+1} \to x_i)$ or $L(x_{i+1} \to x_i) G(x_i \leftrightarrow x_{i+1})$ and sample x_{i+1} accordingly

2. Sample x_{i+1} proportional to $f_s(x_{i+1} \to x_i \to x_{i-1}) G(x_i \leftrightarrow x_{i+1})$

The strategies are visualized in figure 2.4. The gray shape represents the local BSDF. In the left image, the outgoing direction is sampled proportional to the BSDF, while in the right image, the outgoing direction is sampled according to the estimated incoming radiance L. Both strategies are used in the PT sampler from last section. Most radiance is expected to come directly from light sources, so explicit connections are a form of importance sampling according to L. The second method of importance sampling is used during path extension, where the outgoing direction is usually chosen proportional to f_s. Note that these strategies

[1] In section 2.8, the Metropolis-Hastings algorithm is used to generate a sequence of correlated samples proportional to f.

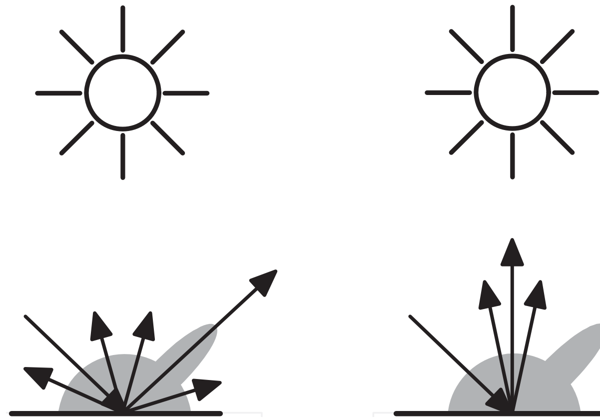


Figure 2.4: Importance sampling along to the local BSDF and towards the light source.

are very rough, as explicit connections only account for direct light while extensions only account for local reflection.

Importance sampling is also combined with Russian roulette to terminate paths. During sampling, the termination probability at path vertex x_i is chosen proportional to the estimated amount of incoming radiance that is absorbed. Usually, the absorption estimate is based on the local BSDF.

Another related requirement for a good probability density function is that for any path $\mathbf{X} \in \Omega$ with $p(\mathbf{X}) > 0$, $\frac{f(\mathbf{X})}{p(\mathbf{X})}$ is bounded. If this property is not satisfied due to singularities in $f(\mathbf{X})$, the probability of sampling these singularities will be infinitesimally small. Hence, these singularities will in practice not contribute to the estimate, causing the absence of important light effects in the final rendering [2]. For a PT sampler, it is enough to require that during path extension at any vertex x_i, it holds that $\frac{f_s(x_{i+1} \to x_i \to x_{i-1})}{P_{\sigma^\perp}(x_i \to x_{i+1})} \in [0, \infty)$. As singularities in the local BSDF are usually modeled using Dirac delta functions, this practically means that $P_{\sigma^\perp}$ must contain Dirac delta functions centered at the same outgoing directions as the Dirac delta functions in the local BSDF.

When performing importance sampling according to f_s, this restriction is already satisfied. Hence, singularities are a special case where importance sampling is not only desirable, but required to capture all light effects.

2.6 Multiple Importance Sampling

The PT sampler from last sections uses partitioning to select between two importance sampling strategies for sampling the last path vertex. On paths with signature $E(S|D)^*DL$, the last path vertex is sampled by making an explicit connection to the light source, assuming that the light source geometry is a good approximation for incoming radiance. For all other paths, the last path vertex is sampled according to the local BSDF, resulting in an implicit

[2] Light effects due to singularities are those effects caused by perfect reflection and refraction such as caustics.

light transport path. However, it is not always evident which importance sampling strategy is most optimal in a certain situation. For highly glossy materials, sampling the last vertex according to the local BSDF often results in less variance than when making an explicit connection. This is because the glossy BSDF is highly irregular, but explicit connections do not take the shape of the BSDF in consideration.

By dropping the path space partition restriction, multiple sampling techniques may be combined to form more robust samplers. Multiple Importance Sampling (MIS) seeks to combine different importance sampling strategies in one optimal but unbiased estimate [51]. A sampler may generate paths according to one of n sampling strategies, where $p_i(\mathbf{X})$ is the probability of sampling path $\mathbf{X}$ using sampling strategy i. Per sampling strategy i, a sampler generates n_i paths $\mathbf{X}_{i,1} \cdots \mathbf{X}_{i,n_i}$. Hence, path $\mathbf{X}_{i,j}$ is sampled with probability $p_i(\mathbf{X}_{i,j})$. These samples are then combined into a single estimate using

$$I_k = \sum_{i=1}^{n} \frac{1}{n_i} \sum_{j=1}^{n_i} w_i(\mathbf{X}_{i,j}) \frac{f(\mathbf{X}_{i,j})}{p_i(\mathbf{X}_{i,j})} \tag{2.11}$$

In this formulation, w_i is a weight factor used in the combination. For this estimate to be unbiased, it is enough that for each path $\mathbf{X} \in \Omega$ with $f(\mathbf{X}) > 0$, $p_i(\mathbf{X}) > 0$ whenever $w_i(\mathbf{X}) > 0$. Furthermore, whenever $f(\mathbf{X}) > 0$, the weight function must satisfy

$$\sum_{i=1}^{n} w_i(\mathbf{X}) = 1 \tag{2.12}$$

In other words, there must be at least one strategy to sample each contributing path and when some path may be sampled with multiple strategies, the weights for these strategies must sum to one. A fairly straightforward weight function satisfying these conditions is

$$w_i(\mathbf{X}) = \frac{n_i p_i(\mathbf{X})}{\sum_{j=1}^{n} n_j p_j(\mathbf{X})} \tag{2.13}$$

This weight function is called the *balance heuristic*. Veach showed that the balance heuristic is the best possible combination in the absence of further information [50]. In particular, they proved that no other combination strategy can significantly improve over the balance heuristic. The general form of the balance heuristic, called the *power heuristic*, is given by

$$w_i(\mathbf{X}) = \frac{n_i p_i(\mathbf{X})^{\beta}}{\sum_{j=1}^{n} n_j p_j(\mathbf{X})^{\beta}} \tag{2.14}$$

2.7 BiDirectional Path Tracing

In the PT sampler, all but the last path vertex are sampled by tracing a path backwards from the eye into the scene. This is not always the most effective sampling strategy. In scenes with mostly indirect light, it is often hard to find valid paths by sampling backwards from the eye. Sampling a part of the path forward, starting at a light source and tracing forward into the scene, can solve this problem.

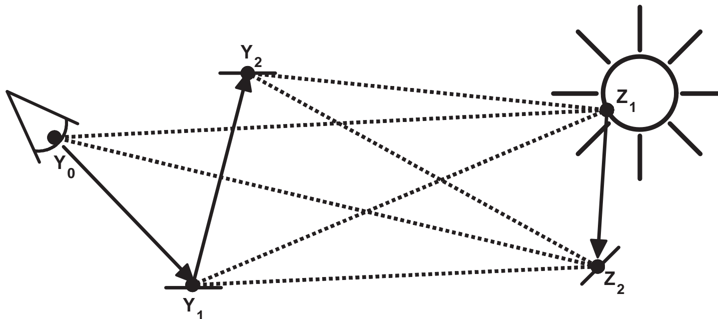


Figure 2.5: Bidirectional path tracing sample.

This is exactly what the BiDirectional Path Tracing (BDPT) sampler does. BDPT was independently developed by Veach and Lafortune [50, 33]. It samples an eye path and a light path and connects these to form complete light transport paths. The eye path starts at the eye and is traced backwards into the scene, like in the PT sampler. The light path starts at a light source and is traced forward into the scene. Connecting the endpoints of any eye and light path using an explicit connection results in a complete light transport path from light source to eye.

Like the PT sampler, for efficiency, instead of generating a single path per sample, the BDPT sampler generates a collection of correlated paths per sample. Figure 2.5 shows a complete BDPT sample. When constructing a sample, an eye path and a light path are sampled independently, wherafter all the vertices of the eye path are explicitly connected to all the vertices of the light path to construct valid light transport paths. Each connection results in a light transport path.

Let $\mathbf{X} \in \Omega$ be a light transport path of length k with vertices $x_0 \cdots x_k$, that is part of a BDPT sample. This path can be constructed using $k+1$ different bidirectional sampling strategies by connecting an eye path $\mathbf{Y}^s = y_0 \cdots y_s$ of length $s \geq 0$ with a light path $\mathbf{Z}^t = z_1 \dots z_t$ of length $t \geq 0$, where $k = s + t$ ($y_i = x_i$ and $z_i = x_{k-i+1}$). Note that both the eye and light path may be of length zero. In case of a zero length light path, the eye path is an implicit path and should end at a light source. In case of an eye path of length 0, the light path is directly connected to the eye. We will not deal with light paths directly hitting the eye[3]. For convenience, we will leave the first vertex y_0 on the eye path implicit in the rest of our discussion of BDPT, as it is always sampled as part of the eye path anyway. Hence, an eye path of length s contains the vertices $\mathbf{Y}^s = y_1 \cdots y_s$. When necessary, we will refer to a complete light transport path sampled using an eye path of length s as $\mathbf{X}_s$.

The probability of generating a (partial) path $\mathbf{X}'$ using either forward or backward tracing equals $p(\mathbf{X}')$. The probability of bidirectionally generating a complete path $\mathbf{X}$ by connecting an eye path of length s with a light path of length $t = k - s$ equals $\hat{p}_s(\mathbf{X})$.

Each bidirectional sampling strategy represents a different importance sampling strategy and has a different probability distribution over path space. Hence, by combining all

[3]A light path can only direclty hit the eye when a camera model with finite aperture lens is used. The contribution of such paths is usually insignificant.

sampled paths using MIS, the strengths of these different sampling strategies are combined in a single unbiased estimator. By incorporating the termination probabilities into the path sampling probabilities $p(\mathbf{X}')$ and $\hat{p}_s(\mathbf{X})$ (similar to the PT sampler), the number of samples per strategy n_i becomes 1. Applying the power heuristic to the BDPT sampler results in a MIS weight function of

$$w_s(\mathbf{X}) = \frac{\hat{p}_s(\mathbf{X})^\beta}{\sum_{i=0}^{k} \hat{p}_i(\mathbf{X})^\beta} \tag{2.15}$$

We will refer to the denominator of this equation as $D(\mathbf{X}) = \sum_{i=0}^{k} \hat{p}_i(\mathbf{X})^\beta$.

Now let us turn to the computation of the probability $\hat{p}_i(\mathbf{X})$ for any $0 \leq i \leq k$. This probability equals the probability of sampling all the individual vertices on the path. Each vertex is either sampled as part of the eye path or the light path. In this case, the first i vertices are sampled as part of the eye path $\mathbf{Y}^s$ $(s = i)$. The remaining $t = k - s$ vertices are sampled as part of the light path $\mathbf{Z}^t$. Therefore, the bidirectional sampling probability can be expressed as the product of the sampling probabilities for the eye and light path separately: $\hat{p}_i(\mathbf{X}) = p(\mathbf{Y}^s)\, p(\mathbf{Z}^t)$.

What is left is the computation of the sampling probabilities for the eye and light path. These probabilities are similar to the sampling probability for paths in the PT sampler.

For some eye or light path $\mathbf{X}' = x_1 \cdots x_k$, the sampling probability $p(\mathbf{X}')$ equals

$$p(\mathbf{X}') = \prod_{i=0}^{k-1} P_A(x_i \rightarrow x_{i+1}) \tag{2.16}$$

In this equation, we use the special notation $P_A(x_0 \rightarrow x_1)$ to indicate the probability of sampling x_1 per unit area. Using this special case keeps the overall notation clean. On an eye path, $P_A(x_0 \rightarrow x_1)$ is the probability of sampling the first eye vertex x_1, depending on the lens model used. On a light path, $P_A(x_0 \rightarrow x_1)$ is simply the probability of sampling the vertex x_1 on a light source. In this case, x_0 can be thought of as the virtual source of all light in the scene.

Remember that the eye vertex y_0 was left implicit. So, the real probability of sampling $\mathbf{X}_s$ with bidirectional sampling strategy s equals $P_A(y_0)\,\hat{p}_i(\mathbf{X})$. The MIS weights do not change due to y_0 as $P_A(y_0)^\beta$ will appear in all probability terms and hence will be a common factor in both the denominator and numerator of $w_s(\mathbf{X})$.

BDPT requires many correlated connection rays to be traced. It is possible to reduce the number of connection rays by applying Russian roulette to the connections. Instead of making all connections, each connection is made with a certain probability, based on the contribution this connection would make to the estimator if it would succeed. By correcting for this probability, the estimate remains unbiased. Note that this optimization is also a form of importance sampling; Connections are sampled according to their estimated contribution. The estimate is based on the assumption that the connection will actually succeed.

For more details on generating and evaluating BDPT samples, see appendix E.

2.8 Metropolis sampling

As explained in section 2.5, perfect importance sampling requires sampling proportional to the function f. We stated that this was not generally possible without knowing f beforehand. However, in the context of unbiased rendering, f is known beforehand but because it is so complex and irregular, it is hard to sample proportional to f. The *Metropolis-Hastings* algorithm is capable of sampling proportional to any function as long as it can be evaluated. The algorithm generates a sequence of samples $X_0 \cdots X_i \cdots$ with $p_i : \Omega \to \mathbb{R}$ being the probability density function for X_i. The sequence is constructed as a Markov chain, so each sample only depends on its predecessor in the sequence. X_{i+1} is constructed from X_i by randomly mutating X_i according to some *mutation strategy*. The algorithm may use any convenient mutation strategy, as long as it satisfies ergodicity. Some mutation strategies are however more effective than others. In section 2.9, we will discuss specific mutation strategies in the context of light transport. We will discuss ergodicity shortly. A mutation strategy is described by its *tentative transition function* $T(y|x)$, which is the probability density function for constructing y as a mutation of x.

The desired sample distribution is obtained by accepting or rejecting proposed mutations according to a carefully chosen acceptance probability. If a mutation is rejected the next sample will remain the same as the current sample ($X_{i+1} = X_i$). Let $a(y|x)$ be the acceptance probability for accepting mutation y as X_{i+1}, given $X_i = x$. The acceptance probability is chosen so that when $p_i \propto f$, so will p_{i+1}. Hence, the equilibrium distribution for the sample distribution sequence $p_0 \cdots p_i \cdots$ is proportional to f. This is achieved by letting $a(y|x)$ satisfy the *detailed balance* condition

$$f(y)T(x|y)a(x|y) = f(x)T(y|x)a(y|x) \tag{2.17}$$

When the acceptance probability satisfies the detailed balance and the mutation strategy satisfies ergodicity, the probability density sequence will converge to the desired equilibrium distribution. In order to reach the equilibrium distribution as quickly as possible, the best strategy is to make the acceptance probability as large as possible. This results in the following acceptance probability:

$$a(x|y) = \min\left(1, \frac{f(x)T(y|x)}{f(y)T(x|y)}\right) \tag{2.18}$$

As mentioned earlier, ergodicity must be satisfied in order for the sequence to reach the equilibrium distribution. Ergodicity means that the sequence converges to the same distribution, no matter how X_0 was chosen. In practice, it is sufficient that $T(y|x) > 0$ for any $x, y \in \Omega$ with $f(x) > 0$ and $f(y) > 0$. In other words, all paths are reachable from all other paths through a single mutation. This is to prevent the sequence from getting stuck in a part of path space, unable to reach another part.

The sample sequence produced by the Metropolis algorithm is used for *perfect* importance sampling, proportional to f. Each sample contributes $\frac{f(X_i)}{p_i(X_i)}$ to the Monte Carlo estimator. As p_i is assumed to be proportional to f, $\frac{f(X_i)}{p_i(X_i)} = c$, however, it is usually not

possible to analytically determine c. Integrating both sides of the equation results in

$$c = \int_{\Omega} f(x) d\Omega(x) \tag{2.19}$$

This equation can be used to estimate c. In the context of unbiased rendering, c is usually estimated using a small number of PT or BDPT samples.

Note that if X_0 is not sampled according to f, that is $p_0 \not\propto f$, then X_i will have the desired distribution only in the limit as $i \rightarrow \infty$. The bias introduced by the difference between p_i and $\frac{f}{c}$ is called startup bias and will result in bias in the Monte Carlo estimate. The startup bias is often reduced by discarding the first k samples, but it is difficult to choose an appropriate k. In section 2.10, we discuss an approach to eliminate startup bias altogether.

2.9 Metropolis Light Transport

The Metropolis algorithm can be applied to the light transport problem to reduce the variance in the Monte Carlo estimate for each pixel. The Metropolis algorithm is used to generate a sequence of light transport paths, sampling proportional to some function f. In the context of rendering, the Metropolis algorithm is usually not directly applied to estimate the measurement equation of an individual pixel. Instead, samples are generated proportional to the combined measurement contribution function for all m pixel sensors

$$f(\mathbf{X}) = \sum_{j=1}^{m} f_j(\mathbf{X}) \tag{2.20}$$

These samples are then shared to estimate the individual integrals I_j. This has several advantages; First of all, because the measurement functions for nearby pixels are often very similar, applying small mutations to hard-to-find paths often results in similar paths that contribute to nearby pixels. Second, as there is only one Metropolis sequence instead of m, the impact of startup bias is reduced and the normalization constant c has to be estimated only once. A disadvantage is that the number of samples contributing to some pixel j becomes proportional to I_j. When there are large variations in the brightness over the image plane, dark areas in the image become relatively undersampled compared to brighter areas.

When estimating multiple integrals at once, the Metropolis Monte Carlo estimators may be further improved by not only letting accepted mutations contribute, but the rejected mutations as well. Assume $\mathbf{X}_i = \mathbf{X}$ and let $\mathbf{Y}$ be the proposed mutation. Then, $\mathbf{Y}$ is accepted as $\mathbf{X}_{i+1}$ with probability $a(\mathbf{Y}|\mathbf{X})$ and $\mathbf{X}$ is accepted ($\mathbf{Y}$ is rejected) as $\mathbf{X}_{i+1}$ with probability $1 - a(\mathbf{Y}|\mathbf{X})$. Hence, the expected contributions of $\mathbf{X}$ and $\mathbf{Y}$ equal $c(1 - a(\mathbf{Y}|\mathbf{X}))$ resp. $ca(\mathbf{Y}|\mathbf{X})$. Instead of only letting $\mathbf{X}_{i+1}$ contribute c to the estimates, it is possible to let $\mathbf{X}$ and $\mathbf{Y}$ both contribute their expected contributions instead. Because different paths may contribute to different pixels, the average number of samples contributing to a pixel increases, especially for relatively dark pixels having on average lower acceptance probabilities.

Algorithm 1 shows a general Metropolis Light Transport (MLT) sampler. The sampler generates a sequence of N samples using the Metropolis-Hastings algorithm. Of these, the first k are discarded to eliminate startup bias. The algorithm's initial path $\mathbf{X}$ and estimated

normalization constant c are passed as parameters to the sampler. For an estimator with low bias, N should be significantly large.

Algorithm 1 : Metropolis($\mathbf{X}, N, k, c$)

```
X_0 ← X
for i = 1 to N + k do
  Y ←mutate(X_{i−1}) ∝ T(Y|X_{i−1})
  a ← a(Y|X_{i−1})
  if i ≥ k then
    contribute a c/N to image along X_{i−1}
    contribute (1 − a) c/N to image along Y
  end if
  if a ≥ U(0,1) then
    X_i ← Y
  else
    X_i ← X_{i−1}
  end if
end for
```

What is left to discuss is the mutation strategy used to construct the sample sequence. As mentioned earlier, the mutation strategy should satisfy ergodicity. This is achieved by sampling a *fresh* path using either PT or BDPT with non-zero probability. For the remainder of mutations, the path vertices are perturbed in order to locally explore path space. Veach proposed two simple perturbation methods, the lens and caustic mutations [52]. Caustic mutations are applied to paths of the form $EDS(S|D)^*L$, these paths are responsible for caustic effects, hence the name. Figure 2.8 shows an example of a caustic mutation. Lens mutations are applied to all remaining paths[4]. Figure 2.6 shows an example of a lens mutation. Both mutation types only perturb the first i vertices of the path $x_0 \cdots x_k$ with $0 < i \leq k$. When $i < k$, the mutated path is formed by concatenating the perturbed subpath $x'_0 \cdots x'_i$ and the original subpath $x_{i+1} \cdots x_k$ to form the complete path $x'_0 \cdots x'_i x_{i+1} \cdots x_k$. If $i = k$, then $x'_0 \cdots x'_i$ already forms a complete path. The signature and length of the path are unaffected by these mutations. When the signature of the path does change during mutation, the mutation is immediately rejected.

Lens mutation: The lens mutation creates a new path from an existing path, beginning at the eye. The mutation is started by perturbing the outgoing eye direction $x_0 \to x_1$, which usually means the mutation will contribute to a different pixel. The new eye subpath is propagated backwards through the same number of specular bounces as the original path, until the first diffuse vertex x_j is reached. If the next vertex x_{j+1} on the original path is diffuse, the mutated vertex x'_j is explicitly connected to x_{j+1} to form the complete mutation $x'_0 \cdots x'_j x_{j+1} \cdots x_k$ [5]. However, if the next vertex x_{j+1} is specular, the outgoing direction

[4] Veach differentiates between lens and multi-chain perturbations, we do not make this distinction [52].

[5] For an explicit connection to make sense, the two vertices involved both need to be diffuse. Therefore, the next vertex must be diffuse.

$x_j \rightarrow x_{j+1}$ is perturbed by a small angle and the eye subpath is further extended through another chain of specular vertices until the next diffuse vertex is reached. This process is repeated until either two consecutive diffuse vertices are connected, or the light source is reached. Figure 2.6 shows the mutation of a path of the form *ESDSDL*. First, the outgoing direction from the eye is perturbed. The mutation is then extended through the first specular bounce. The outgoing direction of the first diffuse vertex is also perturbed and the mutation is again extended through another specular bounce. Finally, the path is explicitly connected to the light source to complete the mutation.

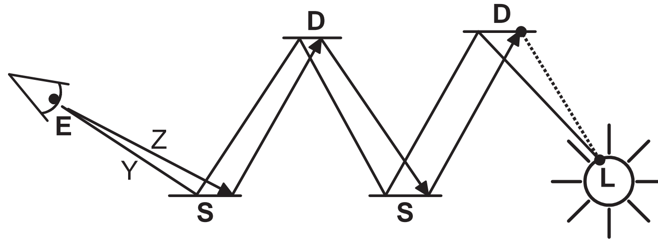


Figure 2.6: Lens mutation of a path of the form *ESDSDL*. The mutation starts at the eye. Further path vertices are mutated backwards until an explicit connection can be made.

Lai Lens mutation: Lai proposed an alternative lens mutation that requires only perturbing the outgoing eye direction $x_0 \rightarrow x_1$ [57]. Just like the original lens mutation, the new eye subpath is propagated backwards through the same number of specular bounces as the original path, until the first diffuse vertex x_j is reached. However, if the next vertex x_{j+1} is specular, instead of perturbing the outgoing direction $x_j \rightarrow x_{j+1}$ as in the original lens mutation, an explicit connection is made to the specular vertex x_{j+1} on the original path. If this connection succeeds, the mutation is extended through the remaining specular vertices until the next diffuse vertex is reached. Again, this process is repeated until either two consecutive diffuse vertices are connected, or the light source is reached. Figure 2.7 shows the mutation of a path of the form *ESDSDL*. First, the outgoing direction from the eye is perturbed. The mutation is then extended through the first specular bounce. Instead of perturbing the outgoing direction, the first diffuse vertex is connected to the next specular vertex and the mutation is again extended through another specular bounce. Finally, the path is explicitly connected to the light source to complete the mutation.

Caustic mutation: Caustic mutations are very similar to lens mutations, but perturb the path forward towards the eye, instead of starting at the eye and working backwards to the light source. The caustic perturbation creates a new path from an existing path, beginning at the second diffuse vertex x_i from the eye. The mutation starts by perturbing the outgoing direction $x_i \rightarrow x_{i-1}$. The new subpath is propagated forwards through $i-2$ specular bounces until the first diffuse vertex x_1 is reached. This vertex is then explicitly connected to the eye vertex x_0 to complete the mutation. Figure 2.8 shows the mutation of a path of the form *EDSDL*. The mutation starts by mutating the outgoing direction at the second diffuse

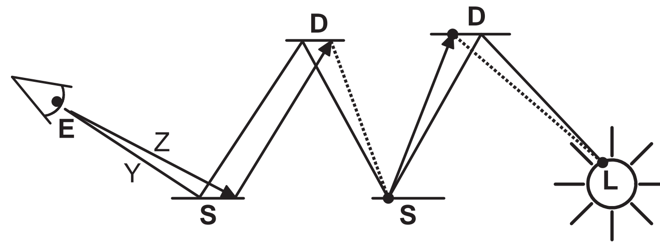


Figure 2.7: Lai Lens mutation of a path of the form *ESDSDL*. The mutation starts at the eye. Further path vertices are mutated backwards until an explicit connection can be made. Diffuse vertices are connected directly to specular vertices while extending the path.

vertex. The mutation is then extended through one specular bounce. Finally, the first diffuse vertex is explicitly connected to the eye to complete the mutation.

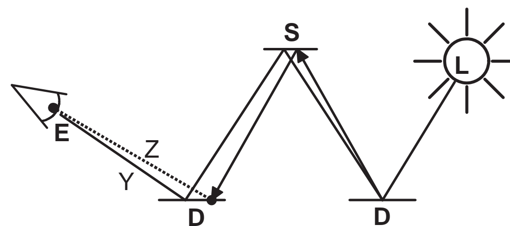


Figure 2.8: Caustic mutation of a path of the form *EDSDL*. The mutation starts at the second diffuse vertex from the eye. All preceding vertices are perturbed forward and an explicit connection to the eye is made.

When using more advanced camera models with a finite aperture lens, both mutations should also perturb the eye vertex x_0 itself. For simple models such as the pinhole camera, there is only one valid x_0, so perturbing x_0 is not necessary. Aside from these perturbation mutations, Veach also proposed mutations that substitute any subpath by a completely new subpath of possibly different length and signature, using bidirectional mutations [52]. We will not discuss these mutations here, and direct the interested reader to the original paper on Metropolis light transport [52]. Because the original paper is found to be difficult to understand by many people trying to implement the Metropolis Light Transport algorithm, Cline presented a comprehensive tutorial on Metropolis Light Transport and its implementation [10].

For more details on generating lens and caustic mutations and evaluating their acceptance probability, see appendix F.

2.10 Energy Redistribution Path Tracing

The Energy Redistribution Path Tracing (ERPT) algorithm as proposed by Cline, is closely related to the MLT algorithm from last section [11]. However, ERPT does not suffer from startup bias. Also, ERPT does not require an accurate and unbiased estimate of the normalization constant c, although it can increase performance. Furthermore, the mutation strategy used in ERPT does not need to satisfy the ergodicity constraint.

MLT suffers from startup bias because the initial path $\mathbf{X}_0$ for the mutation chain is not sampled proportional to f. The ERPT algorithm solves this problem and *does* sample $\mathbf{X}_0$ proportional to f. This is done by sampling multiple mutation chains per sample, all with equal length N. Path tracing is used to generate the initial path $\mathbf{X}_0$ for these mutation chains. Using PT, a path $\mathbf{X}_0$ is sampled with probability $p(\mathbf{X}_0)$, thus the probability of having $\mathbf{X}_0$ as the initial path of a chain is proportional to $p(\mathbf{X}_0)$. However, this number should be proportional to $f(\mathbf{X}_0)$. Hence, the number of mutation chains starting at $\mathbf{X}_0$ is off by a factor $\frac{f(\mathbf{X}_0)}{p(\mathbf{X}_0)}$. This is resolved by starting multiple chains per initial path $\mathbf{X}_0$. When on average, the number of chains for $\mathbf{X}_0$ is proportional to $\frac{f(\mathbf{X}_0)}{p(\mathbf{X}_0)}$, the initial mutation chain paths $\mathbf{X}_0$ are sampled proportional to f and startup bias is eliminated. This is realized by making the number of mutation chains $\mathit{numChains}(\mathbf{X}_0)$ per path $\mathbf{X}_0$ equal to:

$$\mathit{numChains}(\mathbf{X}_0) = \left\lfloor U(0,1) + \frac{1}{N \times e_d} \frac{f(\mathbf{X}_0)}{p(\mathbf{X}_0)} \right\rfloor \tag{2.21}$$

In this equation, $U(0,1)$ is a uniform random number between 0 and 1 and e_d is the amount of energy that is contributed to the image by each mutation in a chain, also called the algorithm's energy quantum.

To see why ERPT does not require an unbiased estimate of c, let us compute the expected number of contributions to the image plane for MLT and ERPT. For MLT, the total number of contributions to the image plane always equals N. For an ERPT sample with initial path $\mathbf{X}_0$, the average number of contributions equals $\frac{1}{e_d}\frac{f(\mathbf{X}_0)}{p(\mathbf{X}_0)}$. Therefore, the expected number of contributions per ERPT sample equals $\int_\Omega \frac{1}{e_d}\frac{f(\mathbf{X})}{p(\mathbf{X})} p(\mathbf{X}) d\Omega(\mathbf{X}) = \frac{1}{e_d}\int_\Omega f(\mathbf{X}) d\Omega(\mathbf{X}) = \frac{c}{e_d}$. Compared to MLT, the expected number of contributions per ERPT sample is off by a factor of $\frac{c}{e_d}\frac{1}{N}$. So, to keep the ERPT estimator unbiased, instead of contributing $\frac{c}{N}$ per mutation, as in MLT, an ERPT mutation should contribute $\frac{c}{N}\frac{e_d}{c}\frac{N}{1} = e_d$. Hence, the ERPT algorithm does not require an explicit estimate of c.

The energy quantum e_d may be chosen freely without introducing bias. However, it does influence its performance. The number of mutation chains per ERPT sample is inversely proportional to both N and e_d. The expected number M of mutation chains per ERPT sample can be regulated by using $e_d = \frac{c}{N \times M}$. This again requires an estimate of c. However, this estimate does not need to be very accurate and may even be biased, as it only influences the performance of the ERPT algorithm. In practice, the ERPT algorithm is not very sensitive to the value of e_d.

In ERPT, generating mutation chains for a PT sample is called energy redistribution. Algorithm 2 shows the energy redistribution part of an ERPT sampler. As input, it takes an initial PT path $\mathbf{X}_0$, the mutation chain length N and the energy quantum e_d.

Algorithm 2 : EnergyRedistribution($\mathbf{X}_0, N, e_d$)

numChains $\leftarrow \left\lfloor U(0,1) + \frac{f(\mathbf{X}_0)}{p(\mathbf{X}_0)} \frac{1}{N \times e_d} \right\rfloor$
for $i = 1$ to *numChains* **do**
  $\mathbf{Y} \leftarrow \mathbf{X}_0$
  for j = 1 to N **do**
    $\mathbf{Z} \leftarrow$mutate($\mathbf{Y}$)
    $a \leftarrow a(\mathbf{Y} \rightarrow \mathbf{Z})$
    deposit ae_d energy at $\mathbf{Z}$
    deposit $(1-a)e_d$ energy at $\mathbf{Y}$
    if $a \geq U(0,1)$ **then**
      $\mathbf{Y} \leftarrow \mathbf{Z}$
    end if
  end for
end for

Finally, a useful property of the ERPT algorithm is that its mutation strategy does not necessarily need to satisfy ergodicity for the outcome to be unbiased. The reason is that mutation chains already have their initial samples distributed proportional to f. Note however that although the outcome will remain unbiased when using only a few ERPT samples, the error may become unacceptably large. We will discuss this in more detail in section 9.2.2. Using many ERPT samples with relatively short mutation chains solves this problem.

Although N should not be chosen too large, N should not be chosen too small either. In ERPT the value of N determines the amount of energy redistribution. For effective energy redistribution, N should be reasonably large. If N is too small, ERPT effectively deteriorates into a quantized PT algorithm. To see why, let us look at the extreme case without any energy redistribution; each mutation chain only contains its initial sample. The number of chains per initial sample is proportional to f, so each sample will contribute on average $e_d \left\lfloor U(0,1) + \frac{1}{e_d} \frac{f(\mathbf{X}_0)}{p(\mathbf{X}_0)} \right\rfloor = \frac{f(\mathbf{X}_0)}{p(\mathbf{X}_0)}$. All this energy is contributed to a single pixel; the pixel corresponding to $\mathbf{X}_0$. Hence, because each chain contributes a single energy quantum, the remaining algorithm is a quantized PT algorithm, performing worse than normal PT.

The ERPT algorithm presented in [11] uses only the lens and caustic mutation types from the MLT algorithm and an N in the order of 100. Note that because lens and caustic mutations do not change the length or signature of a path, all paths in a mutation chain have the same length and signature.

Chapter 3

GPGPU

3.1 Introduction

Modern General Purpose GPU's (GPGPU) are many-core streaming processors. A uniform streaming processor takes a data stream as input, applies a series of operations (a kernel) to all elements in the data stream and produces an output data stream. Usually, all data elements are processed independently, without explicit communication or synchronization. This allows the streaming processor to apply the kernel to multiple data elements in parallel.

In this work, we will focus on the CUDA parallel computing architecture developed by NVIDIA [13]. Other frameworks for GPGPU programming are OpenCL [21], Microsofts DirectCompute [38], and ATI's Stream [3]. On CUDA GPU's, the requirement of independence is less strict and various forms of explicit and implicit communication and synchronization are available. This gives room for implementing a broader class of almost-streaming algorithms on the GPU and further improving the performance of streaming algorithms. Stream elements are arranged in a thread hierarchy, where each level in the hierarchy provides different forms of communication, memory access and synchronization.

In this chapter, we will discuss the CUDA thread hierarchy and corresponding memory hierarchy. We will further explain the forms of synchronization available. In our discussion we will mainly focus on how to achieve high parallelism and memory bandwidth, which translates to high overall performance. Finally, we will discuss the parallel scan, a very useful parallel processing primitive [47].

3.2 Thread Hierarchy

In CUDA, when a kernel is applied to a stream, a single CUDA thread is executed for each stream element. All threads run in parallel. These threads are arranged in a three-level thread hierarchy (See figure 3.1). At the highest level, the threads are arranged in a large 1,2 or 3-dimensional grid. At the second level, the grid is subdivided in equal sized blocks of threads. These blocks are further subdivided in warps, each consisting of 32 threads. The grid, blocks and threads are exposed to the programmer through the CUDA API. The dimensions of the grid and blocks are explicitly specified by the programmer. Warps are

not explicitly exposed to the programmer and blocks are implicitly subdivided in warps of 32 threads each. Warps however have important features and to achieve high performance, it is important to take these into account.

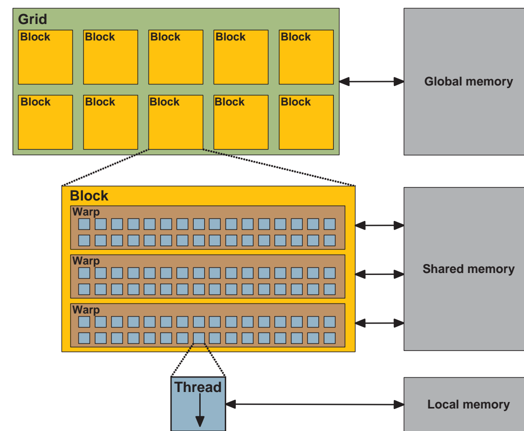


Figure 3.1: CUDA thread hierarchy.

The size of the grid may be chosen freely. The maximum size of a block is limited, depending on the kernel and CUDA compute capability. Each block is mapped to a single Streaming Multiprocessor (SM) on the GPU. All warps in the block are executed on this SM. Modern GPU's have multiple SM's. The grid blocks are distributed over the available SM's. For maximum parallelism and efficient load balancing, the grid must consist of a reasonable amount of blocks, enough to fully utilize all SM's on the GPU.

Multiple warps are executed concurrently on a single SM. The warps share the SM using a form of hyper threading. An SM handles one warp at a time, executing all threads in the warp in parallel[1]. Whenever the warp is suspended, for example awaiting a memory transfer, the SM resumes the next warp. This method is used to hide memory latencies. To effectively hide all latencies, it is important to execute as many concurrent warps per SM as possible.

SIMT

Each SM on the GPU consists of multiple processors sharing one instruction unit (see figure 3.2). Each thread in a warp maps to one of these processors. Therefore, threads in a single warp execute in SIMT (Single Instruction, Multiple Threads). SIMT is an implicit form of SIMD, handled by the hardware. All threads in the warp execute one common instruction at a time. If the control flow of threads in a warp diverges due to data-dependent conditional branching, the hardware serially executes each branch path taken, disabling threads that are not on the path. When code paths eventually converge again, the threads are again executed in parallel. Full efficiency is realized when all threads in the warp follow the same code path and agree on the next instruction, so that all threads are executed in parallel. Because

[1] On older GPU models, the 32 threads are executed in blocks of 8 threads running in parallel.

diverging code paths are executed serially, efficiency is decreased whenever threads follow a different code path, reducing parallelism. To realize high parallelism, it is vital to keep the number of different code paths taken by threads in a warp to a minimum.

3.3 Memory Hierarchy

Figure 3.1 also shows the CUDA memory hierarchy. Each thread has a private local memory, containing registers. Each thread block has a shared memory visible to all threads of the block. Finally, all threads have access to the same global memory. Shared memory is used for communication between threads in a block and global memory may be used to communicate between threads in different blocks (see section 3.4). Global memory is a large memory storage with relatively low bandwidth and high latency. Local thread memory and shared memory are small memory stores with relatively high bandwidth and low latency. Figure 3.2 shows where these memory banks are located on the device.

Each SM has its own on-chip local memory and shared memory banks. Besides these, each SM also possesses a small L1-cache for caching global memory[2]. The shared memory and L1-cache share the same memory bank. The total amount of local memory and shared memory per SM is limited, depending on the device. These resources must be divided between all blocks concurrently executing on an SM. Each thread requires an amount of local memory, depending on the kernel. This restricts the maximum number of warps in a block. Furthermore, each block requires an amount of shared memory and L1-cache, restricting the maximum number of blocks concurrently running on an SM. As the number of warps concurrently running on an SM must be maximized to hide memory latency, shared memory and local memory usage per kernel should be minimized. For more details on block sizes and their relation to available local and shared memory, see [13, 12].

Besides the L1-cache, figure 3.2 also shows an L2-cache. The L2-cache is shared by all SM's on the device. All global memory access is loaded through the L2-cache. On devices with CUDA compute capability 1.x, only explicitly specified read-only memory (containing textures and constants) was cached in the L2 cache, but on more recent devices with at least compute capability 2.0, all memory access, including memory writes, are cached in the L2 cache[14]. Whenever the cache is unable to serve a memory request, it is served by the device memory.

Coalesced memory access

Device memory handles requests serially, serving one memory transaction at a time. For maximum memory bandwidth, memory requests should be served in the least number of memory transactions. Simultaneous memory requests by multiple threads in a warp can be coalesced into one or a few memory transactions. For this, a warp is subdivided in two half-warps of 16 threads each[3]. When the memory access pattern of threads in a half-warp satisfies certain conditions, the memory requests can be coalesced. For older devices with

[2] L1-caches are only available on devices with at least compute capability 2.0.

[3] On devices with at least compute capability 2.0, a warp is no longer subdivided in two half-warps. Instead, all 32 threads are considered together.

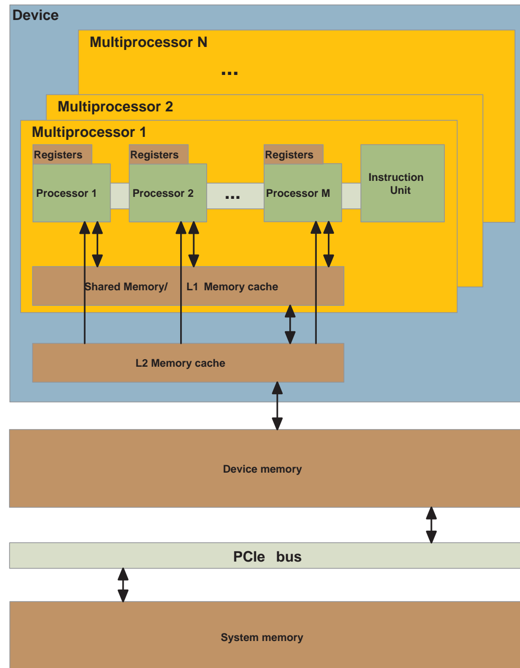


Figure 3.2: CUDA device architecture.

compute capability 1.0 and 1.1, these conditions are much stricter than for newer devices. We will first discuss the more strict requirements. Figure 3.3 shows a coalesced memory transaction for devices of compute capability 1.0 or 1.1.

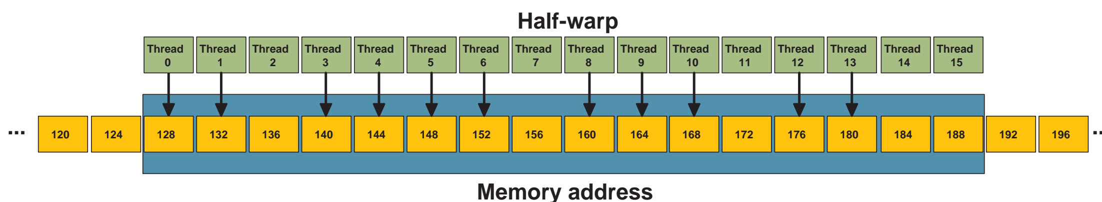


Figure 3.3: Coalesced memory transaction on device with compute capability 1.0 or 1.1. All memory is served in 1 memory transfer.

For memory requests to be coalesced on devices with compute capability 1.0 or 1.1, the threads in the half warp must all access either 4-byte, 8-byte or 16-byte words. Furthermore, all 16 words must lie in the same memory segment of 16 times the word size and equal sized memory alignment. Finally, all threads must access the words in sequence, that is, the k'th thread in the half warp must access the k'th word in the memory segment. These conditions must be met by all active threads in the half warp. Any threads that are temporarily disabled by the hardware due to SIMT are not considered and cannot violate the conditions. If all conditions are met, the whole memory segment is served in coalesced memory transfers

(1 transaction for 4 and 8 byte words, 2 transfers for 16 byte words). Note that when the k'th thread is disabled, the k'th word is still loaded. This reduces the *effective bandwidth*. The effective bandwidth is the amount of transfered memory per second that was actually requested by active threads. When one of the conditions for coalesced memory access is not met, one memory transaction is issued for each thread requesting memory, which significantly reduces memory bandwidth. So, to reach high effective bandwidth, all memory access should be coalesced and all threads in a half-warp should participate in the memory request.

On devices with compute capability 1.2 or higher, the requirements for coalesced memory access are less strict. Figure 3.4 shows a coalesced memory transaction on such a device. On these devices, memory requests are served with the least amount of transfers

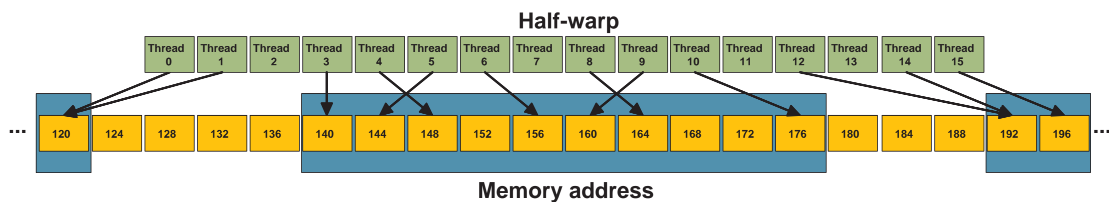


Figure 3.4: Coalesced memory transaction on device with compute capability 1.2. All memory is served in 3 memory transfers.

possible. Threads in the half-warp may have any memory access pattern. All memory requests to any memory segment are coalesced in a single memory transaction. So, when m different memory segments are addressed, m memory transactions will be issued ($2m$ if the word size is 16-byte). In figure 3.4, 3 memory segments are addressed with a word size of 4 bytes, so memory access is served in 3 memory transactions. So, to maximize memory bandwidth, threads in a half-warp should try to minimize the number of different memory segments addressed during a memory request. Unused words due to disabled threads may still be read by the hardware, reducing effective memory bandwidth. To further reduce this waste of bandwidth, the hardware issues the smallest memory transaction that contains all requested words in a memory segment (as demonstrated in figure 3.4).

For maximum shared memory bandwidth, shared memory requests by threads in a half-warp should also adhere to certain access patterns to prevent bank conflicts. These patterns are however less strict and the performance degradation is much less severe compared to non-coalesced global memory access. For more information on coalesced memory access and shared memory bank conflicts, the reader should consult [13].

System memory

In figure 3.2, device memory is attached to system memory via the PCIe (PCI Express) bus. Whenever the CPU and GPU are working together, they can communicate by transferring memory from system memory to device memory and back over the PCIe bus. Note that PCIe bandwidth is usually not symmetrical. Transferring from system memory to device memory is significantly faster than the other way around. For maximum PCIe bandwidth,

memory should be transfered in large chunks at a time. When system memory is allocated in page-locked memory, PCIe memory transactions can be executed concurrent with CPU and GPU kernel execution. For maximum GPU utilization, it is important to perform the GPU kernel execution and memory transaction concurrently whenever possible.

3.4 Synchronization and Communication

CUDA supplies different forms of synchronization and communication between threads in the thread hierarchy. We will first discuss thread synchronization.

Synchronization

At warp level, all threads run in lock-step due to SIMT. Therefore, threads in a warp are implicitly synchronized. At block level, threads in a block can be explicitly synchronized using special synchronization barriers. As all threads in a warp are already synchronized, when a barrier is issued the hardware only synchronizes all warps in the block [56]. Synchronization between threads in different blocks is not possible because CUDA gives no guarantees on the order in which blocks are scheduled for execution and which blocks will be concurrently active on the GPU. Finally, the host system can execute concurrently with the GPU and may synchronize with the GPU by awaiting the completion of kernel execution. CUDA does not provide mechanisms to synchronize the GPU with the host system.

Communication

There are three forms of communication between threads: at warp level, at block level and at grid level. On devices with compute capability 2.0, CUDA supplies voting primitives at warp level, used to make all threads in a warp agree on something. Threads in the same block may communicate through shared memory. When threads in a warp all write the same shared memory address, CUDA write semantics guarantee that one of these writes will succeed, without specifying which. This mechanism can be used to implement some simple voting primitives on older devices.

Finally, all threads can communicate through global memory, regardless of the blocks they belong to. On devices with compute capability 1.1 and higher, global memory can be modified through atomic instructions. Atomic instructions accessing the same memory address (or any address in the same cache-line) collide and must be serialized, reducing parallelism. Atomic memory access without collisions is almost as fast as normal non-coalesced memory access. So, for maximum performance, atomics should be used scarcely and should address different cache lines to reduce collisions.

3.5 Parallel scan

In this section, we will discuss the parallel scan as an important parallel programming primitive. A scan computes the partial prefix sums $s_0 = \sum_{i=0}^{0} x_i, s_1 = \sum_{i=0}^{1} x_i, \cdots, s_{n-1} = \sum_{i=0}^{n-1} x_i$

Algorithm 3 : ParallelScan$(x_0, \cdots, x_{n-1})$

for $i = 0$ **to** $n-1$ **in parallel do**
  $s_i \leftarrow x_i$
end for
for $i = 0$ **to** $\lceil \log_2(n) \rceil - 1$ **do**
  for $j = 0$ **to** $n-1$ **in parallel do**
    if $j \geq 2^i$ **then**
      $s_j \leftarrow s_j + s_{j-2^i}$
    end if
  end for
end for

over a sequence of numbers $x_0 \cdots x_{n-1}$. A parallel scan computes these prefix sums in parallel. Algorithm 3 shows the general parallel scan algorithm. Efficient CUDA implementations of this algorithm are available for computing the prefix sum over all threads in a warp, block or grid, where thread i delivers x_i and wants to compute s_i [47]. The parallel scan over all threads in the grid requires the execution of multiple kernels, however the parallel scan on warp and block level only require the execution of a single kernel and can easily be embedded within more complex kernels as an algorithmic step. Sengupta showed that the parallel scan can be used to implement several parallel algorithms on the GPU, such as parallel sorting [47]. In this work, the parallel scan is used for compaction.

During compaction, each thread wants to output x_i elements in an output stream. The parallel scan is used to uniquely determine the position in the output stream where each thread may store its output elements. After applying the parallel prefix sum, s_i equals the total number of output elements for all threads preceding thread i, itself included. So, by storing the x_i elements at positions $s_i - x_i$ to $s_i - 1$ in the output stream, all output elements will be tightly packed in the output stream.

Stream compaction can be very useful when each thread generates at most n elements for some small n, usually 1. By using compaction, the output stream can by tightly packed, removing any inactive elements from the output stream. This output stream may then serve as the input stream for another kernel. See section 7.4 for an application of stream compaction to ray tracing.

Chapter 4

Related Work

In recent years, much research has been targeted at improving unbiased Monte Carlo rendering methods and implementing efficient ray traversal algorithms on the GPU. In this chapter, we will give a short overview of these developments and the most relevant results.

4.1 GPU ray tracing

In this section, we will give an overview of the most important research, targeted at ray traversal and unbiased rendering on the GPU.

4.1.1 Ray traversal

Since the introduction of GPGPU programming, many researchers have attempted to implement efficient ray traversal algorithms on the GPU. Of these, Purcell was the first to publish an efficient GPU traversal algorithm [45]. His algorithm ran in multiple passes and used a uniform grid as underlaying spatial structure. Because uniform grids are not well suited to handle non-uniform geometry, several researchers proposed traversal algorithms using a KD-tree as spatial structure. Foley proposed two stackless multi-pass KD-tree traversal algorithms, called kd-restart and kd-backtrack [19]. Horn improved the work of Foley and implemented the kd-restart algorithm as a single-pass algorithm [27]. Popov proposed an alternate stackless KD-tree traversal algorithm using links between adjacent tree nodes to steer traversal [44].

The introduction of NVidia's GPGPU framework CUDA allowed for the implementation of efficient stack based GPU traversal algorithms. Guenther proposed a packet traversal algorithm, using the Bounding Volume Hierarchy (BVH) as a spatial data structure [22]. In the algorithm, all rays in the packet share a single stack. For reasonable efficiency, all rays in a packet should have a similar origin and direction, called ray coherence. Aila elaborately studied the SIMT efficiency of stack based BVH traversal on the GPU and improved the efficiency through the use of persistent GPU threads [1]. Garanzha proposed a stackless breadth-first packet traversal algorithm. The algorithm traverses packet frusta through a BVH to locate ray-leaf intersections. The rays are then tested against all triangles in these leafs [20].

All packet traversal algorithms require packets of reasonably coherent rays in order to achieve high performance [41]. Garanzha proposed to construct coherent ray packets on the GPU by spatially sorting the rays and grouping them into packets [20]. Aila found that even though the bvh traversal algorithm does not use packets, it still benefits significantly from ray coherence. When the rays traced by different threads in a GPU warp are relatively coherent, SIMT efficiency is increased and the GPU caches become more effective, increasing traversal performance [1].

Several GPU algorithms are developed to construct spatial structures. The most notable of these are the KD-tree construction algorithm by Zhou [59] and the BVH-tree construction algorithm by Lauterbach [35].

4.1.2 Unbiased GPU rendering

Although a lot of research has been dedicated to efficient ray traversal algorithms on the GPU, relatively little research has been targeted at the development of complete unbiased rendering solutions on the GPU. Novak proposed a GPU path tracer with high SIMT efficiency through path regeneration [40]. In the path tracing method, paths are stochastically terminated causing varying path lengths. This results in gaps in the path tracing sampler stream, reducing SIMT efficiency. By immediately regenerating terminated paths, Novak keeps the average SIMT efficiency high. For an unbiased result, his method requires a cool down period in which all remaining paths are completed while no new paths are regenerated. During this cool down period, SIMT efficiency decreases. Novak also presents the first GPU implementation of a BDPT using path regeneration.

For a long time, the impact of shading on the overall performance of rendering was deemed insignificant (5%) compared to ray traversal (95%) [55]. However, due to the increase in ray traversal performance and the use of more complex materials, shading often takes up a significant part of the method's computations. Hoberock proposed a deferred shading method to increase shading SIMT efficiency in a GPU path tracer. By sorting the PT samplers based on the shader they require next, shading efficiency increases [26]. This method is especially useful for complex and procedural materials.

NVidia provided a real time ray tracing engine based on the CUDA framework, called OptiX [37]. OptiX delivers a flexible framework for GPU ray tracing, allowing for user defined programmable intersection routines and shader models. Although highly flexible, the performance of OptiX is relatively low for divergent rays. This makes it less suitable for unbiased rendering methods such as path tracing.

4.1.3 Hybrid architecture

Besides complete GPU rendering solutions, several attempts have been made to utilize the GPU's computational resources alongside conventional CPU's in a hybrid renderer. Initially, these algorithms relied on the CPU for ray generation, traversal and shading, while the GPU was used as a ray-triangle intersection test co-processor. Cassagnabere presented such an architecture, balancing the work over all available GPU's and CPU's in the system [8]. After the introduction of efficient ray traversal on the GPU, ray tracing and shading

could be performed on either the GPU or CPU, allowing for easier work balancing over available system resources. Budge presented an out-of-core path tracing renderer having implemented all but path generation, propagation and termination on both the GPU and CPU. The rendering tasks are efficiently balanced over all available CPU and GPU resources in the system [7]. His method focused on rendering models that do not completely fit in system memory.

4.2 Unbiased rendering

Most of the important methods concerning unbiased rendering we already discussed in chapter 2. In this section, we will discuss some important related research on unbiased rendering using Monte Carlo estimates.

4.2.1 Variance reduction

After Kajiya introduced the rendering equation and used Monte Carlo Markov chains to estimate its value [30], a lot of research has been targeted at reducing the variance and improving the performance of such estimators. We discussed the most important variance reduction techniques in chapter 2. In this section we will discuss some related methods, not discussed in chapter 2. Jensen proposed an alternative importance sampling method, based on photon mapping. First, the incoming radiance on all surface points is approximated by casting photons into the scene and storing the photons into a photon map. Then, during path tracing, the scattering direction is sampled according to the incoming radiance approximation from the photon map [28]. Because the photon map is assumed to be a good approximation for the incoming radiance, this method should reduce the variance in the estimator. Bekaert proposed to accelerate path tracing by combining multiple PT samples through nearby screen pixels, allowing paths to contribute to multiple pixels. This method increases the sample speed at the cost of extra correlation between screen pixels [5]. Kitaoka introduced the replica exchange light transport method. RELT combines multiple sampling methods with different distributions. A sequence of path is sampled from each distribution as a stationary distribution using the Markov chain Monte Carlo method. All path samples are combined using MIS [31]. Wächter showed how overall variance of the Monte Carlo estimator can be reduced by using stratified quasi random numbers to generate samples [53].

4.2.2 Mutation strategies

Since the introduction of Metropolis Light Transport [50], several researchers have tried to improve and simplify the Metropolis algorithm. One important improvement, the ERPT algorithm [11], has already been discussed in chapter 2. In this section we will discuss some variations, and alternate mutation strategies.

Lai and Fan used the population Monte Carlo framework to adapt the rendering method based on information gathered so far during sampling. They applied the method to the

ERPT rendering algorithm to adapt the perturbation sizes for mutations, based on the acceptance rate of mutations [58, 18]. Liris proposed an alternative mutation strategy, based on Multiple-Try Metropolis sampling. Instead of generating a single mutation as a proposed next sample, the method generates a collection of mutations and selects one. This mutation is accepted according to a modified acceptance, resulting in an increased average acceptance probability. The collection of mutations are usually highly coherent, allowing for packet tracing and increased ray traversal performance [46].

The bidirectional mutation strategy as presented by Veach [50] is generally found to be difficult to understand and implement in practice. Kelemen proposed a much simpler mutation strategy. Instead of mutating the path in path space, the mutation works on the unit hypercube of pseudo-random numbers from which the sample was constructed. He shows that because fixed perturbation sizes in pseudo-random space corresponds to varying perturbation sizes in path space, based on importance sampling, this method increases the average acceptance probability [31].

Pauly showed how to render participating media by modifying Veach's mutation strategy [43].

Part II

GPU Tracers

Chapter 5

The Problem Statement

In this thesis we are addressing the following problem: *How can we adapt unbiased physically based rendering algorithms to run efficiently on the GPU?* In this chapter, we will further elaborate on the problem, before actually addressing its solution in subsequent chapter.

To address the problem we must first specify what it means for a GPU implementation to *run efficiently on the GPU*. In chapter 3 we saw that for an implementation to fully utilize all GPU resources, it must take the GPU thread and memory architecture into account. In particular, for a GPU implementation to be efficient, it must adhere to three basic strategies [12]:

- *Maximizing parallel execution*

  If the implementation does not expose enough parallelism, the GPU will not have enough work, preventing the work from being evenly distributed over all available computation resources. An efficient implementation exposes enough parallelism to fill all GPU resources.

- *Optimizing memory usage to achieve maximum effective memory bandwidth*

  When memory access is not coalesced, memory bandwidth is wasted and the effective memory bandwidth drops. An efficient implementation uses memory access patterns that allow for coalesced memory access, increasing effective memory bandwidth

- *Optimizing control flow to achieve maximum SIMT efficiency*

  If the SIMT efficiency of the implementation is low due to divergent control flow, the GPU will be under-utilized and instruction throughput will drop. In an efficient implementation, threads in a warp follow similar code paths, resulting in high SIMT efficiency and thus high instruction throughput.

Therefore, we will asses the efficiency of our GPU implementations on the basis of these three strategies.

The other important aspect of the problem, *adapting unbiased physically based rendering algorithms*, also requires some clarifications. Because unbiased physically based rendering algorithms can be very diverse, it is difficult to come up with a general method to

adapt any such algorithm to the GPU. We will therefore focus on the most well known algorithms: Path Tracing, BiDirectional Path Tracing and Metropolis Light Transport [50]. All other unbiased algorithms are essentially variations of these three. Instead of implementing the MLT directly, which is actually a biased algorithm, we focus on the ERPT algorithm [11] which is a simple extension of the MLT algorithm that solves the start-up bias problem (section 2.10).

Taking these algorithms as a starting point, the goal is to adapt these algorithms to allow for efficient GPU implementations. In this context, *adapting* means making any change to the original algorithm or its implementation with the goal to increase the efficiency of the GPU implementation. These changes should not compromise the unbiased physically based quality of the algorithm. Furthermore, it is important to assess the impact of any changes to the convergence characteristics of the algorithm.

The problem statement does not specify whether or not the algorithm runs *solely* on the GPU. This leaves open the possibility for a hybrid solution, where the GPU is used only for certain algorithmic tasks while the remaining tasks are executed on the CPU. We will investigate both the possibility of a hybrid solution (chapter 6) and the possibilities for GPU-only rendering algorithms (chapters 7, 8 and 9).

To validate our implementations and compare their performance, we use a wide range of test scenes with varying complexity. Figure 5.1 shows the test scenes and table 5.1 gives a short motivation why these scenes are included in the test set. Note that all scenes are of reasonbale size and will fit in device memory. In this thesis we restrict ourselves to the rendering of models that fit entirely in device memory.

All tests are performed on a single test platform containing an Intel®Core™2 Quad CPU and NVIDIA®GeFource®GTX 470 GPU. Although the GTX 470 supports CUDA 2.0 compute capability, our algorithms only require CUDA compute capabilities 1.1 or 1.2. Furthermore, our algorithms do not depend on the presence of L1 and L2 caches for performance. Besides backwards compatibility, we chose to focus on compute capabilities 1.1 and 1.2 without the presence of caches because this emphasize the streaming characteristics of the architecture. This makes our algorithms equally suitable for other GPGPU solutions such as DirectCompute and ATI Stream [38, 3].

5.1 Related work

All relevant related work is already discussed in chapter 4. In this section, we will shortly explain why previous work did not cover the research problem properly.

The work of Novak is most closely related to ours [40]. We borrow heavily from his work on path regeneration to improve the GPU utilization in GPU PT and BDPT implementations. Novak also presented such a PT and BDPT algorithm. However, their work did not use stream compaction to further increase GPU efficiency and speed up ray traversal. Similarly, Hoberock used stream sorting to improve the performance of complex shaders, but did not use stream compaction to improve GPU efficiency and ray traversal performance either [26]. Furthermore, the BDPT implementation by Novak sacrifices effective importance sampling to realize an efficient GPU implementation, thereby significantly reducing

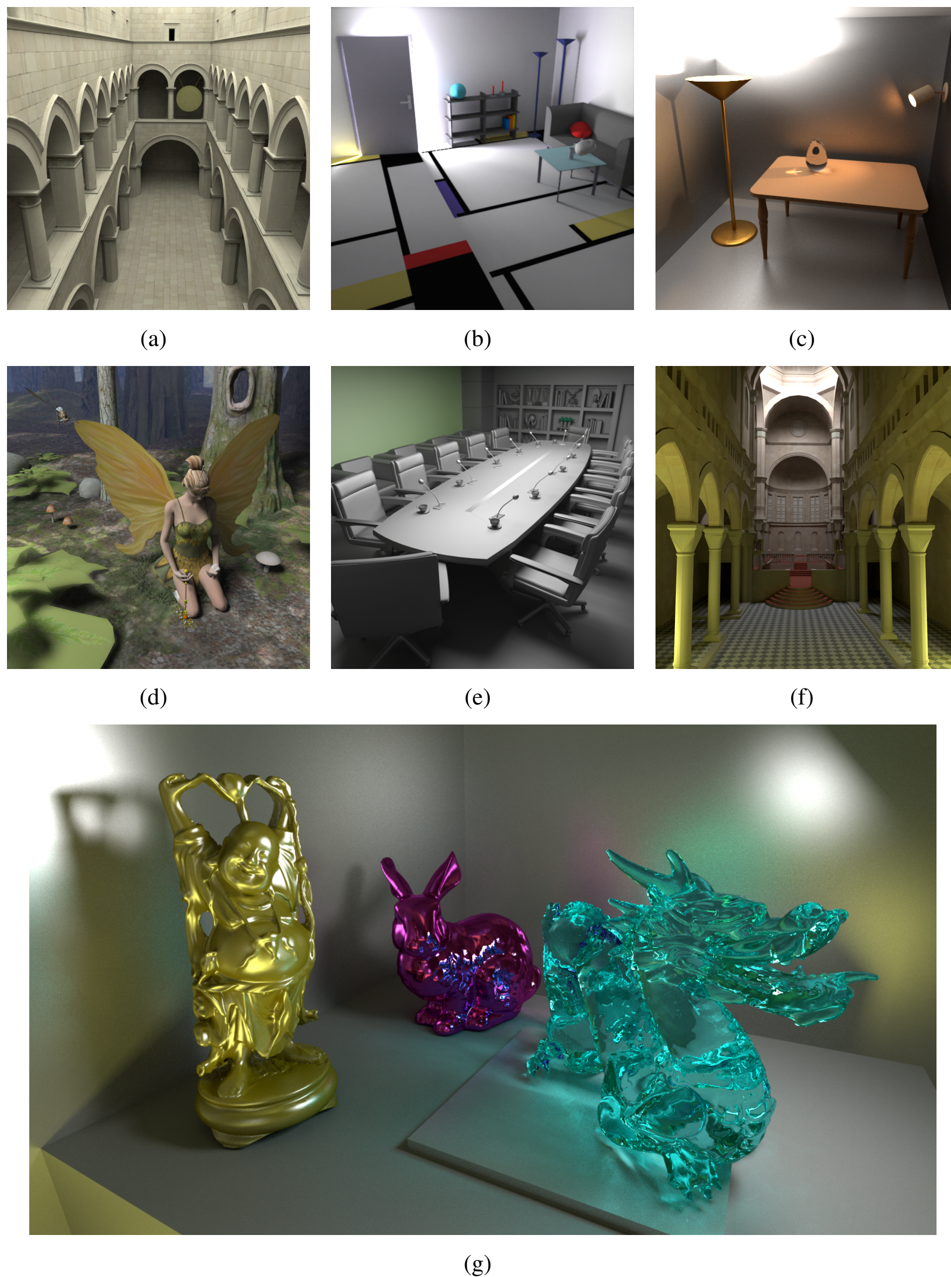


Figure 5.1: Test scenes: (a) Sponza (b) Invisible date (c) Glass egg (f) Sibenik cathedral (d) Fairy forest (e) Conference room (g) Stanford collection.

the strengths of the BDPT algorithm (see section 8.1.2). There have been no attempts to implement the ERPT algorithm on the GPU that we know of.

Apart from GPU-only rendering, we also mentioned some work on hybrid algorithms in chapter 4; most notably, the work of Cassagnabere [8] and Budge [7]. Cassagnabere

uses the GPU for ray-triangle intersection only, performing ray traversal on the CPU. In our work however, ray traversal is implemented on the GPU aswell. Budge also uses the GPU for ray traversal, but their method aims at out-of-core path tracing, while we focus on the rendering of models that fit device memory.

5.2 Context

In this thesis, we focus on the implementation of unbiased samplers. We do not focus on specific ray traversal algorithms or material models. Our samplers are developed within the framework of the Brigade ray tracing engine. In Brigade, the scene is represented by a Bounding Volume Hierarchy to allow for interactive scene manipulation. The BVH's for static models are optimized using spatial splits [48]. We used the GPU ray traversal algorithm published by Aila [1] with the triangle intersection method published by Havel [23]. Note however that the rendering algorithms presented in this thesis are not tied to any specific ray traversal algorithm or spatial data structure. By strictly separating the ray traversal execution from sampler execution, our algorithms are easily combined with ray traversal algorithms requiring large batches of input rays to achieve reasonable performance, such as proposed by Aila and Garanzha [1, 20].

Name	**Triangles** (10^3)	**Motivation**
SPONZA	67	Architectural scene with large area light, a combination of indoor and outdoor lighting effects.
SIBENIK	80	Indoor architectural scene with uneven geometry distribution.
STANFORD	370	Reasonably high primitive count with many complex lighting effects due to specular and glossy materials.
FAIRY FOREST	174	Outdoor scene with many high resolution textures.
GLASS EGG	12	Indoor scene with various materials, mainly illuminated by indirect light.
INVISIBLE DATE	9	Small scene with difficult indirect illumination due to small openings in the geometry.
CONFERENCE ROOM	1180	Detailed model with high primitive count.

Table 5.1: Test scenes, their sizes in kilo-triangles and a motivation why these scenes were used.

Chapter 6

Hybrid Tracer

6.1 Introduction

In this chapter, we will investigate a hybrid architecture where the sampler is implemented on the CPU, using the GPU for ray traversal and intersection. We will show that such an architecture is very flexible, allowing as its basis many different ray tracing based samplers. We will also show that the CPU side easily becomes the bottleneck, limiting the performance gained by using a GPU. This hybrid architecture will reveal some inherent limitations of GPU based unbiased tracers and serve as a reference for the GPU-only tracers in later chapters.

6.2 Hybrid architecture

6.2.1 Sampler

An obvious way to harvest the ray traversal performance of modern day GPU's, without sacrificing flexibility in sampler implementation, is by using the GPU as a ray-traversal co-processor. The sampler is implemented on the CPU, generating rays that require tracing. These rays are then traced on the GPU and the results are used in the CPU sampler. This requires a strict separation between sampler and ray traversal code. The flowchart for such a sampler looks something like figure 6.1. Because the sampler often cannot proceed before the ray intersection results are available, the sampler is suspended as soon as a ray requires tracing. When the ray traversal results are in, the sampler is resumed. We will refer to the process of resuming a sampler until it generates the next ray and gets suspended as *advancing* the sampler.

Often, a sampler could generate more than a single ray before the intersection results of any of these rays is required. For example, when connecting all eye and light vertices in a BDPT sampler, multiple rays may be generated at once. This could increase performance for specific samplers at the cost of complicating the general architecture, especially when a sampler may output any number of rays instead of some fixed number. Because the restriction of a single ray output per sampler advance does not fundamentally restrict the type of sampler used, we will stick to this format for simplicity.

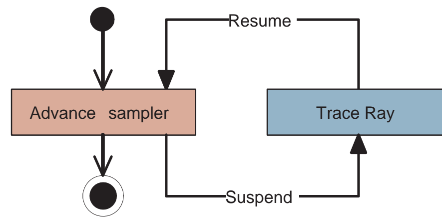


Figure 6.1: General sampler flowchart. The sampler is advanced until a ray requires tracing. Then, the sampler is suspended and the ray is traced. After tracing, the sampler is resumed.

6.2.2 Pipeline

Because the GPU is a massively parallel processor, it performs best when tracing many rays at once. Furthermore, the PCIe bus, used for communicating between CPU and GPU, requires large memory transfers to reach peak performance. Because a sampler only generates a single ray at a time, batches of multiple samplers are used. By advancing all samplers in a batch, a batch of rays is generated. This batch is sent to the GPU as one block, achieving high PCIe performance. The rays are then traced together on the GPU, allowing for enough parallelism to fully utilize the GPU's resources. When finished, the intersection results are send back over the PCIe bus to host memory. The intersection results are used by the samplers during their next advance.

The hybrid tracer is implemented as a 4-stage pipeline of *sampler advance, ray copying, ray traversal* and *result copying* (Figure 6.2). Remember from chapter 3 that the CPU and GPU can execute concurrently. Also remember that CPU and GPU execution may overlap with memory transfers over the PCIe bus. Hence, the four stages of the pipeline can execute concurrently, except for the ray and result copying, which compete for the same resource; the PCIe bus. Notice the implicit dependencies between jobs in the CPU stream in figure 6.2, shown by the dotted arrows: A *sampler advance* job applying to a batch of samplers is dependent on the ray traversal results from the preceding *sampler advance* job applying to the same batch. It is not possible to advance a batch of samplers before the previous ray intersection results are available. These dependencies can be hidden by using at least as many independent batches of samplers as there are stages in the pipeline. In this case, four batches of samplers is enough to keep the pipeline filled.

6.2.3 Details

To utilize all cores of a multicore CPU, we advance the samplers in a batch in parallel using OpenMP [9]. Each sampler outputs exactly one ray, so the output position in memory does not depend on the other samplers in the batch. Furthermore, samplers only use their own sampler and ray intersection data. Hence, the samplers run virtually independent of one another. Because each batch contains many more samplers as there are CPU cores in the system, it is easy to utilize all cores and achieve good load balancing. In our implementation, the only interactions between samplers were caused by contributions to the image plane and use of a shared random generator. These exceptions where handled by creating

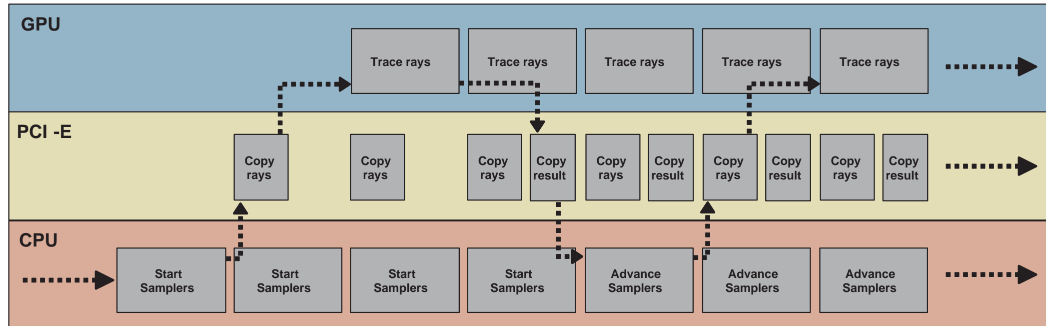


Figure 6.2: CPU-GPU pipeline. Samplers are advanced on the CPU, each generating a single output ray. All rays are copied over the PCIe bus to the GPU, where the rays are traced. The results are copied back. This process is repeated. Multiple independent sampler batches are used to hide dependencies within the stream.

a separate image plane and random generator for each CPU thread. All image planes are combined after tracing is complete. Usually, the number of threads equals the number of CPU cores, so only a few image planes are required.

For each sampler batch, enough host memory is allocated to hold the rays and intersection results for sampler advancing. Extra memory must be allocated to hold any persistent sampler data, needed for the next advance. Also, on the GPU, memory must be allocated to hold all rays and ray traversal results. In most CPU-only tracers, only a single sampler is run per core. Unless the scene is very large, running only a few samplers means that most sampler data still resides in memory caches after tracing. However, in the hybrid tracer, because each batch contains many samplers, all sampler memory is definitely flushed from the cache between two advances of a sampler. The same holds for ray and intersection data. Therefore, the hybrid tracer architecture is less cache friendly than most CPU-only tracers.

6.2.4 Sample regeneration

Until now, we assumed that each sampler always outputs a single ray after being advanced. However, some samples require more rays to be traced than other, for example due to Russian roulette. This problem is solved by using sample regeneration [40]. When a sample is finished, the corresponding sampler immediately starts a new sample. During the cool down period, required for an unbiased result, only unfinished samples are advanced while no samples are regenerated. We use a flag to mark output rays from disabled samplers. During the cool down period, the number of rays per batch reduces per iteration, reducing GPU ray traversal performance. This is an inherent problem with GPU tracers: Because samplers can only generate one or a few rays before their tracing results are required, many parallel samplers are required to achieve high GPU utilization. Furthermore, as the number of rays per sample is a stochastic variable, some samplers will take longer than others, requiring a cool down period for an unbiased result. Depending on the scene and sampler used, the duration of the cool down period will vary.

6.3 Results

In this section we present measurements on the performance of our hybrid architecture and discuss our findings. We tested the hybrid tracer using the path tracing sampler (see section 2.4). We will first study the PT performance within the generic sampler framework. Then we will investigate the performance of the hybrid tracer.

6.3.1 Generic sampler performance

We start by assessing the CPU performance of the generic sampler framework. As explained earlier, when a generic sampler is advanced it must generate a single output ray. Due to this restriction, the implementation of a sampler within the generic sampler framework is usually suboptimal. To determine the performance degradation caused by using the generic sampler framework, we compared the performance of a generic PT and an optimized PT, both fully implemented on the CPU. Figure 6.3 shows the number of PT samples per second for the optimized and generic PT samplers. Both implementations ran on a single core, including ray traversal. The figure shows that the overall performance suffers slightly from using the generic sampler.

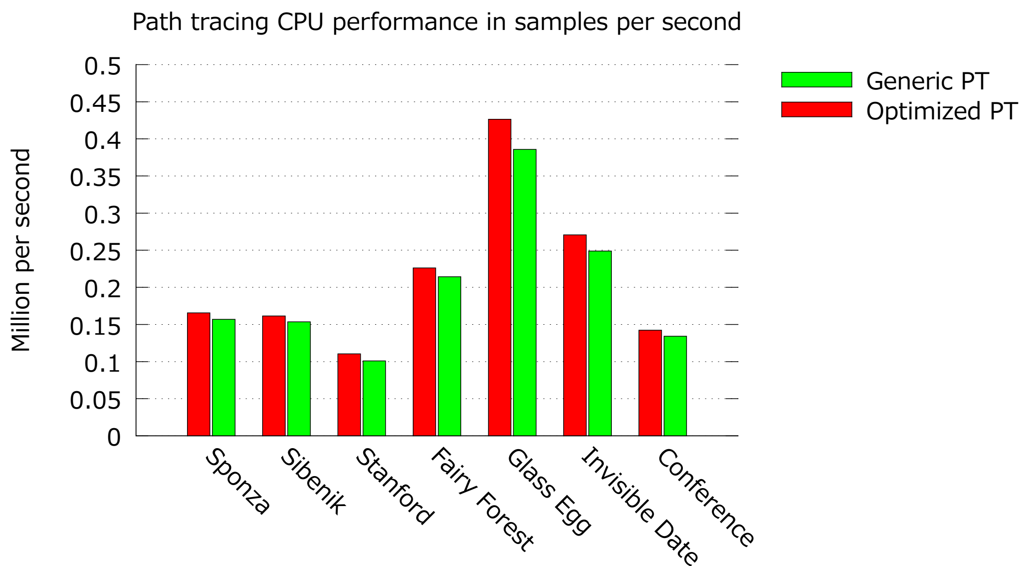


Figure 6.3: Optimized vs. generic PT sampler performance in samples per second on a single CPU core.

6.3.2 Hybrid PT performance

In this section, we will study the performance of the Hybrid PT. In order to realize high GPU performance and PCIe throughput, we used large batches of 256K samplers each. Figure 6.4 shows the hybrid PT performance for increasing number of CPU cores. The figure

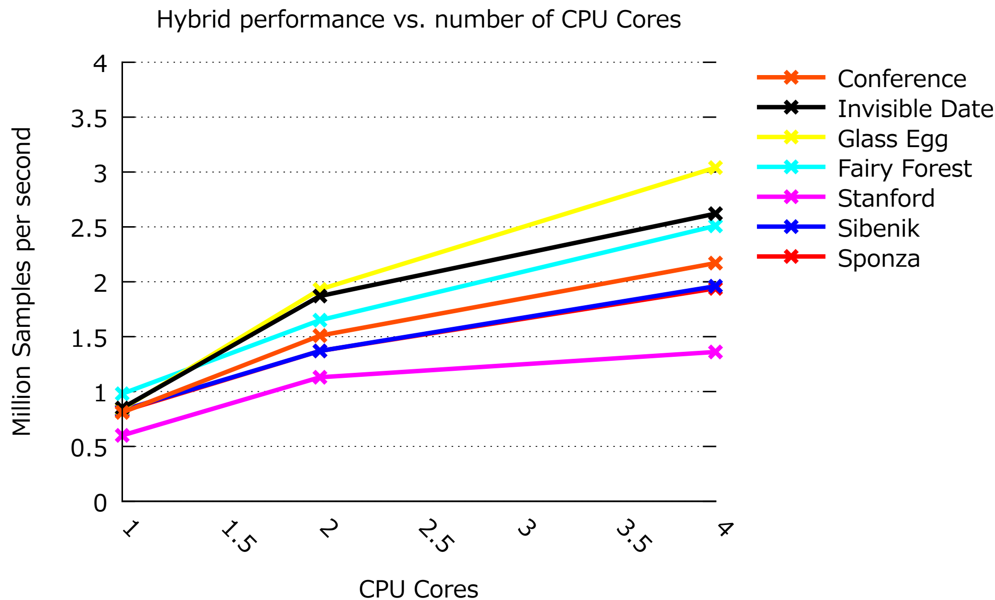


Figure 6.4: Hybrid tracer performance in samples per second depending on the number of CPU cores.

clearly shows that the performance increases sub linear in the number of cores, indicating a bottleneck, besides raw CPU processing power.

To further investigate the performance of the pipeline stages, figure 6.5 shows the tracer performance for parts of the pipeline. The measured performance for a part of the pipeline indicates the overall performance if this part would be the bottleneck. For convenience, performance is measured in rays per second. Besides measuring the performance for the whole pipeline (**CPU+PCIe+GPU**), we measure the performance for ray transfer and traversal (**PCIe+GPU**) and for ray traversal independently (**GPU**). During measurements, all four CPU cores are used for sampler advancing.

The figure clearly shows that advancing the samplers on the CPU constitutes a large bottleneck. The independent **GPU** ray traversal performance is an order of magnitude higher than the combined **CPU+PCIe+GPU** performance. For **PCIe+GPU**, the performance decreases somewhat and seems to reach a maximum performance at about 40M rays per second. This limit is caused by the maximum PCIe throughput and is as expected; in our implementation, storage for a single ray requires 28 bytes and the corresponding intersection result requires 16 byte. So, at 40M rays per second, this results in a total PCIe

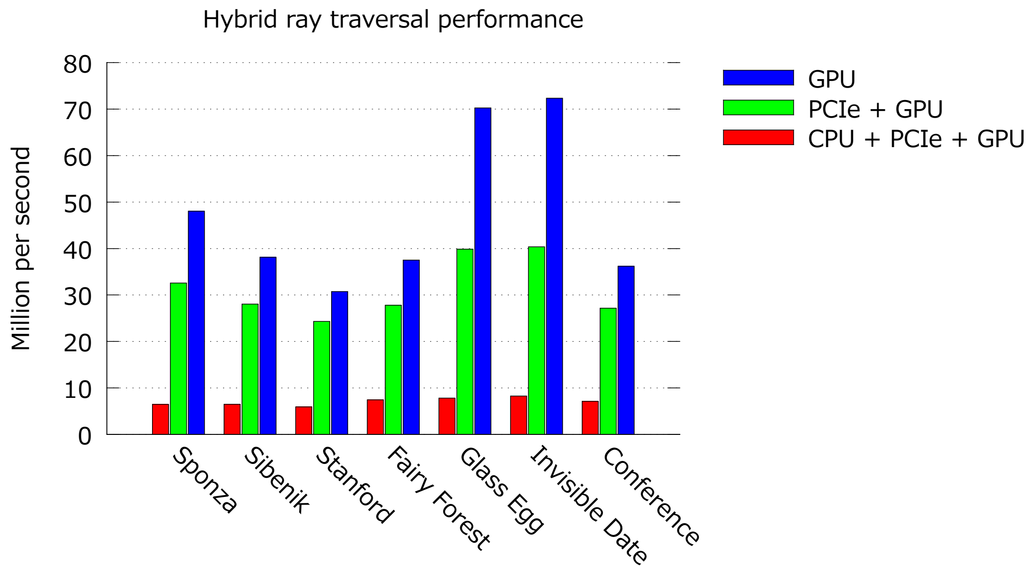


Figure 6.5: Hybrid tracer performance in MRays per second with 4 CPU threads for sampler advancing, the combined PCIe and GPU performance and the isolated GPU performance.

data traffic per second of 1120MB upstream and 640MB downstream; about 75% of the theoretical peak performance for our test platform.

Note that even when the independent **GPU** performance is less than the maximum of 40M rays/s, the **PCIe** stage reduces performance somewhat. This is because the rays produced by a PT sampler are highly divergent and GPU ray traversal of divergent rays is memory bandwidth bound [1]. Because the **GPU** and **PCIe** stages both compete for device memory bandwidth, they will degrade each others performance somewhat.

So far, measurements indicate the CPU as the main bottleneck in our architecture. Furthermore, we saw that sampler performance does not increase linear in the number of CPU cores. This indicates another bottleneck at the CPU side. Besides CPU computing power, the only resource of significance to our tracer is system memory. Further profiling revealed that system memory bandwidth forms the main bottleneck on our test platform.

There are two reasons for this. First of all, just as with device memory, PCIe data transfers take up system memory bandwidth. Second, as explained in section 6.2.3, running generic samplers in large batches causes a lot of cache trashing. Each cache miss results in a system memory transfer, so cache trashing increases the demands on system memory. For these reasons, the hybrid tracer does not scale well with the number of cores and far from utilizes all available GPU processing power.

6.3.3 Conclusion

On our test platform, the performance of the hybrid architecture is bound by the system memory bandwidth and is therefore not very scalable. Figure 6.5 gives an indication of the enormous amount of GPU performance that is wasted. Because the sampler is only implemented on the CPU, work balancing is difficult when system memory bandwidth becomes the bottleneck. This problem could be targeted by using much faster system memory. However, advancing a large stream of generic samplers remains cache unfriendly and therefore does not fit the CPU memory architecture very well. In contrast, the GPU is designed for processing large streams of data in parallel. Therefore, in the remainder of this thesis we will try to implement the sampler on the GPU, moving the complete rendering algorithm to the GPU and effectively eliminating the CPU and PCIe as possible bottlenecks. As an added advantage, this makes it much easier to fully utilize all available GPU's in the system with multiple GPU's.

In the following chapters, we will implement variations of several well known unbiased samplers on the GPU. The work flow of these samplers will closely resemble the generic sampler framework of iteratively advancing samplers and traversing corresponding output rays.

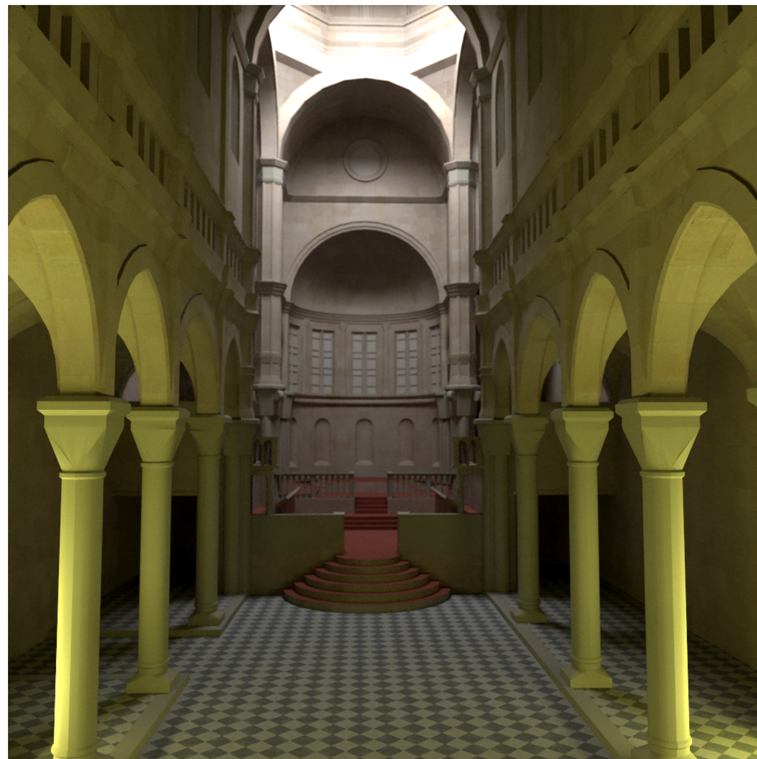


Figure 6.6: Sibenik cathedral rendered with hybrid PT.

Chapter 7

Path Tracer (PT)

7.1 Introduction

In the last chapter, we saw that for a hybrid CPU-GPU tracer, the CPU becomes a significant bottleneck. Therefore, in the remainder of this thesis, we will try to implement the complete sampling algorithm on the GPU. We will start with PT in this chapter, moving to BDPT and ERPT in later chapters. First, the PT algorithm is described as a flowchart. This flowchart is the basis for a two phased GPU PT implementation which forms a starting point for the GPU BDPT and EPRT implementations in later chapters. We will further show how to improve the performance of the GPU PT through stream compaction.

7.2 Two-Phase PT

7.2.1 PT Flowchart

The PT sampling algorithm can be described using the flowchart in figure 7.1. The sampling algorithm is split into processing steps. Ray traversal is performed between processing steps. A processing step may generate an output ray which must be traced before the next processing step starts. The tracing itself is left implicit and not incorporated in the flowchart.

As described in section 2.4, a PT sample is created by sampling a path starting from the eye and explicitly connecting each path vertex directly to a light source. This is usually done by repeatedly extending the eye path with one vertex, each time making a connection to a light source, until the eye path is terminated. The flowchart describes such a method, including path regeneration. So when a sample is complete, the sampler is regenerated and the generation of a new sample is started immediately.

During path extension, an extension ray is generated. This ray is then traced to find the next path vertex. When a new PT sample is started, the first extension ray is generated in the *Regenerate* step, according to the used camera model. If the path already contains vertices, the next extension ray is generated in the *Extend* step, sampling a random outgoing direction for the last path vertex based on its incoming direction and local BSDF. The extension ray is then traced and the intersection is handled in the *Intersect* step where the new eye path vertex is constructed. If no intersection was found, the sample terminates and a new

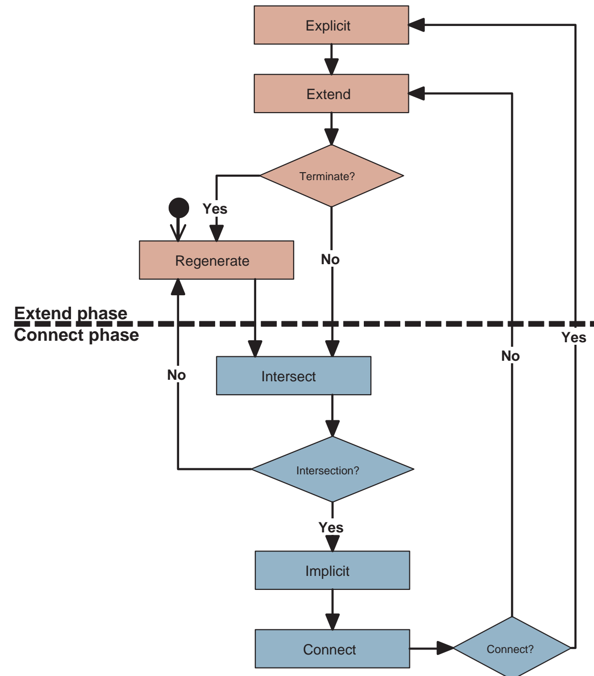


Figure 7.1: Two-Phase PT flowchart. The two phases are handled iteratively one after the other. Each time a thread changes its phase, it may output an optional output ray that will be traced before the next phase is started.

sample is regenerated. If the next vertex is a light source, an implicit light transport path is found and is handled in the *Implicit* step. In the *Connect* step, an explicit connection ray between eye vertex and light source is generated. Note that the *Connect?* condition does not check whether the connection is blocked, but only if a connection ray could be generated (for example, generation fails if the selected light source points away from the path vertex). If generating a connection ray was successful, the connection ray is traced. Only then, the *Explicit* step checks if the connection was obscured. If not, the connection is successful and a valid light transport path is found. When no connection ray was generated, the *Explicit* step is skipped and the eye path is immediately extended. During the *Extend* step, Russian roulette is performed to determine if the path must be terminated. When the path is terminated, the PT sample is complete and a new sample is generated.

7.2.2 Two-Phase PT

Note that the steps in the flowchart are divided into two phases: an **Extend** phase and a **Connect** phase. During the **Extend** phase, the extension ray is generated. During the **Connect** phase, an explicit connection ray is generated. After these rays are generated and traced, their results are handled in the opposite phases, causing the handling of the extension ray intersection to be done in the **Connect** phase and the handling of the connection ray

intersection in the **Extend** phase. At first sight, this might seem like a strange separation of steps in two phases, but it has an interesting property: Whenever a processing step generates an output ray, the next processing step lies in the opposite phase. Hence, rays only need tracing when the control flow passes a phase border. This property gives rise to *Two-Phase Path Tracing* (TPPT). In TPPT the two phases are executed one after the other. During phase execution, any number of steps belonging to this phase may be executed. After one phase is finished and just before the next phase starts, an optional output ray requires tracing. So TPPT repeatedly executes the two phases, one after the other, interleaved with optional ray tracing. This method lends itself for a flexible GPU PT implementation and forms the basis for the more advanced GPU tracers in later chapters.

7.3 GPU PT

In the GPU implementation of TPPT, many independent PT samplers are run in parallel. The tracer is divided in three kernels, one kernel for each phase and a ray traversal kernel. The ray traversal kernel is very similar to the one used in the hybrid tracer from chapter 6.

During each iteration, both phases are executed. Each phase is executed for all active samplers in parallel, so each GPU thread corresponds to a single PT sampler. After each phase, all output rays are traced using the ray traversal kernel, before the next phase starts. Hence, the ray traversal kernel is executed twice every iteration. The more iterations are performed, the better converged the end result will be. The intermediate images can be used to display progressive results. Note that it takes multiple iterations to form a single PT sample.

During phase execution, all threads execute independent samplers, therefore no explicit synchronization is required between threads in blocks. However, to obtain high GPU efficiency, threads in a warp should follow the same code path as much as possible. Note that in figure 7.1, steps are always visited in the same order and no step is ever executed twice during the same phase. Each thread simply executes all relevant steps in this fixed order (top-down in the flowchart), skipping any steps not required by the sampler. This way, all threads in a warp executing the same step will automatically run in parallel using SIMT. All threads skipping this step are idle until the step is complete. Only when all threads in the warp skip a certain step will the step be skipped by the warp altogether.

Each sampler requires storage for a single vertex (the last path vertex). It may also read the output ray from the last phase and write an optional output ray. To achieve high memory bandwidth, these rays and vertices are stored as a structure of arrays. Threads in a warp always handle samplers that are laid out alongside each other in memory. Because all threads in a warp execute the same step using SIMT, any memory access by these threads can be serviced using coalesced memory transactions, significantly increasing effective memory bandwidth.

Note that, because path regeneration is used to increase efficiency, TPPT also requires a cool down phase to turn the consistent progressive results into an unbiased result.

7.3.1 Pixel allocation and measurements

When a sampler is regenerated, a new image pixel must be selected from which to start the next sample. Furthermore, methods for measuring the algorithm's progress and performance are required. Implementing these tasks is not trivial in a highly parallel environment. We implemented these tasks using atomic instructions at a low frequency. Each sampler locally accumulates its measurement counters. Only once every few iterations, a warp scans the local counters of its samplers and adds the contribution of all threads in the warp to the global measurement counters using atomic instructions. If the frequency of global accumulation is chosen low enough (every 10 or more iterations), this does not significantly impact the overall performance.

A similar method is used for pixel allocation. Instead of globally allocating a single new pixel when a sampler is regenerated, n new pixels are allocated using atomic instructions only every n regenerations, reducing the number of atomic instructions.

7.4 Stream compaction

The TPPT implementation from last section forms a starting point for the SBDPT and ERPT implementations of later chapters. In this section, we will further improve on the TPPT implementation using stream compaction and show that merging the two phases results in a more efficient GPU PT implementation, called *Sample Stream PT*(SSPT).

7.4.1 Ray output stream compaction

In the TPPT implementation, all samplers may output an optional ray after each phase. Because all samplers are processed in parallel, these output rays are also generated in parallel, therefore it is not possible to sequentially pack all output rays in a compact stream of rays. A simple solution is to have an output buffer contain enough space to hold exactly as many rays as there are samplers. After a phase, each sampler writes its output ray at its corresponding location in the output buffer. Because ray output is optional, each sampler has to output an *activation flag* indicating whether a ray has been output or not. The ray tracing kernel only needs to trace rays having their corresponding flag raised.

This means that during ray tracing, when a GPU warp loads a continuous batch of 32 rays from the output buffer, only a subset of the rays require actual tracing. To handle this, either new rays need to be loaded, of which again only a subset is active, or some threads in the warp are disabled during the tracing of the rays. The first solution seems fairly complex but the second reduces the GPU efficiency as some threads in the warp will become idle.

Compaction scan

A possible solution is to use a compaction kernel to compact the output buffer. After each phase, a scan (parallel prefix sum) is applied to the activation flags, computing the number of active rays preceding each ray in the buffer. This number is then used to compact the output buffer into a continuous stream of rays. Compacting the actual ray stream would require reading and writing of all rays, so instead of packing the ray buffer itself, a continuous

stream of indices to active rays in the original buffer is built. During ray traversal, each warp reads 32 ray indices which are dereferenced to obtain 32 active rays. Although this increases the GPU efficiency during ray traversal, running a full scan requires several kernel executions and often offsets most performance gains during ray traversal.

Another problem is that, because the output stream contains many inactive rays, coalesced memory access becomes less efficient, reducing effective memory bandwidth (see section 3.3 for an explanation on coalesced memory access). This is most pronounced for CUDA architecture models 1.0 and 1.1, due to their strong requirements for coalesced memory transfers. Neighboring indices in the packed index stream do not necessarily point to rays that are stored continuous in memory. This results in mostly non-coalesced memory transactions during ray access, significantly reducing the memory bandwidth. This problem is reduced for newer CUDA models, as the hardware serves each memory access in the least amount of coalesced memory transactions possible. However, the total amount of memory accessed exceeds that which is needed due to inactive rays that are unnecessarily loaded by hardware during coalesced memory transactions (see picture 3.3).

Immediate packing

We use a different method, which makes use of a parallel scan per block and atomic instructions. Because CUDA 1.0 does not support atomics, this method only works for architectures 1.1 and later. Instead of packing the output rays after each phase, rays are packed before they are written to the buffer during each phase. A single counter is used to keep track of the number of rays written to the output buffer during a phase. The counter is initialized to zero. Each time a batch of rays is written to the buffer, the counter is increased by the size of this batch, effectively allocating buffer space for the batch. Atomic instructions on the same memory are serialized by the hardware, so it is important to keep atomics to a minimum. As threads within a block can communicate efficiently through shared memory and explicit synchronization, all output rays of a single block are first locally packed using a parallel scan just before actual output. This results in a single, large batch of rays per block. Again, instead of actually packing the rays, a list of relative batch indices is generated. Then, buffer space is allocated using only a single atomic instruction per block. Using this global space and the local batch indices, each thread writes its output ray to the buffer. Using this method, the final ouput buffer is one continuous stream of active rays. Because the compaction is performed during each phase, no extra kernels are executed for this method, significantly increasing performance. Note that each sampler must store the global index for its output ray, so that it can access the intersection results during the next phase.

With a little work, memory access is also handled efficiently for all CUDA architectures because each warp accesses only batches of active rays stored in continuous memory. A warp (half-warp) never accesses more than 32 (16) continuous rays during memory access. On CUDA architecture 1.2 and later, each memory access is therefore served with at most 2 (4) coalesced memory transfers. The extra transfer is needed because each coalesced memory access can only access one aligned memory line while the accessed rays may cross one memory line boundary. Because no inactive rays are present in the buffer, no unneces-

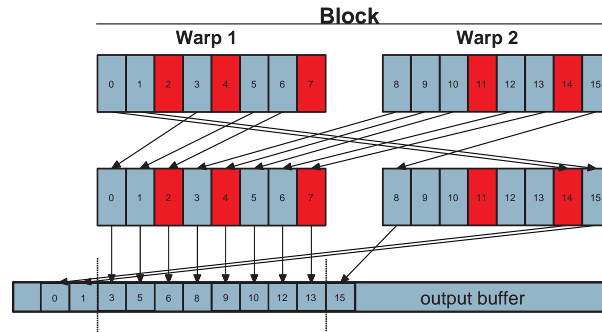


Figure 7.2: Rearranging output rays between threads in a block using shared memory achieves coalesced ray output on CUDA 1.1 architecture. Rays are rearranged to respect alignment requirements (dotted lines) and coalesced access requirements, before they are written to the buffer.

sary data is loaded by the hardware. For efficient memory access on CUDA architecture 1.1, a little extra effort is required. To allow coalesced memory access, the accessed memory must not only be continuous and aligned, but the threads must also access this continuous memory in a predefined order (see section 3.3). To achieve this, the output rays in each block are rearranged through shared memory. Then the rays are written by different threads in the block to allow coalesced memory transfers (see figure 7.2). For a block trying to store n rays, this method requires no more than $\lceil \frac{n}{16} \rceil + 1$ coalesced memory transfers. The performance gain due to coalesced memory access greatly exceeds the cost of rearranging the rays through shared memory.

7.4.2 Sampler Stream PT

The output compaction method from last section is also applicable to the tracers in later chapters. In this section, we will use stream compaction to further improve the performance of the GPU PT algorithm. This method however only applies to the GPU PT. The improvements reduce the required memory bandwidth and take advantage of higher ray tracing performance for coherent rays. We will refer to the improved PT as a *Sample Stream PT* (SSPT).

The first change is to partially merge the **Extend** and **Connect** phases into a large **Extend** phase. Instead of generating only a single optional ray per **Extend** phase, we will generate two optional rays, an optional connection ray and an optional extension ray. In addition to the **Extend** and simplified **Connect** phases a new **Generate** phase is added. Each phase still corresponds to a single kernel. Figure 7.3 shows what the flowchart of this tracer looks like in comparison to the two phase PT.

Each sampler starts in the **Generate** phase, generating the primary path ray starting at the eye. When the sampler is terminated, no new sample is regenerated. We will explain this shortly. During the **Extend** phase, the next path vertex is constructed. Then a possible connection to the light source is made and finally an extension ray is generated. All ex-

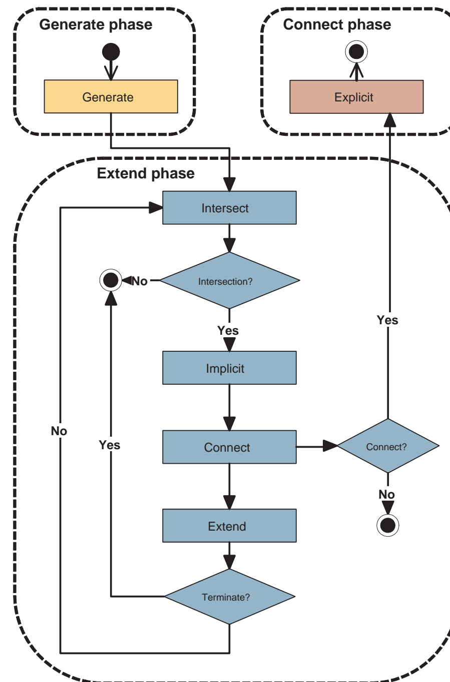


Figure 7.3: Flowchart for SSPT. During the extend phase, an optional connection ray and extension ray are generated. Note that the flow of control is split in the *connect* step. Only if a connection ray is generated is a **Connect** phase executed for this sampler. The sampler starts by generating a primary ray in the **Generate** phase. No path regeneration is used.

tension and regenerated rays are traced just before the **Extend** phase, all connection rays are traced just before the **Connect** phase. During each iteration, the **Generate**, **Extend** and **Connect** phases are executed, in that order, for all applicable samplers only.

In TPPT, the path vertex is constructed in the **Connect** phase, stored in global memory and read again during the **Extend** phase to generate the extension ray. By merging the **Extend** and **Connect** phase, it is no longer necessary to store and load the path vertex to/from memory, reducing the required memory bandwidth for the algorithm.

The most important distinction between TPPT and SSPT is that not only the output rays are compacted before output, but the actual stream of samplers itself is compacted as well (see figure 7.4). During the **Extend** phase, all sampler data is modified, so all data is once read and then written by the kernel. Instead of writing back the sampler data to the same location it was read from, the sampler data is compacted before output, removing all terminated samplers from the stream. This is the reason why samplers do not regenerate, terminated samplers are removed from the stream instead. This means that the stream of samplers becomes shorter after each iteration. Therefore, at the start of each iteration, new samplers are generated at the end of the stream by executing the **Generate** phase. This way the total number of active samplers remains constant. The most important advantage of this method is that it can take advantage of primary ray coherence. Because all regenerated

samplers are placed at the end of the stream, their primary rays will be traced together. These primary rays all have similar origin and direction which is called primary ray coherence [54]. Coherent rays will often visit similar parts of the scene. When coherent rays are traced together by different threads in a GPU warp, average SIMT efficiency is increased and GPU caches become effective, this results in a significant increase in ray tracing performance [1]. To obtain good primary ray coherence within warps, samplers should be regenerated using 4x4 tiles or along a **Z**-curve on the image plane [39]. Besides primary ray coherence, any secondary ray coherence due to specular reflection and small light sources is also exploited by this method: when coherent primary rays all hit the same mirror, the coherent secondary reflection rays are compacted in the output stream. Hence, the coherence between samplers in the stream is preserved. The same holds for connection rays. Therefore, SSPT also performs well for regular Whitted style ray tracing.

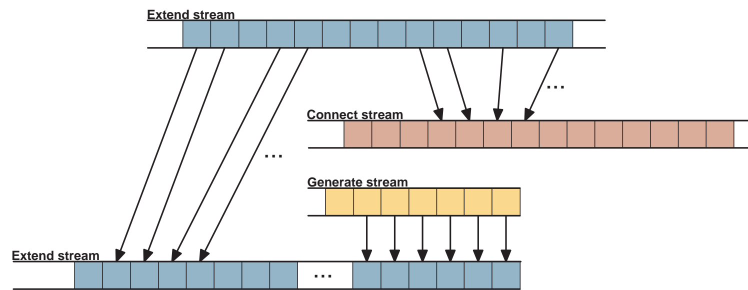


Figure 7.4: In SSPT, each **Extend** phase takes a single **Extend** stream as input and generates two compact output streams, a new **Extend** stream and a **Connect** stream. The **Connect** stream contains all connection rays and corresponding connection data. The **Extend** stream contains all extension rays and corresponding sampler data. All terminated samplers are regenerated at the end of the **Extend** stream.

Note that in the *connect* step, the flow of control is split. This indicates that the **Connect** phase is only executed for all samplers that generated a connection ray. This is also achieved through stream compaction. When a connection is made during the **Extend** phase, the energy traveling along the corresponding light transport path is computed immediately. This energy is then stored with the connection ray as well as the pixel the path contributes to. This way, the connection ray stream contains enough information to contribute any explicit path energy found, without having to access the corresponding sampler. Because the connection ray stream is compacted, by executing the **Connect** phase for all connection rays in the connection stream, the phase is effectively executed only for samplers that actually generated a connection ray. This further increases the GPU efficiency during the **Connect** phase.

7.5 Results

In this section we will present the results of our implementation of the TPPT and SSPT methods. We will first asses the SIMT efficiency of both algorithms and then study the

performance of our implementation.

7.5.1 SIMT Efficiency

The GPU performance of the implementation is largely determined by the SIMT efficiency of the algorithm. To give some insight in the SIMT efficiency of the PT implementations, we measured the average efficiency and occurrence of algorithmic steps during an iteration. The occurrence of a step is the percentage of warps that execute this step during an average iteration. Note that a warp skips a step only when all threads in the warp skip the step. The efficiency of a step is the average percentage of threads in a warp that execute the step in SIMT whenever the step occurs. Tables 7.1 and 7.2 show the efficiency and occurrence of the algorithmic steps of TPPT and SSPT tracers. Note that because the PT algorithm is stochastic, the occurrence percentage corresponds to the occurrence probability.

Results												
	Regenerate		**Extend**		**Intersect**		**Explicit**		**Implicit**		**Connect**	
	Ef%	Oc%	Ef%	Oc%	Ef%	Oc%	Ef%	Oc%	Ef%	Oc%	Ef%	Oc%
SPONZA	51	100	99	100	100	100	78	100	4	31	99	100
SIBENIK	50	100	99	100	100	99	72	100	3	7	99	99
STANFORD	33	100	99	100	100	100	48	100	3	3	99	100
FAIRY FOREST	58	100	82	100	100	99	77	100	3	11	82	99
GLASS EGG	51	100	96	100	100	99	34	100	3	0	96	99
INVISIBLE DATE	49	99	99	99	100	100	59	99	3	0	99	100
CONFERENCE	50	100	99	100	100	100	69	100	3	12	99	100

Table 7.1: SIMT efficiency and occurrence of algorithm steps in an average TPPT iteration.

We start with some observations on TPPT. Except for the **Implicit** step, all steps occur with high probability. Implicit paths occur in general with such low probability that the probability of having at least one implicit path in a warp is still relatively small for most scenes. Scenes with larger area-lights (SPONZA, CONFERENCE) have a larger probability of finding implicit paths, therefore having a higher **Implicit** occurrence probability.

Regenerate has around 50% percent efficiency for most scenes, due to Russian roulette terminating about half the paths with each iteration. Because the STANFORD scene has a lot of specular materials, paths are less likely to terminate, reducing **Regenerate** efficiency.

The efficiency for the **Extend**, **Intersect** and **Connect** steps is close to 100%, except for the FAIRY FOREST scene. This is because the FAIRY FOREST is an outdoor scene where many rays do not hit any geometry. Whenever an extension ray does not hit any geometry, the **Extend** and **Connect** steps are not performed and the path is regenerated. This also explains the slight increase in **Regenerate** efficiency for the FAIRY FOREST.

Finally, the **Explicit** efficiency varies significantly with the scenes. The **Explicit** step is executed each time an explicit connection is created. Whether a connection is created is very scene dependent. For example, the STANFORD scene has a lot of specular materials which fail to make an explicit connection, reducing the **Explicit** efficiency. Furthermore,

most geometry of the GLASS EGG scene lies behind a bright light source, reducing the number of created connections and thereby the **Explicit** efficiency.

Results												
	Regenerate		**Extend**		**Intersect**		**Explicit**		**Implicit**		**Connect**	
	Ef%	Oc%	Ef%	Oc%	Ef%	Oc%	Ef%	Oc%	Ef%	Oc%	Ef%	Oc%
SPONZA	100	52	96	99	100	99	100	78	8	18	96	99
SIBENIK	100	50	99	99	100	99	100	73	3	6	99	99
STANFORD	100	28	99	99	99	99	100	40	3	3	76	86
FAIRY FOREST	100	59	82	99	100	99	100	78	4	9	82	99
GLASS EGG	100	50	96	99	100	99	100	34	3	0	95	99
INVISIBLE DATE	100	43	99	99	99	99	100	52	3	0	99	99
CONFERENCE	100	50	99	99	100	99	100	70	5	8	99	99

Table 7.2: SIMT efficiency and occurrence of algorithm steps in an average SSPT iteration.

Turning to the SSPT tracer, there are a few key differences. To better understand these, note that the probabilities for executing a step for a sampler are the same for both TPPT and SSPT as they implement the same sampler. Therefore, decreased occurrence must always mean increased efficiency. Most notable, in SSPT the **Explicit** and **Regenerate** steps both have 100% efficiency but reduced occurrence. Compared to TPPT, it seems that the occurrence and efficiency measures are exchanged for these steps. This is actually more or less true. Instead of regenerating samplers in place, the SSPT generates new samplers at the end of the stream. Similar, SSPT generates a compact input stream for the **Explicit** step. Because both these steps operate on all samplers in a continuous stream, maximum efficiency is realized. Apart from this, the efficiency and occurrence for both algorithms are virtually the same, except for a slight increase in **Implicit** step efficiency for the SSPT tracer due to more coherent ray traversal.

These efficiency results show that the TPPT and SSPT methods both achieve high efficiency for most steps, which will result in high SIMT efficiency when executing these steps on the GPU, increasing performance.

7.5.2 Performance

In this section, we present the performance of our GPU path tracers. Figure 7.5 shows the overall performance in samples and rays per second for TPPT, TPPT with output stream compaction and SSPT. The performance is somewhat higher for the smaller scenes, but overall the performance does not depend heavily on the scene[1].

Packing the output rays for TPPT gives a slight increase in performance for all scenes. However, packing the whole stream as in SSPT shows a very significant increase in performance. Interestingly, its performance is structurally higher than the ray traversal speed as measured in figure 6.5. This is because the SSPT exploits primary ray coherence, resulting

[1]This is as expected, as ray traversal time only increases logarithmically with the number of scene primitives.

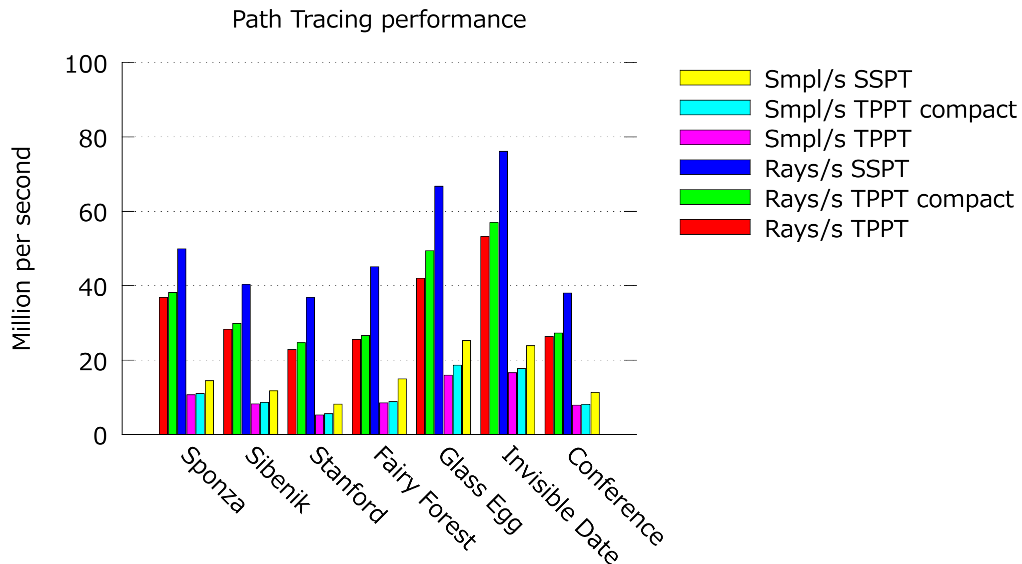


Figure 7.5: General PT performance in rays and PT samples per second for TPPT with and without output stream compaction and the SSPT.

in increased ray traversal performance. Figure 7.6 shows the ray traversal performance for extension rays in TPPT with output stream compaction and SSPT. Ray traversal is up to 50% faster for SSPT due to ray coherence. There is also a measurable increase in traversal speed for connection rays, but the increase is less pronounced compared to extension rays. The reason we presented the TPPT sampler in this chapter, even though SSPT is significantly faster, is because TPPT forms the bases for the samplers in the next two chapters.

Figures 7.7 and 7.8 show a time partition of algorithmic phases for respectively an average TPPT and SSPT iteration, including ray traversal. The time partitions for both algorithms look very much alike. The most notable difference is that the **Connect** phase in SSPT takes a lot less time because most of its work has been moved to the **Extend** phase. The figures show that the vast majority of iteration time is spent at ray traversal, while on average about 18% of the iteration time is spent advancing the samplers. These results show that the GPU samplers are not significantly less efficient than their CPU counterparts. Note that about 5% of the time is not partitioned. This time is spent on iteration overhead such as progressive display.

The performance of the GPU implementations depends on the size of the sampler stream. If the stream is too short, there will not be enough parallelism to fully utilize the GPU. In particular, the GPU in our test platform contains 16 SM's. On each SM, up to 16 warps are executed concurrently using hyper threading. Therefore, the stream must at least contain 8192 threads or 256 warps to be fully utilized. However, to allow for effective load balancing, the GPU should be filled several times. To give an indication of

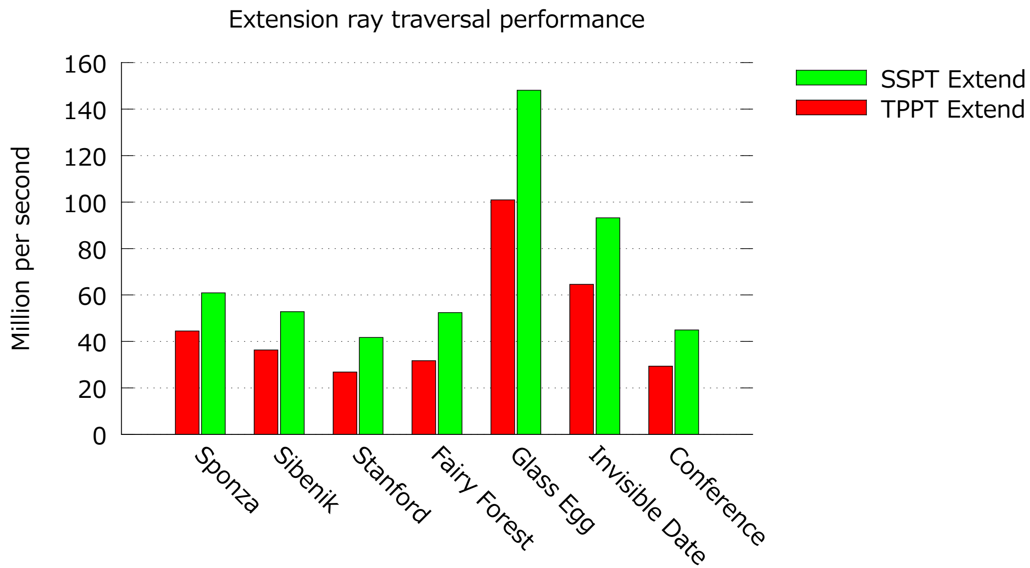


Figure 7.6: Extension ray traversal for TPPT and SSPT (sampling not included in figures).

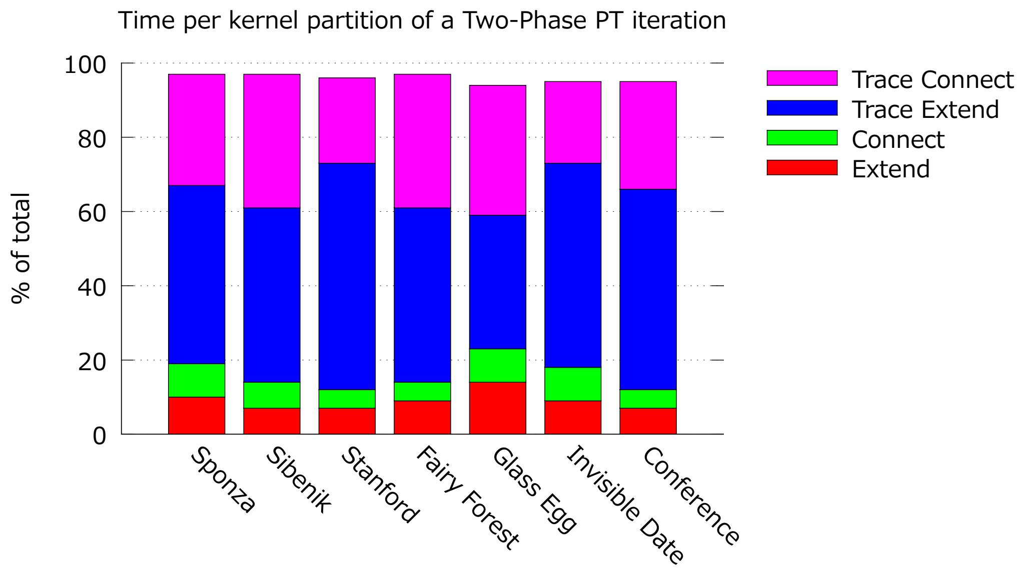


Figure 7.7: TPPT procentual time partition of an iteration.

how performance depends on stream size, figure 7.9 and 7.10 show the performance of the TPPT and SSPT algorithms for increasing stream sizes. Note that the stream size increases

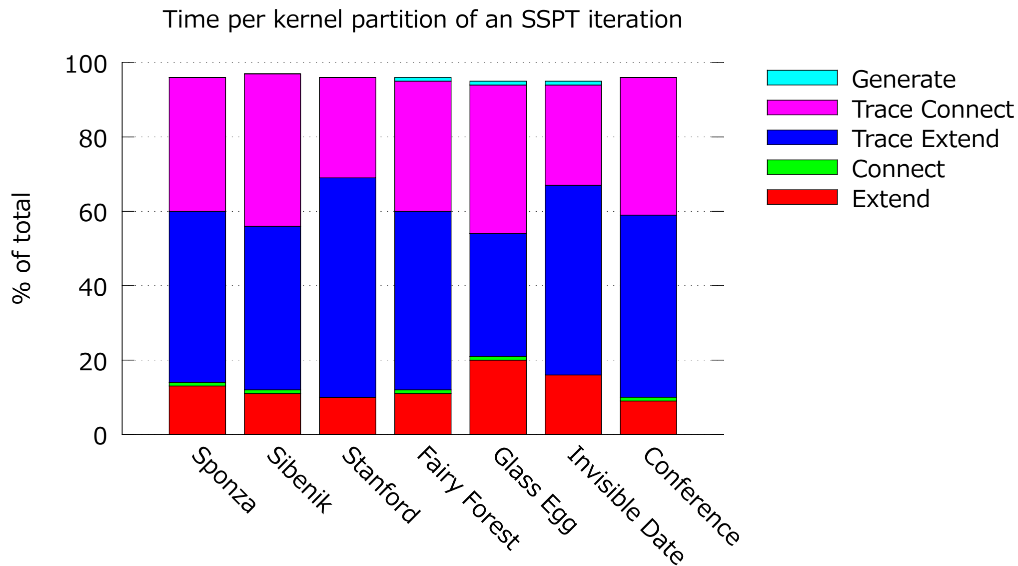


Figure 7.8: SSPT procentual time partition of an iteration.

exponentially. The figures show that performance increases as the stream size increases, but that it eventually stabilizes. In our experiments we used a stream size of 8192 warps, filling the GPU 32 times. In later chapters we will not repeat these graphs for the new samplers as their shape remains virtually the same as for PTTP and SSPT. In practice, one should choose the largest stream size that still fits in memory. This depends on the scene size and available GPU memory.

Finally, table 7.3 shows the memory footprint of both methods for 256 warps. Note that TPPT has about twice the memory consumption as SSPT. This is because SSPT requires much less persistent sampler state. Where TPPT has to store extra information for computations during the *Connect* phase, the SSPT does most of the work during the *Extend* phase, requiring only little persistent data per explicit connection.

	Memory usage
TPPT	2560 Kb
SSPT	1216 Kb

Table 7.3: Memory usage of both PT tracers for sampler and ray storage per 256 warps.

The results from this section have shown that especially the SSPT algorithm results in an efficient GPU path tracer, having an acceptable memory footprint and high performance. However, in chapter 2 we saw that much better samplers than PT exist, resulting in less variance in the image estimate. In the next two chapters, we will show how the TPPT method can be extended to implement the more advanced BDPT sampler and ERPT sampler.

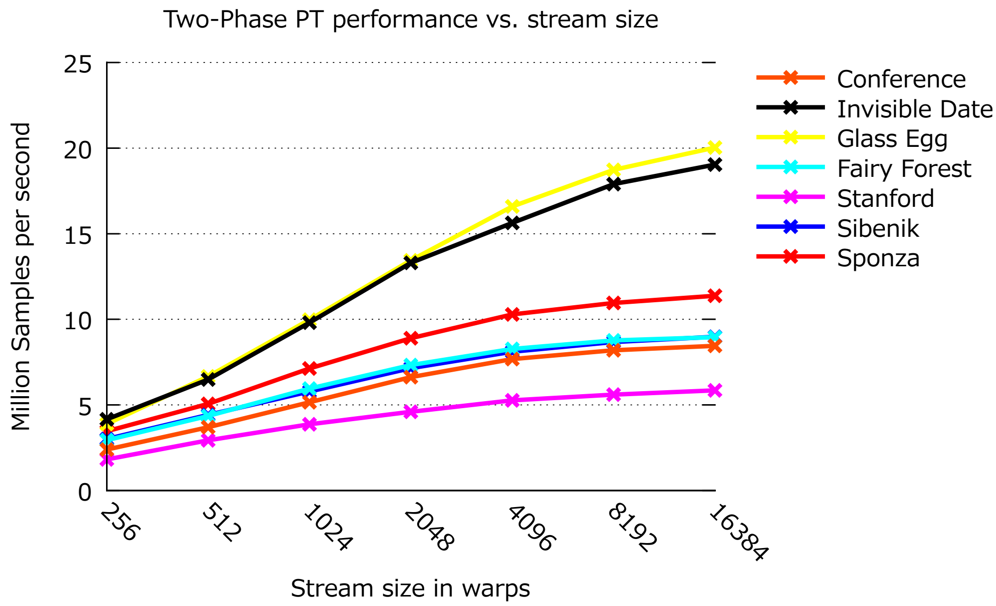


Figure 7.9: TPPT performance in samples per second as a function of the sampler stream size in warps.

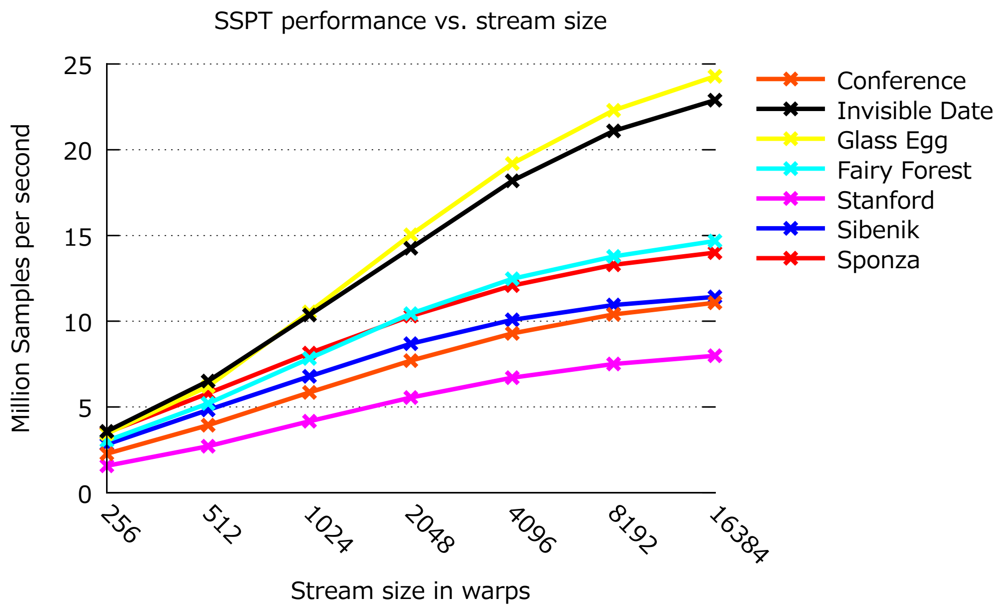


Figure 7.10: SSPT performance in samples per second as a function of the sampler stream size in warps.

Figure 7.11: Conference room rendered with SSPT.

Chapter 8

Streaming BiDirectional Path Tracer (SBDPT)

In this chapter, we will present the SBDPT algorithm, a streaming BDPT algorithm for the GPU. In section 8.1, we give an introduction to the problem and the SBDPT algorithm. In section 8.2, a recursive formulation for MIS is derived which is used in section 8.3 where the SBDPT algorithm and its GPU implementation are presented.

8.1 Introduction

In this section, we explain why BDPT is not well suited for a GPU implementation. We will further discuss a partial solution as presented by Novak [40], before giving an overview of our own solution: the SBDPT algorithm.

8.1.1 Problems with GPU BDPT

In the original BDPT algorithm, both a complete eye path and light path are constructed first. All eye vertices are then connected to all light vertices to obtain a collection of light transport paths. These light transport paths are weighted using multiple importance sampling, and their contribution is added to the image. Directly implementing this algorithm on the GPU proves to be challenging for several reasons: first, unlike normal path tracing, the algorithm requires the full storage of both the light and eye path. As the length of these paths are not yet known in advance, either a dynamic allocation scheme is required, or enough memory must be allocated per sample beforehand to reach some predetermined maximum path length. Having a maximum path length causes bias in the resulting image. By setting a large maximum path length, this bias might often be unnoticeable small, but as most paths would be significantly shorter than the maximum length, a lot of memory would be waisted.

Another major challenge is how to achieve high GPU performance. As seen earlier, for high GPU performance we require both coalesced memory access and coherent code execution within GPU warps. Assuming each GPU thread handles a single bidirectional sample, these goals turn out to be difficult to achieve. The problems are caused mainly by

the difference in path length between samples. It can take significantly longer for one thread to finish its light and/or eye path than for other threads in the same warp. This problem reappears during the connection of the light and eye paths. The number of connections is proportional to the lengths of both the eye and light paths. Hence, some threads will have to make many more connections than others. This problem is amplified when computing correct MIS weights, requiring iteration over all eye and light vertices for each connection. So, in a trivial implementation some threads will have to wait many iterations for others to finish, resulting in low GPU efficiency and threads within a warp will have very different memory access patterns, resulting in a low effective memory bandwidth.

8.1.2 Per warp Russian roulette

A simple and attractive solution might be to perform Russian roulette per warp, instead of per thread. So, during path construction, either all threads in a warp extend their path, or they all terminate, guaranteeing that all eye resp. light paths in a warp have the same length. This solves the GPU efficiency problems; all threads will make equally many connections and have vary similar memory access patterns, allowing for high efficiency and memory bandwidth. This is exactly the solution as presented by Novak [40], albeit their implementation did not contain MIS. Although achieving high performance, this solution has two significant drawbacks: the paths still need to be fully stored in memory, resulting in a maximum path length, and performing Russian roulette per warp does not allow for effective importance sampling w.r.t. path length. When applying per-warp Russian roulette, a fixed termination probability must be used instead of a probability that depends on the local material and proposed extension direction. This makes importance sampling using Russian roulette impossible. For simple scenes with relatively uniform diffuse materials, this poses no problem, but when materials start to vary significantly, containing for example highly reflective materials, importance sampled Russian roulette is much more effective. Bidirectional sampling with MIS receives its strength from optimally combining different importance sampling techniques, each effective in different situations. By using per-warp Russian roulette, the difference between these importance sampling techniques is reduced, thereby reducing the strength of their combination.

8.1.3 Algorithm overview

We propose a solution that does not restrict path lengths, has a fixed memory footprint per bidirectional sample and allows for per-warp Russian roulette. First, we will give an alternative method for computing MIS weights. Veach provided a formulation for constructing provable optimal MIS weights [50]. This construction requires iteration over all vertices in the path for each connection. We propose an alternative method for computing these optimal weights. By recursively computing two extra quantities in each vertex of the light and eye path, we show that for each connection the optimal weights can be computed using a fixed amount of computations and only data from the two vertices directly involved in the connection, independent of the actual path length. Using this, we slightly alter the BDPT algorithm to achieve high GPU efficiency. Instead of connecting a single eye path to a sin-

gle light path, a new eye path is generated for each light vertex. Using this method, in each iteration either the light- or eye path is extended by a single vertex and a single connection is made between the last vertices on both paths, resulting in high GPU efficiency. Because only the last vertices are involved in each connection and MIS weight computations only requires these vertices, we no longer have to store all vertices on the path, just the last vertices suffices. This significantly reduces the memory usage and at the same time allows for high effective memory bandwidth.

8.2 Recursive Multiple Importance Sampling

Whenever an eye path and light path are connected, we need to compute the weight for this newly constructed sample. Veach proposed a method for constructing weights according to the power heuristic [50]. Their method iterates over all vertices in the path to construct the probabilities needed to calculate this weight. We propose a different schema for computing the power heuristic weights. During the construction of the eye and light paths, we recursively compute two quantities in each path vertex. Using these quantities, the weights can be constructed more locally, without having to iterate over all vertices for each connection. This proves to be advantageous when connecting paths of varying lengths in a data parallel context, such as on the GPU.

8.2.1 Recursive weights construction

In this section, we will present our recursive computation schema for computing the power heuristic weights. We will give a more formal derivation in the next section. Remember from section 2.7 that applying the power heuristic to a light transport path $\mathbf{X}_s$ of length k, sampled using an eye path of length s, results in a MIS weight function of

$$w_s(\mathbf{X}_s) = \frac{\hat{p}_s(\mathbf{X}_s)^\beta}{\sum_{i=0}^{k} \hat{p}_i(\mathbf{X}_s)^\beta} \tag{8.1}$$

Our main concern is with the construction of the denominator $D(\mathbf{X}_s) = \sum_{i=0}^{k} \hat{p}_i(\mathbf{X}_s)^\beta$. The i'th term in $D(\mathbf{X}_s)$ represents the probability of sampling $\mathbf{X}_s$ using a sampling strategy with i eye vertices and $k-i$ light vertices. Based on common factors in the terms, we can split $D(\mathbf{X})$ into three parts, as shown in figure 8.1.

$$D(\mathbf{X}_s) = \sum_{i=0}^{s-1} \hat{p}_i(\mathbf{X}_s)^\beta + \hat{p}_s(\mathbf{X}_s)^\beta + \sum_{i=s+1}^{k} \hat{p}_i(\mathbf{X}_s)^\beta \tag{8.2}$$

1. The first term represents all sampling strategies with less than s eye vertices. These strategies all sample the vertices $x_k \cdots x_s$ from the light. Hence, the probability of sampling these vertices is a common factor between the terms within the summation. What is left for each term is the probability of sampling the remaining eye path vertices using the corresponding sampling strategy.

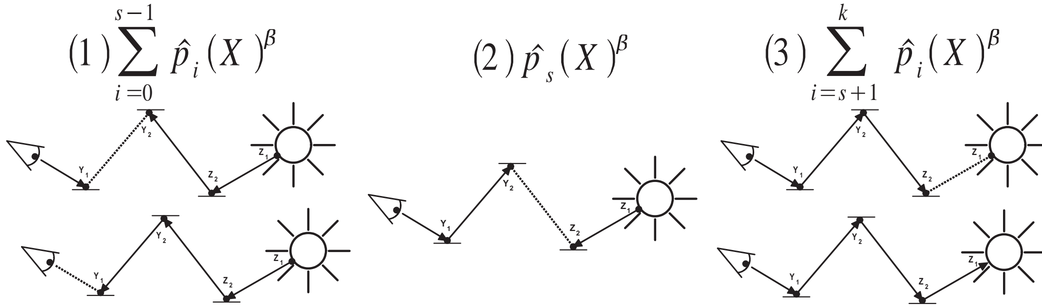


Figure 8.1: The weight's denominator $\sum_{i=0}^{k} \hat{p}_i(\mathbf{X})^\beta$ for a path $\mathbf{X}_s$ can be split into three parts: 1) All sampling strategies having the connection edge within the original eye path, 2) The sampling strategy with which $\mathbf{X}_s$ was sampled, and 3) all sampling strategies having the connection edge within the original light path. The figure shows its application to a path of length $k = 4$, sampled using $s = 2$ eye vertices.

2. This is the strategy with exactly s eye vertices. Hence, this is the strategy actually used to sample this path.

3. The last term represents all sampling strategies with more than s eye vertices. These strategies all sample the vertices $x_1 \cdots x_{s+1}$ from the eye. Hence, the probability of sampling these vertices is a common factor between the terms within the summation. What is left for each term is the probability of sampling the remaining light path vertices using the corresponding sampling strategy.

Remember that the vertex x_0 on the image plane is always sampled as part of the eye path and is left implicit. Note that, when factoring out the common factors in the first term, what is left is a summation over the probabilities of sampling the first $x_1 \cdots x_{s-1}$ vertices using any bidirectional sampling strategy. The same holds for the third term. If we are able to recursively construct these sums while tracing the eye/light paths and storing these sums within the path vertices, we no longer have to iterate over all path vertices when computing the weight. Therefore, we propose to compute two recursive quantities in each vertex x_i while constructing the eye and light paths:

- p_i: The probability with which the path $x_1 \cdots x_i$ was sampled so far. This quantity can be constructed recursively by multiplying p_{i-1} with the probability of generating the next vertex x_i on the path.

- d_i: The sum of the probabilities for sampling the path $x_1 \cdots x_i$ using any bidirectional sampling method, under the assumption that the first light path vertex x_i is given (with a probability of 1). Expressing this quantity recursively is slightly more involved. The sum of probabilities can be divided in two parts:

    1. The probability of sampling $x_1 \cdots x_{i-1}$ as an eye path.

2. The sum of probabilities of sampling at least the last two vertices $x_i x_{i-1}$ as a light path.

The first part simply equals p_{i-1}. The second part is the product of the probability for sampling x_{i-1} backward from x_i, and the sum of the probabilities for sampling the remaining $x_1 \cdots x_{i-2}$ vertices using any bidirectional sampling method, given light path vertex x_{i-1}. Hence, this can be expressed recursively as $d_{i-1} P_A (x_i \rightarrow x_{i-1})$.

When connecting a light and eye path, the weights can be constructed using only these quantities in the neighboring vertices of the connection edge. We will just state the recursive construction of these quantities here. In the next section, we will provide a more formal derivation. For the eye path, the quantities are labeled $p_i^{\mathbf{Y}}$ and $d_i^{\mathbf{Y}}$. For the light path, the quantities are labeled $p_i^{\mathbf{Z}}$ and $d_i^{\mathbf{Z}}$. Using the following recursive definitions, the quantities are computed during eye/light path construction:

$$\begin{aligned} p_0^Y &= 1 \\ p_i^Y &= p_{i-1}^Y P_A (y_{i-1} \rightarrow y_i)^\beta \end{aligned} \tag{8.3}$$

$$\begin{aligned} p_0^Z &= 1 \\ p_i^Z &= p_{i-1}^Z P_A (z_{i-1} \rightarrow z_i)^\beta \end{aligned} \tag{8.4}$$

$$\begin{aligned} d_0^Y &= 0 \\ d_i^Y &= p_{i-1}^Y + d_{i-1}^Y P_A (y_i \rightarrow y_{i-1})^\beta \end{aligned} \tag{8.5}$$

$$\begin{aligned} d_0^Z &= 0 \\ d_i^Z &= p_{i-1}^Z + d_{i-1}^Z P_A (z_i \rightarrow z_{i-1})^\beta \end{aligned} \tag{8.6}$$

Then, when connecting an eye path $\mathbf{Y}_s$ and a light path $\mathbf{Z}_t$ to form $\mathbf{X}$, the corresponding weights become:

$$\begin{aligned} \hat{p}_s (\mathbf{X})^\beta &= p_s^Y p_t^Z \\ D(\mathbf{X}) &= p_t^Z P_A (z_t \rightarrow y_s)^\beta d_s^Y \\ &\quad + p_s^Y p_t^Z \\ &\quad + p_s^Y P_A (y_s \rightarrow z_t)^\beta d_t^Z \\ w_s (\mathbf{X}) &= \frac{\hat{p}_s (\mathbf{X})^\beta}{D(\mathbf{X})} \end{aligned} \tag{8.7}$$

Note that during this construction, we consider the path $y_1 \cdots y_s z_t \cdots z_1$. Unless only perfect diffuse and perfect specular materials are used, to compute $P_A (y_s \rightarrow y_{s-1})$ the vertex z_t is also required (to determine the incoming direction). Because z_t is not known during the construction of the eye path, d_s^Y can only be computed when the actual connection is made. The same hold for d_t^Z.

After the light and eye paths are constructed, the amount of computations required to compute a weight is independent of the length of the path; Only quantities from the vertices

y_{s-1},y_s,z_t and z_{t-1} are required. This fixed amount of computations and memory requirements per connection will prove useful when implementing the BDPT on the GPU.

Appendix E shows how the recursive weight computations are inserted into the BDPT algorithm.

8.2.2 Formal derivation

In this section, we will give a more formal derivation of the recursive formulation for the power heuristic weights. Given a path $\mathbf{X} = x_1 ... x_k$, let $\mathbf{Y}^i$ be the prefix path $x_1 ... x_i$ and $\mathbf{Z}^j$ be the postfix path $x_k ... x_{k-j}$. From section 2.7 we know that the sampling probability $p\left(\mathbf{X}'\right)$ for some (partial) eye or light path $\mathbf{X}' = x_1 \cdots x_k$ equals

$$p\left(\mathbf{X}'\right) = \prod_{i=0}^{k-1} P_A\left(x_i \rightarrow x_{i+1}\right) \tag{8.8}$$

Furthermore, the probability $\hat{p}_s\left(\mathbf{X}\right)$ is the probability of sampling the path $\mathbf{X}$ by connecting the eye path $\mathbf{Y}^s$ of length s and the corresponding light path $\mathbf{Z}^t$ of length $t = k - s$. This probability equals

$$\hat{p}_s\left(\mathbf{X}\right) = p\left(\mathbf{Y}^s\right) p\left(\mathbf{Z}^t\right) = \left(\prod_{i=0}^{s-1} P_A\left(x_i \rightarrow x_{i+1}\right)\right) \left(\prod_{i=s+1}^{k} P_A\left(x_{i+1} \rightarrow x_i\right)\right) \tag{8.9}$$

Using these probabilities, the denominator for the weights equals

$$D(\mathbf{X}) = \sum_{i=0}^{k} \hat{p}_i\left(\mathbf{X}\right)^{\beta} \tag{8.10}$$

Sampling probabilities

Before going on with the proof, we derive a few useful equalities related to sampling probabilities. First, note that:

$$p\left(\mathbf{Y}^s\right) = \left(\prod_{i=0}^{s-1} P_A\left(x_i \rightarrow x_{i+1}\right)\right) = \hat{p}_s\left(\mathbf{Y}^s\right) \tag{8.11}$$

$$p\left(\mathbf{Z}^t\right) = \left(\prod_{i=k-t+1}^{k} P_A\left(x_{i+1} \rightarrow x_i\right)\right) = \left(\prod_{i=s+1}^{k} P_A\left(x_{i+1} \rightarrow x_i\right)\right) = \hat{p}_{k-s}\left(\mathbf{Z}^{k-s}\right) = \hat{p}_t\left(\mathbf{Z}^t\right) \tag{8.12}$$

Using these equalities, we can express the probability of sampling a bidirectional path in terms of the eye and light paths as follows:

$$\hat{p}_s\left(\mathbf{X}\right) = p\left(\mathbf{Y}^s\right) p\left(\mathbf{Z}^t\right) = \hat{p}_s\left(\mathbf{Y}^s\right) \hat{p}_t\left(\mathbf{Z}^t\right) \tag{8.13}$$

It is easy to see that for some $\mathbf{X}$, the probabilities $\hat{p}_m(\mathbf{X})$ and $\hat{p}_n(\mathbf{X})$ with $m \neq n$ often have common factors. Let us investigate these common factors a little further. For any

two probabilities $\hat{p}_m(\mathbf{X})$ and $\hat{p}_n(\mathbf{X})$ with $0 \leq m \leq n \leq k$, we can write the probabilities as follows:

$$\begin{aligned}\hat{p}_m(\mathbf{X}) &= \prod_{i=0}^{m-1} P_A(x_i \to x_{i+1}) \prod_{i=m+1}^{n} P_A(x_{i+1} \to x_i) \prod_{i=n+1}^{k} P_A(x_{i+1} \to x_i) \\ \hat{p}_n(\mathbf{X}) &= \prod_{i=0}^{m-1} P_A(x_i \to x_{i+1}) \prod_{i=m}^{n-1} P_A(x_i \to x_{i+1}) \prod_{i=n+1}^{k} P_A(x_{i+1} \to x_i)\end{aligned} \tag{8.14}$$

We can further express these probabilities in products of probabilities on a prefix and a postfix path.

$$\begin{aligned}\hat{p}_m(\mathbf{X}) &= \hat{p}_m(\mathbf{Y}^m) \left(\prod_{i=m+1}^{n} P_A(x_{i+1} \to x_i) \right) \hat{p}_{k-n}\left(\mathbf{Z}^{k-n}\right) \\ &= \hat{p}_m(\mathbf{Y}^n)\, \hat{p}_{k-n}\left(\mathbf{Z}^{k-n}\right) = \hat{p}_m(\mathbf{Y}^n)\, p\left(\mathbf{Z}^{k-n}\right) \\ &= \hat{p}_m(\mathbf{Y}^m)\, \hat{p}_{k-m}\left(\mathbf{Z}^{k-m}\right) = p(\mathbf{Y}^m)\, p\left(\mathbf{Z}^{k-m}\right)\end{aligned} \tag{8.15}$$

$$\begin{aligned}\hat{p}_n(\mathbf{X}) &= \hat{p}_m(\mathbf{Y}^m) \left(\prod_{i=m}^{n-1} P_A(x_i \to x_{i+1}) \right) \hat{p}_{k-n}\left(\mathbf{Z}^{k-n}\right) \\ &= \hat{p}_n(\mathbf{Y}^n)\, \hat{p}_{k-n}\left(\mathbf{Z}^{k-n}\right) = p(\mathbf{Y}^n)\, p\left(\mathbf{Z}^{k-n}\right) \\ &= \hat{p}_m(\mathbf{Y}^m)\, \hat{p}_{k-n}\left(\mathbf{Z}^{k-m}\right) = p(\mathbf{Y}^m)\, \hat{p}_{k-n}\left(\mathbf{Z}^{k-m}\right)\end{aligned} \tag{8.16}$$

When applying equation 8.9 to a sequence of prefix paths $\mathbf{Y}^i$, we can identify some recursive relations. Note that for $i > s > 1$ it holds that

$$\hat{p}_s\left(\mathbf{Y}^i\right) = \hat{p}_s\left(\mathbf{Y}^{i-1}\right) P_A(x_{i+1} \to x_i) \tag{8.17}$$

Furthermore, from equation 8.11, it is trivial that

$$p\left(\mathbf{Y}^i\right) = \hat{p}_i\left(\mathbf{Y}^i\right) = \hat{p}_{i-1}\left(\mathbf{Y}^{i-1}\right) P_A(x_{i-1} \to x_i) = p\left(\mathbf{Y}^{i-1}\right) P_A(x_{i-1} \to x_i) \tag{8.18}$$

Obviously, similar results hold for a sequence of postfix paths $\mathbf{Z}^i$.

Recursive balance heuristic

In this section, we will show how to construct $D(\mathbf{X})$. We will start by showing how to construct a related quantity $D'(\mathbf{X})$ recursively on the prefix path sequence $\mathbf{Y}^i$. $D'(\mathbf{X})$ is defined as

$$D'(\mathbf{Y}^i) = \sum_{j=0}^{i-1} \hat{p}_j\left(\mathbf{Y}^{i-1}\right)^{\beta} \tag{8.19}$$

We start with the base case; the empty prefix path $\mathbf{Y}^0$.

$$D'(\mathbf{Y}^0) = 0 \tag{8.20}$$

Turning to the general case $D'(\mathbf{Y}^i)$ with $i > 0$. We can rewrite $D'(\mathbf{Y}^i)$ using equations 8.17 and 8.18:

$$\begin{aligned} D'(\mathbf{Y}^i) &= \hat{p}_{i-1}\left(\mathbf{Y}^{i-1}\right)^{\beta} + \sum_{j=0}^{i-2} \hat{p}_j\left(\mathbf{Y}^{i-1}\right)^{\beta} \\ &= p\left(\mathbf{Y}^{i-1}\right)^{\beta} + \sum_{j=0}^{i-2} \hat{p}_j\left(\mathbf{Y}^{i-2}\right)^{\beta} P_A\left(x_i \to x_{i-1}\right)^{\beta} \\ &= p\left(\mathbf{Y}^{i-1}\right)^{\beta} + D'\left(\mathbf{Y}^{i-1}\right) P_A\left(x_i \to x_{i-1}\right)^{\beta} \end{aligned} \tag{8.21}$$

We now have a recursive expression for $D'(\mathbf{Y}^i)$. What is left is a recursive expression for $p\left(\mathbf{Y}^i\right)^{\beta}$. Using equation 8.11, this turns out to be quite trivial:

$$\begin{aligned} p\left(\mathbf{Y}^0\right)^{\beta} &= 1 \\ p\left(\mathbf{Y}^i\right)^{\beta} &= p\left(\mathbf{Y}^{i-1}\right)^{\beta} P_A\left(x_{i-1} \to x_i\right)^{\beta} \end{aligned} \tag{8.22}$$

Using these recursive definitions, we can construct $D(\mathbf{X})$ in a bidirectional way. First, let us split the path $\mathbf{X}$ in a prefix path $\mathbf{Y}^s$ and postfix path $\mathbf{Z}^t$ with $t = k - s$, as generated by some bidirectional sampler.

$$D(\mathbf{X}) = \sum_{i=0}^{k} \hat{p}_i\left(\mathbf{X}\right)^{\beta} = \sum_{i=0}^{s-1} \hat{p}_i\left(\mathbf{X}\right)^{\beta} + \hat{p}_s\left(\mathbf{X}\right)^{\beta} + \sum_{i=s+1}^{k} \hat{p}_i\left(\mathbf{X}\right)^{\beta} \tag{8.23}$$

Using equations 8.15 and 8.16, we can rewrite these terms further:

$$\begin{aligned} \sum_{i=0}^{s-1} \hat{p}_i\left(\mathbf{X}\right)^{\beta} &= \hat{p}_{t+1}\left(\mathbf{Z}^{t+1}\right)^{\beta} \sum_{i=0}^{s-1} \hat{p}_i\left(\mathbf{Y}^{s-1}\right)^{\beta} = p\left(\mathbf{Z}^t\right)^{\beta} P_A\left(x_{s+1} \to x_s\right)^{\beta} D'(\mathbf{Y}^s) \\ \sum_{i=s+1}^{k} \hat{p}_i\left(\mathbf{X}\right)^{\beta} &= \hat{p}_{s+1}\left(\mathbf{Y}^{s+1}\right)^{\beta} \sum_{i=0}^{t-1} \hat{p}_i\left(\mathbf{Z}^{t-1}\right)^{\beta} = p\left(\mathbf{Y}^s\right)^{\beta} P_A\left(x_s \to x_{s+1}\right)^{\beta} D'(\mathbf{Z}^t) \end{aligned} \tag{8.24}$$

Using equation 8.13, we can finally rewrite $D(\mathbf{X})$ into

$$D(\mathbf{X}) = p\left(\mathbf{Z}^t\right)^{\beta} P_A\left(x_{s+1} \to x_s\right)^{\beta} D'(\mathbf{Y}^s) + p\left(\mathbf{Y}^s\right)^{\beta} p\left(\mathbf{Z}^t\right)^{\beta} + p\left(\mathbf{Y}^s\right)^{\beta} P_A\left(x_s \to x_{s+1}\right)^{\beta} D'(\mathbf{Z}^t) \tag{8.25}$$

Now let us apply this to the bidirectional path $\mathbf{X}$, constructed by connecting an eye path $\mathbf{Y}^s = y_1 \cdots y_s$ with a light path $\mathbf{Z}^t = z_1 \cdots z_t$. We define two recursive sequences p_i and d_i:

$$\begin{aligned} p_0^Y &= p_0\left(\mathbf{Y}^0\right)^{\beta} = 1 \\ p_i^Y &= p_i\left(\mathbf{Y}^i\right)^{\beta} = p_{i-1}^Y P_A\left(y_{i-1} \to y_i\right)^{\beta} \end{aligned} \tag{8.26}$$

$$\begin{aligned} p_0^Z &= p_0\left(\mathbf{Z}^0\right)^{\beta} = 1 \\ p_i^Z &= p_i\left(\mathbf{Z}^i\right)^{\beta} = p_{i-1}^Z P_A\left(z_{i-1} \to z_i\right)^{\beta} \end{aligned} \tag{8.27}$$

$$\begin{aligned} d_0^Y &= D'(\mathbf{Y}^0) = 0 \\ d_i^Y &= D'(\mathbf{Y}^i) \\ &= p\left(\mathbf{Y}^{i-1}\right)^\beta + D'\left(\mathbf{Y}^{i-1}\right) P_A\left(y_i \to y_{i-1}\right)^\beta \\ &= p_{i-1}^Y + d_{i-1}^Y P_A\left(y_i \to y_{i-1}\right)^\beta \end{aligned} \tag{8.28}$$

$$\begin{aligned} d_0^Z &= D'(\mathbf{Z}^0) = 0 \\ d_i^Z &= D'(\mathbf{Z}^i) \\ &= p\left(\mathbf{Z}^{i-1}\right)^\beta + D'\left(\mathbf{Z}^{i-1}\right) P_A\left(z_i \to z_{i-1}\right)^\beta \\ &= p_{i-1}^Z + d_{i-1}^Z P_A\left(z_i \to z_{i-1}\right)^\beta \end{aligned} \tag{8.29}$$

Using these recursive quantities, we can write $p_s\left(\mathbf{X}\right)^\beta$ and $D(\mathbf{X})$ as

$$\begin{aligned} p_s\left(\mathbf{X}\right) &= p_s^Y p_t^Z \\ D(\mathbf{X}) &= p_t^Z P_A\left(z_t \to y_s\right)^\beta d_s^Y \\ &\quad + p_s^Y p_t^Z \\ &\quad + p_s^Y P_A\left(y_s \to z_t\right)^\beta d_t^Z \end{aligned} \tag{8.30}$$

Hence, we have constructed the power heuristic weight $w_s\left(\mathbf{X}\right) = \frac{\hat{p}_s(\mathbf{X})^\beta}{D(\mathbf{X})}$.

8.2.3 Further details

In this section we will study some details and special cases of this formulation. First we will investigate some restrictions on the method used to terminate a path during construction. Then we will show how to handle BSDF's containing singularities, like perfect mirrors. Furthermore, we show that it is possible to guarantee at least one diffuse bounce on the eye path before the path is terminated.

Path termination

In the recursive formulation of the eye path quantity $d_i^{\mathbf{Y}}$, we need to evaluate the reverse sampling probability $P_A\left(y_{i-1} \to y_{i-2}\right)$. At the time of evaluation, no information about the light path is available. Therefore, this evaluation must be independent of the light path. Similar holds for evaluating $P_A\left(z_{i-1} \to z_{i-2}\right)$ in $d_i^{\mathbf{Z}}$. Because of these restrictions, any sampling decisions at a vertex may only depend on local information such as the incoming and outgoing direction and the surface properties. However, it may not depend on global path information, such as the vertex index on the path, because the light path length is not known while evaluating these reverse probabilities. Note that the probability to terminate the path at a vertex during construction is part of its sampling probability. Usually, Russian roulette is used to decide termination. Because of the former restriction, the Russian roulette method may also only depend on local vertex information and not on the global vertex index in the path.

Specular scattering

When dealing with specular bounces, the sampling probabilities may contain one or more Dirac delta functions. Any vertex on such a surface is called a specular vertex. A specular bounce is a specular vertex that is not used in the connection between light and eye path, so it is not the endpoint of either of these paths. For every specular bounce on a path, a Dirac delta function appears in the path probability. These Dirac delta functions can not be evaluated directly. However, as we are only interested in the final weight function, we divide out the Dirac delta functions from the weight's numerator and denominator accordingly. When, after division, there is still a Dirac delta function left in the denominator (implying there is at least one sampling strategy with more specular bounces for this path), the weight is forced to zero, otherwise all Dirac delta functions have vanished and the weight can be evaluated. When a path is constructed by connecting a specular vertex to another vertex, the alternative sampling strategy of sampling the same path using zero light vertices (omitting a connection) always contains at least one more Dirac delta function. Therefore, the weight of the connected path is forced to zero. Reversely, every sampling strategy in the denominator connecting a specular vertex to another vertex (specular or not), will contribute nothing to $D(\mathbf{X})$. These conditions must be handled explicitly:

- When trying to connect a specular vertex to another vertex, the corresponding path weight is zero and the connection should be skipped.
- When evaluating d_i where either x_i or x_{i-1} is specular, the first term of d_i connects a specular vertex to another vertex and is forced to zero. So, for this special case, the recursive relation becomes: $d_i = d_{i-1}^Z P_A(x_i \to x_{i-1})^\beta$.

Note that for common camera models the probability distribution of outgoing sampling directions from the eye does not contain a dirac function. Therefore, the eye is not considered a specular vertex. This does not hold for all light sources. For example, a pure directional light source has only a single outgoing direction, therefore its probability distribution contains a dirac function and the vertex on the light source is specular.

Specular prefix path

Finally we turn our attention to an interesting special case: the first non-specular vertex y_i on an eye path. Because connecting to/from a specular vertex results in zero probability, all sampling strategies sampling any of the preceding vertices $y_1 \cdots y_{i-1}$ as part of the light path result in zero probability. Therefore, the probability $P_A(y_i \to y_{i-1})$ is never evaluated during light path construction. For this reason, this first diffuse vertex poses an exception to the sampling restriction mentioned at the beginning of this section. Non-local Russian roulette is allowed for this vertex. This is useful, because we usually like to force at least one diffuse bounce. through similar reasoning, the light path can be forced to have at least one edge. Whether the next vertex is specular or not does not matter, because the light emission direction is already a diffuse 'bounce', unless purely directional light sources are used.

8.3 SBDPT

8.3.1 Algorithm overview

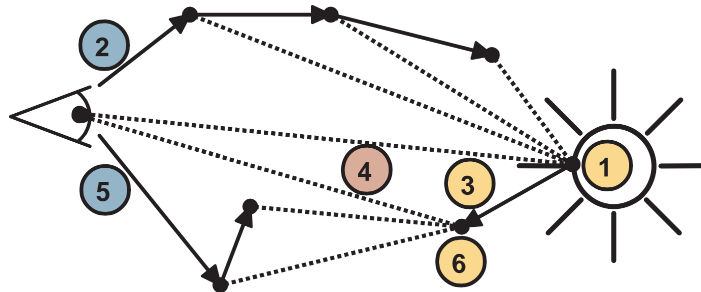


Figure 8.2: Construction of an SBDPT sample. 1) Light vertex is constructed 2) Eye path is generated and connected to light 3) Light path is extended 4) Light path is connected to eye 5) Eye path is generated and connected to light path 6) Light path is terminated.

In the original BDPT algorithm, a single eye path and light path are generated and connected to form a collection of light transport paths. In section 8.1, we explained why this method is not well suited for an efficient GPU implementation. We therefore propose to alter the BDPT algorithm to make it fit the streaming architecture of a GPU, which we will call *Streaming BDPT* or SBDPT. Instead of generating a single eye path, we generate a new eye path for each vertex on the light path. We start out with a light path containing a single vertex on some light source. After this vertex is generated, the first eye path is generated. During construction of the first eye path, each vertex is directly connected to the light vertex resulting in complete light transport paths. So far the algorithm is similar to the PT algorithm described in section 7.2, except now all vertices on the eye path connect to the same light vertex. When the eye path terminates, the light path is extended by a single vertex. This vertex is directly connected to the eye first, before a new eye path is generated. The vertices on this new eye path now connect to the new endpoint of the extended light path. This process is repeated until the light path is terminated and the SBDPT sample is complete (See Figure 8.2). Using this method, connections are no longer made after either of the paths is complete, but instead during the construction of the light and eye paths. During this construction, only the endpoints of the paths are connected. Hence there is no need to store the whole path.

8.3.2 Statistical differences

In this section we will discuss the statistical similarities and differences between BDPT and SBDPT. First, we will show that the probability of sampling a light transport path as part of a sample is the same for both methods except for implicit paths, which require special handling. Further, we explain how correlation between light transport paths is reduced by SBDPT w.r.t. BDPT. Finally, we will discuss the expected number of steps required per

sample in both methods, showing that SBDPT requires no more than twice the number of steps per sample required by BDPT.

Sample probability

To see that SBDPT does not differ fundamentally from BDPT, we show that the probability of sampling a light transport path $\mathbf{X}_{s,t}$ with some bidirectional strategy s,t (connecting eye path $\mathbf{Y}^s$ with s vertices and light path $\mathbf{Z}^t$ with t vertices) as part of an SBDPT sample remains the same. The only exceptions are implicit paths (paths with $t = 0$), which we will handle in the next section.

Each complete SBDPT sample contains at most one light transport path $\mathbf{X}_{s,t}$ for any $s \geq 0$ and $t > 0$. To see this is true, remember that eye paths only connect to the endpoint of the light path. So, there is only a single eye path connecting to the t' light vertex, resulting in light transport paths with t light vertices. Each vertex on this eye path only connects once to the light path (the eye is also an eye vertex). Hence, there is only one way to generate a path with s eye vertices and t light vertices. Going from this, it is easy to show that the probability of sampling such a transport path has not changed. The method for sampling a separate eye or light path has not changed, so the sample probabilities $p(\mathbf{Y}^s)$ and $p(\mathbf{Z}^t)$ remain the same for SBDPT. As there is only one way of sampling $\mathbf{X}_{s,t}$, the corresponding probability remains simply the product of sampling the connected eye and light paths, $p_s(\mathbf{X}) = p(\mathbf{Y}^s)\, p(\mathbf{Z}^t)$, which equals the probability of generating $\mathbf{X}_{s,t}$ as part of a BDPT sample. Therefore, the MIS formulation from the previous sections is still valid for SBDPT.

Implicit paths

Implicit paths, or light transport paths with $t = 0$, require special attention. These paths are not constructed using an explicit connection, but are found when an eye path *accidentally* hits a light source. Each SDBPT sample may contain more than one eye path, so it is possible to find more than a single implicit path $X_{s,0}$ per SDBPT sample for some $s > 0$. A simple solution would be to only count implicit paths found on the first eye path, discarding all other implicit paths. Although valid, this solution wastes valuable information, especially when rendering reflected caustics (paths of the form $ES^+(DS^+)^+L$). These effects usually are a worst case scenario for BDPT. Implicit paths are the only bidirectional strategy capable of finding these light transport paths. Therefore, it is undesirable to discard such implicit paths. A better solution for handling implicit paths is to record the contribution of all found implicit paths in a separate image, called the implicit image. All other contributions are recorded on the explicit image. These images are then combined to form the final unbiased estimate. In this combination, we have to correct the implicit image as it overestimates the contribution of implicit paths. In the original BDPT algorithm, only a single eye path was sampled per BDPT sample, possibly finding an implicit path. In SBDPT, multiple eye paths are sampled per SBDPT sample, so to correct for this, we have to multiply each pixel i in the implicit image by $\frac{1}{N_i}$, with N_i being the number of eye paths sampled for this pixel. The explicit image requires a usual correction for the number of SBDPT samples

per pixel. Adding the corrected images gives an unbiased estimate without wasting implicit path information. Compared to BDPT, the probability of sampling implicit paths as part of an SBDPT sample is higher because more eye paths are generated. Therefore, the MIS weights from section 8.2 are a little favored towards explicit paths. This does not introduce any bias, but the MIS weights are no longer considered optimal. In practice, this is not really a problem, because implicit paths usually are an inferior sampling strategy anyways, already having very low MIS weights. As already explained, the main purpose of implicit paths is to sample implicit caustics. However, because implicit paths are the only valid sampling strategy for these paths, the MIS weights will remain optimal.

Correlation

Although the probabilities for generating explicit light transport paths is the same for BDPT and SBDPT, there is a statistical difference between both methods resulting in less correlation between light transport paths in SBDPT at the cost of extra algorithmic steps per sample. In BDPT, one eye and light path is used to generate a collection of light transport paths by connecting all vertices of both paths. This causes a lot of correlation between light transport paths within a sample. In SBDPT, multiple eye paths are used per sample, reducing the correlation between eye paths. Still only a single light path is used per sample. To reduce the effect of correlation between light paths, the different eye paths should sample the complete image plane, not just the same pixel. This way, the correlation between light transport paths is distributed over the image plane, further reducing its visibility.

Expected steps per sample

As mentioned in the previous section, a consequence of generating multiple eye paths per sample is an increase in the expected number of algorithmic steps per sample. However we can easily show that SBDPT on average requires no more than twice the number of steps per sample as BDPT, where sampling a path vertex and connecting an eye and light path both constitute one step.

For BDPT, each sample consists of a light path and an eye path. Let us say that, for some sample, the eye path has length n and the light path has length m. Generating these paths requires $n+m$ steps. Then all light vertices are connected to all eye vertices, including the eye, requiring an extra $(n+1)\,m$ connection steps. Hence, a total of $r_{bdpt} = n+m+(n+1)\,m$ steps are required for this sample.

For SBDPT, each sample generates one light path and multiple eye paths. Let us say the light path has length m, thus m eye paths are sampled, each of length n_i with $i = 1\cdots m$. Sampling these paths requires $m+\sum n_i$ steps. All light vertices are connected to the eye and all eye vertices are connected to a light vertex, requiring an extra $m+\sum n_i$ steps. Hence, a total of $r_{sbdpt} = 2\,(m+\sum n_i)$ steps is used for this sample.

Let N and M be random variables representing resp. eye path and light path lengths. So n and n_i are realizations of N and m is a realization of M. As eye and light paths are sampled independently, N and M are independent variables. Hence, $E\left[r_{bdpt}\right] = \overline{n}+\overline{m}+(\overline{n}+1)\,\overline{m}$ and

$E[r_{sbdpt}] = 2(\overline{m} + \overline{mn})$, with $\overline{n} = E[N]$ and $\overline{n} = E[M]$. Therefore, on average an SBDPT sample requires no more than twice the number of steps required by a BDPT sample.

8.4 GPU SBDPT

In this section, we will discuss the GPU implementation of the SBDPT algorithm. First, we present a flowchart for generating a single SBDPT sample. Then, further details on efficiently implementing SBDPT on the GPU are given. Finally, we will discuss the performance of the algorithm, both in terms of speed and convergence.

8.4.1 SBDPT flowchart

In this section, we will present a flowchart for generating SBDPT samples. SBDPT can be viewed as an extension of TPPT, described in section 7.2. Sampling a single eye path and making connections to a light vertex remains the inner part of the SBDPT algorithm. The method is extended to generate a light path as well. The generation of a light path is similar to generating an eye path, with the addition that each time, right before the light path is extended, a complete eye path is sampled. So, just before the light path is extended, a new eye path is regenerated and only as soon as the eye path terminates is the light path actually extended. When the light path terminates, the SBDPT sample is complete and a new sample is started by regenerating the light path. Figure 8.3 shows the corresponding flowchart. The flowchart combines two PT-like flowcharts (figure 7.1), one for generating light paths (left) and one for generating eye paths (right). The eye path generation is inserted between the light path connection and light path extension steps. Again, the flowchart is divided in two phases: an **Extend** phase and a **Connect** phase. In the **Extend** phase either the eye or light path is extended. In the **Connect** phase either the light path is directly connected to the eye, or the eye path is connected to the light path. The two phases are executed repeatedly, one after the other. Between phase execution, rays are traced. Therefore, only transitions between processing steps in different phases may require a ray to be traced. Note that, unlike for eye paths, generating the first light vertex does not require tracing a ray, because the vertex is picked directly on a light source. Therefore, *Regenerate Light* is placed in the **Connect** phase. This is advantageous because now, when no intersection is found in *Intersect Light* (the light path is terminated), a new light path is generated immediately, without having to await the next phase. Finally, a word on the transition from *Extend Eye* to *Extend Light*; both reside in the same phase. However, for efficiency reasons explained later, only a single extension is allowed per execution of the **Extend** phase. Therefore, when *Extend Eye* is executed and causes a transition to *Extend Light*, the execution of *Extend Light* must await the next **Extend** phase.

8.4.2 GPU implementation

In this section, we will describe the CUDA implementation in more detail. The setup is very similar to TPPT from section 7.3. A single SBDPT sample is generated by each CUDA thread in parallel. All threads execute the **Extend** and **Connect** phases repeatedly, one after

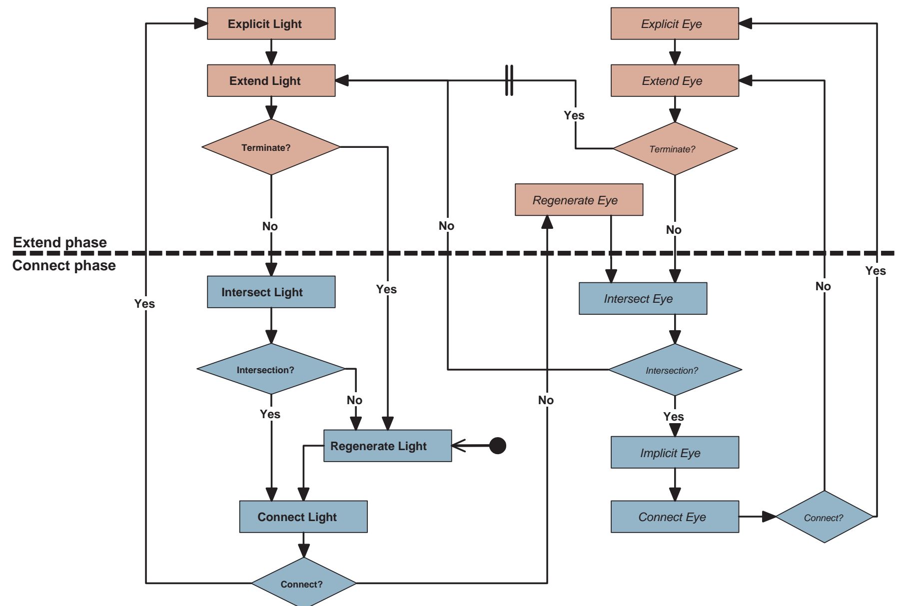


Figure 8.3: GPU SBDPT flowchart. The two phases are handled iteratively one after the other. Each time the flowchart changes phase, an optional output ray may be traced before the next phase is started. Note that for the transition from *Extend eye* to *Extend Light*, a new **Extend** phase must be awaited.

the other. During phase execution, a thread may execute all states belonging to this phase at most once, and in a fixed order. Hence, all threads in a warp executing the same step will follow the same code path and therefore execute in parallel using SIMT. All other threads wait for the step execution to finish before they can proceed.

In every phase, each thread may output an optional ray that is traced before the next phase is executed. Again, separate kernels are used for phase execution and ray traversal. Like TPPT, ray traversal performance is increased by packing all output rays in a single continuous ray stream using immediate packing (see section 7.4.1).

Code sharing

To achieve high SIMT efficiency, the threads in a warp must follow the same code path. Because the lengths of eye and light paths vary between threads in a warp, usually some threads are handling their light path while at the same time all other threads are handling their eye paths. This would effectively result in a 50% upper bound on SIMT efficiency. To resolve this, whenever possible, code paths for handling the light and eye path are shared. The *Explicit*, *Extend*, *Intersect* and *Connect* processing steps are very similar for eye and light paths, thus their code paths are shared. Collapsing these processing steps into shared processing steps effectively gives the flowchart for the actual GPU implementation. In figure 8.4, a simplified version of such a collapsed flowchart is shown. To prevent cluttering

in the diagram all conditions are removed and multiple conditional arrows may start at the same processing step. Whenever a phase is executed, the processing steps are executed in a

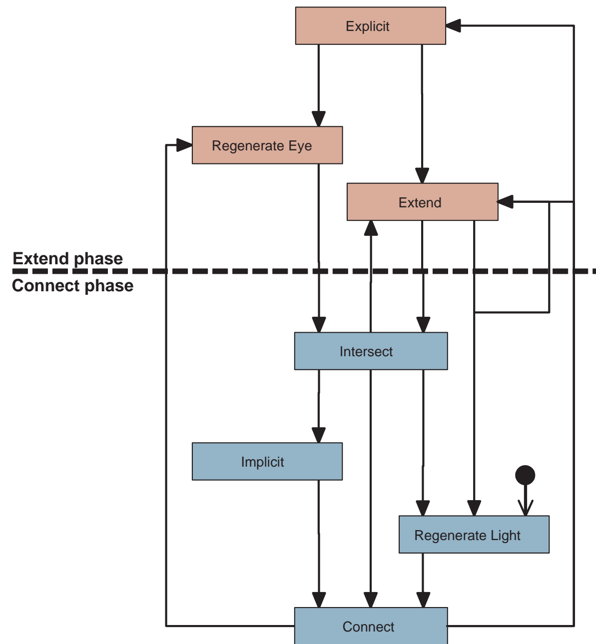


Figure 8.4: Simplified, collapsed GPU SBDPT thread flowchart. The code for steps *Explicit*, *Extend*, *Intersect* and *Connect* is shared for the eye and light paths, allowing for GPU efficiency. Each **Extend** phase, either the eye or light path is extended. Each **Connect** phase, a single connection is made between light and eye path.

fixed order (top-down in figure 8.4). As eye and light paths now share the same code paths, SIMT efficiency is significantly increased.

Because a step is never executed twice per phase, it is not possible to both extend the eye and light path in the same phase. Therefore, the transition from *Extend Eye* to *Extend Light* must await the next execution of the **Extend** phase. To visualize this, the edge from *Extend* to itself in figure 8.4 passes through the **Connect** phase.

Single vertex per path

As mentioned earlier, SBDPT only requires global memory storage for a single eye and light vertex per SBDPT sample. This reduces the global memory footprint, allowing more GPU threads to be active at once, thus increasing parallelism. Furthermore, it allows for coalesced global memory access per warp, increasing effective memory bandwidth. The eye and light vertices are stored separately as a structure of arrays. During *Connect* and *Explicit*, both the eye and light vertex are needed and accessed through coalesced memory transfers. However, during *Extend* and *Intersect*, either the light or eye vertex is needed, depending on the path that is being extended by each thread. Figure 8.5 shows the possible memory

access for an 8-thread warp. Because during each access to vertex data some threads access their eye vertex while others access their light vertex, per access two coalesced memory transfers are required. Also note that a coalesced memory transfer may load memory for threads that do not need it, as seen in figure 8.5. In total, at least half the loaded memory is actually used, so the effective memory bandwidth is at most halved due to accessing eye or light vertices by different threads at the same time.

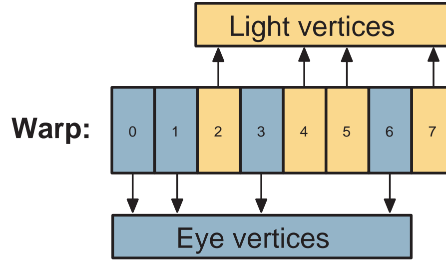


Figure 8.5: Memory access during *Extend*. Some threads in a warp extend their light path while others extend their eye path, resulting in two coalesced transfers per warp.

In our above discussion, we have been claiming that only a single eye and light vertex is required during SBDPT sample construction. However, an important detail was disregarded which we will discuss now. The MIS formulation from section 8.2 suggests that at least two vertices per eye path are required in order to evaluate $P_A(y_s \to y_{s-1})$ during connection. Furthermore, we noted that $P_A(x_i \to x_{i+1})$ could be written as $P_A(x_{i-1} \to x_i \to x_{i+1})$ to make explicit that this probability usually depends on the incoming direction. This also suggests that an extra path vertex is required. These problems are partially solved by storing the incoming direction with each vertex. After generating an extension direction, we now know both the incoming and outgoing direction, so we can compute $P_{\sigma^\perp}(x_{i-1} \to x_i \to x_{i+1})$ and $P_{\sigma^\perp}(x_{i+1} \to x_i \to x_{i-1})$ (*note that probabilities are per unit projected solid angle!*). The same holds during connection: after the connection is set up, all probabilities for the connected vertices can be computed per unit projected solid angle. What is left is the conversion from unit projected solid angle to unit area. The key observation here is that these conversion terms do not need to be computed during the same step or even the same phase as the probability per projected unit solid angle. A single conversion term may be split up further, computing each part when all required data is available. For example, during extension at vertex x_i, we can compute $P_{\sigma^\perp}(x_{i-1} \to x_i \to x_{i+1})$, but not until the next intersection do we know the actual vertex x_{i+1}, allowing us to compute the conversion factor $G(x_i \leftrightarrow x_{i+1})$. Furthermore, during extension of x_{i-1}, we can already compute one cosine term, required for $G(x_i \leftrightarrow x_{i-1})$. The square distance can computed during the next intersection step when x_i is known, while $P_{\sigma^\perp}(x_i \to x_{i-1})$ can not be computed until after the extension of vertex x_i. By carefully splitting up these computations so as to compute each part as soon as the required data is available, only the last vertex on both paths is ever needed during the computations. This has the added advantage of further reducing memory usage, because data used in these computations is usually already loaded for other purposes.

Scattered light path connections

In a normal path tracer, all light transport paths from a single sample contribute to the same pixel. This is no longer true for a BDPT. Each time when a light vertex is connected directly to the eye, the path may contribute to any pixel on the image plane. This poses a problem because multiple, parallel running threads may try to contribute energy to the same pixel. We solved this problem by using atomics when recording energy on the image plane. Because CUDA did not support floating point atomics until GPU architecture 2.0, all data is stored in a fixed point integer format and integer atomics are used to contribute energy. Our experiments showed that the usage of atomics did not significantly influence performance. Obviously, performance would suffer when most contributions were made to only a few pixels, but this is very unlikely in practice (and if it happens, it is probably not a very interesting rendering).

Light path sharing

Connecting a complete eye path to a single light vertex causes correlation between light transport paths. This correlation can be reduced by sharing light vertices between threads in a warp. Each time a thread tries to connect an eye vertex to a light vertex, it connects the eye vertex to the light vertex of any of the other threads in the warp trying to make a similar connection. It is important to exclude any threads that do not try to make such a connection to keep the method unbiased. This method generally reduces correlation, at the cost of reduced stratification in sampling strategies contributing to each pixel. Note that applying this method to light paths directly connecting to the eye would increase correlation; Whenever a light path would connect with the eye path of a thread without an eye path (terminated and not yet regenerated), the light path would be directly connected to the eye. If this happened twice for the same light vertex, exactly the same light transport path would be generated twice, significantly increasing visible noise. We want each light vertex to be connected to the eye exactly once. Therefore, threads directly connecting a light vertex to the eye are excluded from light vertex sharing.

Now, all that is needed is a method for sharing vertices, excluding threads that do not want to participate. Using a parallel prefix sum, all participating threads can determine the number of participating threads preceding this thread (incl. itself) in the warp, called n_i for thread i, and the total number of participants n. Then, each participating thread i writes its thread ID i in a shared buffer at location n_i. Finally, each participating thread picks a thread index j from any of the valid locations in the shared buffer. The participating thread i then connects to the light vertex of this selected thread j. A good method for selecting a location in the buffer is for each thread to use its own n_i plus some fixed stride as a cyclic buffer location, causing each participating thread to be picked exactly once. The shared stride should be incremented each time the **connect** phase is started, thereby stratifying the sharing of light vertices between threads. Listing 8.1 shows a small code snippet showing for selecting a participating thread to connect to. Note that his method only applies to devices with compute capability 1.2 or higher. On older devices, sharing light vertices would prevent coalesced memory access when loading light vertices.

```
__device__ __constant__ int c_stride;

__device__ int get_light_vertex_idx( bool participate )
{
        __shared__ volatile int s_n;
        __shared__ volatile int s_location_buffer[32];

        // inclusive prefix sum over warp participants
        int n_i = PREFIX_SUM( participate? 1: 0 );

        // store total participants
        if( threadIdx.x == 31 ) s_n = n_i;

        if( participate )
        {
                // load total participants
                int n = s_n;
                // share location
                s_location_buffer[n_i - 1] = threadIdx.x;
                // get connection thread
                return s_location_buffer[(n_i+c_stride) % n];
        }
        return threadIdx.x;
}
```

Listing 8.1: Code snippet for selecting a light vertex during light path sharing

Sample stream compaction

We did not apply sampler stream compaction to the SBDPT algorithm, as we did with the PT algorithm in chapter 2.4. Sampler stream compaction could further increase the performance of the SBDPT method. An SBDPT sampler requires significantly more persistent local storage than a PT sampler because the SBDPT must store two path vertices, including some extra temporary values for MIS weight construction. Therefore, packing the SBDPT sampler stream would require a lot of extra data copying. However, most of this data is loaded during extension and connection anyway. Note from section 8.4.2 that although only one of the two vertices is actually needed by the sampler, the hardware usually accesses both vertices due to coalesced memory transactions, so not much extra data access is required for stream compaction.

An important difference between the PT sampler and the SBDPT sampler is that the PT sampler is terminated as soon as the eye path is terminated, while the SBDPT sampler is only terminated after the light path has been terminated. So, when an eye path is terminated, the samplers light path vertex must persist. To accomplish this, the samplers' output stream

could be divided in three parts. All samplers whose eye and light paths did not terminate are packed at the beginning of the output stream, similar to the SSPT method. All samplers whose eye path is terminated but whose light path has not yet been terminated, are packed at the very end of the buffer. The size of the gap in the middle, between the samplers at the beginning and the end of the stream, equals the number of samplers having their light path terminated.

Now, all samplers at the beginning of the stream need to extend their eye paths, all samplers at the end of the stream need to extend their light paths and all samplers in the middle need to regenerate their light path. After the next **Extend** iteration, all samplers that extended their light path and have not terminated now need to regenerate their eye path. Because all terminated samplers are removed from the stream, these samplers still form a continuous part in the stream. Hence, the regenerated primary rays will be traced together, resulting in improved ray tracing performance due to primary ray coherence.

Note that splitting the output stream like this will also increase the SIMT efficiency during explicit connection construction, because all samplers that connect a light path to the eye will be at one end of the stream while all samplers connecting an eye path to a light path will be at the other end. Therefore, all samplers in a warp are more likely to take similar code paths.

We expect that this method will increase SIMT efficiency of the various phases, similar to the SSPT method. Furthermore, we expect an increase in performance due to primary ray coherence. We leave the implementation of sampler stream compaction for SBDPT to further research.

8.4.3 Results

In this section we present the results of our SBDPT implementation. We start by assessing the SIMT efficiency of the method, followed by some performance results.

Efficiency

Table 8.1 shows the efficiency and occurrence of the algorithmic steps in the SBDPT algorithm (see section 7.5.1 for an explanation of efficiency and occurrence). Because eye and light path regeneration must be handled separately, often requiring algorithmic steps to be skipped by a sampler, efficiency is significantly lower compared to the PT methods from the last chapter. Still, the efficiency for most steps is reasonable, allowing for good SIMT efficiency and corresponding GPU performance. The most notable exception is the *Regenerate Light* step, responsible for regenerating the light path. Because multiple eye paths may be generated before the light path is terminated, light paths are regenerated much less often than eye paths. Therefore, this step has a very low efficiency. Luckily, regenerating light paths is not very complicated and does not require a lot of computations. Therefore, performance is not degraded too much by this lack of efficiency.

Performance

As expected from the reduced SIMT efficiency, the SBDPT performance is lower than that of the PT methods from last chapter. Figure 8.6 shows the SBPT performance in rays and SBDPT sampler per second with and without output ray packing. Note that, in contrast to the TPPT method from last chapter, packing output rays causes a much more significant increase in performance. This is because creating bidirectional connections failes more often then creating explicit connections in PT[1], resulting in more gaps in the output ray stream. Removing those gaps through packing increases connection ray traversal performance significantly.

The overall performance in rays per second is significantly less than for the TPPT and SSPT methods. Besides reduced SIMT efficiency, there are several reasons for this performance reduction. First, while PT connection rays usually show some degree of coherence because all connections end in a few light sources, bidirectional connection rays are very divergent and therefore have lower ray traversal performance. Furthermore, the SBDPT sampler is much more complex than the PT sampler, requiring much more computations and memory accesses due to MIS weight construction. This decreases the sampler performance further. The increase in sampler complexity is made visible in figure 8.7, showing a time partition for the phases in an SBDPT iteration. Compared to the PT methods, the SBDPT sampler advancing phases take up a much more significant part of the total iteration time, with an average of 37%. This is almost twice that of the PT samplers.

As shown in table 8.2, the SBDPT requires significantly more memory because it has to store both a light and eye path vertex per sampler, including some auxiliary terms used for MIS computations. Still, the memory footprint remains acceptable, allowing for reasonably sized stream sizes on current generation GPU's.

[1]The light source is often placed outside the scene, directed towards the scene. Therefore, eye vertices seldom lie behind a light source, which would cause an explicit connection to fail. Because light vertices do necessarily lie on the light source in BDPT but may lie anywhere in the scene, this no longer holds and connections are more likely to fail.

Results														
	Explicit		**Regen Eye**		**Extend**		**Intersect**		**Implicit**		**Regen Light**		**Connect**	
	Ef	Oc	Ef	Oc	Ef	Oc	Ef	Oc	Ef	Oc	Ef	Oc	Ef	Oc
SPONZA	32	100	24	100	75	100	65	100	48	100	9	94	73	98
SIBENIK	36	100	24	100	75	99	66	100	49	100	9	97	74	100
STANFORD	29	100	19	100	80	99	73	100	58	100	7	94	78	100
FAIRY FOREST	37	100	29	100	70	99	74	100	41	100	19	100	70	100
GLASS EGG	37	100	25	100	74	100	67	100	48	100	9	92	74	98
INVISIBLE DATE	36	100	24	100	75	100	66	100	50	100	8	92	75	98
CONFERENCE	37	100	25	100	74	100	66	100	49	100	9	92	74	98

Table 8.1: SIMT efficiency and occurrence of algorithm steps in an average SBDPT iteration.

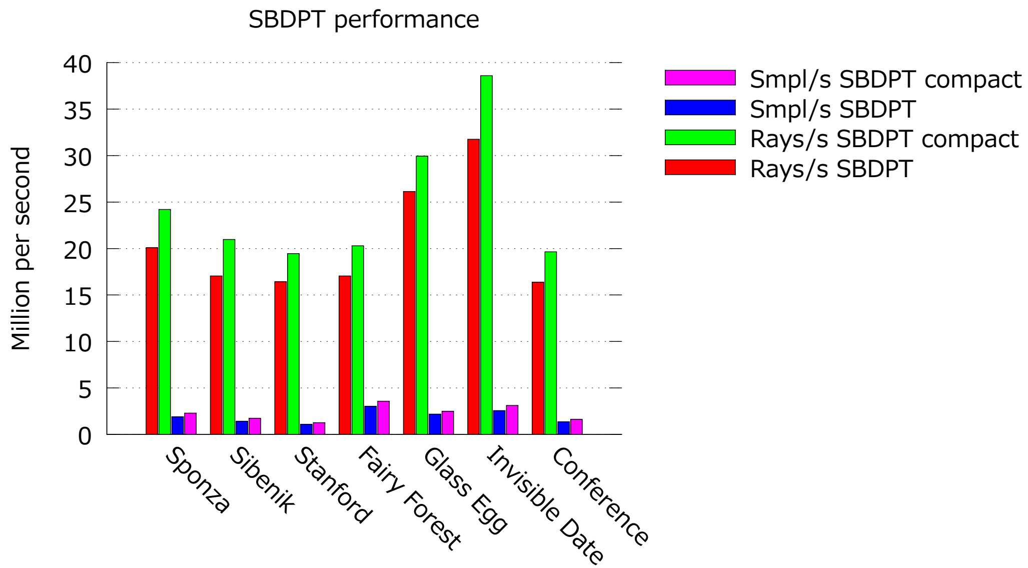


Figure 8.6: General SBDPT performance in rays and SBDPT samples per second without and with output ray stream compaction.

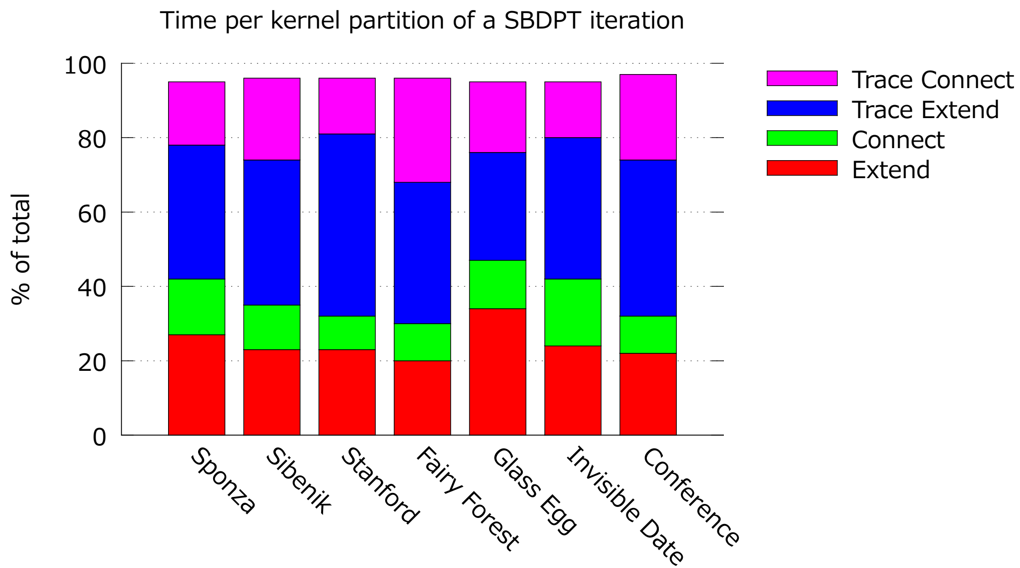


Figure 8.7: SBDPT procentual time partition of an iteration.

Convergence

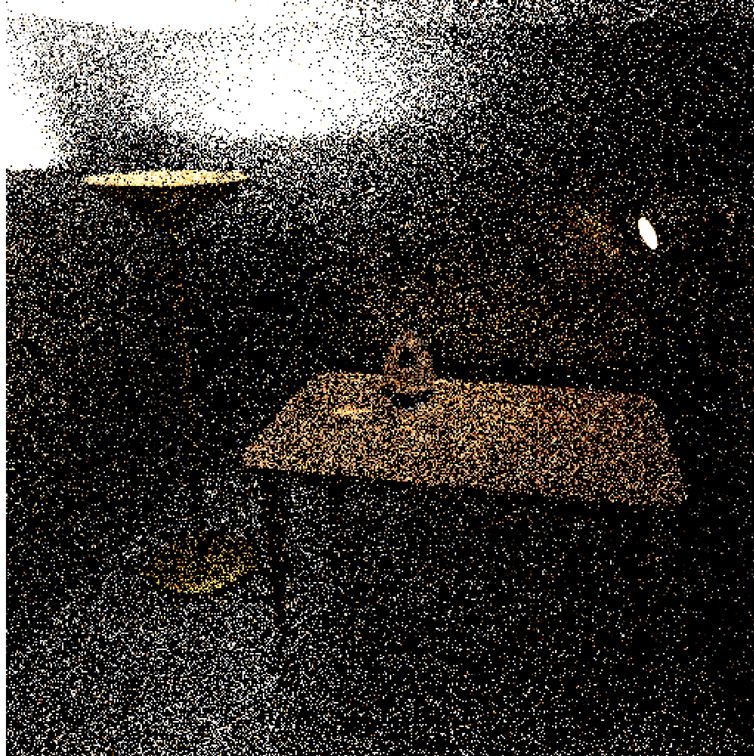
Although SSPT reaches higher performance in terms of rays per second compared to SB-DPT, SBDPT makes up for this through reduced variance in the image estimate. Especially for scenes with many complex materials and much indirect light, BDPT often performs much better than PT. Figure 8.8 shows that this also holds for our SBDPT implementation. The upper image is computed using SSPT while the lower image was computed with SB-DPT. Both images took 5 seconds to render. Note that PT has a very hard time rendering this scene. The reason is that both light sources are covered by glass plates, so the whole scene is lit by caustics. The figure shows that SBDPT is much better at capturing these light effects than SSPT. Note however that there is still some high frequency noise in the glass egg. As explained earlier, this is because BDPT has a hard time sampling reflected caustics. These paths are sampled through implicit eye paths with low probability. This property is inherited by SBDPT, although its effects are somewhat reduced because the SBDPT algorithm generates more eye paths per sample than standard BDPT, effectively increasing the probability of finding the reflected caustics.

Figure 8.9 shows the contributions of some of the bidirectional sampling strategies to the final image. As expected, these images show that MIS causes different sampling strategies to sample different light effects[2]. The results from this section show that the high performance of the PT method often cannot compensate for the increased variance compared to BDPT. Therefore, SBDPT is a valuable alternative to SSPT for scenes with complex illumination. In the next chapter we will go on and implement an ERPT sampler on the GPU, even further reducing variance for certain difficult illumination effects.

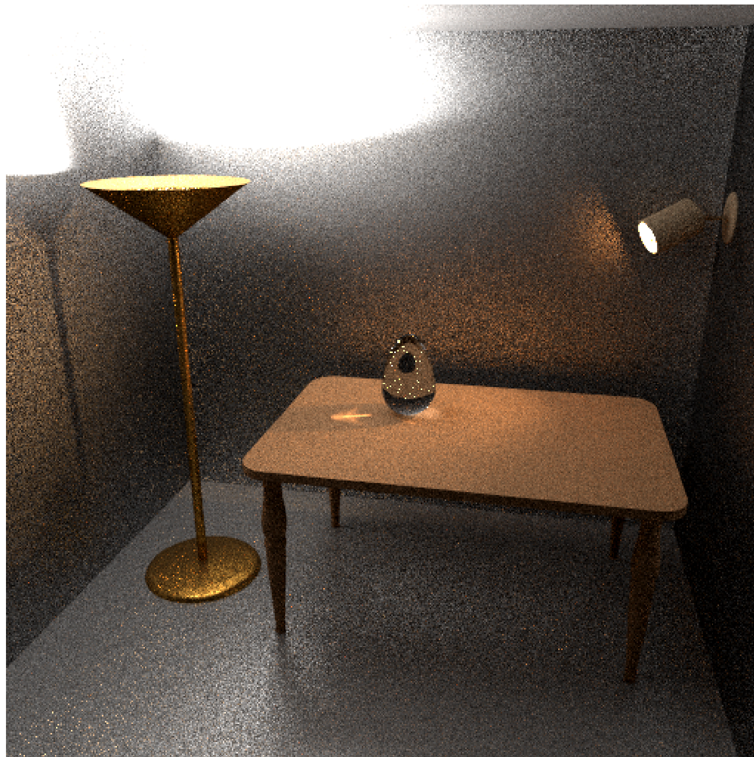
[2]For these images, the GLASS EGG scene was altered by removing the glass plates covering the light sources in order to shorten the average path length. Results are similar for the original scene, but because of the glass plates, the images on the first two rows in the image pyramid have no contribution.

Memory usage per 256 warps	3328 Kb

Table 8.2: SBDPT Memory usage for sampler and ray storage per 256 warps.



(a)



(b)

Figure 8.8: Comparison between SSPT (a) and SBDPT (b). Images are 512×512 and both took 5 seconds to render.

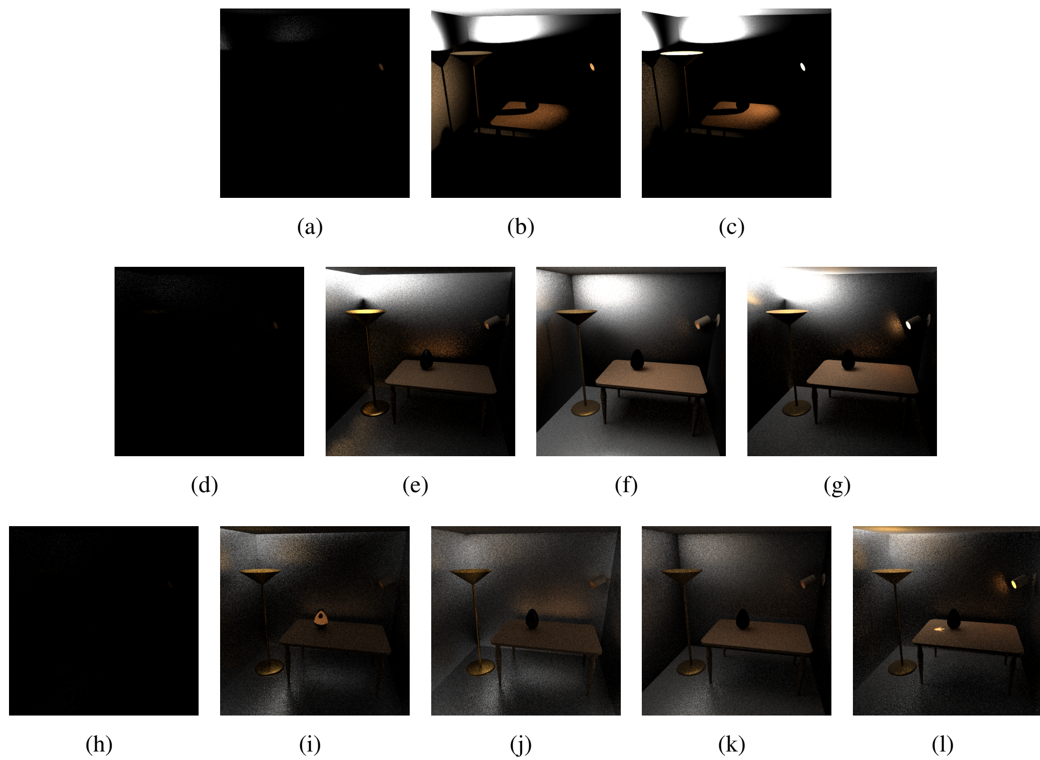


Figure 8.9: Contributions of various bidirectional strategies. Rows represent path lengths, starting with length 3; columns represents the number of vertices sampled on the light path, starting left with 0 light path vertices. Notice how different strategies sample different light effects.

Chapter 9

Energy Redistribution Path Tracer (ERPT)

9.1 Introduction

In this chapter we will present our GPU ERPT implementation. We will start by studying the characteristics of the ERPT mutation strategy using the concept of mutation features. We will then propose an improvement mutation strategy, which we used in our implementation. Next, we will describe the workings of a single ERPT sampler and discuss our GPU implementation. Finally, we will present our findings.

9.2 ERPT mutation

9.2.1 Mutation features

In this section we will introduce the notion of mutation features and their relation to ERPT. Simply put, for a given scene and mutation strategy, a mutation feature is a set of paths all reachable from one another by finite length mutation chains. More formally, for a given path space Ω, measurement contribution function $f : \Omega \to \mathbb{R}$ and mutation strategy with tentative transition function $T : \Omega \times \Omega \to \mathbb{R}$, we will give a partition of Ω into a unique feature set F.

Singularity constraint

For ease of presentation, we impose a *singularity constraint* on the tentative transition function: For each $\mathbf{X}, \mathbf{Y} \in \Omega$ with $f(\mathbf{X}) > 0$ and $T(\mathbf{X}|\mathbf{Y}) > 0$, we assume $\frac{T(\mathbf{X}|\mathbf{Y})}{f(\mathbf{X})} \in (0, \infty)$. In the context of light transport, this practically means that $f(\mathbf{X})$ must contain equally many non-zero Dirac functions as the tentative transition function $T(\mathbf{X}|\mathbf{Y})$. In the context of importance sampling (section 2.5), we saw that to effectively sample specular lighting effects, the sampling probability distribution must contain Dirac delta functions centered at the same positions on the domain as the Dirac delta functions in the measurement contribu-

tion function. This applies equally to the mutation probability distributions. Therefore, this condition is not overly restrictive and is satisfied by any practical mutation strategy.

Reachable relation

To define a feature, we will first define the binary *reachable* relation on path space. For a given scene and mutation strategy, a path $\mathbf{Y}$ is reachable from $\mathbf{X}$ if there is some finite mutation chain, starting at $\mathbf{X}$ and ending at $\mathbf{Y}$, having strictly positive acceptance probabilities.

Definition 9.2.1. *For a given Ω, $f : \Omega \to \mathbb{R}$ and $T : \Omega \times \Omega \to \mathbb{R}$, $\mathbf{Y} \in \Omega$ is* reachable *from $\mathbf{X} \in \Omega$ with $f(\mathbf{X}) > 0$ iff there is some finite sequence $\mathbf{Z}_0 \cdots \mathbf{Z}_n, \mathbf{Z}_i \in \Omega$ with $n \geq 2$, $\mathbf{Z}_0 = \mathbf{X}$, $\mathbf{Z}_n = \mathbf{Y}$, $T(\mathbf{Z}_{i+1}|\mathbf{Z}_i) > 0$ and $a(\mathbf{Z}_{i+1}|\mathbf{Z}_i) > 0$ for all $0 \leq i < n$. We will refer to such a sequence as a mutation chain from $\mathbf{X}$ to $\mathbf{Y}$. The reachable relation is not defined when $f(\mathbf{X}) = 0$.*

Lemma 9.2.1. *For any mutation chain $\mathbf{Z}_0 \dots \mathbf{Z}_n$ from $\mathbf{X}$ to $\mathbf{Y}$ with $\mathbf{Y}$ reachable from $\mathbf{X}$, it holds that $f(\mathbf{Z}_i) > 0$ for all i.*

Proof 9.2.1. *By definition, $f(\mathbf{X}) > 0$. Assume $f(\mathbf{Z}_j) = 0$ for some $0 < j < n$, then there must also be some $0 < i \leq j$ with $f(\mathbf{Z}_{i-1}) > 0$ and $f(\mathbf{Z}_i) = 0$. By definition, $a(\mathbf{Z}_i|\mathbf{Z}_{i-1}) = \min\left(1, \frac{f(\mathbf{Z}_i)T(\mathbf{Z}_{i-1}|\mathbf{Z}_i)}{f(\mathbf{Z}_{i-1})T(\mathbf{Z}_i|\mathbf{Z}_{i-1})}\right) > 0$. However, because $T(\mathbf{Z}_i|\mathbf{Z}_{i-1}) > 0$ for the mutation chain to be valid and either $T(\mathbf{Z}_{i-1}|\mathbf{Z}_i) = 0$ or $\frac{T(\mathbf{Z}_{i-1}|\mathbf{Z}_i)}{f(\mathbf{Z}_{i-1})} \in [0, \infty)$ due to the singularity constraint, the acceptance $a(\mathbf{Z}_i|\mathbf{Z}_{i-1}) = 0$, so by contradiction, $f(\mathbf{Z}_i) > 0$ for all i.*

Using this, we can prove that the *reachable* relation is both symmetric and transitive.

Lemma 9.2.2. *The* reachable *relation is symmetric.*

Proof 9.2.2. *Assume there are some $\mathbf{X}, \mathbf{Y} \in \Omega$ with $\mathbf{Y}$ reachable from $\mathbf{X}$ through mutation chain $C = \mathbf{Z}_0 \cdots \mathbf{Z}_n$, but with $\mathbf{X}$ not reachable from $\mathbf{Y}$. For this to be true, $\mathbf{Z}_n \cdots \mathbf{Z}_0$ must not be a valid mutation chain from $\mathbf{Y}$ to $\mathbf{X}$, so there is at least one $0 \leq i < n$ with either $T(\mathbf{Z}_i|\mathbf{Z}_{i+1}) = 0$ or $a(\mathbf{Z}_i|\mathbf{Z}_{i+1}) = 0$.*

We claim that $T(\mathbf{Z}_i|\mathbf{Z}_{i+1}) = 0$ can never be. To see this, remember that $f(\mathbf{Z}_{i+1}) > 0$ and $T(\mathbf{Z}_{i+1}|\mathbf{Z}_i) > 0$, implying $\frac{f(\mathbf{Z}_{i+1})}{T(\mathbf{Z}_{i+1}|\mathbf{Z}_i)} \in (0, \infty)$. Furthermore, $f(\mathbf{Z}_i) > 0$, so it must be true that $T(\mathbf{Z}_i|\mathbf{Z}_{i+1}) > 0$ for $a(\mathbf{Z}_{i+1}|\mathbf{Z}_i) > 0$ to hold.

Going from here, we reason that $a(\mathbf{Z}_i|\mathbf{Z}_{i+1}) = 0$ implies $T(\mathbf{Z}_{i+1}|\mathbf{Z}_i) = 0$: Assume $a(\mathbf{Z}_i|\mathbf{Z}_{i+1}) = 0$ and $T(\mathbf{Z}_{i+1}|\mathbf{Z}_i) > 0$. From the singularity constraint and $f(\mathbf{Z}_{i+1}) > 0$, we then have that $\frac{f(\mathbf{Z}_{i+1})}{T(\mathbf{Z}_{i+1}|\mathbf{Z}_i)} \in (0, \infty)$. Furthermore, we established that $T(\mathbf{Z}_i|\mathbf{Z}_{i+1}) > 0$ and $f(\mathbf{Z}_i) > 0$, implying that $\frac{f(\mathbf{Z}_i)}{T(\mathbf{Z}_i|\mathbf{Z}_{i+1})} \in (0, \infty)$. Hence, $a(\mathbf{Z}_i|\mathbf{Z}_{i+1}) > 0$, contradicting the initial assumption.

Because $T(\mathbf{Z}_{i+1}|\mathbf{Z}_i) > 0$ must hold for C to be a valid mutation chain from $\mathbf{X}$ to $\mathbf{Y}$, $a(\mathbf{Z}_i|\mathbf{Z}_{i+1}) > 0$ must also hold. Hence, $\mathbf{Z}_n \cdots \mathbf{Z}_0$ is a valid mutation chain from $\mathbf{Y}$ to $\mathbf{X}$.

Lemma 9.2.3. *The* reachable *relation is transitive.*

Proof 9.2.3. *For some $\mathbf{X}, \mathbf{Y}, \mathbf{Z} \in \Omega$ with $\mathbf{Y}$ reachable from $\mathbf{X}$, $\mathbf{Z}$ reachable from $\mathbf{Y}$, $\mathbf{V}_0 \dots \mathbf{V}_n$ a mutation chain from $\mathbf{X}$ to $\mathbf{Y}$ and $\mathbf{W}_0 \cdots \mathbf{W}_m$ a mutation chain from $\mathbf{Y}$ to $\mathbf{Z}$. The concatenated sequence $\mathbf{V}_0 \dots \mathbf{V}_n \mathbf{W}_0 \cdots \mathbf{W}_m$ forms a mutation chain from $\mathbf{X}$ to $\mathbf{Z}$. Hence, $\mathbf{Z}$ is reachable from $\mathbf{X}$.*

Feature partition

Using the *reachable* relation, we define a feature Φ as follows:

Definition 9.2.2. *A feature $\Phi \subseteq \Omega$ is a subset satisfying that for any $\boldsymbol{X} \in \Phi$, $f(\boldsymbol{X}) > 0$ and for any $\boldsymbol{X}, \boldsymbol{Y} \in \Omega$ with $\boldsymbol{X} \in \Phi$, $\boldsymbol{Y} \in \Phi$ iff $\boldsymbol{Y}$ is reachable from $\boldsymbol{X}$.*

Lemma 9.2.4. *For each $\boldsymbol{X} \in \Omega$ with $f(\boldsymbol{X}) > 0$, there exists one unique feature $\Phi \subseteq \Omega$ with $\boldsymbol{X} \in \Phi$.*

Proof 9.2.4. *Take Φ as the set of $\boldsymbol{X}$ and all $\boldsymbol{Y} \in \Omega$ with $\boldsymbol{Y}$ reachable from $\boldsymbol{X}$. By the symmetry and transitivity of the reachable relation, this also means that for any $\boldsymbol{Y} \in \Phi$, Φ contains all $\boldsymbol{Z} \in \Omega$ that are reachable from $\boldsymbol{Y}$ and that any $\boldsymbol{Z} \in \Phi$ is reachable from $\boldsymbol{Y}$. Hence, the subset Φ is a feature containing $\boldsymbol{X}$. To see that this feature is unique, assume $\boldsymbol{X}$ belongs to two distinct features, Φ and Φ'. Then, by the transitivity of the reachable relation, any element in Φ is reachable from any element in Φ' via $\boldsymbol{X}$ and visa versa. Hence, from the definition of features it follows that $\Phi = \Phi'$.*

Theorem 9.2.1. *There exists a unique feature set F of non-empty features in Ω that forms a partition of the set $\{\boldsymbol{X} \in \Omega | f(\boldsymbol{X}) > 0\}$.*

Proof 9.2.5. *Let $S_\Phi(\boldsymbol{X})$ be the unique feature corresponding to $\boldsymbol{X} \in \Omega$ with $f(\boldsymbol{X}) > 0$. It follows from lemma 9.2.4 that for F to be a partition of $\{\boldsymbol{X} \in \Omega | f(\boldsymbol{X}) > 0\}$, F must contain $\{S_\Phi(\boldsymbol{X}) | \boldsymbol{X} \in \Omega, f(\boldsymbol{X}) > 0\}$. Any other feature $\Phi \notin F$ must be empty. If such a feature would not be empty but contain some path $\boldsymbol{X} \in \Omega$, then by definition $f(\boldsymbol{X}) > 0$ and according to lemma 9.2.4, $\Phi = S_\Phi(\boldsymbol{X})$, contradicting that $\Phi \notin F$.*

By adding a special null-feature $\Phi_\emptyset = \{\mathbf{X} \in \Omega | f(\mathbf{X}) = 0\}$, we obtain a partition $F \bigcup \{\Phi_\emptyset\}$ for Ω. Whether the number of features in F is finite depends on the scene and mutation strategy used.

The energy $E(\Phi)$ for some feature Φ is the amount of energy reaching the eye over paths in Φ and equals $E(\Phi) = \int_\Phi f(\mathbf{X}) d\Omega(\mathbf{X})$.

ERPT features

In Metropolis sampling, a mutation chain is a Markov chain of paths, constructed through iterative mutation. Hence, each path in the chain must be *reachable* from the first path in the chain. Therefore, each chain can explore only a single mutation feature. We saw in section 2.8 that to satisfy ergodicity in Metropolis Light Transport, it is sufficient to ensure that $T(\mathbf{X}|\mathbf{Y}) > 0$ for any $\mathbf{X}, \mathbf{Y} \in \Omega$ with $f(\mathbf{X}) > 0$ and $f(\mathbf{Y}) > 0$. This means that for any such $\mathbf{X}$ and $\mathbf{Y}$, $\mathbf{Y}$ is reachable from $\mathbf{X}$. Hence, besides the null feature, there is at most a single non-empty feature containing all paths $\mathbf{X}$ with $f(\mathbf{X}) > 0$. In ERPT however, ergodicity is achieved by using path tracing to generate new mutation chains, thus the mutation strategy itself does not have to satisfy ergodicity. In case ergodicity is not satisfied by the mutation strategy, there may be many (possibly infinite) features, depending on the scene. The type of features within a scene determines the types of errors visible in the final rendering. In the next section, we will study this further in the context of the ERPT mutation strategies as

proposed in the original ERPT paper. Remember that these errors are a form of noise; the image remains unbiased.

9.2.2 ERPT mutation strategy

Remember from section 2.10 that the mutation chain length N should be neither too small nor too large. When N becomes too small, energy redistribution becomes less effective and ERPT deteriorates into quantized PT. We would therefore like to choose N as large as possible. However, as explained in the last section, each mutation chain can only explore a single mutation feature. All N mutations in the chain will lie within this single mutation feature. Having large N can therefore result in noticeable artifacts. In this section, we will study these artifacts using the notion of mutation features. To emphasize the artifacts in our experiments, we used an N in the order of 1000[1].

Feature exploration

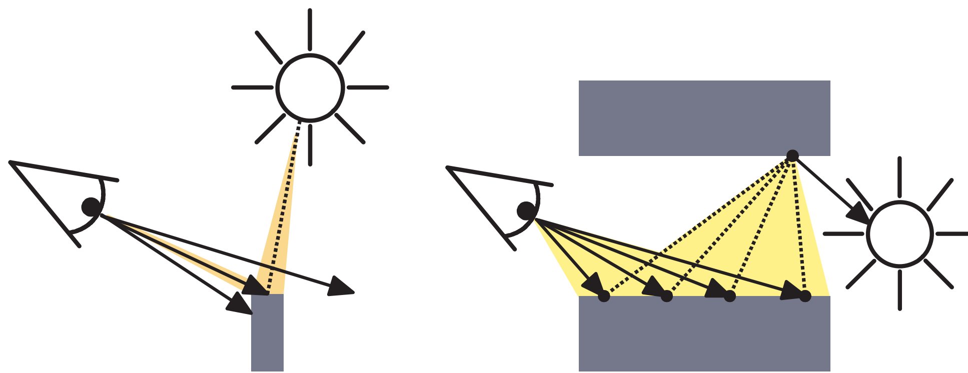
Probably the most noticeable error in ERPT images due to features is that light effects may be completely absent, even when a significant number of ERPT samples is generated. In a progressive ERPT, it seems that a light effect may be absent for a long time until at some point, it seems to be *found* almost instantly. This is a direct consequence of the fact that a mutation chain can only explore a single feature. So, as long as no mutation chain is generated for a certain feature, the feature will be completely absent in the image. This effect becomes more pronounced when the path tracer does a bad job at sampling initial mutation chain paths proportional to their energy. It may take many samples before an initial path is generated in some feature, but when it is, mutation chains are generated for this initial path with high probability, effectively exploring the feature. This causes the feature to appear instantly. Good examples are small highlights on refractive objects. Due to importance sampling, the probability of refraction is often much higher than for reflection. However, when the reflected paths directly hit a bright light source, they result in a very high energy feature. Hence, this is a distinctly visible feature but it may take a while for it to be found. A similar example is a caustic as seen through a mirror. In general, path tracers have a hard time finding these types of paths while they tend to carry a lot of energy.

Feature shape

In the last section, we saw that because each mutation chain only explores a single feature, features may be absent in the final rendering. We will now focus on how a mutation chain explores a feature. This is largely determined by the feature's shape. To see the influence of feature shape on error, take a look at figure 9.1(a). The figure shows a feature containing a path leaving the eye and reaching the light source through one diffuse bounce. The complete feature obtained using lens mutations is shown in yellow. It consists of all paths of length 2,

[1] When using N in the order of 100, these artifacts are already recognizable and significantly weaken the ERPT algorithm. In [11] the authors propose a special post processing filter to reduce these artifacts, causing extra bias in the estimate.

ending at the same point on the light source. When applying the lens mutation, mutations are generated by perturbing the primary ray and connecting the new intersection point to the light source. However, because the feature is so small, most perturbations will miss the top of the small object and fail. When a mutation fails, the mutation chain remains at its current path. This causes the chain to get stuck on the same path for many mutations resulting in increased noise in the image. In this example, the main cause is that the average mutation perturbation size is relatively large compared to the size of the feature that is being explored, causing most mutations to fail. Hence, if perturbations are too large, small features will be extra noisy.



(a) When a feature is small w.r.t. the perturbation size, most mutations will fail.

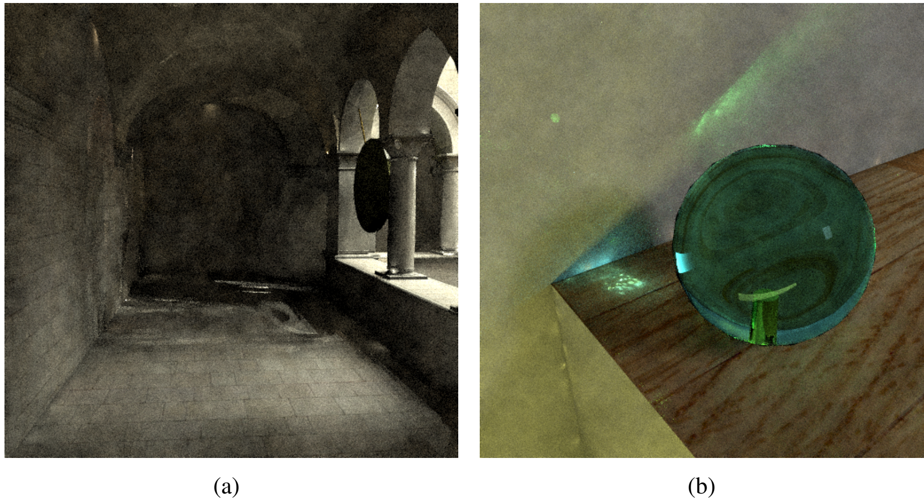
(b) Mutating only a part of the path causes high correlation between mutations.

Figure 9.1: The shape of a feature can cause noise and visible artifacts.

Another property of lens mutations is that for explicit light transport paths, the last vertices on the path are never mutated. The lens mutation always mutates the least amount of vertices, starting from the eye, and makes an explicit connection with a vertex on the original path to complete the mutation. This results in a lot of correlation between mutations, as all mutations in such a mutation chain share the last path vertices (see figure 9.1(b)). Figure 9.2(a) gives an example of this in the Sponza scene. As you can see, instead of soft area-light shadows, one gets multiple sharp shadows on the floors and walls. The effect looks a bit like a rendering using a few Virtual Point Lights (VPL) to approximate global illumination by point lights [32]. This is exactly what happens: during each mutation, only the first vertex from the eye (excluding the eye itself) is perturbed, so the second vertex becomes a shared VPL for all paths in the mutation chain. A similar effect appears when the caustic mutation is used (see figure 9.2(b)); All mutations in the mutation chain share the same light vertex, so instead of a single uniform caustic caused by one large area-light, there seem to be multiple caustics from multiple point lights.

Feature energy distribution

In the previous paragraph we saw that small features can cause extra noise and correlation between samples. This however does not mean that large, well shaped features cause less



(a) (b)

Figure 9.2: High correlation between mutations in a mutation chain cause distinct errors. (a) Correlation between lens mutations cause sharp shadows. (b) Correlation between caustic mutations cause splotchy caustics.

apparent errors in the final image. The type of visible errors also depend on the distribution of energy within the feature. For example, a local optimum in the energy distribution of a feature results in a small region within the feature that is easy to find through mutations but hard to escape. An important example of this are paths in corners (see figure 9.3). The shown feature is relatively large thus most mutations will result in valid light transport paths. However, the energy is very unevenly distributed over the feature. The further the first path vertex lies from the second vertex, the smaller will be the energy flowing over this path. This is mainly due to the cosine term in the rendering equation on the angle between the normal of the horizontal surface and the connection edge. A consequence is a local optimum in the energy distribution around the unconnected path (unbroken line) in figure 9.3. Therefore, mutation chains will tend to mutate towards this local optimum and will have a hard time escaping the corner. Actually, the real problem here is not the local optimum within the feature, but the fact that the total energy of the feature is quite small. Each mutation chain has a fixed amount of energy to redistribute. All this energy will be distributed over paths within a single feature. If the total feature energy is very small, the probability of generating a mutation chain for this feature is also small, but if one is generated, all chain energy is distributed over this low energy feature. If many low energy features are present in the scene, the result is that only a few such features appear in the image, but they will seem to be oversampled and too bright. In the example case, this becomes visible as bright splotchy errors in corners. Figure 9.4 shows an example of such errors. This shows that not only the size of the features, but also the energy distribution within the features determines the kind of errors in the resulting image.

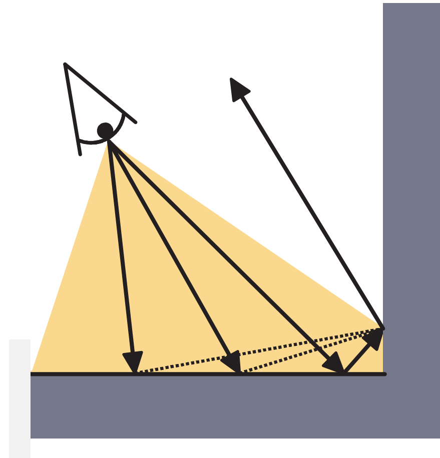


Figure 9.3: Mutations have a hard time escaping small corners due to a local optimum in mutation acceptance probabilities.

Figure 9.4: Splotchy errors in the corners due to low energy feature.

Image space feature

We use ERPT to compute an image, so it seems natural to look at feature properties in image space. The feature energy $E(\Phi)$ is the amount of energy reaching the image plane along the feature paths. This energy is distributed over the pixels in the image plane. However, a feature does not contribute equally to all image pixels. The image space contribution of a feature is defined by the distribution of feature energy over the image plane. Let us say

that $I_i(\Phi)$ is the energy contribution to pixel i from all paths in feature Φ. We can now for example define the size of a feature in image space as $|\{i|I_i(\Phi) > 0\}|$; the number of pixels this feature contributes energy to. Most of the feature characteristics described in previous sections apply directly to image space. When an feature is only a few pixels in size or its distribution is very uneven, contributing mostly to a few pixels, most of the mutation chain energy is distributed over these few pixels. Furthermore, when the feature's image space size is small, the probability of finding initial mutation chain paths for a feature through path tracing decreases because only a few path tracing samples are generated per pixel. This can cause a large increase in noise as by chance, some small features are explored exhaustively while others are completely ignored. Complex caustic paths form a good example. Caustic paths are hard to find through path tracing and the corresponding features are often small and complex in shape. This results in low probability mutation chains contributing to one or two pixels. These chains appear as bright speckles on the image plane.

9.2.3 Mutation improvements

As we will see later, to achieve high GPU efficiency the GPU implementation of ERPT uses many mutations per initial path to explore a feature. To reduce artifacts caused by small features, we use an improved mutation strategy, significantly increasing the average feature size.

First of all, as already suggested in the original presentation of ERPT [11], we mix the lens and caustics mutations whenever possible. Some features are better sampled using one or the other, but in our experience, it is not completely evident when to use which. Furthermore, using both mutation types together can only increase feature sizes, which it often does with success.

The second, more important extension is, instead of always mutating the least amount of path vertices, to determine the number of path vertices to mutate using Russian roulette. We start with mutating the least number of path vertices, just like the original mutation strategy, and repeatedly use Russian roulette to determine whether more path vertices require mutating. For lens mutations, this means that we start by mutating the primary ray and propagate these mutations until either the whole path is mutated, or a vertex is reached that may be connected directly to the next path vertex on the original path. In the last case, instead of immediately making an explicit connection, we use Russian roulette to determine whether to stop here and make a connection or go on mutating more path vertices. This process is repeated until either Russian roulette terminates the mutation or the whole path is mutated. In the last case, the light vertex itself is also mutated and no explicit connection is made.

A similar method is used for caustic mutations. Because caustic mutations are no longer only used to sample caustics, we refer to them as *light mutations*, as they mutate the path backwards from the light towards the eye. Because light mutations mutate backwards, we first need to determine where to start the mutation before the vertices are actually mutated. Except for the first path vertex, each diffuse vertex on the path is a candidate to start the light mutation. We start with the first candidate along the path as seen from the eye. We again use Russian roulette to determine whether to stop and use this candidate or go on to the next. When we reach the light source, the last path vertex is automatically used. When

a mutation vertex is selected, the light mutation is constructed, starting from this selected vertex and going backwards, mutating all path vertices until the first path vertex is mutated and connected to the eye.

Note that for both mutation strategies, the original strategy is a special case of our improved strategy, having a termination probability of 1 during Russian roulette.

Finally, we mixed the original lens mutation with the lens mutation strategy proposed by Lai [57]. Whenever the current vertex on a mutated path is diffuse while the next vertex is specular, one of these strategies is chosen at random and used to extend the mutation. When the original strategy is chosen, the outgoing direction is perturbed and the mutation is extended. When Lai's mutation strategy is chosen, the current path vertex is directly connected to the next specular vertex on the original path and the mutation is extended. Finding small mutations for complex paths is often easier when making such explicit connections to specular vertices.

These improvements increase the feature size at the cost of a small increase in the required number of traversed rays per mutation. Due to Russian roulette, the probability of mutating n path vertices roughly decreases exponentially with n, therefore we expect that the average number of traversed rays per mutation will only increase slightly. To give a simple example: in a closed indoor scene with only diffuse materials and a mutation Russian roulette probability of $\frac{1}{2}$, the expected number of traversed rays per mutation is bounded by $2+\sum\frac{1}{2^n}=3$ when using mutation Russian roulette, only a single ray more compared to not using Russian roulette. We expect this to be a reasonable upper bound for most practical scenes. In section 9.3.4, we will measure the increase in traversed rays per mutation for a few well known scenes, confirming our expected upper bound.

Many of the problems with mutations discussed in this section are significantly reduced by using Russian roulette to determine mutation length. Because all path vertices are mutated with non-zero probability, the feature sizes are significantly increased. At the same time, most mutations only mutate the first few path vertices, which keeps the average sampling cost low and is useful for exploring small features. During lens mutations, all vertices, including the light vertex, may be mutated. Therefore, the aforementioned problems due to correlation between light vertices such as sharp shadows and splotchy caustics are significantly reduced. Note that light mutations are always mixed with lens mutation, so although light mutations never mutate the light vertex, the last path vertex for such paths may be mutated during lens mutations. Finally, because the second path vertex may be mutated, paths are no longer trapped in small corners, removing the splotchy errors in corners. See figure 9.5 for a comparison between the original and improved mutation strategy. As you can see, most problems with the original mutation strategy have vanished or are significantly reduced. Our improved mutation strategy trades structural noise for more uniform noise. This is most visible in figure 9.5(b), where noise appears to have increased compared to figure 9.5(a). Appearances are however deceiving; the noise in figure 9.5(a) is much more structural and is perceived as artifacts instead of noise. Remember however that statistically speaking, these artifacts are still unbiased noise.

We expect that the mutation strategy may be further improved by incorporating ideas from Kelemen [31]. As we will see later, our GPU ERPT implementation requires all paths in a mutation chain to be of the same length in order to achieve high parallelism. The mu-

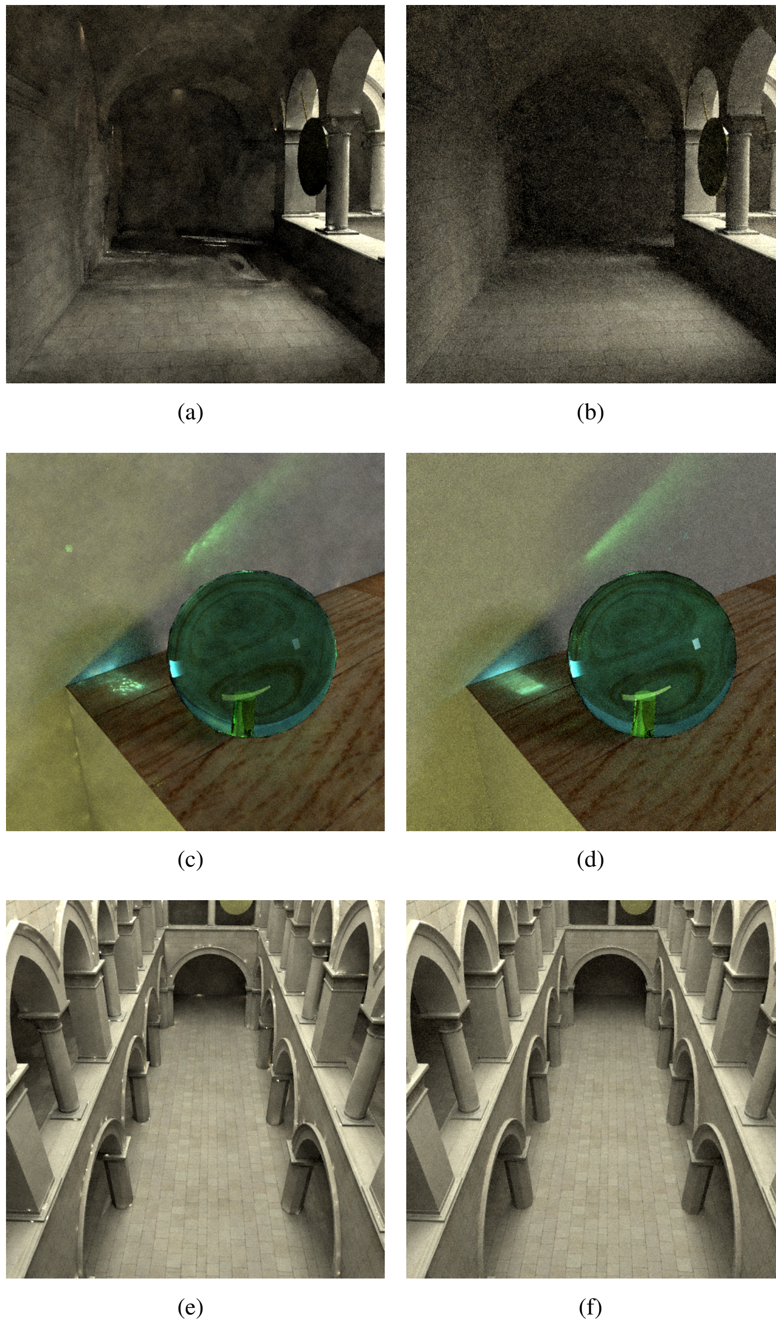


Figure 9.5: Original mutation strategy (a),(c),(e) vs. Improved mutation strategy (b),(d),(f). Computation times were similar for both strategies. Our improvement trades structural noise for more uniform noise, reducing visible artifacts.

tation strategy proposed by Kelemen also mutate the length of the path. Therefore, it is not trivial to efficiently implement the mutation strategy from Kelemen on the GPU. How-

ever, by letting the perturbation size for the outgoing direction of a vertex depend on the incoming direction and the local BSDF of a vertex, the acceptance probability may be further increased. This seems especially beneficial for near-specular materials. Furthermore, in our implementation a specular vertex must remain specular after mutation. This results in smaller features, especially when complex caustics are present. Dropping this constraint will increase feature size even further, reducing speckles which are caused by difficult caustic features.

9.2.4 Filter

For most mutation strategies, including ours, it is unavoidable that some scenes will contain high energy features with a small image space size. As discussed earlier, this may result in high energy speckles. One cause of speckles that is hard to completely prevent is that of floating point imprecisions. For example, in most ray tracing implementations, it is possible (although unlikely) for a ray to pass through two triangles at their connecting edge due to arithmatic imprecisions in the intersection computations. If the energy on such a path is redistributed, this may result in a feature consisting of only a single path causing a one-pixel sized speckle. In the original ERPT paper, consecutive sample filtering is used to solve this problem. The consecutive sample filter prevents a mutation chain from getting stuck to long at the same pixel. If it has been stuck at the same pixel for some fixed number of consecutive mutations, the mutation chain will stop contributing energy until it has moved away from the pixel. This filter reduces speckles, but makes the sampling method irreversibly biased. We use a simple post-processing filter to remove speckles. We compute an unbiased (progressive) ERPT image, and remove speckles if necessary. For each pixel, the filter works on a region of nearby pixels [2]. First, we compute the color intensity average and variance within the region. If the distance between the pixel's intensity and the region's average is at least twice the region's variance, the pixel belongs to a speckle and is ignored. Instead, the region's average color is used.

9.3 GPU ERPT

9.3.1 Overview

In this section, we start with a short overview of our GPU ERPT implementation. The ERPT implementation is an extension of the TPPT implementation from chapter 7. Besides path tracing, an ERPT sampler also performs energy redistribution. Whenever a sampler finds a light transport path during path tracing, its energy is redistributed using a multiple of 32 mutation chains running in parallel. ERPT samplers are run in groups of 32 samplers. Each group maps to one GPU warp. During path tracing, each thread runs its own sampler, similar to the TPPT implementation. However, during energy redistribution, all threads in a warp work together on a single sampler from the sampler group, all other samplers in the group are temporarily suspended until energy redistribution for the one sampler is complete.

[2] The region size depends on the maximum speckle size we want to remove. We used a region of 3×3 pixels around each pixel.

To allow high SIMT parallelism, all threads in a warp follow the same code path. All warps initially start path tracing. Whenever any thread in a warp completes a light transport path, path tracing is suspended and the path is distributed to all threads in that warp. The threads start redistributing the path's energy in parallel. During a mutation, mutation type selection and Russian roulette choices are shared between threads in a warp, causing all threads to simultaneously mutate the same path vertices. This allows for high SIMT efficiency and coalesced memory access. When energy redistribution for the path is finished, the warp resumes path tracing.

In the following section, we will discuss our ERPT sampler. The GPU implementation is discussed in section 9.3.3. Finally, in section 9.3.4, we will discuss the results of our implementation.

9.3.2 ERPT sampler

In this section, we present the ERPT sampler. The ERPT sampler is an extension of the TPPT sampler. The flowchart for the sampler is shown in figure 9.6. It consists of a path tracing sub chart(left) and a mutation subroutine(right). The path tracing sub chart looks very similar to the TPPT flowchart in figure 7.1, with a few changes. Most noticeable are the invocations of the mutate routine; each time a light transport path is constructed, the mutate routine is executed to redistribute the paths energy. Complete light transport paths may be found during the *explicit* and *implicit* steps, so these steps are immediately followed by the mutate routine.

Another change is the merging of the *extend* and *generate* steps. Furthermore, the transitions from *intersect* to *regenerate* and from *connect* to *extend* are removed. All steps are executed in a cycle without any shortcuts. Whenever a step in the cycle is not applicable to a sampler (for example, the *explicit* step is inapplicable when no connection was made), the step is ignored. This setup has the advantage that the sampler flowchart more closely resembles a GPU thread's control flow. We will discuss this further in section 9.3.3. Note that the flowchart is no longer divided in two phases. Each transition that possibly requires ray tracing is indicated by a double bar through the transition. Like TPPT, the ERPT sampler uses sample regeneration after a sample is complete.

Mutation routine

Now let us turn to the mutate routine. Whenever a sampler finds some light transport path $\mathbf{X}$, the mutation routine distributes its energy using mutation chains. Mutation chains are generated in batches of 32 chains. Remember from section 2.10 that the number of chains *numChains* for a light transport path $\mathbf{X}$ depends on the path energy $f(\mathbf{X})$ and equals $numChains = \left\lfloor U(0,1) + \frac{1}{N \times e_d} \frac{f(\mathbf{X})}{p(\mathbf{X})} \right\rfloor$, with N the number of mutations per chain and e_d the energy quantum. In order to generate chain batches of 32 chains each, while remaining unbiased, we use $numChainBatches = \left\lfloor U(0,1) + \frac{1}{32 \times N \times e_d} \frac{f(\mathbf{X})}{p(\mathbf{X})} \right\rfloor$.

The mutation chains in a batch are generated in parallel. A complete mutation chain is generated by repeatedly extending the chain by one mutation, until the chain consists of N mutations. The sampler generates all mutation chains in parallel, so it repeatedly generates

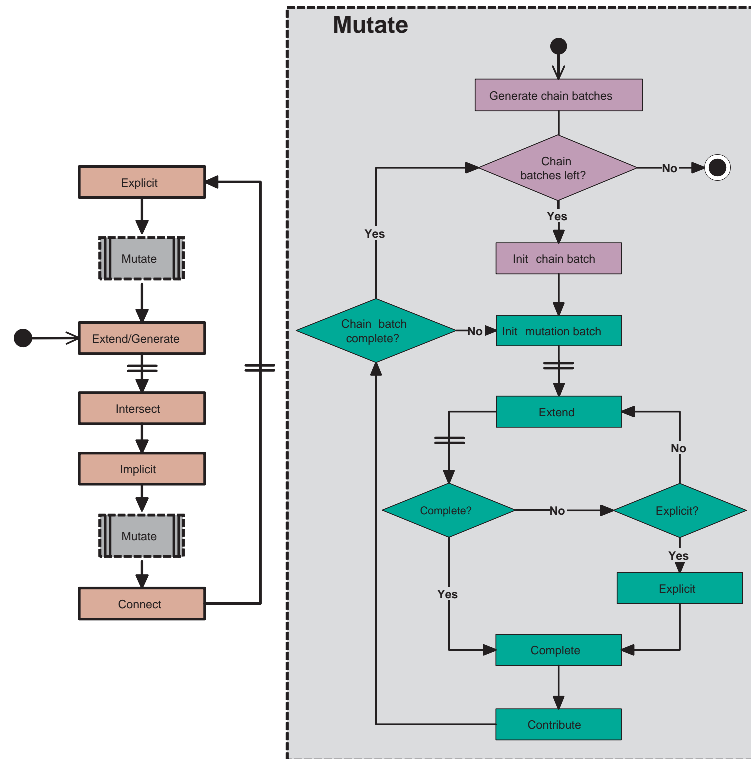


Figure 9.6: ERPT sampler flowchart. Extension of the PT sampler. Each time a light transport path may be created, the **mutate** sub-flowchart is executed. No distinction is made between phases, any transitions requiring ray tracing are indicated by a double bar through the transition arrow.

mutation batches of 32 mutations in parallel, one mutation for each chain in the chain batch. Repeating this N times gives 32 mutation chains of N mutations each. Algorithm 4 presents an overview of the mutate routine; it closely resembles the original ERPT method (see algorithm 2) except now, batches of 32 mutation chains are generated in parallel. Note that all chains in a batch start with the same light transport path. This introduces some correlation between mutation chains, especially between the first few mutations in the chains. This problem is mitigated by discarding the first M mutations of each mutation chain. We found that discarding the first 2 or 3 mutations is usually enough, unless the average acceptance probability is very low for mutations around the initial path. However, in such cases the mutation chains will have problems exploring the mutation feature anyway, causing speckles in the image. Using a despeckle filter solves this problem.

Another consequence of generating mutation chains in large batches from a single initial sample is that all mutation chains will explore the same mutation feature, resulting in $32N$ mutations per batch exploring a single feature. As explained in section 9.2.3, using our improved mutation strategy significantly increases feature sizes and reduces noticeable artifacts due to long mutation chains. This reduction applies equally well to the case of

multiple mutation chains exploring the same feature because chains tent to spread out rather fast. Using our improved mutation strategy therefore significantly reduces artifacts caused by generating mutation chains in batches.

Algorithm 4 : Mutate($\mathbf{X}$,N,e_d)

$numChainBatches \leftarrow \times \left\lfloor U(0,1) + \frac{1}{32 \times N \times e_d} \frac{f(\mathbf{X})}{p(\mathbf{X})} \right\rfloor$
for $i = 1$ to $numChainBatches$ **do**
  for $j = 1$ to 32 **in parallel do**
    $\mathbf{Y}_j \leftarrow \mathbf{X}$
  end for
  for j = 1 to N + M **do**
    for k = 1 to 32 **in parallel do**
      $\mathbf{Z}_k \leftarrow$mutate($\mathbf{Y}_k$)
      $a \leftarrow a(\mathbf{Y}_k \rightarrow \mathbf{Z}_k)$
      if $j > M$ **then**
        deposit ae_d energy at $\mathbf{Z}_k$
        deposit $(1-a)e_d$ energy at $\mathbf{Y}_k$
      end if
      if $a \geq U(0,1)$ **then**
        $\mathbf{Y}_k \leftarrow \mathbf{Z}_k$
      end if
    end for
  end for
end for

Now look back at the mutate routine in the flowchart in figure 9.6. The steps responsible for generating mutation batches are green and the steps for generating chain batches are purple. After a chain batch is initialized, the sampler starts the first mutation batch.

During mutation initialization, either a light or lens mutation is selected. The number of vertices to mutate is determined through Russian roulette. These choices are shared by all mutations in the mutation batch. To see why this sharing is possible, note that all mutation chains in a batch start with the same path. Also note that our mutation strategy neither mutates the path length, nor its signature (specularity of a vertex is preserved under mutation). Hence, all mutations in the batch have the same path signature. Because the choices of mutation type and number of vertices to mutate only depend on the path signature, it is possible to share the same choice between all mutations in the batch.

After initialization, the mutations are repeatedly extended by one extra vertex until the selected number of vertices is mutated. Note that for most mutation batches, an explicit connection is required to complete the mutations. When the mutation batch is complete, acceptance probabilities are computed, energy is deposited to the image and each mutation is either accepted or rejected. This process is repeated until N such mutation batches are generated after which the chain batch is complete. If there are unprocessed mutation chain batches left, the sampler starts the next chain batch. If not, the mutation routine finishes and

path tracing is resumed.

9.3.3 Implementation details

In this section, we describe the implementation details of our GPU implementation. We start by giving a general idea of our implementation before going into further details.

First off, in contrast to the previous tracers, the ERPT implementation restricts the maximum path length, introducing some bias into the image. This is necessary because the complete light transport path is needed during energy redistribution.

From chapter 3 we know that for high GPU performance, we require both parallelism in the form of SIMT efficiency and coalesced memory access. To allow high SIMT efficiency, all threads in a GPU warp should follow the same code path where possible. In our implementation, all threads in a warp execute the same step from the ERPT sampler flowchart (figure 9.6) in parallel. To realize this, samplers are grouped in groups of 32 samplers. The flow of control for samplers is generally handled per group instead of per sampler, so all samplers in a group follow the same steps in the flowchart. In practice, each sampler group maps to a single GPU warp. The first thread in a warp dictates the general control flow; all other threads follow. Whenever a step is not applicable to one of the samplers in the group, the corresponding thread remains idle until the warp finished this step. All memory access is coalesced because the sampler data in stored as a structure of arrays in memory, similar to the PT and SBDPT implementations.

The ERPT sampler is no longer divided in separate phases. Therefore, besides the general ray tracing kernel, the implementation consists of a single sampler kernel. Each iteration, the sampler kernel is executed for all sampler groups, generating a (possibly compacted) stream of output rays. Then, these rays are traced. The ray tracing results are used in the next iteration. As already mentioned, all samplers in a group share their flow of control. Different groups may however follow different code paths. For example, during an iteration, some groups may perform path tracing while others are distributing energy. During each iteration, each warp advances its corresponding sampler group until at least one of the samplers in the group requires ray tracing. The rays are output to the output stream and the warp execution is suspended. Looking at the flowchart, all transitions where ray output may occur are indicated by double bars (figure 9.6).

In the PT and SBDPT implementations, each sampler matches a single GPU thread, so threads in a warp did not explicitly work together. All parallelism was implicitly realized by the hardware. For the ERPT implementation, this is no longer true. Now, each sampler group maps to a single warp. During path tracing each sampler in a group still maps to a single thread in the corresponding warp, similar to the other GPU tracers. However, during energy redistribution all threads in the warp work on a single sampler, chosen from the sampler group. All other samplers in the group are temporarily suspended until energy redistribution for the selected sampler is complete. During energy redistribution, the threads in a warp work closely together to maintain parallelism.

In the next sections, we will give further details on the implementation of path tracing and energy redistribution.

Path Tracing

The path tracing part of the ERPT implementation accomplishes its GPU efficiency in exactly the same way the TPPT implementation does: the algorithmic steps are executed in a fixed order (top-down in the flowchart). If a step is not applicable to some threads, they remain idle until the other threads in the warp finish this step. At first sight, it may seem that the removal of shortcuts in the sampler flowchart significantly changes the execution flow, but this is not true. To see this, remember that on the GPU, threads in a warp always follow the same code path anyway, although some threads may be temporarily disabled. Therefore, when some threads take a shortcut, they still have to remain idle until all other threads not taking the shortcut have finished the remaining steps. Hence, the shortcuts are implicitly removed by the GPU hardware.

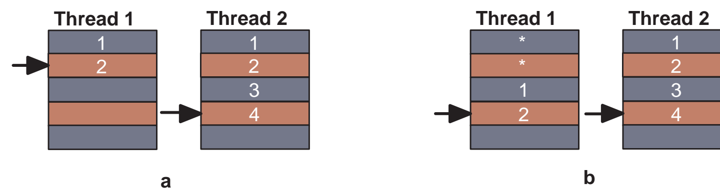


Figure 9.7: Path vertex access. **Thread 1** has regenerated its sampler after the first two iterations, resulting in a) Non-coalesced access in the vertex buffers b) Coalesced access in the cyclic vertex buffers. In a cyclic vertex buffer, the path does not need to start at the beginning of the buffer. By moving to the next vertex in the buffer after each *Extend/Generate* step, whether the path was regenerated or not, memory access remains coalesced after path regeneration.

Coalesced path vertex storage requires some extra work compared to TPPT. The reason for this is that during energy redistribution, the complete light transport path is mutated. Therefore, it is no longer possible to only store the last path vertex, as done in TPPT; All path vertices must be stored during path tracing. Because dynamic memory allocation per sampler does not seem like a feasible solution, we impose a maximum path length on the sampler, introducing some bias. Remember that during path tracing, each sampler maps to a single GPU thread. Instead of having enough memory to store a single vertex per thread, each thread now has a local vertex buffer to store all path vertices. The buffers for threads in a warp are stored as a structure of arrays so that when all threads in a warp access local vertices with the same index in their local vertex buffers, this results in one coalesced memory transaction serving all threads at once. Note that when the threads access different vertices at the same time, one coalesced memory transfer is required for each unique index used, resulting in reduced effective memory bandwidth.

Because Russian roulette is performed independently by all samplers, some samplers will be terminated and regenerated earlier than others. Therefore, during some iteration, threads in a warp may be working on paths of different lengths. As shown in figure 9.7a, this results in threads accessing different vertices, breaking coalesced memory access. Note however, that there is no need to store the first path vertex at the beginning of the vertex

```
__device__ int select_sampler( bool redistribute_energy )
{
        __shared__ volatile int index;

        index = -1;

        if( redistribute_energy )
                index = threadIdx.x;

        return index;
}
```

Listing 9.1: Code snippit for selecting a sampler from a group for energy redistribution

buffer. After each *Extend/Generate* step, a new path vertex is generated, whether the path was regenerated or not. By treating the vertex buffer as a cyclic buffer and storing the next vertex at the next spot in memory, even after regeneration, memory access remains coalesced (see figure 9.7b).

Energy Redistribution

During path tracing, each sampler requires only a single thread. However, during energy redistribution, each sampler requires 32 threads to process all mutation chains in a batch in parallel. Therefore, during energy redistribution, only a single sampler in the group is active. All 32 threads in the corresponding warp work together on this single sampler, allowing for parallelism during mutation chain batch generation. During the *Init chain batch* step, the warp checks if some samplers in the group require energy redistribution. If so, the threads collectively select one of the samplers to work on. Listing 9.1 shows a small CUDA code snippet for selecting a sampler for energy redistribution.

In this code, we make use of CUDA's write semantics for shared memory. Remember from section 3.4 that when multiple threads in a warp simultaneously write to the same shared memory, one of the writes is guaranteed to succeed. All samplers requiring energy redistribution write their index into the same shared variable. Afterwards, the variable will contain the index of one of these samplers. This sampler is selected for energy redistribution. After energy redistribution is complete, the selection procedure is repeated until no more samplers in the group require energy redistribution. Only then is path tracing resumed.

When a sampler is selected for energy redistribution, each thread in the warp starts processing one mutation chain. This takes multiple iterations to complete. All mutation chains start with the same path: the light transport path found by the selected sampler. During chain initialization, this path is distributed to all threads in the warp. After the mutation chains are initialized, all threads start generating mutations. During mutation initialization, the first thread in the warp determines the mutation type and number of vertices to mutate. These choices are communicated to the other threads through shared memory. Then, all

threads generate their next mutation in parallel. Although the mutation chains start with the same path, the chains will diverge due to randomization.

As explained in section 9.3.3, all threads in a warp follow the same code path because mutation choices are shared between mutations in a batch. This results in high GPU efficiency. Furthermore, because all threads will access the same path vertices during mutation, memory access is coalesced, allowing for high effective memory bandwidth. Note that efficiency is reduced when mutations fail early in their construction. When a mutation fails during construction, there is no reason to finish the mutation. Hence, the corresponding threads will remain idle until all other mutations in the batch are complete.

After a mutation batch is complete, energy is deposited on the image plane and each mutation is either rejected or accepted. Energy contributions to the image plane require scattered atomic writes to global memory, similar to the SBDPT implementation. Remember that scattered writes are relatively expensive compared to coalesced memory access. We can half the number of scattered writes by locally accumulating energy for the accepted mutation and only depositing accumulated energy for rejected mutations. In algorithms 2 and 4, two deposits are made after each mutation. However, one of the corresponding paths is used during the next mutation. By postponing the deposit for this path until later and accumulating the energy locally, only a single deposit is required per mutation, cutting the number of scattered writes in half.

Memory allocation

As explained earlier, because the whole path is required during energy distribution, each sampler requires enough memory to hold a path of maximum length. This memory is statically allocated beforehand for all samplers. However, this is not the only required vertex storage. During energy redistribution, a new mutation is constructed before it is either accepted or rejected. Hence, enough memory is needed to store both the old and new mutation per mutation chain. Statically allocating enough memory per sampler group for energy redistribution would increase the total memory footprint by almost 200%. Luckily, we can dynamically allocate memory storage for energy redistribution. Dynamic memory allocation during path tracing was prohibitively expensive because paths in a sampler group may have different lengths and are regenerated at a high frequency. However, this is not true for energy redistribution. During energy redistribution, all mutations in a batch have the same length. Furthermore, energy redistribution spans many iterations, as each mutation chain consists of many mutations and each mutation requires multiple iterations. Hence, dynamic memory allocation for energy redistribution will occur much less frequently, making it a practical solution.

Vertices are allocated in blocks of 64 vertices; two for each mutation chain in the batch, stored as a structure of arrays. During mutation chain initializations, the sampler allocates as many blocks as there are vertices on the mutation path. Dynamic allocation is performed using atomic instructions. A large pool of vertex blocks is statically allocated. Pointers to all vertex blocks are stored on a FIFO list. Then, vertex blocks are allocated by removing pointers from the head of the list and blocks are released by adding their pointers back at the tail of the list. We found that for most scenes, the total required memory for energy

redistribution is reduced by about 60% compared to static allocation. Due to its infrequent occurrence, dynamic memory allocation did not measurably influence the overall performance.

Specular prefix path

Having a maximum path length introduces bias into the result. Usually, this bias is hardly visible. However, one noticeable exception concerns paths starting with many specular vertices. For example, when looking directly at a complex reflective object, some paths require many reflections before reaching a diffuse surface. Restricting the path length may remove such paths, leaving parts of the scene completely black. Luckily, we can make an exception for this kind of paths. First of all, note that paths starting with a specular vertex only allow lens mutations. Furthermore, note that these specular vertices are not used during mutation. The only information required is the number of specular vertices at the beginning of a path. We call the sequence of specular vertices at the beginning of a path its specular prefix path. During path tracing, we do not store the specular prefix path, but only record the length of the specular prefix path. During energy redistribution, each mutation is expected to start with equally many specular vertices. This way, the number of specular vertices at the beginning of a light transport path is unbounded. Similar tricks could be used to allow any number of specular vertices on the whole path, only restricting the number of diffuse vertices, but at the cost of breaking coalesced memory access. We therefore only implemented specular prefix paths.

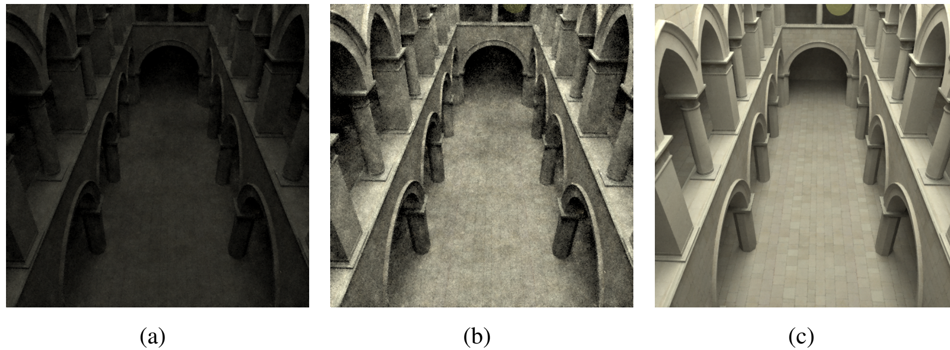
Progressive result

The ERPT sampler uses sample regeneration to fully utilize the GPU. As explained in section 6.2.4, a cool down period is required to obtain an unbiased result. The length of the cool down period depends on the number of iterations required to finish one sample. Compared to one PT sample, it may take a sampler many more iterations to complete one ERPT sample. Each mutation chain batch found by the sampler takes many iterations to complete. Also, whenever any of the other samplers in the corresponding sampler group finds a mutation chain batch, the sampler is suspended many iterations until this mutation chain batch is complete. Hence, the number of iterations to finish an ERPT sample may be large and varies significantly. This usually results in a long cool down period in order to obtain an unbiased result[3].

For an interactive tracer, it is important to visualize the progressive results every few iterations. However, the varying number of iterations to complete an ERPT sample poses a serious problem for estimating the progress. In order to display progressive results, an estimate of the total number of samples is required. A naïve estimate would simply count either the number of regenerated or terminated samples. Because samples may take many iterations to finish, these estimates will be very bad during the first few tens of iterations. The problem is that most samples will be somewhere halfway under construction. However, samples already contribute energy to the image plane during their construction. When only

[3]Disregarding the fact that the result will never be fully unbiased due to the restricted path length.

counting at termination, the sample progress is underestimated and the progressive result will be too bright. When counting at regeneration, the reverse is true and the progressive results will be too dark. This problem can be alleviated by roughly estimating the progress of samples. We count the total number of found mutation chains C_{chains}, including mutation chains that are not yet completed. Furthermore, we count the total number of completed mutations $C_{mutations}$. We already know how many mutations are generated per mutation chain, namely $32N$, so we can now estimate the progress of all found mutation chains as $\frac{C_{mutations}}{32NC_{chains}}$. We initially estimate the average number of samples by counting the number of regenerated samples, then we correct this estimate using the estimated mutation chain progress. The final estimate is still very rough, because there is no way of estimating the number of mutation chains that are still to be found for an unfinished sample. Although being a very rough estimation, we found it to be a much better estimation for progress. In practice, the remaining error in the estimation is much less visible, removing the heavy under-/overestimation of early progressive results. See figure 9.8 for a comparison.



(a) (b) (c)

Figure 9.8: ERPT progress estimation after a few iterations (a) without mutation chain progress correction (b) with mutation chain progress correction. (c) Reference image.

9.3.4 Results

In this section we present the results of our ERPT implementation. We start by comparing the original and improved mutation strategy. Then, we assess the SIMT efficiency of our implementations, followed by some performance results and a discussion on the convergence characteristics of the implementation.

9.3.5 Improved mutation

We start by comparing the original mutation strategy, which we will now call the short mutation strategy, and the improved mutation strategy, or Russian roulette mutation strategy. First, we compare the average number of traversed rays required per mutation. Table 9.1 shows the results. As expected in section 9.2.3, on average the mutations require no more than one extra ray traversal per mutation. For most scenes, the average number of rays only increases marginally. The average number of rays increases most for the INVISIBLE

DATE scene. This is because the scene is mostly lit by very long paths, so consequently the corresponding mutations also mutate more path vertices on average.

Average rays per mutation		
	Short	**Russian roulette**
SPONZA	2.00	2.18
SIBENIK	2.00	2.16
STANFORD	3.12	3.25
FAIRY FOREST	2.00	2.03
GLASS EGG	2.01	2.30
INVISIBLE DATE	2.00	2.77
CONFERENCE	2.00	2.06

Table 9.1: Average number of traversed rays per mutation for short mutations and Russian roulette mutations.

Table 9.2 shows the average acceptance probability of the short and Russian roulette mutation strategy. It is interesting to see that the average acceptance probability is decreased for the Russian roulette mutation. An important reason for this decrease is probably because mutating multiple vertices causes larger changes in the overall path, reducing the probability of generating a valid mutation candidate. For example, in the INVISIBLE DATE scene, large changes to multiple path vertices can easily cause the mutation to hit the door instead of the crack in the door. This affectively reduces the acceptance probability.

Even though the acceptance probability is reduced, section 9.2.3 showed a significant reduction in objectional noise in the end result. This shows that a high average acceptance probability does not guarantee an effective mutation strategy for ERPT. Especially when long mutation chains are used, having large mutation features is at least as important for an effective mutation strategy as the average acceptance probability.

Average mutation acceptance		
	Short	**Russian roulette**
SPONZA	0.83	0.69
SIBENIK	0.80	0.64
STANFORD	0.58	0.44
FAIRY FOREST	0.84	0.60
GLASS EGG	0.83	0.70
INVISIBLE DATE	0.83	0.51
CONFERENCE	0.84	0.73

Table 9.2: Average mutation acceptance for short mutations and Russian roulette mutations.

9.3.6 Efficiency

Tables 9.3 and 9.4 show the efficiency and occurrence of the ERPT sampler steps. The sampler steps are separated in path tracing steps (table 9.3), used to generate initial samples for energy redistribution, and energy redistribution steps (table 9.4), generating the actual mutation chains. Because most time is spent in energy redistribution, the occurrence of path tracing steps is very low. However, because all threads in a warp are either redistributing energy or tracing paths, this does not reduce the average efficiency of path tracing steps, which is roughly comparable to the TPPT efficiency from chapter 7. Note that the **Implicit** step is removed and incorporated in the **Intersect** step.

Results										
	Regenerate		**Extend**		**Intersect**		**Explicit**		**Connect**	
	Ef	Oc	Ef	Oc	Ef	Oc	Ef	Oc	Ef	Oc
SPONZA	52	0	96	0	100	0	78	0	96	0
SIBENIK	50	3	99	3	100	3	72	3	99	3
STANFORD	33	2	98	2	100	2	47	2	99	2
FAIRY FOREST	58	0	82	0	100	0	77	0	81	0
GLASS EGG	51	0	96	0	100	0	33	0	97	0
INVISIBLE DATE	50	0	99	0	100	0	60	0	100	0
CONFERENCE	51	0	98	0	100	0	68	0	99	0

Table 9.3: SIMT efficiency and occurrence of path tracing steps in an average ERPT iteration.

As shown in table 9.4, the energy redistribution steps all have very high efficiency. Again, occurrence is not very high, but this does not reduce efficiency because all threads in a warp follow the same code path. Note that because PT steps have relatively low occurrence compared to energy redistribution steps, the overall ERPT performance is mostly determined by the energy redistribution steps. As these steps show very high efficiency, the ERPT implementation achieves high SIMT efficiency.

Results												
	Bounce		**Connect**		**Init Mutation**		**Init ER**		**Contribute**		**Init Chain**	
	Ef	Oc	Ef	Oc	Ef	Oc	Ef	Oc	Ef	Oc	Ef	Oc
SPONZA	96	55	89	31	100	40	100	0	100	40	100	1
SIBENIK	96	49	89	27	100	35	100	3	100	35	100	3
STANFORD	86	64	81	15	100	23	100	2	100	23	100	2
FAIRY FOREST	98	55	89	33	100	43	100	0	100	43	100	0
GLASS EGG	96	58	95	32	100	39	100	0	100	39	100	0
INVISIBLE DATE	95	62	86	32	100	35	100	0	100	35	100	0
CONFERENCE	97	55	93	33	100	43	100	0	100	43	100	0

Table 9.4: SIMT efficiency and occurrence of mutation steps in an average ERPT iteration.

9.3.7 Performance

The high SIMT efficiency of the ERPT sampler translates in a high performance GPU implementation. Figure 9.9 shows the ERPT sampler performance in rays and mutations per second. It shows that the ERPT sampler achieves a relatively high performance, compared to the SBDPT from last chapter. The reason is mostly determined by the high SIMT efficiency of the ERPT sampler. Besides the high efficiency due to all threads in a warp following the same algorithmic steps, the SIMT efficiency within steps is also relatively high because all threads in a warp work on mutations of the same signature. Therefore, at each bounce they expect the same material type, further increasing SIMT efficiency. Finally, because all threads in a warp start their mutation chain with the same initial sample, rays generated by a warp often show a reasonable amount of coherence, increasing the ray traversal performance.

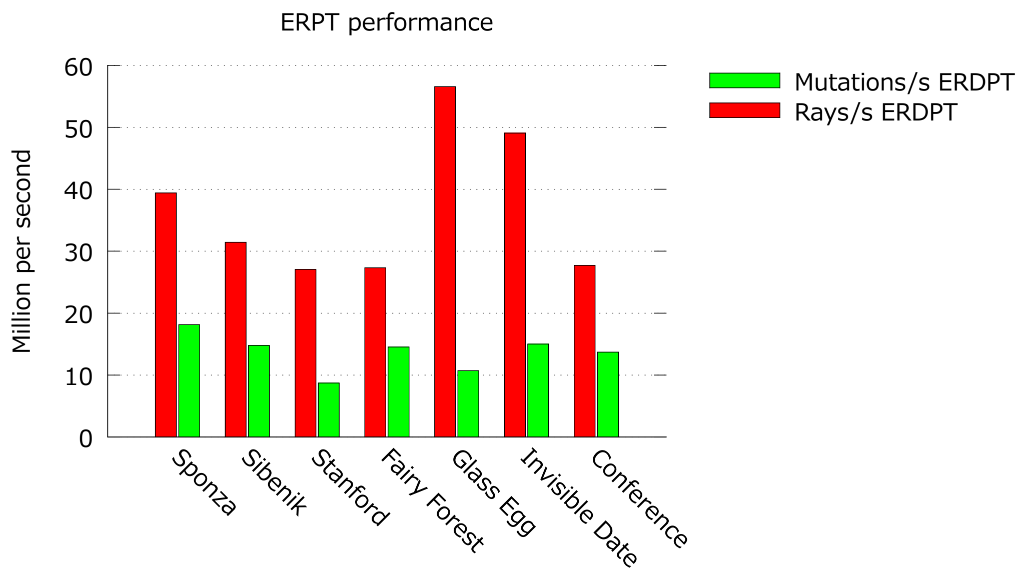


Figure 9.9: General ERPT performance in rays and mutations per second.

Figure 9.10 shows a time partition of an average ERPT iteration in phases. The percentage of time spent on sampler advancing is similar to the SBDPT from last chapter. Still, the overall performance is higher. This further indicates the increase in ray traversal performance due to increased ray coherence.

Finally, table 9.5 shows the memory footprint of the ERPT sampler. Because all path vertices must be stored for energy redistribution, the memory footprint depends on the maximum allowed path length. The table shows the default memory footprint and the extra memory required per allowed path vertex. When using dynamic mutation vertex allocation, less mutation vertices are required. Therefore, the memory footprint for path vertices and dynamic mutation vertices are also shown separately. The ERPT method has a relatively

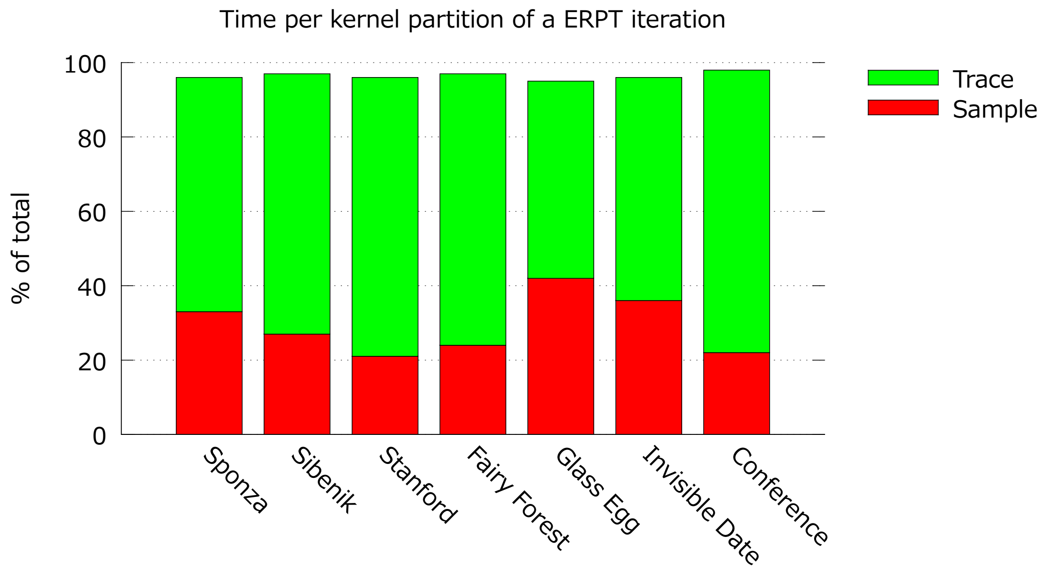


Figure 9.10: ERPT procentual time partition of an iteration.

large memory footprint. For the method to achieve high performance, a significant sampler stream is required. Therefore, this method requires a modern GPU with a significant amount of device memory.

Default memory usage per 256 warps	2752 Kb
Extra memory usage per static vertex	1824 Kb
Extra memory usage per static path vertex	608 Kb
Extra memory usage per dynamic mutation vertex	1217 Kb

Table 9.5: ERPT Memory usage for sampler and ray storage per 256 warps.

9.3.8 Convergence

ERPT renders some light effects much faster than PT or SBDPT. Figure 9.11 shows the caustic from the glass egg in the GLASS EGG scene. On the left, the caustic is rendered with SBDPT, while on the right, it is rendered with ERPT. Both renderings took equally long to render. The ERPT rendering is much smoother, having virtually no high frequency noise. The caustic however, is slightly darker because the progressive rendering result was cut off, skipping the cool down period required to remove all bias.

Figure 9.12 shows a comparison of ERPT with regular PT on the SPONZA scene. This image clearly shows that the ERPT and PT samplers both converge to the same image.

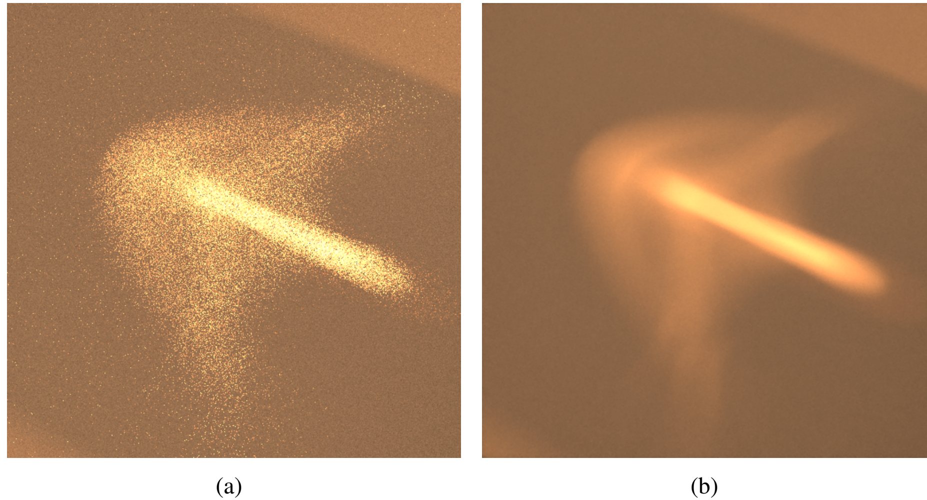


Figure 9.11: Complex caustic rendered with SBDPT (a) and ERPT (b). Computation time was similar for both images. The ERPT image is darker because progressive render was cut off, skipping the cool down period to remove all bias.

While the PT rendering still contains a significant amount of high frequency noise, the ERPT rendering however does not show noticeable noise.

The merits of ERPT are even more pronounced when applied to the INVISIBLE DATE scene. Figure 9.13 shows the INVISIBLE DATE scene rendered with SSPT, SBDPT and ERPT. All renderings took equally long to render. These images show that ERPT is much better at capturing light traveling through small cracks in the scene. For this scene, SBDPT is hardly any better than PT: finding paths through the crack in the door is still very unlikely. ERPT uses energy redistribution to exploit any found light paths, effectively exploring the scene illumination.

Although the ERPT method produces images with little high frequency noise, this does not mean the estimate has a smaller error. Figure 9.14 shows the STANFORD scene, rendered with SBDPT and ERPT. The SBDPT sampler still contains a lot of high frequency noise. On the other hand, many features are still completely missing from the ERPT rendering. This is an inherent property of exploring features through energy redistribution. Note that all missing features correspond to relatively long paths. This is a consequence of how the sampler is implemented on the GPU. During path tracing, short paths are encountered first. Therefore, at the start of the algorithm, mutation chains mostly handle short paths. Only after the energy of these short paths is distributed, is path tracing continued and can longer paths be found. Therefore, the initial progressive results usually lack light contributions from long paths.

Furthermore, when the scene contains very difficult and small features, such as the features corresponding to several reflections and refractions in the Stanford dragon, ERPT has a hard time distributing the energy, resulting in a few high energy speckles. These speckles are easily removed using a despeckle filter, but this means that the contribution of

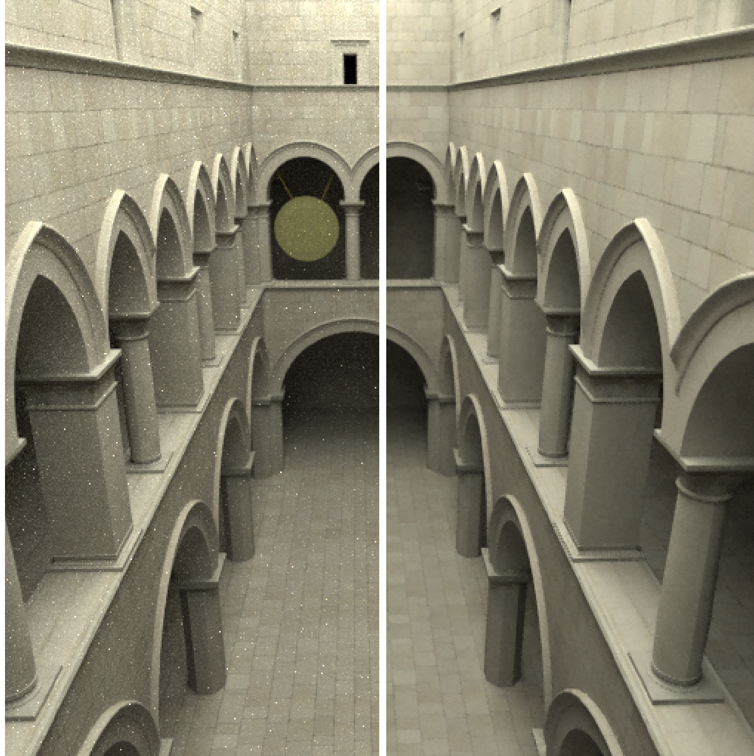
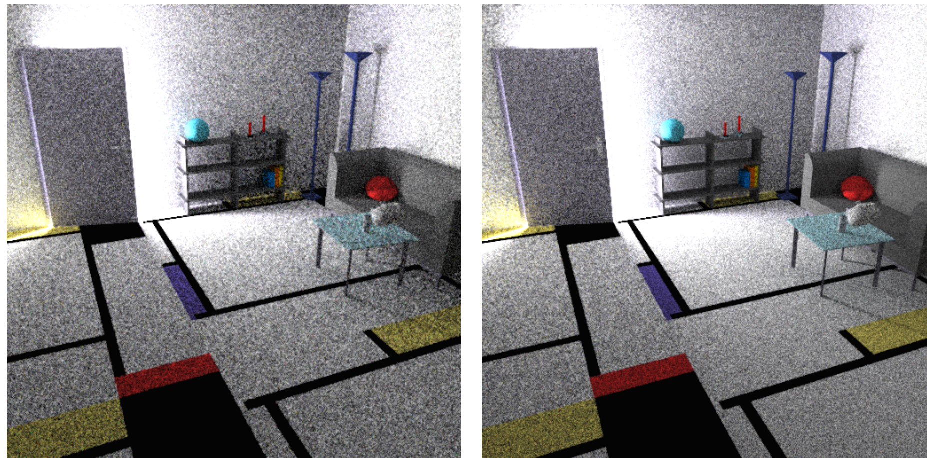


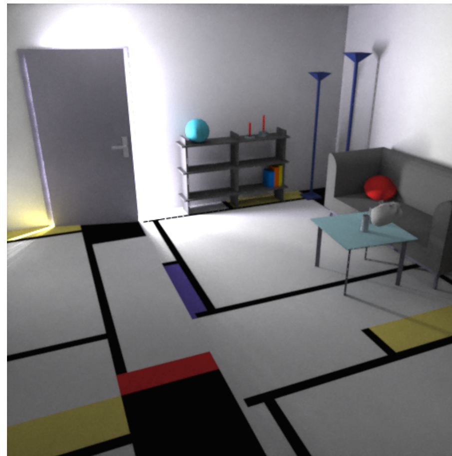
Figure 9.12: Sponza scene progressive rendering. The left half of the image is rendered with SSPT, the right half is rendered with ERPT. Rendering took 20 seconds.

the corresponding features lack from the final rendering.

In this section, we showed that the ERPT implementation achieves high performance. Furthermore, we showed that the ERPT has a relatively high convergence speed compared to PT and SBDPT when it comes to scenes with certain complex illumination effects, especially due to light traveling through small cracks in the scene. Finally, we showed that the lack of high frequency noise in ERPT renderings can sometimes be deceiving, because the image may still lack significant illumination effects.

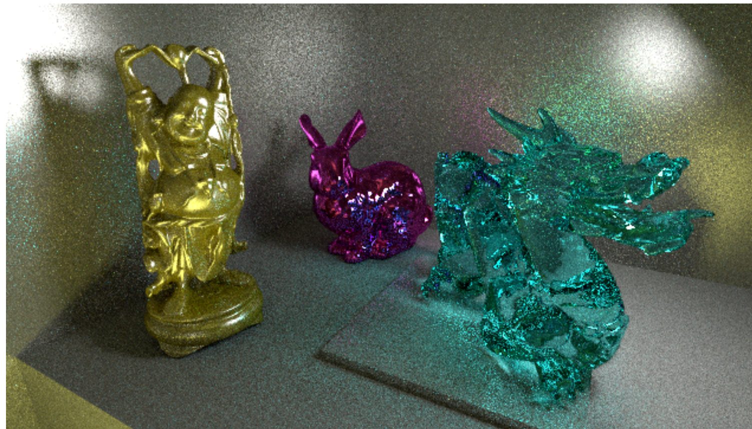


(a) (b)

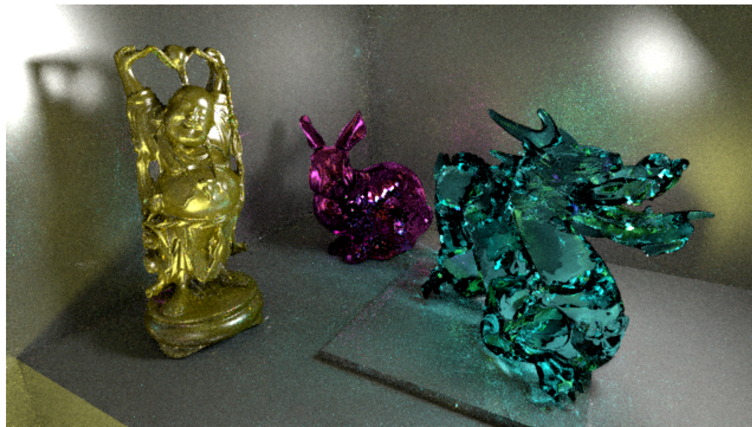


(c)

Figure 9.13: Invisible date scene rendered with SSPT (a), SBDPT (b) and ERPT (c). Computation time was similar for all images. SBDPT is only slightly better than SSPT. ERPT quality is much higher at the cost of a little bias due to maximum path length.



(a)



(b)

Figure 9.14: Stanford scene rendered with (a) BDPT and (b) ERPT. The BDPT rendering suffers from much more high frequency noise, but in the ERPT rendering, many features are still completely absent. Scenes with many small and complex features converge slowly with ERPT.

Part III

Results

Chapter 10

Comparison

In this section we compare the performance of the three GPU-only tracers: SSPT, SBDPT and ERPT. We do not consider TPPT here because it implements the same sampler as SSPT but performs consistently worse.

10.1 Performance

We will first compare the performance of the GPU implementations in executed samplers and traversed rays per second. We compare the ray traversal and sampler iteration performance independently. Figure 10.1 shows the performance in the number of iterations per second, excluding ray traversal times. This figure thus shows the performance of advancing samplers and generating output rays.

Note that the SSPT and SBDPT samplers execute different phases during each iteration and may therefore generate one extension ray and one connection ray during each iteration, while the ERPT sampler only executes a single phase per iteration and generates either one extension or one connection ray. For a fair performance comparison we count both the **Extend** and **Regenerate/Connect** phases as separate iterations in our measurements, effectively doubling the number of iterations per second for SSPT and SBDPT.

Figure 10.1 shows that SSPT performs considerably better than SBDPT and ERPT for all scenes. ERPT performs relatively worse and SBDPT ends last. The high performance of SSPT was to be expected. SSPT requires much less complicated computations (no MIS or mutation probability). Furthermore, the increased primary ray coherence in SSPT results in more coherent shading and thus GPU efficiency, increasing the sampler performance further. The same goes for the ERPT algorithm; because all mutations in a warp have the same length and path signature, ERPT samplers achieve high GPU efficiency, increasing the sampler performance. SBDPT on the other hand requires relatively high memory bandwidth due to coalesced memory access because both the eye and light path vertices are read during each iteration, even though only one of these is actually needed during extension (see section 8.4.2). Furthermore, SBDPT requires more computations in order to construct MIS weights. Hence, the reduced performance of the SBDPT sampler was to be expected.

Because SBDPT does not benefit from any kind of sampler coherence, the performance is very stable and scene-independent. The only exception is the INVISIBLE DATE scene where SBDPT reaches significantly higher performance because the scene's triangle data completely fits into the GPU's L2 cache and most connection rays fail.

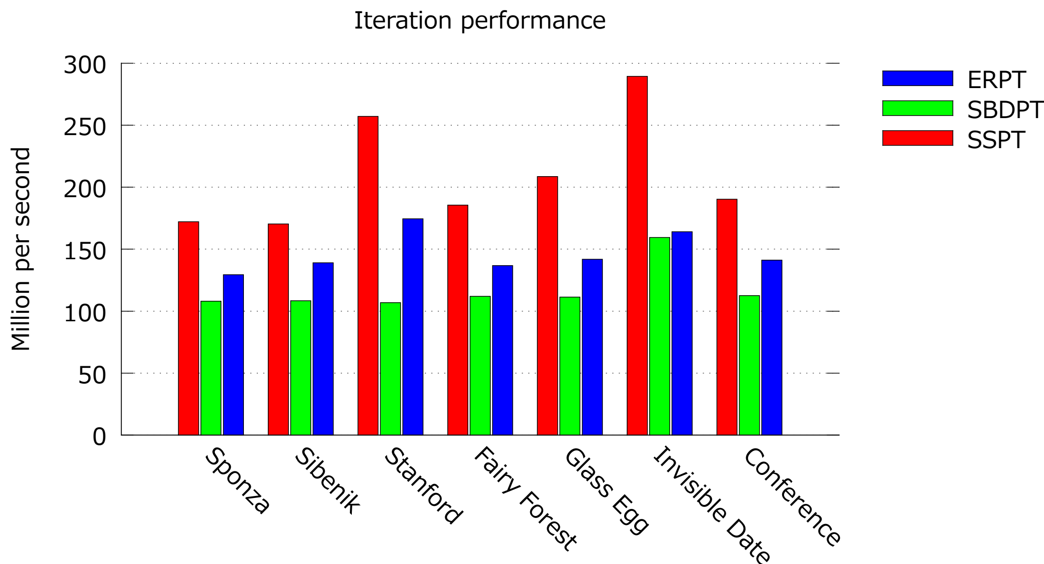


Figure 10.1: Iteration performance comparison in iterations per second. Ray traversal times are not included in these figures.

Even though there are significant differences in sampler performance between the three GPU tracers, the maximum difference is no more than 150% and on average about 70%. Furthermore, for reasonably sized scenes, the performance hardly depends on scene size. Hence, the differences in sampler performance are not that large (significantly less than one order of magnitude) so the overall performance and scalability of these algorithms will be largely determined by their ray traversal performance.

Figure 10.2 shows the ray traversal performance in rays per second, excluding sampler iteration times. We used the same ray traversal kernel in all tracers [1]. Using a different traversal algorithm, for example one using a kd-tree as its spatial structure, would obviously give significantly different results. We believe however that many of the observations drawn from these results will be similar for most traversal algorithms because traversal algorithms usually benefit from ray coherence.

Although the same ray traversal kernel is used, the traversal performance for the different tracers differs by up to 70%. The SSPT and ERPT algorithms achieve roughly the same ray traversal performance, somewhat dependent on the particular scene. Again, the SBDPT algorithm performs worst on all scenes. The reasons for this difference in ray traversal performance are similar to those causing the difference in sampler iteration performance. The

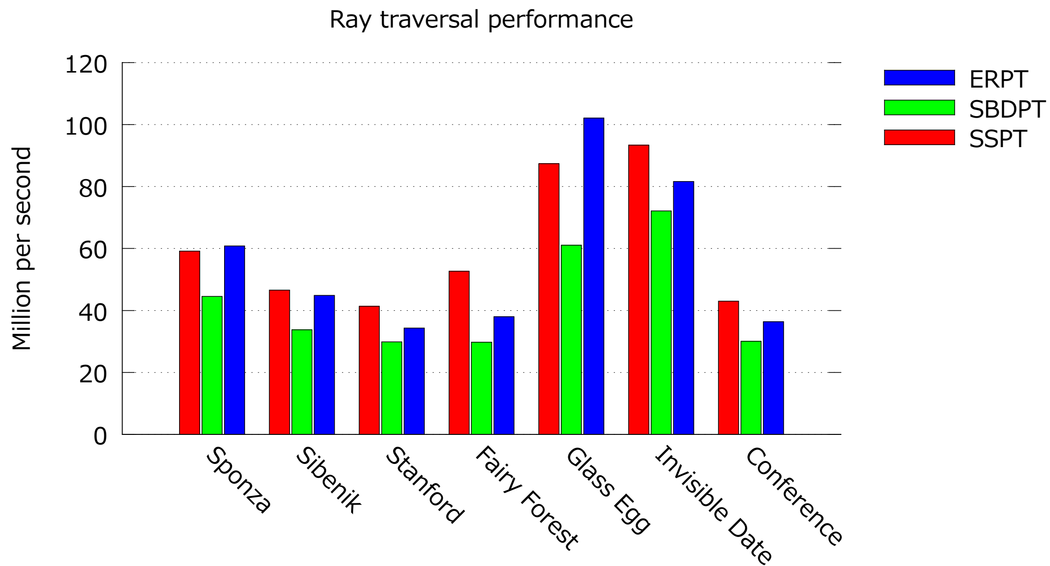


Figure 10.2: Ray traversal performance comparison in rays per second. Sampler iteration times are not included in these figures.

SSPT algorithm benefits heavily from primary ray coherence, achieving high traversal performance. Similar, the ERPT algorithm achieves some ray coherence because all mutation chains in a warp start with the same initial mutation. Therefore mutations in a warp will often contain a reasonable amount of coherence, resulting in high traversal performance. SBDPT does not benefit from such coherence. Furthermore, while explicit connection rays in path tracing all end in the same light sources, causing significant ray coherence, connection rays in BDPT connect eye and light vertices from all over the scene. Therefore, the average ray traversal performance for bidirectional connection rays is lower than for ordinary connection rays in path tracing. This further reduces the ray traversal performance of SBDPT.

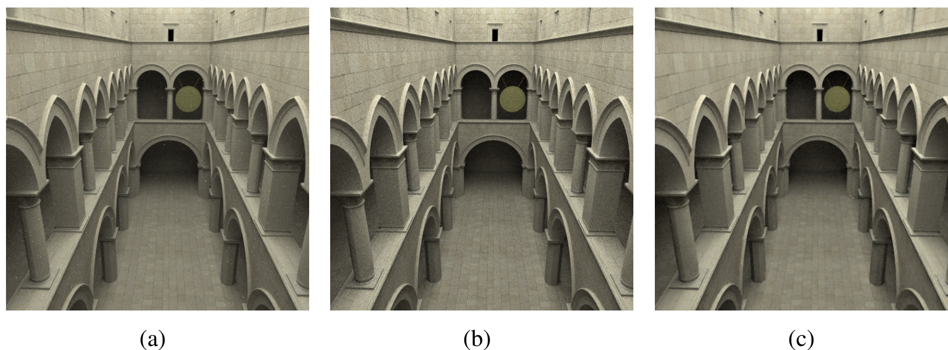
Although ray traversal performance differs significantly from one tracer to another, the differences remain limited. Figure 10.2 shows that ray traversal performance decreases somewhat for larger scenes, but although CONFERENCE is several orders of magnitude larger than INVISIBLE DATE, the ray traversal speed decreases by less than a factor of 3. As sampler performance barely depends on scene size, this indicates that our GPU tracers scale very well with scene size.

Lastly, a word on memory consumption. SSPT has the smallest memory footprint, requiring storage for only a single path vertex per sampler. SBDPT also has a relatively small memory footprint of two path vertices per sampler, independent of path length. ERPT on the other hand requires storage for all path vertices, making its memory footprint dependent on the maximum path length. This restricts the scalability of our ERPT implementation.

Therefore, while SSPT and SBDPT are suitable for older graphics devices with significantly less device memory, an efficient ERPT implementation requires a modern GPU with at least 1G of device memory.

10.2 Convergence

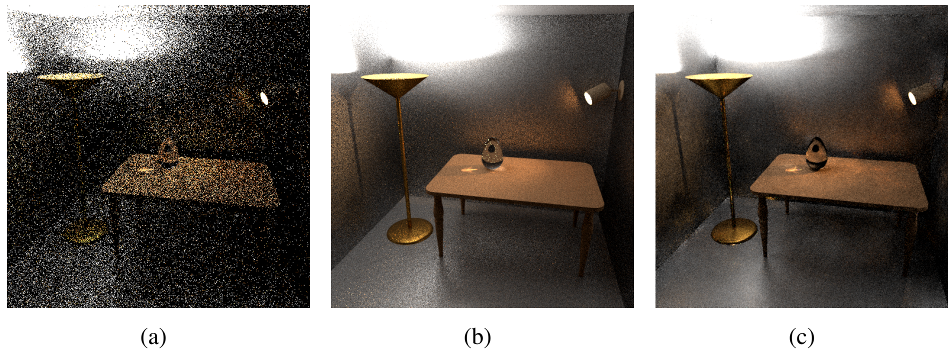
Although the performance measures from last section give some insight into the scalability of the implementations, they are not a very good measure for comparing the implementations. The goal of these algorithms is to compute a converged image. Therefore, to evaluate the implementations, we should compare the convergence speed of the GPU tracers. When it comes to convergence, there are considerable differences between these algorithms. Each algorithm has its strengths and weaknesses and its performance heavily depends on the particular scene. While some light effects are best captured with one algorithm, other effects are better sampled by another. To emphasize the strengths and weaknesses of the three implementations, we compared them using three scenes: SPONZA, GLASS EGG and INVISIBLE DATE. The progressive results for the three scenes are shown in figures 10.3, 10.4 and 10.5.



(a) (b) (c)

Figure 10.3: Sponza scene rendered with SSPT (a), SBDPT (b) and ERPT (c). Computation time was similar for all images (10 seconds).

For the SPONZA scene in figure 10.3, there is only little difference in quality between the three tracers. All three tracers have roughly the same noise levels. The SSPT has the highest noise level, showing significant high frequency noise over the image. The SBDPT and ERPT images have similar noise levels and it depends on personal taste which is preferable. The reason all methods perform equally well on SPONZA is because the scene does not contain any significantly complex illumination effects. The scene has a large accessible light source that is easily sampled backwards from the eye. Furthermore, the scene contains no complex materials, so complex light effects such as caustics are absent in the scene. As sampling light paths is relatively easy in this scene, advanced sampling methods such as BDPT and ERPT do not significantly improve over ordinary PT, resulting in similar convergence speed.



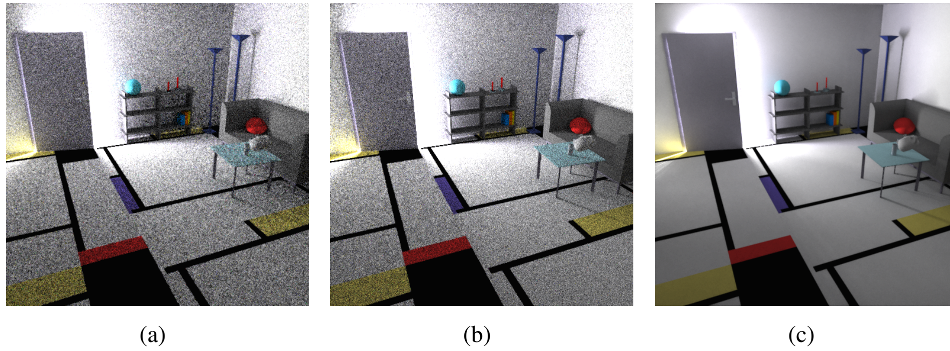
(a) (b) (c)

Figure 10.4: Glass Egg scene rendered with SSPT (a), SBDPT (b) and ERPT (c). Computation time was similar for all images (10 seconds).

On the other hand, the GLASS EGG scene in figure 10.4 shows a much more significant difference between the three tracers, caused by the small and inaccessible light sources and many complex materials in the scene. Both light sources contain small glass lenses, so the entire scene is lit by two large caustics. Hence, all explicit connections to the light source in the SSPT algorithm fail because of these lenses and all light must be sampled through implicit paths. For this reason, SSPT performs exceptionally bad on this scene. ERPT already performs much better than PT, but after only 10 seconds, the result is still noisy and splotchy due to many small features. For this scene, SBDPT performs best, effectively sampling the indirect light by sampling forward from the light sources. Note however that SBDPT still has a problem with rendering caustics seen through the glass egg. As explained in section 8.3.2, this is as expected. SBDPT does not contain any sampling strategy which samples these paths with relative ease; reflected caustics are only sampled through implicit eye paths. Hence, the SBDPT does not perform any better on these effects than the SSPT algorithm. ERPT does not suffer from this problem, sampling a reflected caustic is not more difficult than sampling any other caustic and as soon as an initial path for the reflected caustic is found, energy redistribution is used to explore similar paths in the same mutation feature.

Finally, the INVISIBLE DATE scene in figure 10.5 shows an exceptionally difficult scene. Although the scene only contains simple materials and has relatively low geometric complexity, it is considered very difficult because the entire scene is lit by indirect light passing through a relatively small crack in the scene geometry. Finding light transport paths through this crack is very unlikely, no matter which bidirectional sampling strategy is used. This causes, even after 30 seconds of render time, both SSPT and SBDPT to still produce very noisy images. SBDPT performs slightly better than SSPT, but the difference is not very significant. On the other hand, ERPT performs exceptionally good on this scene compared with the other two tracers. Whenever ERPT finds an initial light transport path through the crack in the scene, small mutations are used to easily produce many mutations passing through the crack, effectively exploring the illumination in the scene.

In general, SSPT performs well on scenes where the light source is easily reachable through backwards sampling from the eye. This includes most scenes with little indirect



(a) (b) (c)

Figure 10.5: Invisible Date scene rendered with SSPT (a), SBDPT (b) and ERPT (c). Computation time was similar for all images (30 seconds).

light and no caustics. SBDPT performs much better for scenes with lots of indirect light and complex materials, as long as there is at least one bidirectional sampling strategy capable of sampling each illumination effect with relative ease. This includes scenes lit by indirect light from light sources in confined spaces and scenes containing complex caustics. Finally, ERPT performs outstandingly for scenes with very inaccessible light sources due to small cracks in the geometry. ERPT performs reasonably well on complex materials, caustics and reflected caustics, but often suffers from small mutation features.

Our findings agree with reported results on CPU implementations of PT, BDPT and ERPT [50, 4]. Hence, the convergence characteristics of these algorithms are preserved in our GPU implementations.

Chapter 11

Conclusions and Future Work

In this thesis we sought to find efficient GPU implementations for unbiased physically based rendering methods. We started by investigating a hybrid architecture, having the sampler implemented on the CPU while using the GPU for ray traversal only. By implementing the sampler within a generic sampler framework, this hybrid architecture can easily be combined with any type of sampler. We showed that although very flexible, the hybrid architecture does not scale well, and advancing large streams of samplers on the CPU is easily limited by the memory bandwidth of the system's main memory.

Therefore, we presented streaming GPU implementations for three well known rendering methods: PT, BDPT and ERPT. Having both sampling and ray traversal implemented on the GPU, these implementations run entirely on the GPU and do not suffer from limited system memory bandwidth. We showed how we achieved high parallelism, SIMT efficiency and coalesced memory access in our implementations, required for an efficient GPU implementation.

We showed that by immediately packing output rays in a compact continuous stream of output rays, GPU ray traversal performance is increased. Furthermore, we showed that immediate sampler stream packing can be used to speed up the GPU PT implementation and increase ray traversal performance further by exploiting primary ray coherence.

The streaming implementation of BDPT (SBDPT) was obtained by using a recursive reformulation of the Multiple Importance Sampling weight computations. Using this formulation, SBDPT requires only storage for a single light and eye vertex in memory at any time during sample evaluation, making the methods memory-footprint independent of the path length and allowing for high SIMT efficiency.

In order to better understand the structural noise patterns produced by the ERPT algorithm, we introduced the notion of mutation features. Using this analysis we presented an improved mutation strategy, trading structural noise for more uniform noise. This allows for longer mutation chains or more mutation chains starting at the same initial sample, without introducing too much objectionable noise. This feature is used to achieve high GPU efficiency in our GPU ERPT implementation. By generating similar mutation chains in batches, all threads in a GPU warp are made to follow similar code paths, achieving high SIMT efficiency and coalesced memory access.

Finally, we showed that the convergence characteristics of the original samplers are preserved in our GPU implementations. Although the PT implementation achieves the highest GPU performance, this does not compensate for the higher variance in its estimate. BDPT often performs much better on scenes lit by indirect light and scenes containing many complex materials. ERPT performs exceptionally good at sampling scenes lit by indirect light passing through small cracks in the geometry and performs reasonably well on scenes with complex materials, although it still suffers from small mutation features. The large differences in variance between these methods shows that high iteration and ray traversal performance for GPU path tracers does not render more advanced sampling methods obsolete. This is especially true for scenes containing complex illumination effects. Therefore, for unbiased GPU rendering to be a valid alternative to CPU rendering, it is important to find efficient GPU implementations for advanced rendering methods. Our work contributes to this goal by presenting two such implementations for the BDPT and ERPT algorithms.

The GPU efficiency of our ERPT implementation is mainly based on the fact that all mutations in a warp have the same path length. This leaves significant freedom in the choice of mutation strategy. We believe that the mutation strategy could be further improved by drawing ideas from Kelemen's mutation strategy in pseudo-random number space [31]. By making perturbation sizes proportional to the local BSDF, the average acceptance probability of the mutation strategy could be further improved. Furthermore, by allowing the path signature to change under mutation, the mutation feature size could be further increased, reducing noise due to small mutation features. Also, similar to Kelemen, mutating complete PT samples instead of one light transport path at a time could further increase the average feature size, improving the mutation strategy [31]. Allowing path length to change under mutation would significantly increase the feature size, but we expect it to be difficult to efficiently implement such a mutation strategy on the GPU without sacrificing SIMT efficiency and coalesced memory access. Improving the mutation strategy requires further research.

We wonder whether it is possible to use bidirectional sampling strategies to construct light transport paths used for energy redistribution. Because bidirectional sampling strategies are much better at sampling many difficult illumination effects, complex mutation features will be more likely to be found. This might lead to a more even exploration of path space over time, reducing the lack of important features in early progressive results.

Because our PT, BDPT and ERPT implementations require a limited number of BSDF evaluations during each sampler phase, we except that these methods could be easily combined with the work of Hoberock on efficient deferred shading in GPU ray tracers [26]. Hoberock reports a significant increase in shading performance when using complex procedural BSDFs. We are curious whether these results would also apply to our BDPT and ERPT algorithms.

It is often beneficial to estimate the variance on the image plane and dedicate more samples to these apparently difficult areas. Adding such methods to our algorithms could further increase their performance [25, 42].

Finally, our implementations did not simulate advanced illumination effects such as subsurface scattering [29], participating media [43, 34] or spectral rendering [49]. For completeness, a physically based renderer should also support these features. Extending our implementation to support these features will require further research.

Bibliography

[1] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on gpus. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 145–149, New York, NY, USA, 2009. ACM.

[2] James Arvo, Philip Dutré, Alexander Keller, Henrik Wann Jensen, Art Owen, Matt Pharr, and Peter Shirley. Monte carlo ray tracing. Technical report, SIGGRAPH 2003 Course Notes, Course #44: Monte Carlo Ray Tracing, 2003.

[3] ATI. Stream technology. www.amd.com/stream.

[4] Christopher Batty. Implementing energy redistribution path tracing. http://www.cs.ubc.ca/ batty/projects/ERPT-report.pdf. Project report.

[5] Philippe Bekaert, Mateu Sbert, and John Halton. Accelerating path tracing by re-using paths. In *Proceedings of the 13th Eurographics workshop on Rendering*, EGRW '02, pages 125–134, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[6] Carsten Benthin. *Realtime Ray Tracing on Current CPU Architectures*. PhD thesis, Saarland University, Saarbrücken, Germany, 2006.

[7] Brian C. Budge, Tony Bernardin, Shubhabrata Sengupta, Ken Joy, and John D. Owens. Out-of-core data management for path tracing on hybrid resources. In *Proceedings of Eurographics 2009*, March 2009.

[8] Christophe Cassagnabère, François Rousselle, and Christophe Renaud. Cpu-gpu multithreaded programming model: Application to the path tracing with next event estimation algorithm. In *Proceedings of ISVC (2)*, pages 265–275, 2006.

[9] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[10] David Cline. A practical introduction to metropolis light transport. Technical report, 2005.

[11] David Cline, Justin Talbot, and Parris Egbert. Energy redistribution path tracing. In *Proceedings of ACM SIGGRAPH 2005*, SIGGRAPH '05, pages 1186–1195, New York, NY, USA, 2005. ACM.

[12] NVIDIA Corporation. Nvidia cuda c programming best practices guide. http://developer.nvidia.com/cuda, January 2009.

[13] NVIDIA Corporation. Nvidia cuda compute unified device architecture programming guide. http://developer.nvidia.com/cuda, January 2009.

[14] NVIDIA Corporation. Tuning cuda applications for fermi. http://developer.nvidia.com/cuda, December 2010.

[15] Julie Dorsey and Leonard McMillan. Computer graphics and architecture: state of the art and outlook for the future. *SIGGRAPH Comput. Graph.*, 32:45–48, February 1998.

[16] Philip Dutré, Kavita Bala, and Philippe Bekaert. *Advanced Global Illumination*. A. K. Peters, Ltd., Natick, MA, USA, 2002.

[17] Philip Dutré, Paul Heckbert, Vincent Ma, Fabio Pellacini, Robert Porschka, Mahesh Ramasubramanian, Cyril Soler, and Greg Ward. Global illumination compendium, 2001.

[18] Shaohua Fan. *Sequential Monte Carlo Methods for Physically Based Rendering*. PhD thesis, University of Wisconsin - Madison, Madison, WI, 2006.

[19] Tim Foley and Jeremy Sugerman. Kd-tree acceleration structures for a gpu raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '05, pages 15–22, New York, NY, USA, 2005. ACM.

[20] Kirill Garanzha and Charles Loop. Fast ray sorting and breadth-first packet traversal for GPU ray tracing. *Computer Graphics Forum*, 29(2), May 2010.

[21] KHRONOS Group. Opencl–the open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/, June 2010.

[22] Johannes Günther, Stefan Popov, Hans-Peter Seidel, and Philipp Slusallek. Realtime ray tracing on GPU with BVH-based packet traversal. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007*, pages 113–118, September 2007.

[23] Jiří Havel and Adam Herout. Yet faster ray-triangle intersection (using sse4). *IEEE Trans Vis Comput Graph*, 16(3):434–8.

[24] Paul S. Heckbert. Discontinuity meshing for radiosity. In *Third Eurographics Workshop on Rendering*, pages 203–226, 1992.

[25] Jared Hoberock and John C. Hart. Arbitrary importance functions for metropolis light transport. *Comput. Graph. Forum*, 29(6):1993–2003, 2010.

[26] Jared Hoberock, Victor Lu, Yuntao Jia, and John C. Hart. Stream compaction for deferred shading. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 173–180, New York, NY, USA, 2009. ACM.

[27] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive k-d tree gpu raytracing. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, I3D '07, pages 167–174, New York, NY, USA, 2007. ACM.

[28] Henrik Wann Jensen. Importance driven path tracing using the photon map. In *Proceedings of Eurographics Rendering Workshop*, pages 326–335. Springer-Verlag, 1995.

[29] Henrik Wann Jensen, Stephen R. Marschner, Marc Levoy, and Pat Hanrahan. A practical model for subsurface light transport. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 511–518, New York, NY, USA, 2001. ACM.

[30] James T. Kajiya. The rendering equation. In *Computer Graphics*, pages 143–150, 1986.

[31] Csaba Kelemen, László Szirmay-Kalos, György Antal, and Ferenc Csonka. A simple and robust mutation strategy for the metropolis light transport algorithm. In *Proceedings of Computer Graphics Forum*, pages 531–540, 2002.

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

[33] 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.

[34] Eric P. Lafortune and Yves D. Willems. Rendering participating media with bidirectional path tracing. In *Proceedings of Eurographics Rendering Workshop*, pages 91–100. Springer-Verlag/Wien, 1996.

[35] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. Fast bvh construction on gpus. *Comput. Graph. Forum*, 28(2):375–384, 2009.

[36] Fabien Lavignotte and Mathias Paulin. Scalable photon splatting for global illumination. In *Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, GRAPHITE '03, pages 203–ff, New York, NY, USA, 2003. ACM.

[37] H. Ludvigsen and A. C. Elster. Real-time ray tracing using nvidia optix. In *Proceedings of the European Association for Computer Graphics 31th Annual Conference: EUROGRAPHICS 2010, short papers*. The European Association for Computer Graphics, 2010.

[38] Microsoft. Directcompute. http://msdn.microsoft.com/directx/, June 2010.

[39] Bochang Moon, Yongyoung Byun, Tae-Joon Kim, Pio Claudio, Hye-Sun Kim, Yun-Ji Ban, Seung Woo Nam, and Sung-Eui Yoon. Cache-oblivious ray reordering. *ACM Trans. Graph.*, 29:28:1–28:10, July 2010.

[40] Jan Novák, Vlastimil Havran, and Carsten Daschbacher. Path regeneration for interactive path tracing. In *Proceedings of the European Association for Computer Graphics 28th Annual Conference: EUROGRAPHICS 2007, short papers*, pages 61–64. The European Association for Computer Graphics, 2010.

[41] Ryan Overbeck, Ravi Ramamoorthi, and William R. Mark. Large Ray Packets for Real-time Whitted Ray Tracing. In *Proceedings of IEEE/EG Symposium on Interactive Ray Tracing (IRT)*, pages 41—-48, Aug 2008.

[42] Ryan S. Overbeck, Craig Donner, and Ravi Ramamoorthi. Adaptive wavelet rendering. In *Proceedings of CM SIGGRAPH Asia 2009*, SIGGRAPH Asia '09, pages 140:1–140:12, New York, NY, USA, 2009. ACM.

[43] Mark Pauly, Thomas Kollig, and Alexander Keller. Metropolis light transport for participating media. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, pages 11–22, London, UK, 2000. Springer-Verlag.

[44] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum*, 26(3):415–424, September 2007. (Proceedings of Eurographics).

[45] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).

[46] Benjamin Segovia, Jean-Claude Iehl, and Bernard Péroche. Coherent Metropolis Light Transport with Multiple-Try Mutations. Technical Report RR-LIRIS-2007-015, LIRIS UMR 5205 CNRS/INSA de Lyon/Université Claude Bernard Lyon 1/Université Lumière Lyon 2/Ecole Centrale de Lyon, April 2007.

[47] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for gpu computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 97–106, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[48] Martin Stich, Heiko Friedrich, and Andreas Dietrich. Spatial splits in bounding volume hierarchies. In *Proceedings of High-Performance Graphics 2009*, 2009.

[49] Yinlong Sun. *A spectrum-based framework for realistic image synthesis*. PhD thesis, 2000. AAINQ61686.

[50] Eric Veach. Robust monte carlo methods for light transport simulation. 1998. Adviser-Guibas, Leonidas J.

[51] Eric Veach and Leonidas J. Guibas. Optimally combining sampling techniques for monte carlo rendering, 1995.

[52] Eric Veach and Leonidas J. Guibas. Metropolis light transport. In *Proceedings of ACM SIGGRAPH 1997*, pages 65–76. Addison Wesley, 1997.

[53] Carsten Alexander Wächter. *Quasi-Monte Carlo Light Transport Simulation by Efficient Ray Tracing*. PhD thesis, Institut für Medieninformatik, Ochsenhausen, Germany, 2007.

[54] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive rendering with coherent ray tracing. In *Proceedings of Computer Graphics Forum*, pages 153–164, 2001.

[55] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23:343–349, June 1980.

[56] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *ISPASS'10*, pages 235–246, 2010.

[57] L. Zhang Y.-C. Lai, F. Liu and C. R. Dyer. Efficient schemes for monte carlo markov chain algorithms in global illumination. In *Proceedings of 4th International Symposium on Visual Computing*, 2008.

[58] S. Chenney Y.-C. Lai, S. Fan and C. R. Dyer. Photorealistic image rendering with population monte carlo energy redistribution. In *Proceedings of Eurographics Symposium on Rendering*, pages 287–296, 2007.

[59] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph.*, 27:126:1–126:11, December 2008.

Appendix A

Glossary

In this appendix we give an overview of frequently used terms and abbreviations.

Balance heuristic: Combination strategy for MIS

BDPT: BiDirectional Path Tracing

BSDF: Bidirectional Scattering Distribution Function

ERPT: Energy Redistribution Path Tracing

GPGPU: General Purpose Graphics Processing Unit

IS: Importance Sampling

MIS: Multiple Importance Sampling

MLT: Metropolis Light Transport

Power heuristic: Generic form of the balance heuristic

PT: Path Tracing

SBDPT: Streaming BiDirectional Path Tracing

SIMT: Single Instruction Multiple Threads

SM: Streaming Multiprocessor on the GPU

SSPT: Streaming Sampler Path Tracing

TPPT: Two-Phase Path Tracing

Warp: Group of 32 threads, executed in SIMT on the GPU

Appendix B

Sample probability

When applying the Monte Carlo method to the measurement equation, the probability of sampling a path must be computed with respect to the area product measure on path space (See section 2.3). Therefore, when sampling a light transport path, the probability of sampling one or a sequence of path vertices with respect to one measure must often be converted to the corresponding probability with respect to another measure in order to compute the contribution or the importance weights of such a path (See sections 2.1 and 2.6). In practice, one often needs to convert between units *solid angle*, *projected solid angle* and *surface area*. In this appendix, we derive some relations between probabilities with respect to these measures and derived product measures, relevant to the PT, BDPT and ERPT algorithms.

The relation between the units *solid angle* $d\sigma(\vec{\omega})$, *projected solid angle* $d\sigma^{\perp}(\vec{\omega})$ and *surface area* $dA(x)$ is depicted in figure B.1 and is given by

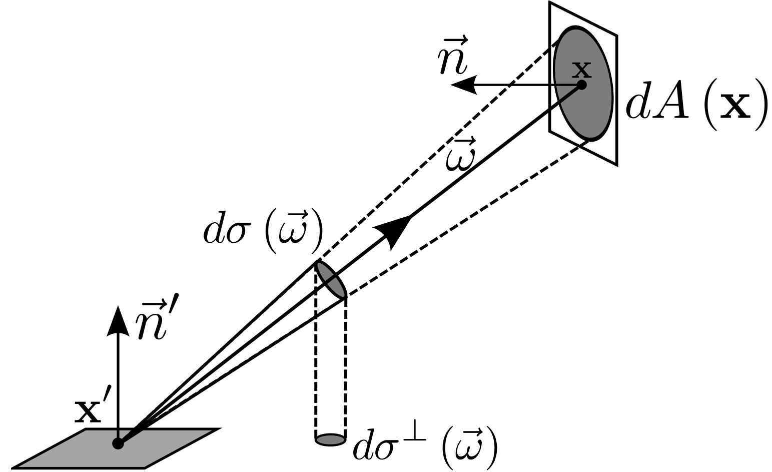


Figure B.1: The geometric relation between the units projected solid angle, solid angle and surface area.

$$d\sigma^{\perp}(\vec{\omega}) = \left|\vec{\omega}\cdot\vec{n}'\right| d\sigma(\vec{\omega}) = G\left(x \leftrightarrow x'\right) dA(x) \tag{B.1}$$

In this relation, $G(x \leftrightarrow x')$ is called the *geometric factor* and equals

$$G\left(x \leftrightarrow x'\right) = V\left(x \leftrightarrow x'\right) \frac{|\vec{\omega} \cdot \vec{n}| \, |\vec{\omega} \cdot \vec{n}'|}{|x - x'|^2} \tag{B.2}$$

where $V(x \leftrightarrow x')$ is called the *visibility factor*, being 1 iff the two surface points x and x' are visible from one another and 0 otherwise.

B.1 Vertex sampling

Figure B.2 shows three common methods for sampling a path vertex x_i, given the path vertices x_{i-1} and x_{i+1}. The first method simply samples x_i directly on the surface geometry according to some probability distribution $P_A(x_i)$ with respect to surface area (upper right). The second method samples x_i forward from x_{i-1}. It does this by sampling an outgoing direction from x_{i-1} per unit projected solid angle and tracing a ray into the scene to find x_i (lower left). Similarly, the third method samples x_i backwards from x_{i+1} (lower right). The

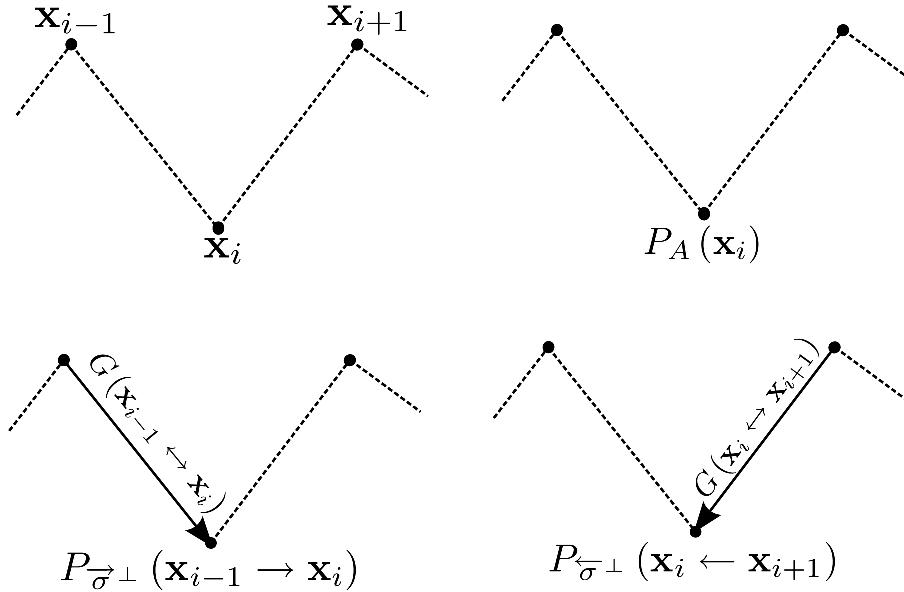


Figure B.2: Three methods for sampling path vertex x_i: Per unit surface area (upper right), through forward sampling from vertex x_{i-1} (bottom left) and through backward sampling from vertex x_{i+1} (bottom right).

probability of sampling x_i per projected solid angle through forward sampling is expressed as $P_{\vec{\sigma}^\perp}(x_{i-1} \to x_i)$. Similarly, the probability of sampling x_i backward per projected solid angle is expressed as $P_{\overleftarrow{\sigma}^\perp}(x_i \leftarrow x_{i+1})$. In order to compute the probability $P_A(x_i)$ of sampling x_i per unit surface area, these probabilities need to be converted from unit projected solid angle to unit surface area. Using equation B.1, these probabilities relate according to

$$G(x_i \leftrightarrow x_{i-1}) P_{\vec{\sigma}^\perp}(x_{i-1} \to x_i) = P_A(x_i) = G(x_i \leftrightarrow x_{i+1}) P_{\overleftarrow{\sigma}^\perp}(x_i \leftarrow x_{i+1}) \tag{B.3}$$

See figure B.2 for a visual representation of the sampling probabilities per unit surface area for forward and backward sampling.

B.2 Subpath sampling

A sequence of consecutive path vertices $x_s \cdots x_t$ is often sampled using either forward or backward sampling. Using the conversion factors from equation (B.3), the probability of sampling such a sequence per unit area can be expressed in terms of the probabilities for sampling the individual vertices per unit projected solid angle.

The probability of sampling the vertices $x_s \cdots x_t$ forward with respect to projected solid angle relates to the probability of sampling all vertices with respect to unit area according to

$$\prod_{i=s}^{t} G(x_i \leftrightarrow x_{i-1}) \prod_{i=s}^{t} P_{\overrightarrow{\sigma}^\perp}(x_{i-1} \rightarrow x_i) = \prod_{i=s}^{t} P_A(x_i) \tag{B.4}$$

Similarly, the probability for backward sampling and surface area sampling relate according to

$$\prod_{i=s}^{t} G(x_i \leftrightarrow x_{i+1}) \prod_{i=s}^{t} P_{\overleftarrow{\sigma}^\perp}(x_i \leftarrow x_{i+1}) = \prod_{i=s}^{t} P_A(x_i) \tag{B.5}$$

Using these two equations, it is possible to relate the backward sampling probability per projected solid angle for a sequence $x_s \cdots x_t$ to the corresponding forward sampling probability per projected solid angle using

$$\begin{aligned} \prod_{i=s}^{t} G(x_i \leftrightarrow x_{i+1}) \prod_{i=s}^{t} P_{\overleftarrow{\sigma}^\perp}(x_i \leftarrow x_{i+1}) &= \prod_{i=s}^{t} G(x_i \leftrightarrow x_{i-1}) \prod_{i=s}^{t} P_{\overrightarrow{\sigma}^\perp}(x_{i-1} \rightarrow x_i) \\ \prod_{i=s}^{t} P_{\overleftarrow{\sigma}^\perp}(x_i \leftarrow x_{i+1}) &= \frac{G(x_s \leftrightarrow x_{s-1})}{G(x_t \leftrightarrow x_{t+1})} \prod_{i=s}^{t} P_{\overrightarrow{\sigma}^\perp}(x_{i-1} \rightarrow x_i) \end{aligned} \tag{B.6}$$

B.3 Bidirectional sampling

In BDPT, some path vertices are sampled using backward sampling while other vertices are sampled using forward sampling or surface area sampling. Sampling a sequence of vertices by combining forward and backward sampling is called *bidirectional sampling*. For example, the sequence $x_r \cdots x_t$ can be constructed by sampling the vertices $x_r \cdots x_s$ using forward sampling and sampling $x_t \cdots x_{s+1}$ using backward sampling. The corresponding sampling probabilities per projected solid angle relate to probabilities per unit surface area according to

$$\prod_{i=r}^{s} G(x_i \leftrightarrow x_{i-1}) \prod_{i=s+1}^{t} G(x_i \leftrightarrow x_{i+1}) \prod_{i=r}^{s} P_{\overrightarrow{\sigma}^\perp}(x_{i-1} \rightarrow x_i) \prod_{i=s+1}^{t} P_{\overleftarrow{\sigma}^\perp}(x_i \leftarrow x_{i+1}) = \prod_{i=r}^{t} P_A(x_i) \tag{B.7}$$

Figure B.3 shows a bidirectionally sampled light transport path. The vertices x_0,x_1 and x_5 are sampled using surface area sampling, x_2 is sampled using forward sampling and vertices x_3 and x_4 are sampled using backward sampling. The figure also shows the sampling

probabilities, including conversion factors to convert from projected solid angle to unit area.

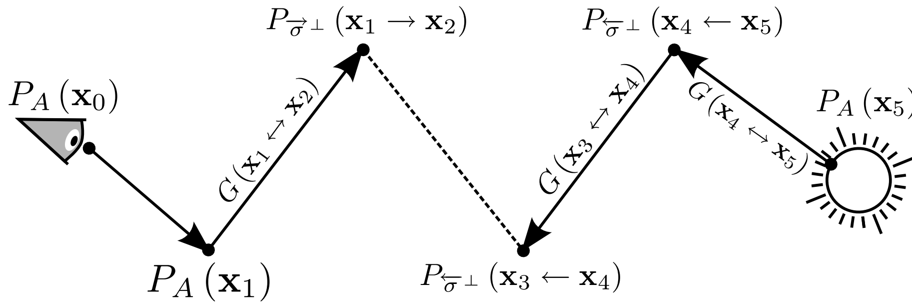


Figure B.3: Bidirectional sampled light transport path, including probability conversion factors.

Appendix C

Camera Model

Usually, the first two path vertices x_0 and x_1 on a light transport path are sampled using special sampling methods corresponding to the sensor sensitivity function $W_j(x_1 \to x_0)$ used. The sensor sensitivity function describes how the image is constructed from incoming radiance and models the camera. For a given camera model and corresponding sampling methods for x_0 and x_1, the sensor sensitivity function for some pixel j is usually not specified explicitly but the *modified sensor sensitivity function* $\hat{W}_j(x_1 \to x_0)$ is specified instead:

$$\hat{W}_j(x_1 \to x_0) = W_j(x_1 \to x_0)\, G(x_0 \leftrightarrow x_1)\, \frac{1}{P_I(x_0)\, P_A(x_1)} \tag{C.1}$$

Where $P_I(x_0)$ and $P_A(x_1)$ are the probability of sampling respectively x_0 and x_1 per unit surface area using the sampling methods corresponding to the camera model. In this thesis, we used a simple finite aperture lens and we will describe this model in more detail in this appendix.

Figure C.1 visualizes the finite aperture camera model. The first vertex x_0 is sampled directly on the camera lens I with probability $P_I(x_0)$ per unit surface area. Vertex x_1 is found by sampling a point $\mathbf{v}$ on the view plane V and tracing a ray from x_0 through v into the scene. The first intersection between the ray and the scene geometry gives x_1. $P_V(x_1)$ is the probability of sampling x_1 per unit view plane area or, in other words, the probability of sampling v on the view plane. When applying the Monte Carlo method to the measurement equation from section 2.1, it is necessary to convert this probability from unit view plane area to unit surface area. Figure C.1 shows the relation between these measures, and the corresponding probabilities relate according to

$$\frac{P_V(x_1)}{(\vec{n}_0 \cdot (x_1 - x_0))^3} = \frac{P_{\vec{\sigma}}(x_0 \to x_1)}{l^2\, |x_1 - x_0|^3} = \frac{P_A(x_1)}{l^2 \vec{n}_1 \cdot (x_0 - x_1)} \tag{C.2}$$

In this equation, l is the distance from lens to view plane, also called the focal length, and $\vec{n}_0$ is the view direction. Figure C.1 also shows how the view plane V relates to screen space S, depending on l, the horizontal and vertical field of view angles θ_h and θ_v, and the image width and height w and h. The point v on the view plane is usually generated by sampling a point s in screen space S and constructing the corresponding point v on the view plane. If

Figure C.1: Camera model for a simple camera with finite aperture lens.

$P_S(x_1)$ is the probability of sampling x_1 per unit screen space area (probability of sampling s in screen space), another conversion factor is required to convert from unit screen space area to unit view plane area. These probabilities relate according to

$$\frac{4l^2 \tan(\theta_h) \tan(\theta_v)}{wh} P_V(x_1) = P_S(x_1) \tag{C.3}$$

Or, using $\rho = \frac{4l^2 \tan(\theta_h) \tan(\theta_v)}{wh}$, shortly

$$\rho P_V(x_1) = P_S(x_1) \tag{C.4}$$

Putting everything together, the probability of sampling x_1 per unit screen space area relates to the corresponding probability per unit surface area according to

$$P_A(x_1) = \frac{l^2 \vec{n}_1 \cdot (x_0 - x_1)}{\rho\left(\vec{n}_0 \cdot (x_1 - x_0)\right)^3} P_S(x_1) \tag{C.5}$$

Hence, for the finite aperture camera model, equation C.1 becomes

$$\hat{W}_j(x_1 \rightarrow x_0) = W_j(x_1 \rightarrow x_0)\, G(x_0 \leftrightarrow x_1)\, \frac{\rho\left(\vec{n}_0 \cdot (x_1 - x_0)\right)^3}{l^2 \vec{n}_1 \cdot (x_0 - x_1)} \frac{1}{P_I(x_0)\, P_S(x_1)} \tag{C.6}$$

For our simple finite aperture camera model, $P_I(x_0)$ and $P_S(x_0)$ are constant. That is; x_0 is sampled uniformly on the lens and s is sampled uniformly in screen space. Furthermore, $\hat{W}_j(x_1 \rightarrow x_0) = c$ whenever $x_1 \rightarrow x_0$ passes through pixel j and 0 otherwise, with c a constant that determines the total sensor sensitivity.

Appendix D

PT Algorithm

This appendix gives a more detailed description of the path tracing algorithm discussed in section 2.4. Remember that a path tracer can sample a light transport path as either an explicit path by making an explicit connection to the light source, or as an implicit path by *accidentally* finding a light source without making an explicit connection. In this appendix, we show how to compute the contribution and MIS weights for explicit and implicit paths. Using these, the general layout of the path tracing algorithm is given in pseudo code.

D.1 Path contribution

The contribution of a sampled path $\mathbf{X} = x_0 \cdots x_k$ to the Monte Carlo estimate equals $\frac{f(\mathbf{X})}{p(\mathbf{X})}$ (see section 2.3). When the path is generated using path tracing, part of the path is sampled using forward sampling. Therefore, the probability of sampling these vertices per projected solid angle needs to be converted to unit surface area using equation B.4. The measurement contribution function $f(\mathbf{X})$ shares several factors with the probability $p(\mathbf{X})$ for sampling explicit and implicit paths per unit path space, which will cancel out in the Monte Carlo contribution $\frac{f(\mathbf{X})}{p(\mathbf{X})}$.

Let $p_I(x_0 \cdots x_k)$ be the probability of sampling $\mathbf{X}$ as an implicit path. After canceling out the common factors, the Monte Carlo contribution of an implicit path equals

$$\frac{f_j(x_0 \cdots x_k)}{p_I(x_0 \cdots x_k)} = \hat{W}_j(x_1 \to x_0) \prod_{i=1}^{k-1} \frac{f_r(x_{i+1} \to x_i \to x_{i-1})}{P_{\vec{\sigma}^\perp}(x_i \to x_{i+1})} \cdot L_e(x_k \to x_{k-1}) \tag{D.1}$$

All geometric factors cancel out because they appear in both equations B.4 and 2.4. Note that we used the modified sensor sensitivity function $\hat{W}_j$ from appendix C.

Furthermore, let $p_E(x_0 \cdots x_k)$ be the probability of sampling $\mathbf{X}$ as an explicit path. After canceling out the common factors, the contribution for an explicit path becomes

$$\begin{aligned} \frac{f_j(x_0 \cdots x_k)}{p_E(x_0 \cdots x_k)} =& \hat{W}_j(x_1 \to x_0) \prod_{i=1}^{k-2} \frac{f_r(x_{i+1} \to x_i \to x_{i-1})}{P_{\vec{\sigma}^\perp}(x_i \to x_{i+1})} \\ & \cdot f_r(x_k \to x_{k-1} \to x_{k-2}) \, G(x_k \leftrightarrow x_{k-1}) \frac{L_e(x_k \to x_{k-1})}{P_A(x_k)} \end{aligned} \tag{D.2}$$

Because the last vertex is not sampled using forward sampling but using surface area sampling, the geometric factor in f_j between the last two vertices x_3 and x_4 does not cancel out and therefore appears in the final contribution.

Figures D.1 and D.2 show an implicit and explicit sampled light transport path, visualizing the factors in their contributions. The figures give an indication of the order in

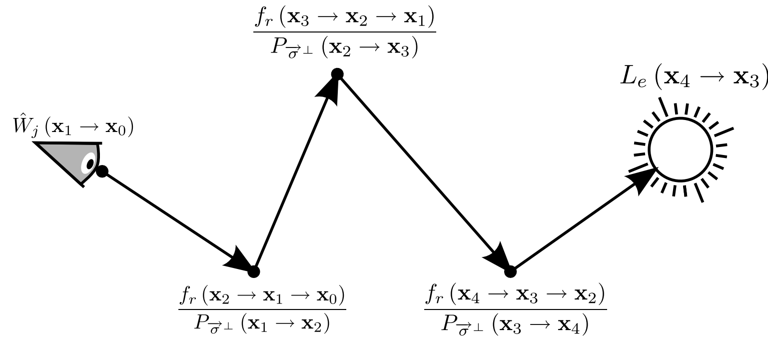


Figure D.1: Monte Carlo contribution of implicit path.

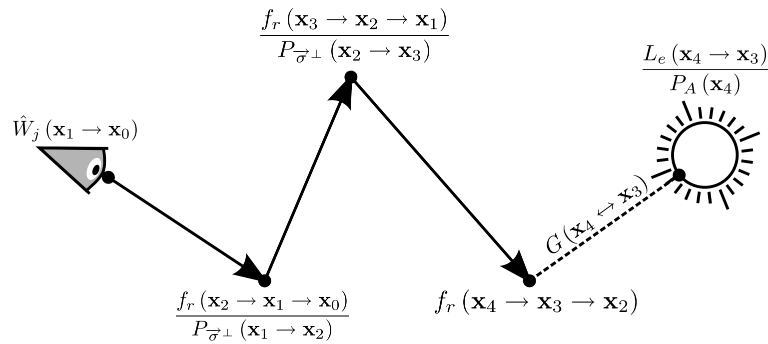


Figure D.2: Monte Carlo contribution of explicit path.

which these contributions are computed during sample construction, starting with evaluating the sensor sensitivity function and iteratively extending the path, evaluating the BSDF and sample probabilities and finally evaluating the light source emittance.

D.2 MIS weights

As explained in section 2.6, when light transport paths may be sampled through multiple sampling strategies, the samples need to be combined using Multiple Importance Sampling. For optimally combining implicit and explicit samples, we need to compute their importance weights according to the power heuristic. All but the last path vertex are sampled according to the same probability distributions for both explicit and implicit paths. Therefore, when computing the weights for MIS, the common factors cancel out and only the probabilities for sampling the last path vertex using forward sampling and surface area sampling

are used. For implicit paths, the power heuristic equals

$$\begin{aligned} w_I(x_0\cdots x_k) &= \frac{p_I(x_0\cdots x_k)^\beta}{p_E(x_0\cdots x_k)^\beta + p_I(x_0\cdots x_k)^\beta} \\ &= \frac{\left(P_{\vec{\sigma}^\perp}(x_{k-1}\to x_k)\,G(x_k\leftrightarrow x_{k-1})\right)^\beta}{P_A(x_k)^\beta + \left(P_{\vec{\sigma}^\perp}(x_{k-1}\to x_k)\,G(x_k\leftrightarrow x_{k-1})\right)^\beta} \end{aligned} \tag{D.3}$$

For explicit paths, the power heuristic equals

$$\begin{aligned} w_E(x_0\cdots x_k) &= \frac{p_E(x_0\cdots x_k)^\beta}{p_E(x_0\cdots x_k)^\beta + p_I(x_0\cdots x_k)^\beta} \\ &= \frac{P_A(x_k)^\beta}{P_A(x_k)^\beta + \left(P_{\vec{\sigma}^\perp}(x_{k-1}\to x_k)\,G(x_k\leftrightarrow x_{k-1})\right)^\beta} \end{aligned} \tag{D.4}$$

D.3 Algorithm

Using the contributions and MIS weights from previous sections, we can construct a path tracing sample and compute its total contribution. Algorithm 5 gives the pseudo code for generating a PT sample. The algorithm returns the combined contribution of all explicit and implicit paths in the generated sample. The algorithm starts by generating vertex y_0 on the lens. Then, y_1 is sampled by tracing a ray through the view plane according to the sampling probability $P_S(y_1)$ per unit screen space area (see appendix C). If the scene is not closed, the ray may miss all geometry, failing to generate vertex y_1. In this case, the path immediately terminates. Otherwise, the sample is extended iteratively.

During extension, it is first checked to see if the path so far is a valid implicit path. If so, the corresponding importance weight and contribution are computed and added to the total sample contribution. In the computation of the importance weight, $P_A(y_i)$ is the probability with which y_i would have been sampled as the light vertex if this path was constructed as an explicit path instead. Note that paths of length 1 can only be sampled using implicit paths and thus their importance weight equals 1.

Then, an explicit connection is made by generating a vertex z on the light source. If the current vertex y_i and the light vertex z are not visible from one another, the connection failed. Otherwise, the importance and contribution of the explicit path are computed and added to the total sample contribution. In the computation of the importance weight, $P_{\vec{\sigma}^\perp}(y_{i-1}\to y_i\to z)$ is the probability with which z would have been sampled as the next path vertex if this path was constructed as an implicit path instead.

Finally, the next path vertex is sampled through forward sampling. This again requires ray tracing. The path may be terminated for two reasons; either it is terminated through Russian roulette or the generated output ray misses all geometry, effectively terminating the path. If the path is not terminated, the path is extended by another vertex during the next iteration. Note that the Russian roulette probability is assumed to be incorporated in the sampling probability $P_{\vec{\sigma}^\perp}(y_i\to y_{i+1})$. Hence, with non-zero probability, no next vertex may be generated. When the path has terminated, the total sample contribution is returned.

Algorithm 5 : PT()

$color \leftarrow$ black

sample $\mathbf{y}_0 \sim P_I(\mathbf{y}_0)$
sample $\mathbf{y}_1 \sim P_S(\mathbf{y}_1)$

$f_X \leftarrow \hat{W}(\mathbf{y}_1 \rightarrow \mathbf{y}_0)$

$i \leftarrow 0$
while path is not terminated **do**
  $i \leftarrow i+1$

  $\{$*Check implicit path*$\}$
  if $L_e(\mathbf{y}_i \rightarrow \mathbf{y}_{i-1}) > 0$ **then**
    if $i > 1$ **then**
      $P_A(\mathbf{y}_{i-1} \rightarrow \mathbf{y}_i) \leftarrow P_{\vec{\sigma}^\perp}(\mathbf{y}_{i-1} \rightarrow \mathbf{y}_i) G(\mathbf{y}_{i-1} \leftrightarrow \mathbf{y}_i)$
      $w_I \leftarrow \frac{P_A(\mathbf{y}_{i-1} \rightarrow \mathbf{y}_i)^\beta}{P_A(\mathbf{y}_i)^\beta + P_A(\mathbf{y}_{i-1} \rightarrow \mathbf{y}_i)^\beta}$
    else
      $w_I \leftarrow 1$
    end if
    $color \leftarrow color + w_I f_X L_e(\mathbf{y}_i \rightarrow \mathbf{y}_{i-1})$
  end if

  $\{$*Make explicit connection*$\}$
  sample $\mathbf{z} \sim P_A(\mathbf{z})$
  if $V(\mathbf{y}_i \leftrightarrow \mathbf{z}) = 0$ **then**
    $w_E \leftarrow \frac{P_A(\mathbf{z})^\beta}{P_A(\mathbf{z})^\beta + \left(P_{\vec{\sigma}^\perp}(\mathbf{y}_{i-1} \rightarrow \mathbf{y}_i \rightarrow \mathbf{z}) G(\mathbf{y}_i \leftrightarrow \mathbf{z})\right)^\beta}$
    $color \leftarrow color + w_E f_X f_r(\mathbf{z} \rightarrow \mathbf{y}_i \rightarrow \mathbf{y}_{i-1}) G(\mathbf{z} \leftrightarrow \mathbf{y}_i) \frac{L_e(\mathbf{z} \rightarrow \mathbf{y}_i)}{P_A(\mathbf{z})}$
  end if

  $\{$*Extend path*$\}$
  sample $\mathbf{y}_{i+1} \sim P_{\vec{\sigma}^\perp}(\mathbf{y}_i \rightarrow \mathbf{y}_{i+1})$
  $f_X \leftarrow f_X \cdot \frac{f_r(\mathbf{y}_{i+1} \rightarrow \mathbf{y}_i \rightarrow \mathbf{y}_{i-1})}{P_{\vec{\sigma}^\perp}(\mathbf{y}_i \rightarrow \mathbf{y}_{i+1})}$
end while

return $color$

Appendix E

BDPT Algorithm

This appendix gives a more detailed description of the BiDirectional Path Tracing algorithm discussed in section 2.7. Remember that BDPT can sample a light transport path through one of several different bidirectional sampling strategies, including the explicit and implicit sampling strategies from the PT algorithm. In the last appendix, we already showed how to compute the contribution for explicit and implicit paths. In this appendix, we show how to compute the contribution for paths sampled using one of the remaining bidirectional sampling strategies. Using these contributions and the recursive MIS weights from section 8.2, the general layout of the BDPT algorithm is given in pseudo code.

E.1 Path contribution

The contribution of a sampled path $\mathbf{X} = x_0 \cdots x_k$ to the Monte Carlo estimate equals $\frac{f(\mathbf{X})}{p(\mathbf{X})}$ (see section 2.3). Just like paths sampled through path tracing, the measurement contribution function $f(\mathbf{X})$ shares several factors with the probability $p(\mathbf{X})$ for bidirectionally sampling paths per unit path space, which again cancel out in the Monte Carlo contribution $\frac{f(\mathbf{X})}{p(\mathbf{X})}$.

Remember from section 2.7 that a path $\mathbf{X} = x_0 \cdots x_k$ of length k can be sampled using either one of $k+1$ sampling strategies. Furthermore, $\hat{p}_s(\mathbf{X})$ is the probability of bidirectionally sampling $\mathbf{X}$ by connecting the eye path $x_0 \cdots x_s$ of length s[1] with the light path $x_{s+1} \cdots x_k$ of length $t = k - s$. Paths sampled using light paths of length $t = 0$ and $t = 1$ correspond to implicit and explicit paths as sampled by a path tracer. We already showed how common factors cancel out in the contribution of implicit and explicit paths in appendix D. What is left are the contributions for paths sampled using any of the remaining bidirectional sampling strategies.

For paths sampled with a zero length eye path ($s = 0$), all vertices except x_0 are sampled as part of the light path. After combining equations B.7, C.6 and 2.4 and canceling out the

[1] Note that because x_0 is also part of the eye path, an eye path of length s contains $s+1$ vertices.

common factors, the Monte Carlo contribution of these paths equals

$$\begin{aligned}\frac{f_j(x_0\cdots x_k)}{p_0(x_0\cdots x_k)} =& \hat{W}_j(\mathbf{x}_1\to\mathbf{x}_0)\frac{l^2\vec{n}_1\cdot(\mathbf{x}_0-\mathbf{x}_1)}{\rho(\vec{n}_0\cdot(\mathbf{x}_1-\mathbf{x}_0))^3}P_S(\mathbf{x}_1)\\ & f_r(\mathbf{x}_2\to\mathbf{x}_1\to\mathbf{x}_0)\\ & \prod_{j=2}^{k-1}\frac{f_r(\mathbf{x}_{j+1}\to\mathbf{x}_j\to\mathbf{x}_{j-1})}{P_{\overleftarrow{\sigma}^\perp}(\mathbf{x}_{j-1}\leftarrow\mathbf{x}_j)}\\ & \frac{L_e(\mathbf{x}_k\to\mathbf{x}_{k-1})}{P_A(\mathbf{x}_k)P_{\overleftarrow{\sigma}^\perp}(\mathbf{x}_{k-1}\leftarrow\mathbf{x}_k)}\end{aligned} \tag{E.1}$$

Note that we used the modified sensor sensitivity function $\hat{W}_j$ from appendix C. For all other bidirectional strategies, using $\hat{W}_j$ removes the conversion factor from unit screen space to unit surface area resulting from sampling x_1 per unit screen space. However, when $s=0$, x_1 is sampled as part of the light path and is therefore not sampled per unit screen space. Hence, no conversion is required. Consequently, using $\hat{W}_j$ introduces an extra correction factor $\frac{l^2\vec{n}_1\cdot(x_0-x_1)}{\rho(\vec{n}_0\cdot(x_1-x_0))^3}$.

Finally for the remaining sampling strategies with $0<s<k-1$, after canceling out the common factors, the Monte Carlo contribution of a path equals

$$\begin{aligned}\frac{f_j(x_0\cdots x_k)}{p_s(x_0\cdots x_k)} =& \hat{W}_j(\mathbf{x}_1\to\mathbf{x}_0)\\ & \prod_{j=1}^{s-1}\frac{f_r(\mathbf{x}_{j+1}\to\mathbf{x}_j\to\mathbf{x}_{j-1})}{P_{\overrightarrow{\sigma}^\perp}(\mathbf{x}_j\to\mathbf{x}_{j+1})}\\ & f_r(\mathbf{x}_{s+1}\to\mathbf{x}_s\to\mathbf{x}_{s-1})G(\mathbf{x}_s\leftrightarrow\mathbf{x}_{s+1})f_r(\mathbf{x}_{s+2}\to\mathbf{x}_{s+1}\to\mathbf{x}_s)\\ & \prod_{j=s+2}^{k-1}\frac{f_r(\mathbf{x}_{j+1}\to\mathbf{x}_j\to\mathbf{x}_{j-1})}{P_{\overleftarrow{\sigma}^\perp}(\mathbf{x}_{j-1}\leftarrow\mathbf{x}_j)}\\ & \frac{L_e(\mathbf{x}_k\to\mathbf{x}_{k-1})}{P_A(\mathbf{x}_k)P_{\overleftarrow{\sigma}^\perp}(\mathbf{x}_{k-1}\leftarrow\mathbf{x}_k)}\end{aligned} \tag{E.2}$$

All geometric factors on the eye and light path cancel out because they appear in both equation B.7 and 2.4. The only remaining geometric factor corresponds to the connection edge between x_s and x_{s+1}.

Figures E.1 and E.2 show light transport paths sampled with bidirectional sampling strategies corresponding to $s=0$ and $s=2$. The figures visualize the factors in these paths' Monte Carlo contributions. The figures also give an indication of the order in which these contributions are computed during sample construction. For the eye path, construction starts with evaluating the sensor sensitivity function and iteratively extending the eye path, evaluating the BSDF and sample probabilities. For the light path, construction starts with evaluating the light source emittance and iteratively extending the light path, evaluating the BSDF and sample probabilities. Finally, the eye and light path are explicitly connected and the connection is evaluated.

The computation of MIS weights for bidirectionally sampled paths using the power heuristic has already been discussed in detail in section 8.2.

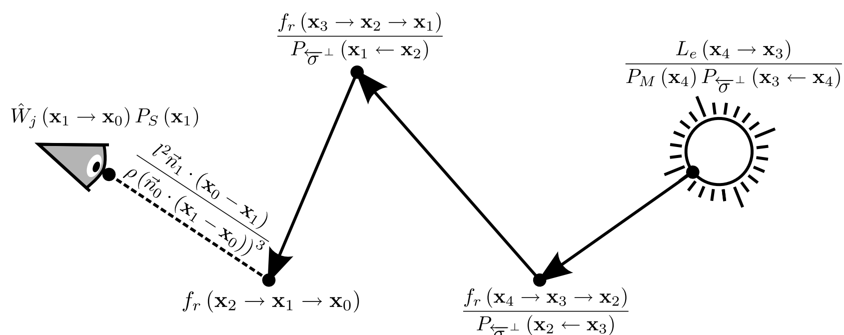


Figure E.1: Monte Carlo contribution of path sampled by connecting a light path directly to the eye.

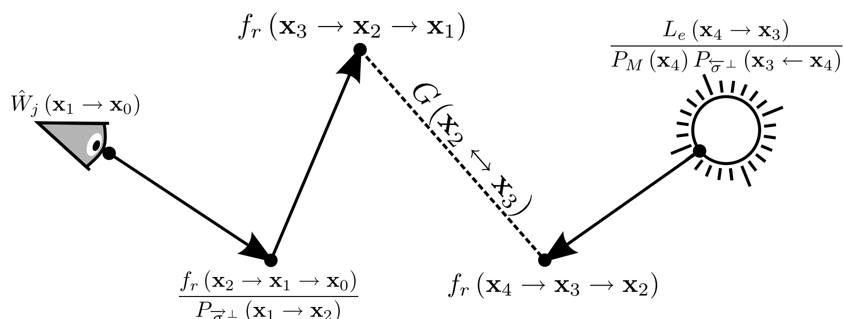


Figure E.2: Monte Carlo contribution of bidirectional sampled path.

E.2 Algorithm

Using the contributions from the previous section and the recursive MIS weights from section 8.2, we can now construct a BDPT sample. The pseudo code for generating such a sample is given by algorithms 6, 7 and 8.

Algorithm 6 generates an eye path $y_0 \cdots y_{N_E}$ and is somewhat similar to the path tracing algorithm (see algorithm 5), but without the evaluation of explicit and implicit paths. Instead, the recursive MIS quantities for the path vertices are computed. The algorithm samples the first two vertices according to the camera model and iteratively extends the path with extra vertices until the path terminates. As with the path tracing algorithm in appendix D, the eye path is terminated either because the extension ray misses all scene geometry or due to Russian roulette. Again, the Russian roulette probability is assumed to be incorporated in the forward sampling probability. Note the special case which occurs while extending vertex x_1. This special case allows us to use the modified sensor sensitivity function $\hat{W}_j$ and relieves us from evaluating $P_{\overleftarrow{\sigma}^\perp}(y_0 \leftarrow y_1)^\beta$.

Algorithm 7 looks a lot like algorithm 6 and generates a light path $z_1 \cdots z_{N_L}$. The main difference is that the first vertex z_1 is sampled on the light source instead of on the lens and that the second vertex z_2 is sampled using backward sampling from the light source. No

Algorithm 6 : SampleEyePath

sample $\mathbf{y}_0 \sim P_I(\mathbf{y}_0)$
sample $\mathbf{y}_1 \sim P_S(\mathbf{y}_1)$

$P_0^Y \leftarrow P_I(\mathbf{y}_0)$
$D_0^Y \leftarrow 0$
$F_0^Y \leftarrow \hat{W}(\mathbf{y}_1 \to \mathbf{y}_0)$

$i \leftarrow 0$
while path is not terminated **do**
  $i \leftarrow i+1$
  sample $\mathbf{y}_{i+1} \sim P_{\overrightarrow{\sigma}^\perp}(\mathbf{y}_i \to \mathbf{y}_{i+1})$
  if i = 1 **then**
    $P_1^Y \leftarrow P_0^Y \cdot P_S(\mathbf{y}_1)$
    $D_1^Y \leftarrow P_0^Y$
  else
    $P_i^Y \leftarrow P_{i-1}^Y \cdot P_{\overrightarrow{\sigma}^\perp}(\mathbf{y}_{i-1} \to \mathbf{y}_i)^\beta G(\mathbf{y}_{i-1} \leftrightarrow \mathbf{y}_i)^\beta$
    $D_i^Y \leftarrow P_{i-1}^Y + D_{i-1}^Y \cdot P_{\overleftarrow{\sigma}^\perp}(\mathbf{y}_{i-1} \leftarrow \mathbf{y}_i)^\beta G(\mathbf{y}_i \leftrightarrow \mathbf{y}_{i-1})^\beta$
  end if
  $F_i^Y \leftarrow F_{i-1}^Y \cdot \frac{f_r(\mathbf{y}_{i+1} \to \mathbf{y}_i \to \mathbf{y}_{i-1})}{P_{\overrightarrow{\sigma}^\perp}(\mathbf{y}_i \to \mathbf{y}_{i+1})}$
end while
$N_E \leftarrow i$

special case is required when extending z_2.

After both the eye and light path are sampled, algorithm 8 evaluates all connections and computes the total sample contribution. The algorithm iterates over all $N_E + 1$ eye vertices, checking for implicit paths. If an implicit path is encountered, the path contribution and importance are computed and added to the image along the first path edge. For each eye vertex, connections are made to all N_L light vertices on the light path. For each pair, a connection is established. If the vertices are not visible from one another, the connection failed. Otherwise, the importance and contribution of the path are computed and added to the image along the first path edge. Note that special care must be taken when the vertices y_0 and z_1 are involved in the connection. Also, instead of accumulating the total sample contribution as done in PT algorithm 5, all contributions are directly accumulated on the image. This is necessary because not all paths in the sample contribute to the same image pixel.

Algorithm 7 : SampleLightPath

sample $\mathbf{z}_1 \sim P_A(\mathbf{z}_1)$
sample $\mathbf{z}_2 \sim P_{\overrightarrow{\sigma}^\perp}(\mathbf{z}_1 \to \mathbf{z}_2)$

$P_1^Z \leftarrow P_A(\mathbf{z}_1)$
$D_1^Z \leftarrow 1$
$F_1^Z \leftarrow L_e(\mathbf{z}_2 \to \mathbf{z}_1)$

$i \leftarrow 1$
while path is not terminated **do**
  $i \leftarrow i+1$
  sample $\mathbf{z}_{i+1} \sim P_{\overrightarrow{\sigma}^\perp}(\mathbf{z}_i \to \mathbf{z}_{i+1})$
  $F_i^Z \leftarrow F_{i-1}^Z \cdot \frac{f_r(\mathbf{z}_{i-1} \to \mathbf{z}_i \to \mathbf{z}_{i+1})}{P_{\overrightarrow{\sigma}^\perp}(\mathbf{z}_i \to \mathbf{z}_{i+1})}$
  $P_i^Z \leftarrow P_{i-1}^Z \cdot P_{\overrightarrow{\sigma}^\perp}(\mathbf{z}_{i-1} \to \mathbf{z}_i)^\beta G(\mathbf{z}_{i-1} \leftrightarrow \mathbf{z}_i)^\beta$
  $D_i^Z \leftarrow P_{i-1}^Z + D_{i-1}^Z \cdot P_{\overleftarrow{\sigma}^\perp}(\mathbf{z}_{i-1} \leftarrow \mathbf{z}_i)^\beta G(\mathbf{z}_i \leftrightarrow \mathbf{z}_{i-1})^\beta$
end while
$N_L \leftarrow i$

Algorithm 8 : Connect

for $i = 0$ to N_E **do**
  {Check implicit path}
  if $i > 0$ and $L_e(\mathbf{y}_i \rightarrow \mathbf{y}_{i-1})$ **then**
    $D \leftarrow P_i^Y + P_A(\mathbf{y}_i) D_{i-1}^Y$
    $F \leftarrow F_{i-1}^Y L_e(\mathbf{y}_i \rightarrow \mathbf{y}_{i-1})$
    $w \leftarrow \frac{P_i^Y}{D}$
    contribute wF along edge $\mathbf{y}_0\mathbf{y}_1$
  end if

  {Connect to light path}
  for $j = 1$ to N_L **do**
    if $\mathbf{y}_i$ is visible from $\mathbf{z}_j$ **then**

      $D \leftarrow P_i^Y P_j^Z$

      if $i > 0$ **then**
        $P_A(\mathbf{z}_j \rightarrow \mathbf{y}_i) \leftarrow P_{\overrightarrow{\sigma}^{\perp}}(\mathbf{z}_j \rightarrow \mathbf{y}_i) G(\mathbf{y}_i \leftrightarrow \mathbf{z}_j)$
        $P_A(\mathbf{y}_{i-1} \leftarrow \mathbf{y}_i) \leftarrow P_{\overleftarrow{\sigma}^{\perp}}(\mathbf{y}_{i-1} \leftarrow \mathbf{y}_i) G(\mathbf{y}_i \leftrightarrow \mathbf{y}_{i-1})$
        $D \leftarrow D + P_j^Z P_A(\mathbf{z}_j \rightarrow \mathbf{y}_i)^{\beta} \left(P_{i-1}^Y + P_A(\mathbf{y}_{i-1} \leftarrow \mathbf{y}_i)^{\beta} D_{i-1}^Y \right)$
        $F \leftarrow F_{i-1}^E f_r(\mathbf{z}_j \rightarrow \mathbf{y}_i \rightarrow \mathbf{y}_{i-1}) G(\mathbf{y}_i \leftrightarrow \mathbf{z}_j)$
      else
        $F \leftarrow \hat{W}(\mathbf{z}_j \rightarrow \mathbf{y}_0) \frac{l^2 \vec{n}_{z_j} \cdot (\mathbf{y}_0 - \mathbf{z}_j)}{\rho(\vec{n}_{y_0} \cdot (\mathbf{z}_j - \mathbf{y}_0))^3} P_S(\mathbf{z}_j)$
      end if

      $P_A(\mathbf{y}_i \rightarrow \mathbf{z}_j) \leftarrow P_{\overrightarrow{\sigma}^{\perp}}(\mathbf{y}_i \rightarrow \mathbf{z}_j) G(\mathbf{y}_i \leftrightarrow \mathbf{z}_j)$
      if $j > 1$ **then**
        $P_A(\mathbf{z}_{j-1} \leftarrow \mathbf{z}_j) \leftarrow P_{\overleftarrow{\sigma}^{\perp}}(\mathbf{z}_{j-1} \leftarrow \mathbf{z}_j) G(\mathbf{z}_j \leftrightarrow \mathbf{z}_{j-1})$
        $D \leftarrow D + P_i^Y P_A(\mathbf{y}_i \rightarrow \mathbf{z}_j)^{\beta} \left(P_{j-1}^Z + P_A(\mathbf{z}_{j-1} \leftarrow \mathbf{z}_j)^{\beta} D_{j-1}^Z \right)$
        $F \leftarrow F \cdot F_{j-1}^L f_r(\mathbf{z}_{j+1} \rightarrow \mathbf{z}_j \rightarrow \mathbf{y}_i)$
      else
        $D \leftarrow D + P_i^Y P_A(\mathbf{y}_i \rightarrow \mathbf{z}_1)^{\beta}$
        $F \leftarrow F \cdot L_e(\mathbf{z}_1 \rightarrow \mathbf{y}_i)$
      end if

      $w \leftarrow \frac{P_i^Y P_j^Z}{D}$
      if i = 0 **then**
        contribute wF along edge $\mathbf{y}_0\mathbf{z}_j$
      else
        contribute wF along edge $\mathbf{y}_0\mathbf{y}_1$
      end if
    end if
  end for
end for

Appendix F

MLT Mutations

This appendix gives a detailed description of the lens and caustic mutations in the MLT algorithm as discussed in section 2.9. Remember that the MLT algorithm generates a sequence of light transport paths through mutations. A mutation transforms a path $\mathbf{X}$ into a new path $\mathbf{Y}$ by making small changes to the original path. Two important mutation strategies are the lens and caustic mutation. Both mutation strategies mutate the first $m+1$ path vertices on a path $\mathbf{X} = x_0 \cdots x_k$ into $y_0 \cdots y_m$ to form the mutated path $\mathbf{Y} = y_0 \cdots y_m x_{m+1} \cdots x_k$. The lens mutation starts from the eye and mutates the vertices through forward sampling. The caustic mutation starts somewhere on the path and mutating backwards towards the eye (for the caustic mutation, $m < k$). Both mutations preserve the path length (See section 2.9 for more details on the mutation strategies). In this appendix, we will show how to compute the acceptance probability for the lens and caustic mutation strategies and show the algorithms for generating such mutations in pseudo code.

F.1 Acceptance probability

Remember from section 2.8 that for some path $\mathbf{Y}$ that is generated from path $\mathbf{X}$ through mutations, the acceptance probability in the MLT algorithm is defined as

$$a = \min\left(1, \frac{f(\mathbf{Y})\, p(\mathbf{X}|\mathbf{Y})}{f(\mathbf{X})\, p(\mathbf{Y}|\mathbf{X})}\right) \tag{F.1}$$

Because usually only a part of the path is mutated, many of the factors in this equation cancel out.

Lets start with the measurement contribution factor $\frac{f(\mathbf{Y})}{f(\mathbf{X})}$ in the acceptance probability. When $0 < m < k$, that is, only the first $m+1$ vertices on the path are mutated and the remaining vertices $x_{m+1} \cdots x_k = y_{m+1} \cdots y_k$ stay the same, some of the factors in both measurement contribution functions cancel out, resulting in

$$\frac{f(y_0 \cdots y_k)}{f(x_0 \cdots x_k)} = \frac{W_i(y_1 \to y_0)}{W_i(x_1 \to x_0)} \prod_{j=0}^{m} \frac{G(y_j \leftrightarrow y_{j+1})}{G(x_j \leftrightarrow x_{j+1})} \prod_{j=1}^{m+1} \frac{f_r(y_{j+1} \to y_j \to y_{j-1})}{f_r(x_{j+1} \to x_j \to x_{j-1})} \tag{F.2}$$

In case the entire path is mutated by a lens mutation ($m = k$), no common factors appear and the the measurement contribution factor becomes

$$\frac{f(y_0 \cdots y_k)}{f(x_0 \cdots x_k)} = \frac{W_i(y_1 \to y_0)}{W_i(x_1 \to x_0)} \prod_{j=0}^{k-1} \frac{G(y_j \leftrightarrow y_{j+1})}{G(x_j \leftrightarrow x_{j+1})} \prod_{j=1}^{k} \frac{f_r(y_{j+1} \to y_j \to y_{j-1})}{f_r(x_{j+1} \to x_j \to x_{j-1})} \tag{F.3}$$

Note that in these equations, we used, for convenience, the special notation $L_e(x_k \to x_{k-1}) = f_r(x_{k+1} \to x_k \to x_{k-1})$. In this case, x_{k+1} can be thought of as an implicit vertex on all light transport paths being the source of all light in the scene [50].

What is left is the probability factor $\frac{p(\mathbf{X}|\mathbf{Y})}{p(\mathbf{Y}|\mathbf{X})}$ in the acceptance probability. Because the lens and caustic mutations are sampled differently, this factor differs for both mutations strategies. Let's start with the lens mutation. The lens mutation mutates the first $m+1$ path vertices of a path through forward sampling. Because part of the mutation is sampled forward, some mutation probabilities must be converted from unit projected solid angle to unit surface area (see appendix D). Now let $P_L^m(\mathbf{Y}|\mathbf{X})$ be the probability of generating $\mathbf{Y}$ from $\mathbf{X}$ per unit path space, using the lens mutation. Then, using the conversion factors from equations B.4 and C.2, we get

$$\begin{aligned} \frac{P_L^m(\mathbf{X}|\mathbf{Y})}{p_L^m(\mathbf{Y}|\mathbf{X})} =& \frac{l^2 \vec{n}_1 \cdot (x_1 - x_0)}{l^2 \vec{n}_1 \cdot (y_1 - y_0)} \frac{\rho(\vec{n}_0 \cdot (y_1 - y_0))^3}{\rho(\vec{n}_0 \cdot (x_1 - x_0))^3} \prod_{i=1}^{m-1} \frac{G(x_i \leftrightarrow x_{i+1})}{G(y_i \leftrightarrow y_{i+1})} \\ &\cdot \frac{P_I(x_0|\mathbf{Y}) P_S(x_1|\mathbf{Y})}{P_I(y_0|\mathbf{X}) P_S(y_1|\mathbf{X})} \prod_{i=1}^{m-1} \frac{P_{\vec{\sigma}^\perp}(x_i \to x_{i+1}|\mathbf{Y})}{P_{\vec{\sigma}^\perp}(y_i \to y_{i+1}|\mathbf{X})} \end{aligned} \tag{F.4}$$

Similarly, the caustic mutation mutates the first $m+1$ path vertices of a path, this time through backward sampling. Because part of the mutation is sampled backward, some mutation probabilities must again be converted from unit projected solid angle to unit surface area (see appendix D). Let $P_C^m(\mathbf{Y}|\mathbf{X})$ be the probability of generating $\mathbf{Y}$ from $\mathbf{X}$ per unit path space, using the caustic mutation. Then, using the conversion factors from equation B.5, we get

$$\frac{P_C^m(\mathbf{X}|\mathbf{Y})}{p_C^m(\mathbf{Y}|\mathbf{X})} = \prod_{i=1}^{m} \frac{G(x_i \leftrightarrow x_{i+1})}{G(y_i \leftrightarrow y_{i+1})} \cdot \frac{P_I(x_0|\mathbf{Y})}{P_I(y_0|\mathbf{X})} \prod_{i=1}^{m-1} \frac{P_{\overleftarrow{\sigma}^\perp}(x_i \leftarrow x_{i+1}|\mathbf{Y})}{P_{\overleftarrow{\sigma}^\perp}(y_i \leftarrow y_{i+1}|\mathbf{X})} \cdot \frac{P_{\overleftarrow{\sigma}^\perp}(x_m \leftarrow y_{m+1}|\mathbf{Y})}{P_{\overleftarrow{\sigma}^\perp}(y_m \leftarrow x_{m+1}|\mathbf{X})} \tag{F.5}$$

The acceptance probability term $\frac{f(\mathbf{Y})p(\mathbf{X}|\mathbf{Y})}{f(\mathbf{X})p(\mathbf{Y}|\mathbf{X})}$ can now be obtained by combining the above equations. Note that geometric factors appear in both the measurement contribution factor $\frac{f(\mathbf{Y})}{f(\mathbf{X})}$ and the probability factor $\frac{p(\mathbf{X}|\mathbf{Y})}{p(\mathbf{Y}|\mathbf{X})}$. Many of these common factors will cancel out in the final acceptance probability.

Figures F.1 and F.2 show how to compute the acceptance probability term for the lens mutation after all common factors are canceled out. Figures F.1 shows a partially mutated path, requiring an explicit connection, while figure F.2 shows a fully mutated path. The figures give an indication of the order in which the acceptance probability term is computed during mutation construction, starting with evaluating the modified sensor sensitivity function and iteratively extending the mutation with one vertex, evaluating the BSDF and sample

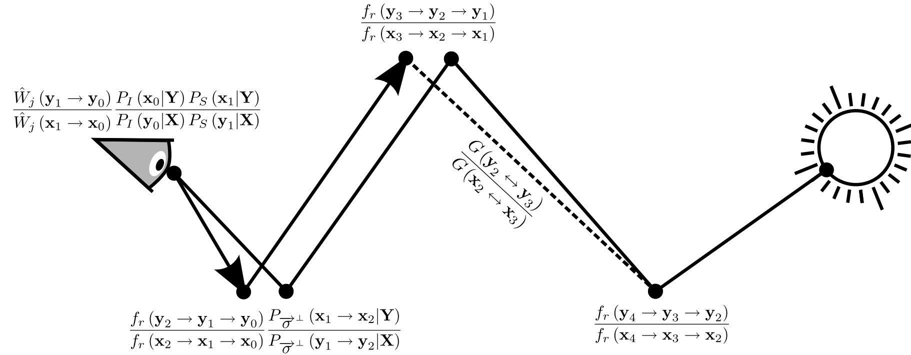


Figure F.1: Acceptance probability for partial lens mutation.

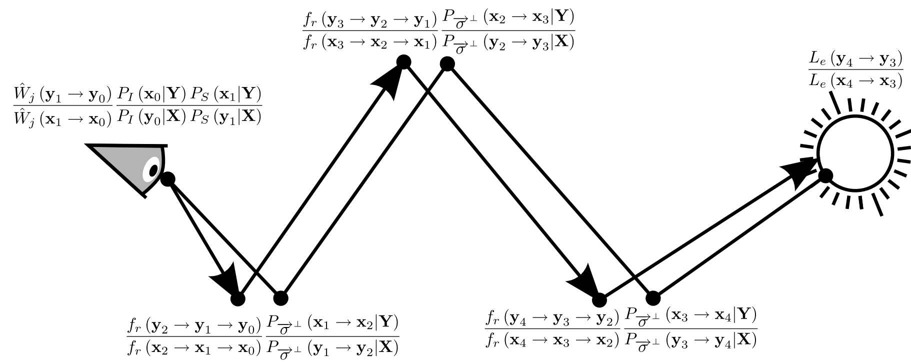


Figure F.2: Acceptance probability for full lens mutation.

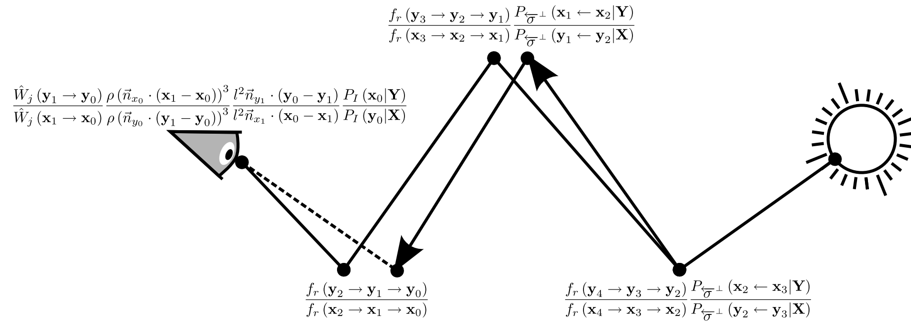


Figure F.3: Acceptance probability for partial caustic mutation.

probabilities and finally evaluating the connection or light source emittance. Note that we used the modified sensor sensitivity function for the finite aperture lens from appendix C.

Figures F.3 and F.4 show how to compute the acceptance probability term for the caustic mutation. Figures F.3 shows a partially mutated path, starting somewhere in the middle of the path, while figure F.4 shows a fully mutated path, starting at the light source. Again, the figures give an indication of the order in which the acceptance probability term is computed

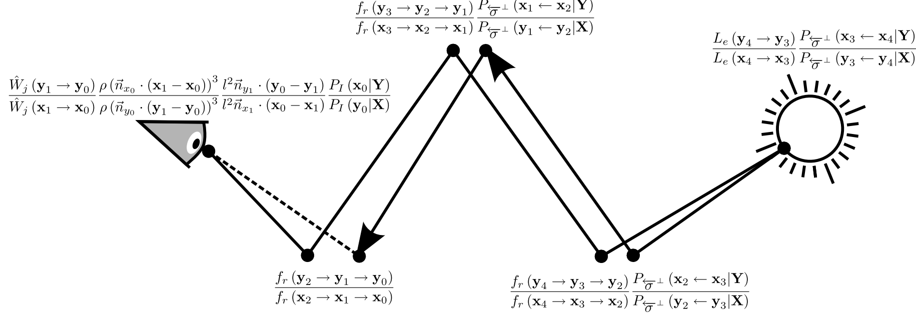


Figure F.4: Acceptance probability for full caustic mutation.

during mutation construction, starting at the light source or an intermediate vertex and iteratively extending the mutation backwards with one vertex, evaluating the BSDF and sample probabilities and finally making an explicit connection to the eye and evaluating the sensor sensitivity function. Similar to the bidirectional sample in figure E.1, the use of the modified sensor sensitivity function in the caustic mutation requires an extra correction factor $\frac{l^2 \vec{n}_1 \cdot (x_0 - x_1)}{\rho(\vec{n}_0 \cdot (x_1 - x_0))^3}$ because vertex x_1 is sampled backward.

In practice, most mutations are symmetrical, so the fraction $\frac{P_I(x_0|\mathbf{Y})}{P_I(y_0|\mathbf{X})}$, $\frac{P_S(x_1|\mathbf{Y})}{P_S(y_1|\mathbf{X})}$, $\frac{P_{\overleftarrow{\sigma}^\perp}(x_i \leftarrow x_{i+1}|\mathbf{Y})}{P_{\overleftarrow{\sigma}^\perp}(y_i \leftarrow y_{i+1}|\mathbf{X})}$ and $\frac{P_{\overrightarrow{\sigma}^\perp}(x_i \to x_{i+1}|\mathbf{Y})}{P_{\overrightarrow{\sigma}^\perp}(y_i \to y_{i+1}|\mathbf{X})}$ all cancel out to 1, significantly simplifying the computations.

F.2 Algorithm

Using the acceptance probability formulations from the previous section, we now give pseudo code for the lens and caustic mutations. Algorithm 9 and 10 give the caustic and lens mutation algorithms in pseudo code. The algorithms take an initial path $\mathbf{X}_k$ of length k and the number m, indicating the number of vertices to mutate, as input and return the mutated path $y_0 \cdots y_k$ and the corresponding acceptance probability.

Both algorithms start with mutating the eye vertex x_0. Note that this is only relevant when using a finite aperture lens. The caustic mutation proceeds with mutating the path vertices backward and finally making an explicit connection to the mutated eye vertex. The lens mutation first mutates path vertex x_1, according to the camera model (see appendix C), before mutating the remaining vertices forward from the eye. If not all vertices are mutated, an explicit connection is established.

Whenever one of the algorithms terminate before completing the mutation, for example because an explicit connection failed or an extension ray misses all scene geometry, effectively terminating the mutation, a zero acceptance probability is returned. No mutated path is returned as it will be rejected anyway.

Algorithm 9 : CausticMutation($\mathbf{X}_k, m$)

{*Mutate the first m+1 vertices*}
sample $\mathbf{y}_0 \sim P_I(\mathbf{y}_0|\mathbf{x}_0)$
$\mathbf{y}_{m+1} \cdots \mathbf{y}_k = \mathbf{x}_{m+1} \cdots \mathbf{x}_k$
$T \leftarrow \frac{P_I(\mathbf{x}_0|\mathbf{y}_0)}{P_I(\mathbf{y}_0|\mathbf{x}_0)}$
for $i = m$ to 1 **do**
  sample $\mathbf{y}_i \sim P_{\overleftarrow{\sigma}^\perp}(\mathbf{y}_i \leftarrow \mathbf{y}_{i+1}|\mathbf{x}_i\mathbf{x}_{i+1})$
  if path is terminated **then**
    return 0
  end if
  $T \leftarrow T \cdot \frac{P_{\overleftarrow{\sigma}^\perp}(\mathbf{x}_i \leftarrow \mathbf{x}_{i+1}|\mathbf{y}_i\mathbf{y}_{i+1})}{P_{\overleftarrow{\sigma}^\perp}(\mathbf{y}_i \leftarrow \mathbf{y}_{i+1}|\mathbf{x}_i\mathbf{x}_{i+1})}$
  if $i+1 = k$ **then**
    $T \leftarrow T \cdot \frac{L_e(\mathbf{y}_k \rightarrow \mathbf{y}_{k-1})}{L_e(\mathbf{x}_k \rightarrow \mathbf{x}_{k-1})}$
  else
    $T \leftarrow T \cdot \frac{f_r(\mathbf{y}_{i+2} \rightarrow \mathbf{y}_{i+1} \rightarrow \mathbf{y}_i)}{f_r(\mathbf{x}_{i+2} \rightarrow \mathbf{x}_{i+1} \rightarrow \mathbf{x}_i)}$
  end if
end for
if $V(\mathbf{y}_0 \leftrightarrow \mathbf{y}_1) = 0$ **then**
  return 0
end if
$T \leftarrow T \cdot \frac{\hat{W}_i(\mathbf{y}_1 \rightarrow \mathbf{y}_0)}{\hat{W}_i(\mathbf{x}_1 \rightarrow \mathbf{x}_0)} \frac{\rho\left(\vec{n}_{x_0} \cdot (\mathbf{x}_1 - \mathbf{x}_0)\right)^3}{\rho\left(\vec{n}_{y_0} \cdot (\mathbf{y}_1 - \mathbf{y}_0)\right)^3} \frac{l^2 \vec{n}_{y_1} \cdot (\mathbf{y}_0 - \mathbf{y}_1)}{l^2 \vec{n}_{x_1} \cdot (\mathbf{x}_0 - \mathbf{x}_1)}$
$T \leftarrow T \cdot \frac{f_r(\mathbf{y}_2 \rightarrow \mathbf{y}_1 \rightarrow \mathbf{y}_0)}{f_r(\mathbf{x}_2 \rightarrow \mathbf{x}_1 \rightarrow \mathbf{x}_0)}$

{*Return acceptance probability and mutated path*}
return $\langle \min(1, T), y_0 \cdots y_k \rangle$

Algorithm 10 : LensMutation($\mathbf{X}_k, m$)

{*Mutate the first m+1 vertices*}
sample $\mathbf{y}_0 \sim P_I(\mathbf{y}_0|\mathbf{x}_0)$
sample $\mathbf{y}_1 \sim P_S(\mathbf{y}_1|\mathbf{x}_1)$
if path is terminated **then**
  return 0
end if
$T \leftarrow \frac{\hat{W}_i(\mathbf{y}_1 \to \mathbf{y}_0)}{\hat{W}_i(\mathbf{x}_1 \to \mathbf{x}_0)} \frac{P_I(\mathbf{x}_0|\mathbf{y}_0) P_S(\mathbf{x}_1|\mathbf{y}_1)}{P_I(\mathbf{y}_0|\mathbf{x}_0) P_S(\mathbf{y}_1|\mathbf{x}_1)}$
for $i = 2$ to m **do**
  sample $\mathbf{y}_i \sim P_{\vec{\sigma}^\perp}(\mathbf{y}_{i-1} \to \mathbf{y}_i | \mathbf{x}_{i-1}\mathbf{x}_i)$
  if path is terminated **then**
    return 0
  end if
  $T \leftarrow T \cdot \frac{f_r(\mathbf{y}_i \to \mathbf{y}_{i-1} \to \mathbf{y}_{i-2})}{f_r(\mathbf{x}_i \to \mathbf{x}_{i-1} \to \mathbf{x}_{i-2})} \frac{P_{\vec{\sigma}^\perp}(\mathbf{x}_{i-1} \to \mathbf{x}_i | \mathbf{y}_{i-1}\mathbf{y}_i)}{P_{\vec{\sigma}^\perp}(\mathbf{y}_{i-1} \to \mathbf{y}_i | \mathbf{x}_{i-1}\mathbf{x}_i)}$
end for

{*Make an explicit connection*}
if $m < k$ **then**
  if $V(\mathbf{y}_m \leftrightarrow \mathbf{y}_{m+1}) = 0$ **then**
    return 0
  end if
  $T \leftarrow T \cdot \frac{G(\mathbf{y}_m \leftrightarrow \mathbf{y}_{m+1})}{G(\mathbf{x}_m \leftrightarrow \mathbf{x}_{m+1})} \frac{f_r(\mathbf{y}_{m+1} \to \mathbf{y}_m \to \mathbf{y}_{m-1})}{f_r(\mathbf{x}_{m+1} \to \mathbf{x}_m \to \mathbf{x}_{m-1})}$
  $\mathbf{y}_{m+1} \cdots \mathbf{y}_k = \mathbf{x}_{m+1} \cdots \mathbf{x}_k$
end if

if $m \geq k-1$ **then**
  $T \leftarrow T \cdot \frac{L_e(\mathbf{y}_k \to \mathbf{y}_{k-1})}{L_e(\mathbf{x}_k \to \mathbf{x}_{k-1})}$
else
  $T \leftarrow T \cdot \frac{f_r(\mathbf{y}_{m+2} \to \mathbf{y}_{m+1} \to \mathbf{y}_m)}{f_r(\mathbf{x}_{m+2} \to \mathbf{x}_{m+1} \to \mathbf{x}_m)}$
end if

{*Return acceptance probability and mutated path*}
return $\langle \min(1, T), y_0 \cdots y_k \rangle$