Interactive Manycore Photon Mapping

by

Dipl.-Inform. Bartosz Fabianowski

A dissertation submitted to the University of Dublin, Trinity College
in fulfillment of the requirements for the degree of
Doctor of Philosophy

January 2011

Declaration

I declare that this thesis has not been submitted as an exercise for a degree at this or any other university and that unless stated, it is entirely my own work.
I agree that the library may lend or copy the thesis upon request. This permission covers only single copies made for study purposes, subject to normal conditions of acknowledgment.

Dipl.-Inform. Bartosz Fabianowski
14th January 2011

Summary

Photon mapping is a state of the art global illumination rendering algorithm. Photons are traced from the light sources in a first pass and their interactions with scene surfaces stored. A second pass reconstructs illumination by density estimation, reproducing a wide range of optical phenomena. This thesis addresses the question how photon mapping can be turned from an offline algorithm to one synthesizing images at interactive frame rates.

An emerging trend in computer architecture is manycore computing. Performance increases no longer result from higher operating frequencies and instruction-level parallelism but from more processing cores working in parallel. Owing to this, photon mapping is investigated in the context of manycore computing, ensuring computations are efficiently distributed to highly parallel processing units. The specific platform employed is CUDA, utilizing commodity NVIDIA GPU hardware.

Techniques enabling efficient parallel execution in CUDA are presented for each component of the photon mapping algorithm. We begin with a focus on the tracing of rays and photons through a scene. A new heuristic for constructing the spatial index that accelerates these operations is introduced and the resulting speedup demonstrated. Spatial index traversal is addressed next, investigating both its stackless and stack-based variants. The latter is found to be more efficient. We then show how to exploit hardware resources not used during traversal as explicitly managed caches, reducing bandwidth requirements. Our findings identify computational cost as a factor whose importance is understated by the CUDA documentation. Reducing bandwidth requirements does improve performance but the overheads of explicit cache management in many cases negate the benefits.

Photon mapping uses k-th nearest neighbor density estimation, locating the k photon interactions nearest a query point to determine a smoothing kernel bandwidth for it. This costly operation is not efficiently possible on the CUDA platform. We therefore employ variable kernel density estimation instead, a priori choosing an individual bandwidth for each photon interaction. Our extension of the photon differentials framework efficiently computes bandwidths adaptive to the local interaction density from information tracked with each photon. We demonstrate image quality matching that obtained with k-th nearest neighbor density estimation.

The final photon mapping component is the photon map, a spatial index over photon interactions that accelerates their retrieval. We show how to replace the kd-tree originally used with a bounding volume hierarchy (BVH). This simplifies density estimation and permits parallel construction via the linear BVH (LBVH) algorithm. We improve the efficiency of LBVH construction, illustrate the quality of the resulting photon maps relative to an expensive heuristic BVH construction and finally present benchmarks of the complete photon mapping algorithm running in CUDA.

An extension to volumetric photon mapping for simulating light transport in participating media forms the final part of the thesis. Photon differentials are further extended to account for extinction and scattering. Volumetric photon mapping leads to more expensive density estimation. We therefore investigate additional improvements by rederiving the efficient beam radiance estimate method, correcting its physical units, and showing that packetization provides a large performance benefit.

Acknowledgments

First and foremost, I thank Suule Soo for entering my life during this PhD, giving it all an additional sense of purpose. I am grateful for the unconditional support I have received from her and from my family throughout the weeks, months and years of this onerous but rewarding journey. My parents and sister have always been the cornerstones of my life and given me the strength I needed.

My flatmates Мария Левчук and Martin Pražák provided me with the cheerful distractions that are so important in life. I am equally grateful to my other friends here in Ireland, back home and elsewhere in the world for enriching my life in the years leading up to this PhD and during the time it took to complete it. Many fellow members of the GV2 group at Trinity College have become good friends over the years. I especially thank Paul Reitsma, Colin Flower and Yann Morvan for numerous long and insightful discussions.

I was very fortunate to have Dr John Dingliana as my supervisor. He provided me with invaluable advice and support whenever I needed it but also allowed me to roam freely and explore ideas. I am glad I had the opportunity to work on my PhD under his supervision. Dr Steven Collins gave me the inspiration that started off this PhD in the right direction while my examiners Dr Erik Reinhard and Dr Michael Manzke were there at the end, providing me with their professional assessment of my work. I thank them all for their involvement.

This thesis is dedicated to Kees Kwak, a dear family friend. While the completion of this PhD marks a happy moment in my life, his current battle with cancer reminds me that good things must never be taken for granted and many difficult moments lie ahead for all of us.

Contents

Contents vii

List of Tables xi

List of Figures xiii

List of Algorithms xv

List of Acronyms xvii

1. Introduction 1
- 1.1. Motivation . . . 1
  - 1.1.1. Photon Mapping . . . 1
  - 1.1.2. Manycore Computing . . . 2
- 1.2. Scope and Limitations . . . 3
- 1.3. Contributions . . . 4
  - 1.3.1. Peer Reviewed Publications . . . 5
- 1.4. Thesis Overview . . . 5

2. Background 7
- 2.1. Light Transport . . . 7
  - 2.1.1. Radiometry . . . 7
  - 2.1.2. Rendering Equation . . . 9
  - 2.1.3. Reflection . . . 10
  - 2.1.4. Participating Media . . . 12
- 2.2. Manycore Computing . . . 14
  - 2.2.1. CPU-GPU Convergence . . . 14
  - 2.2.2. CUDA Programming Model . . . 15
  - 2.2.3. CUDA Hardware Implementation . . . 17
  - 2.2.4. Algorithmic Building Blocks . . . 21

3. Related Work 25
- 3.1. Ray Tracing . . . 25
  - 3.1.1. Recursion . . . 25
  - 3.1.2. Footprints . . . 26
  - 3.1.3. Surface Intersection . . . 28
- 3.2. Spatial Indexing . . . 30
  - 3.2.1. Space Partitioning . . . 31
  - 3.2.2. Primitive Partitioning . . . 31

3.2.3. Construction . . . 32
3.2.4. Traversal . . . 35
3.3. Monte Carlo Rendering . . . 39
3.3.1. Monte Carlo Quadrature . . . 39
3.3.2. Rendering Algorithms . . . 40
3.3.3. Sampling . . . 41
3.4. Photon Mapping . . . 42
3.4.1. Photon Tracing . . . 43
3.4.2. Density Estimation . . . 45
3.4.3. Photon Map . . . 50
3.5. Simplification . . . 52
3.5.1. Virtual Point Lights . . . 52
3.5.2. Photon Mapping . . . 54
3.5.3. Object Space Interpolation . . . 56
3.5.4. Participating Media . . . 58

4. Ray and Photon Tracing **61**
4.1. kd-Tree Construction . . . 61
4.1.1. Geometric Probability . . . 62
4.1.2. Numerical Approximation . . . 63
4.1.3. SIROH . . . 63
4.1.4. Results and Discussion . . . 65
4.2. Stackless kd-Tree Traversal . . . 68
4.2.1. Zero Volume Nodes . . . 68
4.2.2. Traversal Algorithm . . . 70
4.2.3. Extensions . . . 71
4.2.4. Results and Discussion . . . 74
4.3. Stack-Based kd-Tree Traversal . . . 76
4.3.1. Node Caching . . . 76
4.3.2. Stack Caching . . . 78
4.3.3. Results and Discussion . . . 80

5. Density Estimation **85**
5.1. Photon Differentials . . . 85
5.1.1. Initialization . . . 86
5.1.2. Specular Reflection . . . 88
5.1.3. Diffuse Reflection . . . 89
5.1.4. Russian Roulette . . . 92
5.2. Bandwidth Selection . . . 92
5.2.1. Anisotropic Kernel Support Region . . . 93
5.2.2. Dampened Adaptation . . . 95
5.2.3. Results and Discussion . . . 96

6. Photon Map **103**
6.1. BVH Construction . . . 103
6.1.1. Voxel Volume Heuristic . . . 104
6.1.2. Linear BVH . . . 105
6.1.3. Termination Criterion . . . 107
6.1.4. Results and Discussion . . . 107
6.2. BVH Storage . . . 109
6.2.1. Compact Representation . . . 110
6.2.2. Ray Tracing Traversal . . . 111
6.2.3. Photon Mapping Traversal . . . 112
6.2.4. Results and Discussion . . . 113
6.3. Combined Results . . . 115

7. Participating Media **117**
7.1. Beam Radiance Estimate . . . 117
7.2. Photon Differentials . . . 119
7.2.1. Propagation . . . 119
7.2.2. Scattering . . . 121
7.2.3. Russian Roulette . . . 124
7.3. Bandwidth Selection . . . 125
7.3.1. Isotropic Kernel Support Region . . . 125
7.3.2. Dampened Adaptation . . . 125
7.3.3. Spectral Considerations . . . 126
7.3.4. Results and Discussion . . . 126
7.4. Stream Processing . . . 128
7.4.1. Job Queuing . . . 132
7.4.2. Packetization . . . 133
7.4.3. Results and Discussion . . . 133
7.5. Combined Results . . . 134

8. Conclusions and Future Work **137**
8.1. Conclusions . . . 137
8.2. Future Work . . . 139

A. Benchmarks **141**
A.1. Benchmark Environment . . . 141
A.1.1. Code Base I . . . 141
A.1.2. Code Base II . . . 141
A.2. Benchmark Scenes . . . 142
A.2.1. Common Scenes . . . 142
A.2.2. Photon Mapping Scenes . . . 142

Bibliography **147**

List of Tables

2.1. Summary of radiometric terms 8

4.1. Ray tracing frame rates with the numerical approximation from section 4.1.2 and SIROH 66
4.2. Ray tracing frame rates with SIROH for two other ray tracers 67
4.3. n_T and n_I with the numerical approximation from section 4.1.2 and SIROH 67
4.4. Construction timings and kd-tree statistics with SIROH 67
4.5. n_T with stackless traversal 74
4.6. Ray tracing frame rates with stackless traversal 75
4.7. Instructions per ray with stackless traversal 75
4.8. Global memory accesses during inner node traversal with explicitly managed caches . 81
4.9. Instructions per ray with explicitly managed caches 82
4.10. Ray tracing frame rates with explicitly managed caches 83

5.1. Photon tracing and density estimation statistics 96

6.1. Rendering frame rates with different minimal photon map leaf size thresholds 108
6.2. Photon map node counts and construction timings with LBVH 109
6.3. Photon map n_T, n_I and rendering frame rates with LBVH construction 109
6.4. Traversal statistics and rendering frame rates with compact BVH photon map 114
6.5. Global memory accesses, instructions per ray and frame rates with compact BVH . . . 115
6.6. Photon mapping statistics and frame rates with dynamic illumination 116

7.1. Light source type and participating medium properties 127
7.2. Adaptation dampening for different bounce depths 127
7.3. Photon tracing and density estimation statistics 128
7.4. Density estimation instructions per ray with different job queue types and packetization 134
7.5. Density estimation times per frame with different job queue types and packetization . 135
7.6. Volumetric photon mapping statistics and frame rates with dynamic illumination . . . 135

A.1. Properties of the benchmark scenes used for algorithm evaluation 144

List of Figures

2.1. High level view of CUDA . . . 15
2.2. CUDA thread and memory hierarchy . . . 16
2.3. CUDA texture processor cluster on the GT200 chip . . . 18
2.4. Parallel prescan in CUDA . . . 22
2.5. Parallel stream compaction . . . 23
2.6. Split . . . 24
2.7. Parallel prescan order for CUDA radix sort . . . 24

3.1. Linear BVH construction . . . 35
3.2. Two-dimensional slabs test . . . 37

4.1. Probability p_N that a node N is visited by a ray originating in $S \setminus N$ . . . 64
4.2. Information reuse in SIROH . . . 65
4.3. Stackless kd-tree traversal by restarting at the root with shifted ray start . . . 69
4.4. Stackless kd-tree traversal by restarting at the root with flags register . . . 73

5.1. Kernel support region defined by photon differentials . . . 93
5.2. Distribution of skewed ellipsoid semiprincipal axis lengths . . . 97
5.3. Images rendered by ray tracing and photon mapping . . . 98

6.1. Efficient linear BVH construction . . . 106
6.2. BVH bounding plane inheritance in two dimensions . . . 110

7.1. Images rendered by volumetric photon mapping . . . 129
7.2. Rendering by stream processing . . . 132

A.1. Benchmark scenes for algorithm evaluation . . . 143
A.2. Additional benchmark scenes for photon mapping evaluation . . . 145

List of Algorithms

3.1. Möller-Trumbore intersection test for triangle $\triangle(\vec{v}_0, \vec{v}_1, \vec{v}_2)$ and ray $\vec{z}(t) = \vec{x} + t\vec{\omega}$ . . 29
3.2. Precalculation for Wald triangle intersection test 30
3.3. Wald intersection test for a triangle represented by $k, \vec{n}', \vec{e}'_1, \vec{e}'_2$ and ray $\vec{z}(t) = \vec{x} + t\vec{\omega}$. 30
3.4. Slabs test for node $(\vec{m}, \vec{M})$ and ray $\vec{z}(t) = \vec{x} + t\vec{\omega}$ 37
3.5. Photon map test for node $(\vec{m}, \vec{M})$ and point $\vec{x}$ 37
4.1. Stackless kd-tree traversal 71
4.2. Stackless kd-tree traversal with push-down and short-stack extensions 72
4.3. Push operation with stack cache I 78
4.4. Pop operation with stack cache I 78
4.5. Push operation with stack cache II 79
4.6. Pop operation with stack cache II 79
6.1. Slabs test for sibling nodes encoded as $\vec{m}$, $\vec{M}$, $\vec{l}$, $\vec{L}$ and ray $\vec{z}(t) = \vec{x} + t\vec{\omega}$ with $t \in [t_a, t_b]$ 112
6.2. Photon map test for sibling nodes encoded as $\vec{m}$, $\vec{M}$, $\vec{l}$, $\vec{L}$ and point $\vec{x}$ 113

List of Acronyms

AABB	axis-aligned bounding box	31
BFS	breadth-first search	33
BRDF	bidirectional reflectance distribution function	10
BVH	bounding volume hierarchy	31
CTA	cooperative thread array	17
CUDA	compute unified device architecture	15
DFS	depth-first search	34
DPU	double precision unit	18
GPU	graphics processing unit	14
LBVH	linear BVH	34
RSM	reflective shadow map	53
SAH	surface area heuristic	32
SFU	special function unit	18
SIMD	single instruction multiple data	14
SIMT	single instruction multiple thread	15
SIROH	scene-interior ray origin heuristic	63
SM	streaming multiprocessor	18
SP	streaming processor	18
TPC	thread processor cluster	18
VPL	virtual point light	53
VVH	voxel volume heuristic	51

1. Introduction

The physical world is visible because of light being emitted, interacting with objects and the atmosphere and ultimately reaching an observer. Physically based rendering aims to simulate this process and synthesize realistic images of three-dimensional scenes as seen by a virtual camera. Photon mapping [Jen96] is a state of the art algorithm combining efficient simulation of light transport with the ability to reproduce a wide range of optical phenomena. Its computational cost is still substantial, however, placing it in the offline domain. With the advent of the manycore computing paradigm and especially a platform implementing it on commodity hardware [NVI10b], the opportunity arises to turn photon mapping into an interactive rendering algorithm.

1.1. Motivation

Competing goals of ever higher realism and rendering speeds exist in computer graphics. Commodity graphics processing units (GPUs) now routinely synthesize convincing images of three-dimensional scenes at real time frame rates by rasterization. However, only the surface closest to the observer is then visualized and illumination calculated not by simulating physical light transport but by applying fast heuristics yielding visually pleasing results [SA10].

The focus is placed differently in physically based rendering, simulating light transport from source to observer. A global illumination simulation is obtained by accounting for light arriving from sources directly and also that redirected via other parts of the scene. This allows optical effects such as color bleeding from one surface onto another, subtle indirect illumination and highly focused caustics to be reproduced [DBB06]. Numerous rendering algorithms exist, each characterized by the subset of light paths from source to observer it can simulate [Hec90] and the tradeoffs between physical accuracy and speed made in its design.

1.1.1. Photon Mapping

Photon mapping is a two-pass physically based global illumination rendering algorithm. Particles carrying flux are emitted by the light sources and traced through their interactions with the scene first. Information about these interactions is stored in the photon map data structure. To generate an actual image, a second pass determines the surfaces visible from a virtual camera and computes their illumination. The link between the two passes is provided by kernel density estimation [Par62]. A circular kernel is centered around each interaction, spreading flux to nearby regions of the scene and allowing illumination to be reconstructed. Using this approach, all paths from light source to observer can be accounted for. The main tradeoff is between variance (noise) and bias (blurring). Increasing the number of photons emitted reduces both noise and bias at the expense of higher computational cost. Varying the kernel bandwidth allows the tradeoff to be adjusted locally.

In photon mapping, kth-nearest-neighbor searches are used to select kernel bandwidths [LQ65].

Whenever illumination is being reconstructed at a point, the k nearest photon interactions are located. A kernel is then centered around each with bandwidth equal to the distance at which the kth-nearest interaction was found. This approach yields kernel bandwidths adaptive to the local interaction density. In areas of highly focused illumination, small bandwidths and crisp caustics result. Regions of sparse illumination are assigned larger bandwidths, reducing variance.

While elegant in design and general in the range of optical effects it can reproduce, photon mapping remains computationally expensive. Photon map construction is a recursive process that does not efficiently scale with the number of computational cores. Locating the k nearest interactions at each visible point requires the photon map to be searched for candidates and the k nearest to be extracted. Although these searches can be accelerated by optimizing the photon map [WGS04] and some of them eliminated by interpolating between neighboring pixels [WRC88, KGPB05], such techniques introduce their own overheads and do not reduce overall computational complexity.

The question emerges whether bottlenecks in the algorithm can be addressed differently, by replacing parts of it with alternative solutions. Eliminating the need for kth-nearest-neighbor searches would be an important step but requires an alternative method of selecting kernel bandwidths to be found. Could information such as photon differentials [SFES07] traced together with each photon be used for this purpose in a wide range of scenes? What further acceleration opportunities would arise from removing the need to locate neighboring interactions? Is it ultimately possible to achieve interactive frame rates on commodity hardware?

Photon mapping can easily be extended to simulate interactions with participating media [Jen01]. While the algorithm remains elegant and general, computational cost increases even further. The investigation should therefore be extended to also determine whether photon mapping for participating media can be accelerated to reach interactive speeds.

1.1.2. Manycore Computing

Microprocessor technology is famously following Moore's law [Moo65] of exponentially increasing performance. With operating frequencies approaching physical limits and diminishing returns from added instruction-level parallelism, this rate of growth can only be sustained by increasing the number of processing cores working in parallel. Beginning with two cores on one die [TDJ+02], the number of cores has been increasing ever further. This shift in computer architecture is creating new challenges for algorithm design, requiring scalable algorithms to be devised for optimal performance.

As the number of parallel processing cores increases, traditional programming models become ever less efficient. Eventually, the point of manycore computing is reached where all performance stems from parallel processing and serial algorithms are no longer viable. In recent years, there has been a strong trend toward manycore architectures in both CPU [SCS+08] and GPU [Adv09, NVI10b] design. Traditional CPUs are extended by increasing the number of general purpose processing cores. GPUs, historically based on a parallel processing model, are extended by adding better support for general purpose computing to their cores. This convergence of CPU and GPU architectures on the manycore paradigm motivates investigations of the implications for existing algorithms.

CUDA is the first widely available manycore computational platform. Built on commodity NVIDIA GPU hardware, it provides a low-cost entry to manycore computing. As an example, the GTX 280, a member of the second generation of CUDA-capable GPUs, has 240 parallel processing cores and offers a theoretical peak performance of almost 1 TFLOP/s. This speed is achieved by omitting many

components found in traditional CPU designs such as branch prediction and most caches. Instead, all performance is due to a large number of simple cores working in parallel.

Threads executing on the cores are organized in a hierarchical manner. The cost of cooperation and synchronization varies with hierarchy level. At the bottom of the hierarchy, groups of 32 threads execute in lock-step, simultaneously processing the same instruction. Having evolved from earlier NVIDIA GPUs, the CUDA platform also includes legacy features which should be utilized for optimal performance, such as dedicated texture address calculation and interpolation units.

With the manycore paradigm an emerging trend in computer architecture, its efficient use in algorithms is becoming increasingly more important. Photon mapping has the potential to be a fully parallel rendering algorithm. Photons are traced independently of each other and illumination can be computed independently for all pixels. Parallelizing photon map construction would remove the last parallelization bottleneck. More challenges exist, however. The simplification of individual processing cores and the ways in which they cooperate and synchronize call into question established best practices. Can the photon mapping algorithm be made to perform well on a manycore platform with its novel architecture?

Because it is readily and widely available, CUDA is a good target for this work. Although some of its features are specific to CUDA, a large overlap exists with other manycore architectures such as those by Intel and AMD. By adapting photon mapping to the manycore paradigm, the unprecedented processing power provided by this new trend could be harnessed.

1.2. Scope and Limitations

In this thesis, we investigate the challenges of adapting the photon mapping algorithm to obtain interactive rendering speeds on commodity hardware. Motivated by the trend toward manycore computing, our main aims are to remove scalability bottlenecks and ensure the algorithm makes efficient use of the massively parallel processing units provided by a manycore platform.

Being an emerging trend, manycore computing is a volatile and moving target. We concentrate on the CUDA platform, specifically that provided by a GTX 280 GPU. This GPU has been at the top end of NVIDIA's consumer offering for several years, creating a stable platform for our research. At the time of this writing, the next generation of NVIDIA GPU hardware is reaching wide availability. Reevaluating our work on this evolution of the CUDA platform would be interesting. However, due to the many subtle technical challenges in porting forward code and the need to thoroughly investigate any architectural differences, it is not part of our current effort.

A single GPU provides a complete manycore platform with massively parallel processing cores, a full memory subsystem and hierarchical thread cooperation and synchronization mechanisms. Our goal of investigating photon mapping on a manycore architecture can be met with this hardware. Using several GPUs would increase the available processing power but would also add another level of complexity and new, different technical challenges. Multi-GPU computing is therefore a separate, extensive topic outside the scope of our work.

Many extensions of the photon mapping algorithm are orthogonal to our effort. Techniques for interpolating and reusing information can provide acceleration with few visible artifacts. Because such methods are largely independent of the way in which light transport is simulated, our focus is on simulation only. Combining our work with interpolation techniques is largely an implementation problem and therefore not part of this thesis. In the interest of interactive performance, we also do

not use the optional final gather component of the photon mapping algorithm which improves image quality at the expense of an order of magnitude more illumination reconstructions.

As photon mapping is a physically based rendering algorithm, we ensure physical plausibility in our work. Simulating light transport both with and without participating media, we support several surface and participating medium models that follow physical principles, are efficient to evaluate and widely used in computer graphics. In the resulting rendering algorithms, photons are retraced every frame, allowing for interactive light source manipulation. Because illumination is fully recalculated per frame, the scene geometry could also be modified at this rate. However, our focus is on dynamic illumination and the incorporation of dynamic geometry is beyond its scope.

Research Question

Can scalability bottlenecks and inefficiencies in the photon mapping algorithm be overcome and the performance characteristics of the emerging manycore computing paradigm be leveraged for global illumination rendering at interactive frame rates on consumer hardware?

1.3. Contributions

We propose solutions that make the photon mapping algorithm operate efficiently on a manycore platform and allow it to achieve interactive speeds. The first aspect investigated is the problem of tracing photons and rays through a scene. Both operations involve the traversal of a spatial index to find the nearest surface seen in a given direction:

- In cooperation with a fellow researcher, we propose and evaluate SIROH, a novel heuristic for constructing spatial indexes. We demonstrate ray tracing performance often exceeding that obtained using the best currently known method, the surface area heuristic (SAH) [MB90].
- We study the traversal of the resulting spatial indexes without the need for a stack data structure. If nodes of zero volume are present in an index, traversal can be shown to enter an infinite loop. We present a simple modification of the traversal algorithm that avoids this problem.
- On current hardware, stack-based traversal outperforms the stackless variant. We investigate opportunities for further acceleration by using a small pool of fast memory offered by CUDA and registers as explicitly managed caches, benchmarking several methods.

Our next focus is on eliminating the need for kth-nearest-neighbor searches by selecting bandwidths for all photon interactions independently:

- We extend the concept of photon differentials to efficiently handle diffuse reflections and photon termination by Russian roulette. We also reduce storage and bandwidth requirements.
- Photon differentials allow anisotropic kernel support regions adaptive to the local interaction density to be constructed. We combine these with a dampening of the adaptation that avoids excessive support regions where illumination arrives via different paths.

Having constructed a kernel support region for each photon interaction, we replace the kd-tree traditionally used for the photon map with a bounding volume hierarchy (BVH):

- We adapt the voxel volume heuristic (VVH) [WGS04] to BVH photon maps and show that its traversal performance is not significantly better than that achieved by using the linear BVH (LBVH) construction algorithm [LGS+09] which is scalable and efficient in CUDA.
- For storing the BVH, we propose a novel, compact representation that reduces storage and bandwidth requirements and exhibits improved traversal performance.
- We demonstrate the integration of all components into a complete global illumination rendering algorithm that enables photon mapping at interactive speeds on consumer hardware.

Our final focus is on the extension of photon mapping to participating media:

- Basing our work on the concept of the beam radiance estimate [JZJ08a], we rederive the equations with correct physical units.
- We extend photon differentials to take into account extinction and scattering in participating media. Comparing with the results of kth-nearest-neighbor searches, we investigate the amount of adaptation dampening required to adjust for illumination arriving via different paths across a range of scenes and illumination conditions.
- Demonstrating the integration of all components into a rendering algorithm for participating media, we show that illumination reconstruction significantly benefits from grouping threads into packets and compare different grouping methods.

1.3.1. Peer Reviewed Publications

[FD09b] Bartosz Fabianowski and John Dingliana. Interactive Global Photon Mapping. In *Computer Graphics Forum (Proceedings of the 20th Eurographics Symposium on Rendering (EGSR))*, 28(4), pages 1151–1159, 2009.

[FFD09] Bartosz Fabianowski, Colin Flower and John Dingliana. A Cost Metric for Scene-Interior Ray Origins. In *Short Paper Proceedings of the 30th Annual Conference of the European Association for Computer Graphics (Eurographics)*, pages 49–52, 2009.

[FD09a] Bartosz Fabianowski and John Dingliana. Compact BVH Storage for Ray Tracing and Photon Mapping. In *Proceedings of the 9th Irish Workshop on Computer Graphics (Eurographics Ireland)*, pages 1–8, 2009. **Best Paper Award**

1.4. Thesis Overview

Chapter 1: Introduction provides a general introduction to the motivation behind this work, outlines its scope and gives a summary of the contributions, including a list of peer-reviewed publications.

Chapter 2: Background begins with an overview of the physics behind light transport. Manycore computing in general and the CUDA platform in particular are then explained, detailing its architecture, performance characteristics and the principles of efficient manycore algorithms.

Chapter 3: Related Work first introduces the important operation of ray tracing and the techniques devised to accelerate it. Different types of spatial indexes are described along with their efficient construction and traversal. Photon mapping is presented next in the context of Monte Carlo physically based rendering. The chapter concludes with a look at simplifications commonly used to achieve interactive rendering speeds.

Chapter 4: Ray and Photon Tracing is concerned with the tracing of photons and rays through a three-dimensional scene. This key operation is known to be fastest when a spatial index in the form of a kd-tree is used. In this chapter, a novel heuristic for constructing kd-trees is presented and efficient kd-tree traversal in CUDA is addressed. Benchmarks are used to analyze the performance of the discussed methods.

Chapter 5: Density Estimation focuses on reconstructing illumination from stored photon interactions without kth-nearest-neighbor searches. The concept of photon differentials is extended from specular interactions only to diffuse reflections and Russian roulette as used in photon mapping. Its efficiency is also improved by reducing storage and bandwidth requirements. The construction of anisotropic kernel support regions is then described and a dampening of the adaptation to the local interaction density proposed that avoids excessively large support regions. Applicability to a number of scenes with diffuse and specular surfaces is shown.

Chapter 6: Photon Map deals with the final component required for interactive photon mapping. The traditionally used kd-tree photon map is replaced with a BVH. An analysis shows that an efficient CUDA construction method produces BVHs of high quality. A novel compact BVH representation is introduced and shown to reduce storage and bandwidth requirements. All components are then integrated and a complete photon tracing algorithm is obtained. Benchmarks illustrate the performance and demonstrate interactive frame rates.

Chapter 7: Participating Media extends the focus to scenes with participating media. The beam radiance estimate used for efficient density estimation is rederived with correct physical units first, followed by an extension of photon differentials and bandwidth selection to this scenario. A comparison with kth-nearest-neighbor density estimation shows the adaptation dampening required to account for the influence of illumination arriving via different paths on the interaction density. Illumination reconstruction performance is shown to benefit from grouping threads into packets and different methods for doing so are compared. Image quality and rendering speeds are again assessed in benchmarks.

Chapter 8: Conclusions and Future Work summarizes the work presented in the previous chapters, providing an overview and a discussion of the results and suggesting directions for future work.

2. Background

The discussion of physically based rendering at interactive frame rates requires background knowledge from two areas. A model of the physics behind light transport is needed as well as an understanding of the computer architecture being considered. Both are introduced in this chapter.

2.1. Light Transport

At the core of all physically based rendering lies the simulation of light transport in the real world. The behavior of light is studied by the field of *optics* which provides several models for it [BW99]. Physically based rendering typically concerns itself with geometric optics only [DBB06]. This is the simplest model available, making the assumptions that light travels in straight lines, is unaffected by external forces and propagates instantly. Additional effects explained by wave optics, such as rays curving due to gravitational forces [WSE04], are sometimes additionally considered.

2.1.1. Radiometry

Before describing and simulating light transport, physical units must be defined that allow the amount of light present to be quantified. The relevant disciplines are *radiometry* and *photometry* [McC94]. Both define systems of physical units but with different backgrounds. Photometry is based on the perceptual effects light has on the human visual system. Radiometry is independent of a potential human observer. Synthesizing images as captured by a virtual camera, physically based rendering uses radiometric theory and quantities.

2.1.1.1. Solid Angle

A mathematical concept frequently encountered in radiometry is that of the *solid angle* Ω. Given a point $\vec{x}$ and a surface S, the solid angle subtended by S is the area covered by its projection onto the unit sphere around $\vec{x}$. This is useful for expressing portions of directions. For example, a sphere containing all possible directions has solid angle 4π, a hemisphere above or a below a surface, 2π.

2.1.1.2. Radiance

Surfaces on which light originates are *light sources*. A source emits a *radiant power* or *radiant flux* Φ into a solid angle Ω above an area A. Flux is measured in watts (W), solid angle in steradians (sr) and area in square meters (m^2). The principal quantity in physically based rendering is *radiance* L, the differential flux traveling in a direction per unit solid angle per unit projected area,

$$L(\vec{x}, \vec{\omega}) = \frac{d^2\Phi}{dA^{\perp}\, d\Omega}. \tag{2.1}$$

Quantity	Symbol	Unit
Radiance	L	$W\,m^{-2}\,sr^{-1}$
Intensity	I	$W\,sr^{-1}$
Exitance, Radiosity, Irradiance	M, B, E	$W\,m^{-2}$
Power, Flux	Φ	W

Table 2.1.: Summary of radiometric terms

This may vary with position in space $\vec{x}$ and direction $\vec{\omega}$, creating a five-dimensional *radiance field*. Projected area is used as radiance depends on the area orthogonal to its direction. When radiance interacts with a surface of normal $\vec{n}$, the projected area differential becomes $dA^{\perp} = |\vec{n} \cdot \vec{\omega}|\ dA$ and

$$L\left(\vec{x}, \vec{\omega}\right) = \frac{d^2\Phi}{|\vec{n} \cdot \vec{\omega}|\ dA\, d\Omega}. \tag{2.2}$$

Radiance has two essential properties. First, it is invariant along straight lines in a vacuum. If not occluded by a surface between them, the radiance L_i incident at a point $\vec{x}$ from the direction of a point $\vec{y}$ is identical to the radiance L_o emanating from $\vec{y}$ toward $\vec{x}$,

$$L_i(\vec{x}, \widehat{\vec{y} - \vec{x}}) = L_o(\vec{y}, \widehat{\vec{x} - \vec{y}}). \tag{2.3}$$

Its second important property is that both cameras and human observers respond proportionally to incident radiance. A physically based rendering algorithm should therefore compute the radiance reaching a virtual camera.

2.1.1.3. Cumulative Quantities

The radiance field on a surface can be integrated into cumulative quantities. Integrating projected radiance L over the surface area A yields *radiant intensity* I, the total flux emanating or incident per unit solid angle,

$$I\left(\vec{\omega}\right) = \int_A L\left(\vec{x}, \vec{\omega}\right) |\vec{n} \cdot \vec{\omega}|\ dA = \frac{d\Phi}{d\Omega}. \tag{2.4}$$

Integrating projected outgoing radiance L_o over the hemisphere of directions Ω above a point $\vec{x}$ yields *radiant exitance* M or *radiosity* B, the total flux emanating per unit area. The corresponding integral of projected incident radiance L_i yields *irradiance* E, the total flux incident per unit area,

$$M\left(\vec{x}\right) = \int_\Omega L_o\left(\vec{x}, \vec{\omega}\right) |\vec{n} \cdot \vec{\omega}|\ d\Omega = \frac{d\Phi_o}{dA}, \tag{2.5}$$

$$E\left(\vec{x}\right) = \int_\Omega L_i\left(\vec{x}, \vec{\omega}\right) |\vec{n} \cdot \vec{\omega}|\ d\Omega = \frac{d\Phi_i}{dA}. \tag{2.6}$$

By integrating over the area A and the hemisphere of directions Ω, the total radiant power Φ emanating from or incident on a surface can be calculated,

$$\Phi\ = \int_A \int_\Omega L\left(\vec{x}, \vec{\omega}\right) |\vec{n} \cdot \vec{\omega}|\ d\Omega\, dA.$$

Table 2.1 summarizes these radiometric terms. For brevity, the prefix *radiant* is consistently omitted throughout this thesis.

2.1.1.4. Point Light Sources

A popular approximation for small light-emitting surfaces is the *point light*. By reducing the source area to zero, light originates from a single point $\vec{y}$. A point light is physically impossible and introduces singularities that require special case treatment. Because no source area A exists, the integral in equation 2.4 cannot be evaluated and intensity $I(\vec{\omega})$ must be directly specified for all directions. Not associated with any surface, a point light may emit into the full sphere of directions.

The lack of a light source area also means that radiance is undefined and cannot be integrated. However, irradiance due to a point light may be computed in another way. Let $\vec{x}$ be an unoccluded point on a surface. The area differential dA at $\vec{x}$ projects to $dA^{\perp} = |\vec{n} \cdot \widehat{(\vec{x}-\vec{y})}|\, dA$ and subtends a differential solid angle $d\Omega = \|\vec{x}-\vec{y}\|^{-2}\, dA^{\perp} = \|\vec{x}-\vec{y}\|^{-2} |\vec{n} \cdot \widehat{(\vec{x}-\vec{y})}|\, dA$ at $\vec{y}$. Inserting this, $I(\vec{\omega}) = \frac{d\Phi_o}{d\Omega}$ from equation 2.4 and $\Phi_i = \Phi_o$ due to no occlusion into equation 2.6 yields

$$E(\vec{x}) = \frac{d\Phi_i}{dA} = \frac{d\Phi_o}{dA} = \frac{d\Phi_o}{d\Omega} \frac{|\vec{n} \cdot \widehat{(\vec{x}-\vec{y})}|}{\|\vec{x}-\vec{y}\|^2} = I(\widehat{\vec{x}-\vec{y}}) \frac{|\vec{n} \cdot \widehat{(\vec{x}-\vec{y})}|}{\|\vec{x}-\vec{y}\|^2}. \tag{2.7}$$

The simplest point light source is *isotropic*, emitting flux evenly into the sphere of directions with $I(\vec{\omega}) = \frac{\Phi}{4\pi}$. The irradiance due to this type of light source then is

$$E(\vec{x}) = \frac{\Phi |\vec{n} \cdot \widehat{(\vec{x}-\vec{y})}|}{4\pi \|\vec{x}-\vec{y}\|^2}.$$

2.1.1.5. Color

So far, a single color of light has been presumed. In reality, all radiometric quantities have *spectral* variants, defined as differentials of the terms in table 2.1 with respect to wavelength λ. Fully accounting for a continuous spectrum is too computationally expensive. Physically based rendering therefore uses discrete color bands. A simple and popular approximation is the RGB model, computing radiance and related quantities separately for three bands. To better reproduce spectral effects such as dispersion, a higher number of color bands may be chosen [Col94].

2.1.1.6. Photons

Corpuscular theory models light as a stream of elementary particles, *photons*, each carrying an energy $E = hc/\lambda$ [Pla00] where h is Planck's constant, c is the speed of light and λ is the wavelength. Simulating the propagation of actual photons carrying elementary amounts of light is prohibitively expensive. However, it serves as the inspiration for photon mapping, the physically based rendering algorithm used in this thesis. In photon mapping, particles also called photons but carrying larger fractions of the total light source flux Φ are emitted and traced. Color may be accounted for by assigning each photon information for multiple color bands.

2.1.2. Rendering Equation

A concise model of light transport on surfaces is given by the *rendering equation* [Kaj86],

$$L_o(\vec{x}, \vec{\omega}) = L_e(\vec{x}, \vec{\omega}) + \int_{\Omega_i} f(\vec{x}, \vec{\omega}_i, \vec{\omega})\, L_i(\vec{x}, \vec{\omega}_i)\, |\vec{n} \cdot \vec{\omega}_i|\, d\Omega_i. \tag{2.8}$$

Here, L_o is the radiance leaving a surface point $\vec{x}$ in direction $\vec{\omega}$. It is the sum of the radiance L_e emitted by the surface itself and the radiance L_i incident from other surfaces reflected by it. Incident radiance from a hemisphere of directions Ω_i above $\vec{x}$ is projected onto the surface and reflected according to a *bidirectional reflectance distribution function (BRDF)* f [Nic65].

In vacuum and, approximatively, also air, radiance exchange between surfaces follows equation 2.3. Radiance emanating from a surface instantaneously propagates in a straight line to the nearest surface on its path. The radiance incident at a point $\vec{x}$ from direction ω_i is thus identical to that emanating in direction $-\omega_i$ from the nearest surface point $\vec{y}$ seen in this direction. With $\vec{r}$ a helper function that locates $\vec{y}$, the relationship is $L_i(\vec{x},\vec{\omega}_i) = L_o(\vec{r}(\vec{x},\vec{\omega}_i), -\vec{\omega}_i)$. Inserting this into the rendering equation yields the definition of a steady state radiance equilibrium in the scene,

$$L(\vec{x},\vec{\omega}) = L_e(\vec{x},\vec{\omega}) + \int_{\Omega_i} f(\vec{x},\vec{\omega}_i,\vec{\omega})\, L(\vec{r}(\vec{x},\vec{\omega}_i), -\vec{\omega}_i)\, |\vec{n}\cdot\vec{\omega}_i|\, d\Omega_i. \tag{2.9}$$

A useful decomposition of this equation is that into its *direct* and *indirect* components,

$$L(\vec{x},\vec{\omega}) = L_e(\vec{x},\vec{\omega}) + L_{direct}(\vec{x},\vec{\omega}) + L_{indirect}(\vec{x},\vec{\omega}). \tag{2.10}$$

L_{direct} is the reflection of radiance arriving at a surface directly from light sources. $L_{indirect}$ is the corresponding reflection of radiance that has undergone at least one reflection before. With the total radiance reflected by a surface denoted $L_r = L_{direct} + L_{indirect}$, the two terms are

$$L_{direct}(\vec{x},\vec{\omega}) = \int_{\Omega_i} f(\vec{x},\vec{\omega}_i,\vec{\omega})\, L_e(\vec{r}(\vec{x},\vec{\omega}_i), -\vec{\omega}_i)\, |\vec{n}\cdot\vec{\omega}_i|\, d\Omega_i, \tag{2.11}$$

$$L_{indirect}(\vec{x},\vec{\omega}) = \int_{\Omega_i} f(\vec{x},\vec{\omega}_i,\vec{\omega})\, L_r(\vec{r}(\vec{x},\vec{\omega}_i), -\vec{\omega}_i)\, |\vec{n}\cdot\vec{\omega}_i|\, d\Omega_i. \tag{2.12}$$

Equation 2.11 cannot directly be applied to point light sources because their emitted radiance L_e is undefined. With a point light, the BRDF needs to be evaluated for the single incident direction $\vec{\omega}_i = \widehat{\vec{y}-\vec{x}}$ and may be taken out of the integral. The remaining integral is equation 2.6, the irradiance due to a light source. For point lights, this is given in equation 2.7. A point light source not occluded by another surface thus contributes the direct illumination

$$\begin{aligned} L_{direct}(\vec{x},\vec{\omega}) &= f(\vec{x},\widehat{\vec{y}-\vec{x}},\vec{\omega}) E_{\vec{y}}(\vec{x}) \\ &= f(\vec{x},\widehat{\vec{y}-\vec{x}},\vec{\omega}) I(\widehat{\vec{x}-\vec{y}}) \frac{|\vec{n}\cdot(\widehat{\vec{x}-\vec{y}})|}{\|\vec{x}-\vec{y}\|^2}. \end{aligned} \tag{2.13}$$

When multiple point light sources are present, equation 2.13 must be evaluated for each. If all sources are point lights, the surface emission term in equation 2.10 is zero and can be omitted. The equation for indirect illumination is unaffected by the choice of light source type.

2.1.3. Reflection

Light reflection on a surface is modeled by its BRDF. This function relates the differential *outgoing radiance* L_o in a direction $\vec{\omega}$ to the differential *incident irradiance* from a direction $\vec{\omega}_i$,

$$f(\vec{x},\vec{\omega}_i,\vec{\omega}) = \frac{dL_o(\vec{x},\vec{\omega})}{dE(\vec{x},\vec{\omega}_i)}.$$

A BRDF has unit sr^{-1}. By integrating it over the hemisphere of outgoing directions Ω, the dimensionless *reflectivity* ρ is obtained,

$$\rho\left(\vec{x}, \vec{\omega}_i\right) = \int_{\Omega} f\left(\vec{x}, \vec{\omega}_i, \vec{\omega}\right) \, d\Omega.$$

Conservation of energy dictates that reflectivity cannot exceed one. Thus, for any $\vec{\omega}_i \in \Omega_i$,

$$\rho\left(\vec{x}, \vec{\omega}_i\right) \leq 1.$$

A second property is Helmholtz-reciprocity [vH67] which mandates that

$$f\left(\vec{x}, \vec{\omega}_i, \vec{\omega}\right) = f\left(\vec{x}, \vec{\omega}, \vec{\omega}_i\right).$$

2.1.3.1. BRDFs

The simplest surface model is the *Lambertian* or *diffuse* reflector which redirects all irradiance uniformly back into the hemisphere Ω. It has reflectivity $\rho_d = k_d$ and a constant BRDF

$$f_d\left(\vec{x}, \vec{\omega}_i, \vec{\omega}\right) = k_d \frac{1}{\pi}. \tag{2.14}$$

At the other end of the spectrum lies the perfect *specular* reflector which redirects incident radiance into the mirror reflection of $\vec{\omega}_i$ about the surface normal $\vec{n}$ only. Its reflectivity is $\rho_s = k_s$. Using a spherical coordinate system with *zenith* direction $\vec{n}$, *inclination* θ relative to $\vec{n}$ and *azimuth* φ around it, the BRDF can be written using Dirac δ functions as

$$f_s\left(\vec{x}, \vec{\omega}_i, \vec{\omega}\right) = 2k_s\delta\left(\sin^2\theta - \sin^2\theta_i\right)\delta\left(\varphi - \varphi_i \pm \pi\right). \tag{2.15}$$

For glossy reflection around the perfect mirror direction, the Phong model [Pho75] is frequently used. While its original definition violates physical principles, a modified variant [LW94] ensures energy conversation and Helmholtz reciprocity. Sharpness is controlled by an exponent $n \geq 0$ with $n = 0$ yielding diffuse reflection and $n \to \infty$, perfect specularity. Reflectivity depends on the incident angle but is guaranteed to be $\rho_g\left(\vec{\omega}_i\right) \leq k_g$. The reflected direction (α, φ) is expressed in a spherical coordinate system whose zenith is the perfect mirror direction. With Ω the hemisphere of possible outgoing directions, the BRDF is

$$f_g\left(\vec{x}, \vec{\omega}_i, \vec{\omega}\right) = \begin{cases} k_g \dfrac{n+2}{2\pi} \cos^n\alpha & \text{if } \vec{\omega} \in \Omega, \\ 0 & \text{else.} \end{cases} \tag{2.16}$$

2.1.3.2. Generalizations

In analogy to the BRDF, a translucent surface is characterized by its *bidirectional transmittance distribution function (BTDF)*. Conservation of energy and Helmholtz-reciprocity hold for BTDFs in similar form [Vea96]. Using BTDFs, *refraction* can be modeled, the redirection of radiance resulting from a change in *refractive index* η. A surface with both reflective and transmissive characteristics has a BRDF and a BTDF pair for light incident on one side and again for the other. These four functions may be combined into a single *bidirectional scattering distribution function (BSDF)*.

The *bidirectional scattering-surface reflectance distribution function (BSSRDF)* [NRH77] is another generalization. Instead of assuming radiance is reflected at the same point at which it arrived, the BSSRDF $f\left(\vec{x}_i, \vec{x}, \vec{\omega}_i, \vec{\omega}\right)$ establishes a relationship between arbitrary pairs of points on a surface. This allows subsurface scattering to be described but significantly increases model complexity.

2.1.3.3. Light Paths

When comparing physically based rendering algorithms, a classification of the paths from light source to virtual camera is useful. A popular notation [Hec90] uses L for a light source, E for the virtual camera or eye, S for specular and D for diffuse interaction. Paths are described by simple regular expressions, such as LDE for a single diffuse interaction, LS^+DE for one or more specular interactions followed by one diffuse (giving rise to caustics) or $L\left(S|D\right)^* E$ for all possible paths. When discussing further phenomena, additional letters may be used.

2.1.4. Participating Media

Equation 2.3, the invariance of radiance along straight lines, only holds for a vacuum. When particles are suspended in the atmosphere, the radiance transfer from a point $\vec{y}$ to a point $\vec{x}$ is subject to further phenomena [KvH84]. To describe these, the path from $\vec{y}$ to $\vec{x}$ is first parameterized using a direction vector $\vec{\omega} = \widehat{\vec{x} - \vec{y}}$ and a parameter $0 \leq t \leq t_{max} = \|\vec{x} - \vec{y}\|$ as

$$\vec{z}\left(t\right) = \vec{y} + t\vec{\omega}.$$

Absorption A photon on the path from $\vec{y}$ to $\vec{x}$ may encounter a particle and be *absorbed* by it, ceasing to exist. The fraction of photons absorbed per unit distance traveled is given by the *absorption coefficient* σ_a with unit m^{-1}. This leads to a differential change in radiance of

$$\left(\frac{dL}{dt}\left(\vec{z}\left(t\right), \vec{\omega}\right)\right)_{absorption} = -\sigma_a\left(\vec{z}\left(t\right)\right) L\left(\vec{z}\left(t\right), \vec{\omega}\right).$$

Out-scattering When it encounters a particle, a photon may not be absorbed but *scattered* instead, continuing in another direction. The fraction of photons scattered per unit distance traveled is given by the *scattering coefficient* σ_s with unit m^{-1}, yielding the differential change in radiance

$$\left(\frac{dL}{dt}\left(\vec{z}\left(t\right), \vec{\omega}\right)\right)_{outscatter} = -\sigma_s\left(\vec{z}\left(t\right)\right) L\left(\vec{z}\left(t\right), \vec{\omega}\right).$$

In-scattering Photons scattered out of other directions may be redirected onto the path from $\vec{y}$ to $\vec{x}$. The directional distribution of scattered photons is given by a *phase function* p with unit sr^{-1}. Accumulating redirected photons from the entire sphere of directions $\Omega_{4\pi}$, the differential change in radiance caused by in-scattering is

$$\left(\frac{dL}{dt}\left(\vec{z}\left(t\right), \vec{\omega}\right)\right)_{inscatter} = \int_{\Omega_{4\pi}} p\left(\vec{z}\left(t\right), \vec{\omega}_i, \vec{\omega}\right) \sigma_s\left(\vec{z}\left(t\right)\right) L\left(\vec{z}\left(t\right), \vec{\omega}_i\right) d\Omega_i.$$

Emission The final phenomenon is the *emission* of new photons onto the path from $\vec{y}$ to $\vec{x}$ by the atmosphere. This is modeled by an *emittance function* ϵ with unit $W\, m^{-3}\, sr^{-1}$ that leads to a

differential change in radiance of

$$\left(\frac{dL}{dt}\left(\vec{z}\left(t\right),\vec{\omega}\right)\right)_{emission} = \epsilon\left(\vec{z}\left(t\right),\vec{\omega}\right).$$

2.1.4.1. Volume Rendering Equation

The total differential change in radiance is the sum of these four effects. To simplify notation, properties of the atmosphere or *participating medium* dependent on $\vec{z}(t)$ are parameterized using t only. Furthermore, the *extinction coefficient* $\sigma_e\left(t\right) = \sigma_a\left(t\right) + \sigma_s\left(t\right)$ subsumes out-scattering and absorption, resulting in

$$\frac{dL}{dt}\left(\vec{z}\left(t\right),\vec{\omega}\right) = -\sigma_e\left(t\right)L\left(\vec{z}\left(t\right),\vec{\omega}\right) + \sigma_s\left(t\right)\int_{\Omega_{4\pi}} p\left(t,\vec{\omega}_i,\vec{\omega}\right)L\left(\vec{z}\left(t\right),\vec{\omega}_i\right)\,d\Omega_i + \epsilon\left(t,\vec{\omega}\right). \tag{2.17}$$

With the boundary condition $L(\vec{y},\vec{\omega}) = L_o(\vec{y},\vec{\omega})$ and $\tau\left(a,b\right) = e^{-\int_a^b \sigma_e(t)\,dt}$ the *transmittance* between two points in the medium, the solution to this differential equation is

$$\begin{aligned} L(\vec{x},\vec{\omega}) &= \tau\left(0,t_{max}\right)L_o(\vec{y},\vec{\omega}) \\ &\quad + \int_0^{t_{max}} \tau\left(0,t\right)\left(\sigma_s\left(t\right)\int_{\Omega_{4\pi}} p\left(t,\vec{\omega}_i,\vec{\omega}\right)L\left(\vec{z}\left(t\right),\vec{\omega}_i\right)\,d\Omega_i + \epsilon\left(t,\vec{\omega}\right)\right)\,dt. \end{aligned} \tag{2.18}$$

This equation models all effects on the radiance emanating from $\vec{y}$ toward $\vec{x}$, capturing attenuation due to absorption and out-scattering but also intensification due to in-scattering and emission by the medium itself. The resulting radiance incident at $\vec{x}$ from the direction of $\vec{y}$ is

$$L_i(\vec{x},-\vec{\omega}) = L(\vec{x},\vec{\omega}). \tag{2.19}$$

Equations 2.18 and 2.19 describe the relationship between radiance L_o emanating at surfaces, radiance L_i incident at surfaces and field radiance L at any point in the medium. Inserting these instead of equation 2.3 into the rendering equation 2.8 provides the recursive formulation of a steady state radiance equilibrium in a scene containing a participating medium. This is what a rendering algorithm should solve.

2.1.4.2. Simplifications

In order to reduce the complexity of the problem, simplifications may be applied. If the medium does not emit radiance, terms accounting for ϵ drop out. When the medium is made of a single substance with spatially varying density, the phase function is independent of position, the scattering, absorption, extinction coefficients and the emission function are proportional to the scalar medium density $\rho\left(\vec{x}\right)$ and transmittance reduces to $\tau\left(a,b\right) = e^{-\sigma_e\int_a^b \rho(t)\,dt}$. A *homogeneous* medium is a further simplification with constant density $\rho = 1$ and $\tau\left(a,b\right) = e^{-\sigma_e(b-a)}$.

2.1.4.3. Phase Functions

The phase function relates outgoing radiance in a direction $\vec{\omega}$ to that incident from a direction $\vec{\omega}_i$. Like BRDFs, phase functions are Helmholtz-reciprocal. To obey the conservation of energy, a phase function must integrate to one [BLS93]. The fraction of incident radiance redistributed into other

directions depends not on the phase function but on the volume *albedo*,

$$\alpha = \frac{\sigma_s}{\sigma_a + \sigma_s}. \tag{2.20}$$

Phase functions are typically symmetric around the incident direction and parameterized by the cosine of the *phase angle*, $\cos\theta = \vec{\omega} \cdot \vec{\omega}_i$, only. The simplest phase function represents *isotropic* scattering [Bli82], redistributing incident radiance evenly into the hemisphere of directions via

$$p(\cos\theta) = \frac{1}{4\pi}. \tag{2.21}$$

More complex empirically motivated phase functions are efficiently approximated by the Schlick function [BLS93]. Its parameter $k \in (-1, 1)$ controls *anisotropy*. A setting of $k = 0$ produces isotropic scattering. Decreasing k results in progressively stronger backward scattering, increasing it yields forward scattering. The phase function is

$$p(\cos\theta) = \frac{1 - k^2}{4\pi (1 - k\cos\theta)^2}. \tag{2.22}$$

An effect not accounted for in the current model is *elastic scattering*, radiance redistribution not only into different directions but also to other wavelengths. Simulating this in a physically based rendering algorithm is possible but requires the *radiative transfer equation* 2.17 to be replaced with the more complex and computationally expensive *full radiative transfer equation* [GMAS05].

2.2. Manycore Computing

Simulating light transport is computationally expensive. While Moore's law [Moo65] predicts that available processing power will grow exponentially, such gains are no longer transparently achieved. As the returns from added instruction-level parallelism diminish and operating frequencies approach physical limits, increasingly more parallel processing is needed, leading to *manycore* computing and the need for highly scalable algorithms.

2.2.1. CPU-GPU Convergence

Modern CPUs employ *data parallelism*. Using the *single instruction multiple data (SIMD)* [Fly72] processing model, a single operation is simultaneously performed on several data items. The popular x86 architecture has a SIMD *width* of 4 [RCC+06], one instruction affecting up to four data items. *Multicore* CPUs gain further parallelism from independent processing cores working in parallel. After early dual core designs [TDJ+02], CPUs featuring eight cores are now widely available [Int09].

GPUs have gained programmability with the addition of *shader* support [LKM01] to their already highly parallel processing units. Despite initially limited flexibility and access through graphics APIs only, *general purpose computation on GPUs (GPGPU)* has been applied to a wide range of problems, two examples being physically based rendering [PBMH02, PDC+03] and image registration [Fab06]. Support for the automatic translation of general purpose code into graphics API calls and shaders has aided such efforts by hiding some of the complexity [BFH+04].

A step is being taken now in which CPUs and GPUs converge on *manycore* computing with very large numbers of simple processing cores operating in parallel.

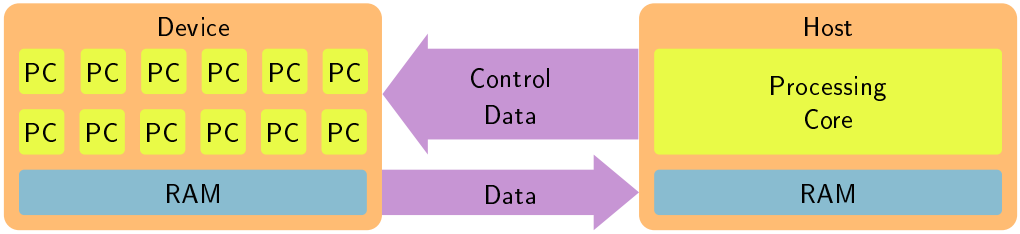


Figure 2.1.: High level view of CUDA: The host initiates memory transfers and invokes kernels. These execute inside the device on an array of highly parallel processing cores.

The *Cell Broadband Engine Architecture* [IST07] features 8 simplified parallel processing cores with 4-wide SIMD, thus operating on 32 data items in parallel. A traditional CPU core handles overall control flow and work distribution. *Intel Larrabee* [SCS+08] uses 10 simplified cores with 16-wide SIMD. Highlighting the convergence, Larrabee employs CPU cores but acts as a GPU, incorporating GPU features such as *texturing units*. While no longer pursued as a stand-alone architecture, the first products incorporating this technology and more than 50 cores have been announced [Int10].

GPU evolution has led to the *NVIDIA CUDA* [NVI10b] platform with up to 512 processing cores in the current generation. Groups of 32 cores run in *lock-step*, executing the same instruction for parallel threads. Termed *single instruction multiple thread (SIMT)*, this is akin to 16 cores with 32-wide SIMD. Competing *AMD Stream* [Adv09] features a current maximum of 1600 cores arranged in a 25 core 64-wide SIMT layout. Implemented on GPUs, both platforms provide access to texturing units and require a CPU for control flow and work distribution. The emergence of the vendor independent *OpenCL* API [Mun10] highlights their similarities.

At the time of this writing, the fastest consumer GPU chips are found in the NVIDIA GTX 480 with 480 processing cores, 1.35 TFLOP/s theoretical peak processing power and 177.4 GB/s peak main memory bandwidth and the AMD Radeon 5870 with 1600 processing cores, 2.72 TFLOP/s and 153.6 GB/s memory bandwidth. Linking multiple GPU chips adds another level of parallelism, providing additional performance.

The GPU used throughout this thesis is an NVIDIA GTX 280 with 240 processing cores executing instructions in a 30 core, 32-wide SIMT layout every four cycles, 933 GFLOP/s and 141.7 GB/s.

2.2.2. CUDA Programming Model

CUDA is the first manycore platform to reach wide availability. It illustrates the concept of manycore computing but also the compromises required in an actual implementation. The CUDA programming model is an abstraction that maps to commodity NVIDIA GPU hardware. Figure 2.1 shows a high level view of the model. For every computation, a large number of threads are launched, each executing an identical *kernel*. Conceptually, all threads operate in parallel with very limited synchronization and communication. This data parallel view of the GPU as a general purpose *stream processing* device efficiently scales across several generations of hardware. Threads are executed on the physically available cores in any order until all have terminated and the kernel launch is completed.

Kernels are invoked by a *host* computer. Once launched, all threads run to completion, accessing only resources resident on the GPU *device*. Threads cannot allocate further resources, spawn new threads or otherwise influence each other except through the use of predefined synchronization primitives. All responsibility for memory allocation, memory transfer and kernel invocation lies with the

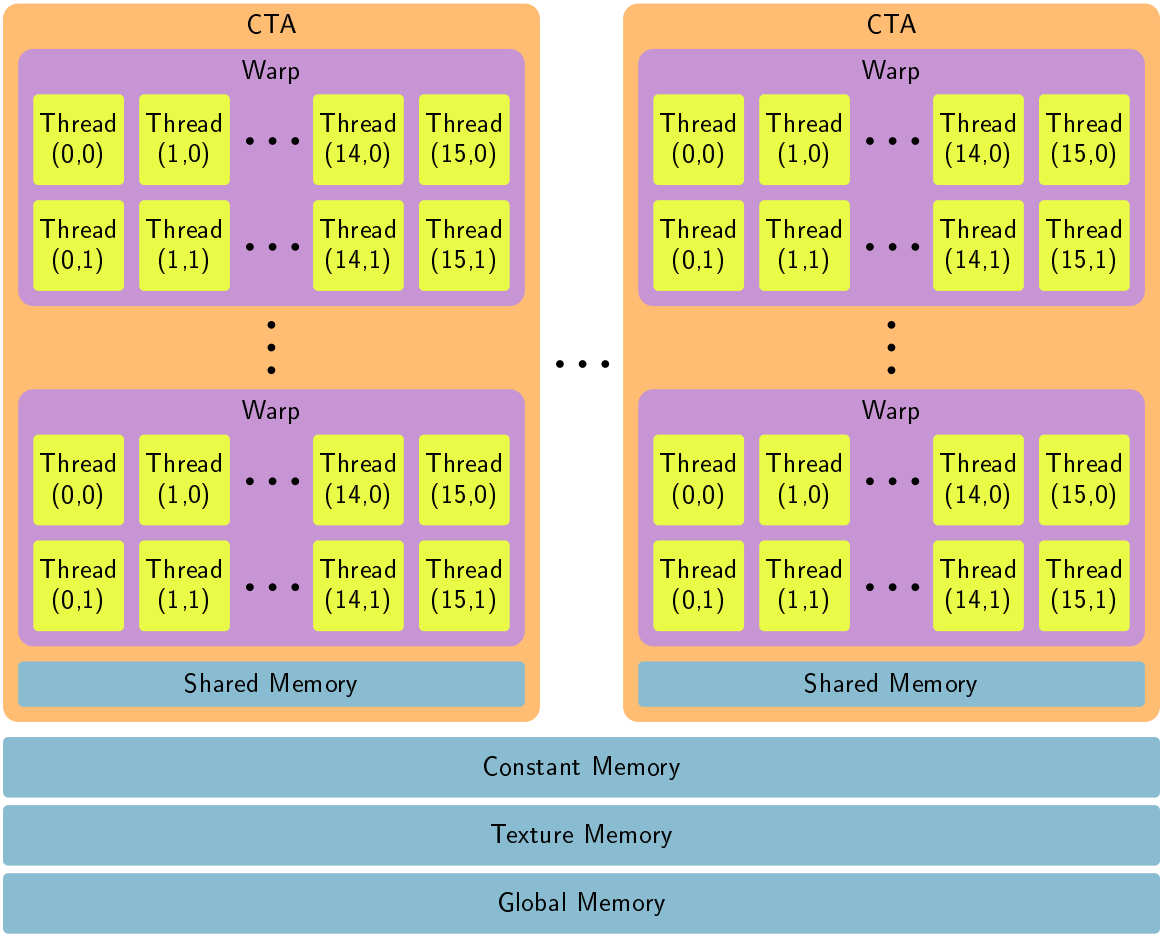


Figure 2.2.: CUDA thread and memory hierarchy: Warps of 32 threads each execute in lock-step, cooperative thread arrays (CTAs) of up to 512 threads cooperate via shared memory and synchronization primitives. All threads can access constant, texture and global memory.

host. A CUDA application consists of a host program and kernels spawned by it on the device.

CUDA kernels are written in a dialect of C++ with minor extensions to accommodate device memory and thread layouts. The `nvcc` utility decomposes source code into host and device components, passes these to separate compilers and links the results into a single binary.

2.2.2.1. Thread and Memory Hierarchy

To facilitate efficient implementation on actual hardware, threads and memory are hierarchically organized in CUDA. Threads further from each other in the hierarchy have looser synchronization and fewer cooperation options. Memory higher in the hierarchy is larger but has higher latency and lower bandwidth. The thread and memory layouts are interlinked as illustrated in figure 2.2.

Warps of 32 threads each form the first hierarchy level. These threads run in lock-step on neighboring processing cores, simultaneously executing the same instruction. When threads within a warp take different code paths, a *divergent branch* results. Execution is serialized by running both branches one after the other with inactive threads masked off. After the divergent section has completed, the entire warp continues in lock-step. Terminated threads are similarly masked off until an entire warp has finished. Warps do not communicate directly but can invoke a *voting* instruction to determine

whether a predicate is fulfilled by *all* or *any* of the 32 threads.

The next hierarchy level is the *cooperative thread array (CTA)* of up to 512 threads. Warps belonging to a single CTA are scheduled onto the same processing cores in a nondeterministic order. Their locality on the hardware allows for inexpensive communication and synchronization. A CTA has access to a small private pool of fast *shared memory*. Values written by one thread can be read by all others. Since warps are scheduled in nondeterministic order, information exchange between them requires the use of a *CTA synchronization* instruction. All warps in the CTA execute until this instruction, memory is synchronized and execution continues in nondeterministic order.

At the top level of the hierarchy, a kernel invocation may contain any number of CTAs. Among these, no guarantees of execution order or synchronization exist. Threads from all CTAs can access constant, global and texture memory. *Constant memory* can be written to by the host only and provides cached access to a small number of run-time constants. *Global memory* is the GPU main memory. It is the largest memory space with read-write access but without caching. By accessing parts of it as read-only *texture memory*, the caches present in GPU texturing units can be used.

For convenience, CTAs and the threads within them may be addressed as one-, two- or three-dimensional arrays.

2.2.2.2. Shared Resources

Processing power increases with the number of available cores. Memory performance does not scale in the same way and is therefore an important bottleneck. With the limited caches available, the core mechanism for latency hiding in CUDA is *hardware multi-threading*. Processing cores are equipped with large register banks that can hold the state of several warps at once. When a warp stalls on a memory access or synchronization instruction, execution seamlessly continues with another warp. To enable this mechanism, the number of registers and the amount of shared memory required by each CTA should be kept low. Significantly more warps should also be launched than can execute at one time, allowing each core to keep a high *occupancy* of resident warps.

There are no preemption or memory protection mechanisms. The programmer must ensure threads do not deadlock by waiting for each other or overwrite important results. *Atomic* instructions may be used to safely implement global job queues and memory pools but by enforcing serialized access, limit scalability. As texture memory is an alias for a subset of the global memory space, textures can be modified by writing to the corresponding global memory locations. However, texture caches are not synchronized. Only when a kernel has completed and control returns to the host is it ensured that all memory writes become visible. For complex algorithms, this requires a *multi-pass* approach where global synchronization is achieved by launching consecutive kernels.

Resources may be shared between CUDA and a graphics API [SA10]. This allows results to be visualized without having to copy them to host memory first. Since all other screen updates stall while CUDA is occupying the GPU, a *watchdog timer* permits an execution time of at most five seconds per kernel for a GPU shared by CUDA with the desktop environment.

2.2.3. CUDA Hardware Implementation

As of this writing, three generations of CUDA hardware exist. The following discussion focuses on the GTX 280 used in the thesis and its second generation GT200 chip. Advances in the third generation are addressed in section 2.2.3.3.

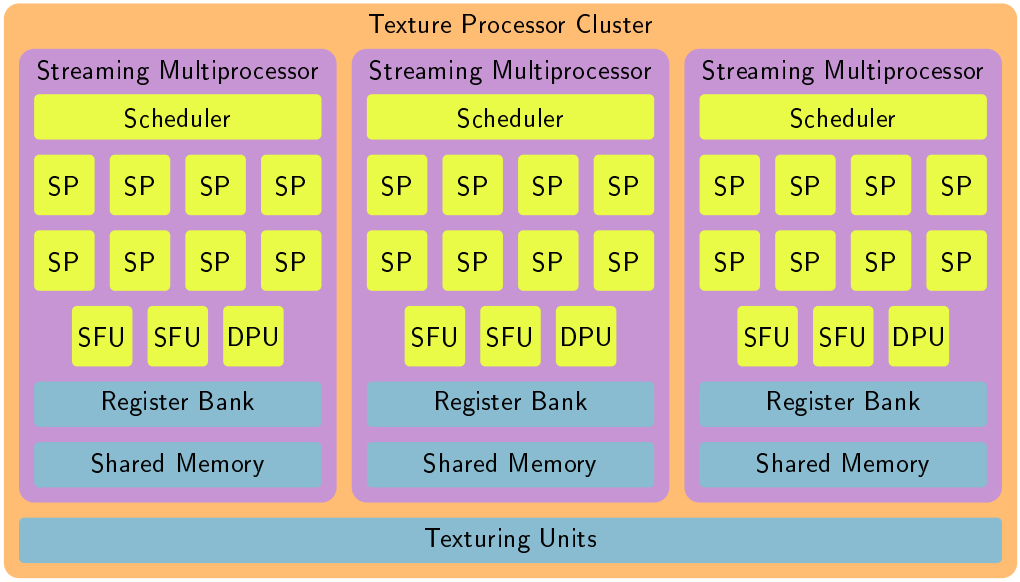


Figure 2.3.: CUDA texture processor cluster on the GT200 chip: A streaming multiprocessor (SM) comprising execution units, scheduler, register banks and shared memory executes warps in lock-step. Three SMs sharing texturing units form a thread processor cluster.

Figure 2.3 provides an overview of the functional unit arrangement on a GT200 chip. Eight general purpose processing cores, the *streaming processors (SPs)*, form a *streaming multiprocessor (SM)*. They share an instruction scheduler and run in lock-step, executing one instruction for a warp of 32 threads every four cycles. Two *special function units (SFUs)* compute transcendentals and divisions, a *double precision unit (DPU)* handles double precision numbers. Complex instructions using the SFUs or DFU or taking more cycles to complete have lower throughput than simple arithmetic. Texturing units and their caches are shared by *thread processor clusters (TPCs)* of three SMs each. A GTX 280 has 10 TPCs, 30 SMs and 240 SPs.

2.2.3.1. Memory

Each SM distributes 16,384 32-bit registers and 16 kB of shared memory among the warps resident on it. A maximum occupancy of 32 resident warps is reached when resource usage does not exceed 16 registers and 16 bytes of shared memory per thread. Higher requirements reduce occupancy, lowering the effectiveness of latency hiding by hardware multi-threading. A single thread can use at most 128 registers, an entire CTA may occupy all 16,384 registers and 16 kB of shared memory.

Shared memory has latency and bandwidth comparable to the register banks, allowing very efficient exchange of information between the threads of a CTA. It is physically composed of 16 *banks*. Maximum throughput is achieved if the addresses simultaneously requested lie in different banks. A *bank conflict* requires serialization of accesses, increasing latency and reducing bandwidth.

Global memory has a size of 1 GB on the GTX 280. It is located off-chip, has significantly higher latency and lower bandwidth. Main memory accesses are identified as the most important bottleneck by the CUDA documentation [NVI10a]. When several threads in a 16-thread *half-warp* address locations that can be serviced by a single 32, 64 or 128 byte transaction, their requests are *coalesced*, improving latency and bandwidth. Despite not being cached, main memory has variable latency. This is explained by the presence of *translation lookaside buffers (TLBs)*, revealed through micro-

benchmarking [VD08, WPSAM10]. Memory working sets of up to 8 MB are covered by an L1 TLB, incurring a latency of 436 to 443 cycles on hit. L2 TLB hits yield latencies of 487 cycles for working sets of up to 32 MB. L2 TLB misses result in latencies of 698 cycles.

Read-only accesses to global memory can be made through the GPU texturing units which provide caching. Details are again revealed by micro-benchmarking. The cache sizes are 5 kB L1 and 256 kB L2. An L1 hit has a latency of 261 cycles, an L2 hit 371 cycles. Cache misses require global memory reads. TLBs with working set sizes of 8 MB and 16 MB are used, producing latencies of 497 cycles on L1 TLB hit, 544 on L2 TLB hit and 753 on TLB miss. Besides caching, the texturing units provide optional *linear interpolation* of neighboring values and *normalization* to a range of $(-1, 1)$ or $(0, 1)$. Disadvantages of the texturing include smaller TLB working sets, high latencies for cache misses and the lack of coalescing. Texturing units are configured by the host. Each texture may be one-, two- or three-dimensional with elements consisting of either one, two or four 32-bit values.

A 64 kB constant memory space is provided through which the host can pass run-time parameters to kernels. This has three cache levels with sizes 2 kB, 8 kB and 32 kB and latencies of 8, 81 and 220 for hits and 476 for a miss. Instructions are cached in a similar hierarchy, sharing L2 and L3 with the constants. L1 caches are located in the SMs, L2 in the TPCs and the L3 cache is global.

Local memory is used for automatic variables that the compiler is unable to place in registers. This occurs for large structures and arrays addressed by an index. Local memory aliases a region of global memory and thus carries its access characteristics.

All memory management occurs on the host. Shared memory is allocated at compile or run-time. Texturing units must be reserved during compilation but texture sizes and parameters are specified at run-time. Independent memory transfers and kernel executions may be placed in different CUDA *streams*. Each stream is processed in order but the kernel execution from one and a memory transfer from another may overlap if non-pageable *pinned* memory is used on the host.

2.2.3.2. Instructions

CUDA device code is written in a dialect of C++ lacking some advanced features such as virtual functions or multiple inheritance. A kernel is expressed as a single, serial control flow. Parallelization automatically occurs when the same kernel is simultaneously executed by a large number of threads. No call stack exists on the device. All function calls are inlined and recursion is not permitted. The thread hierarchy is specified when invoking the kernel from the host.

The threads of a warp cooperate via voting instructions that determine whether a given predicate holds for *any* or *all* of them. Cooperation on the CTA level occurs through shared memory. A synchronization instruction lets all warps advance to the same point in the code, ensuring prior memory writes become mutually visible. Memory fence instructions can extend this visibility assurance across all CTAs but do not imply any synchronization. To maximize performance, lock-step execution in warps should be taken into account, avoiding divergent branches, uncoalesced global memory accesses and shared memory bank conflicts. Each thread can determine its position in the thread hierarchy by querying the `threadIdx` and `blockIdx` variables.

Variables are explicitly placed in the shared, global and constant memory spaces at compile time. They may be accessed directly in a kernel with the compiler generating appropriate instructions. For atomic operations and texture memory reads, library functions are provided. Convenience packed data types group two, three or four integers or floating point values. Aliasing instructions allow a 32-bit floating point number to be stored in a 32-bit integer variable and vice versa.

Floating point support follows the IEEE 754 standard [IEE08] with minor deviations. Signaling NaNs, dynamically configurable rounding modes and denormalized numbers are not supported. Simple arithmetic instructions execute on the SPs. For maximal throughput, the compiler combines multiplications and additions into *multiply-add* instructions. Complex operations may expand into long sequences of instructions and require the SFUs.

Intrinsics provide faster but lower precision variants of transcendentals and division. A reciprocal is faster than a general division. Square root is implemented as the reciprocal of an inverse square root, the latter of which may directly be accessed with an intrinsic. SPs are optimized for 32-bit floating point arithmetic with 24-bit mantissa. Multiplication of two 32-bit integers therefore requires several operations and an intrinsic provides faster 24-bit multiplication. Calculations in double precision are performed by the DPU, providing only $\frac{1}{8}$ the throughput of single precision.

The compiler is able to optimize short loops by unrolling and short sections of conditional code by *predication*, issuing instructions that are always executed but masked off in inactive threads.

2.2.3.3. Extensions

A *compute capability* summarizes the GPU feature set. First generation G80 chips have capability 1.0 or 1.1, second generation GT200 1.2 or 1.3 and third generation GF100, 2.0. The description above is based on a GTX 280 GPU with compute capability 1.3.

Capability 2.0 extends the hardware and programming models. *Graphics processor clusters (GPCs)* of four SMs each replace the TPCs. Every SM has two schedulers, 32 processing cores and four SFUs. Simple integer and single precision instructions are emitted by both schedulers simultaneously for two different warps. Instead of a designated DFU, double precision instructions are executed by pairs of standard cores, increasing double precision throughput to $\frac{1}{2}$ that of single precision. Fast 32-bit integer multiplication and denormalized floating point numbers are supported.

The maximal CTA size is increased to 1024 threads. Each SM has $32,768$ registers and 64 kB fast memory organized in 32 banks. If their resource requirements permit, up to 48 warps from *one or more kernels* can be resident on an SM. This allows different kernel executions to overlap. Memory transfers between host and device in both directions can also happen simultaneously. SMs can be configured to provide either 16 kB or 48 kB of shared memory. The remaining fast memory serves as a *global memory cache*. Global memory accesses are further coalesced across entire warps.

Instruction support is also extended. Call stacks exist, enabling function calls and C++ classes. The synchronization instruction can evaluate a predicate across all threads in a CTA, reporting whether it holds for any or all of them. Warp voting is enhanced to return individual results for the 32 threads of a warp in a bitmask. A `printf` utility function aids debugging.

2.2.3.4. Challenges

CUDA provides unprecedented potential processing power but its efficient use poses a number of challenges. An important aspect is a general opaqueness of the platform. Hardware details are left undocumented and only partially uncovered through reverse engineering. Source code is compiled to the open *PTX* intermediate format but then transformed into binary device code by an undocumented process. Bugs exist in this stage that cannot be worked around. CUDA is also an evolving platform. Not only bug fixes but new features and instructions are provided with each software release. Two APIs exist, the lower level *device* and higher level *runtime* API. Only in the most recent CUDA

revisions have these been made compatible. No CUDA version provides a linker for device code, always requiring full recompilation.

Published peak performance figures include calculations made by texture addressing units and cannot be fully reached in realistic scenarios. General optimization guidelines exist [NVI10a] but with hardware details undisclosed, benchmarking is required to choose between different thread layouts and alternative implementations. Some apparent optimizations may actually reduce performance. For example, the use of texturing units provides caching but precludes coalescing. A new hardware generation requires complete reevaluation of all optimizations.

Tool support is limited. The CUDA profiler only reports aggregate values for an entire TPC. A debugger exists in recent CUDA versions but cannot be used on systems with a single GPU shared by CUDA and the desktop environment. Despite a watchdog timer terminating kernels after five seconds, GPU crashes or deadlocks may render a system unresponsive. *Emulation mode* allows kernels to be cross-compiled into native host instructions but has been deprecated as it is of little help in debugging problems that occur on actual hardware. A GPU simulator that executes PTX code directly [CDP09] is currently limited to emulating the first generation G80 chip only.

Other challenges are inherent to the hardware platform. Nondeterministic scheduling order requires careful algorithm design and an implementation that avoids undefined reads or deadlocks. Complex algorithms must be split into consecutive kernels due to the lack of global synchronization. Texturing units have limited flexibility by operating on predefined data types only. The scheduler that assigns warps to SMs is optimized for uniform workloads. In other scenarios, it should be circumvented by using an explicit job queue [AL09].

An important consideration for all parallel architectures is Amdahl's law [Amd67]. With $0 \leq r_p \leq 1$ the parallelizable fraction of an algorithm, the speedup s achieved by using n processing cores is

$$s = \frac{1}{(1 - r_p) + \frac{r_p}{n}}.$$

Speedup thus crucially depends on the removal of parallelization bottlenecks, increasing r_p.

2.2.4. Algorithmic Building Blocks

Algorithm design for manycore architectures can be simplified and accelerated by employing reusable *building blocks* that efficiently parallelize common operations. Several important building blocks follow from the concept of a *parallel scan*.

2.2.4.1. Scan and Prescan

The *scan* or *inclusive prefix sum* transforms a vector $\vec{a}$ such that the i-th component is the combination of all a_j with $j \leq i$ via an associative binary operator $\oplus$. A *prescan* or *exclusive prefix sum* does the same for $j < i$. With I the identity element of operator $\oplus$,

$$\operatorname{scan}(\vec{a}, \oplus) = (\quad a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus \ldots \oplus a_{n-2}), (a_0 \oplus \ldots \oplus a_{n-1})), \tag{2.23}$$

$$\operatorname{prescan}(\vec{a}, \oplus) = (I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus \ldots \oplus a_{n-2})). \tag{2.24}$$

A trivial implementation that loops over the input in $O(n)$ time cannot be parallelized as each iteration requires previous results to be known. The alternative illustrated in figure 2.4 arranges

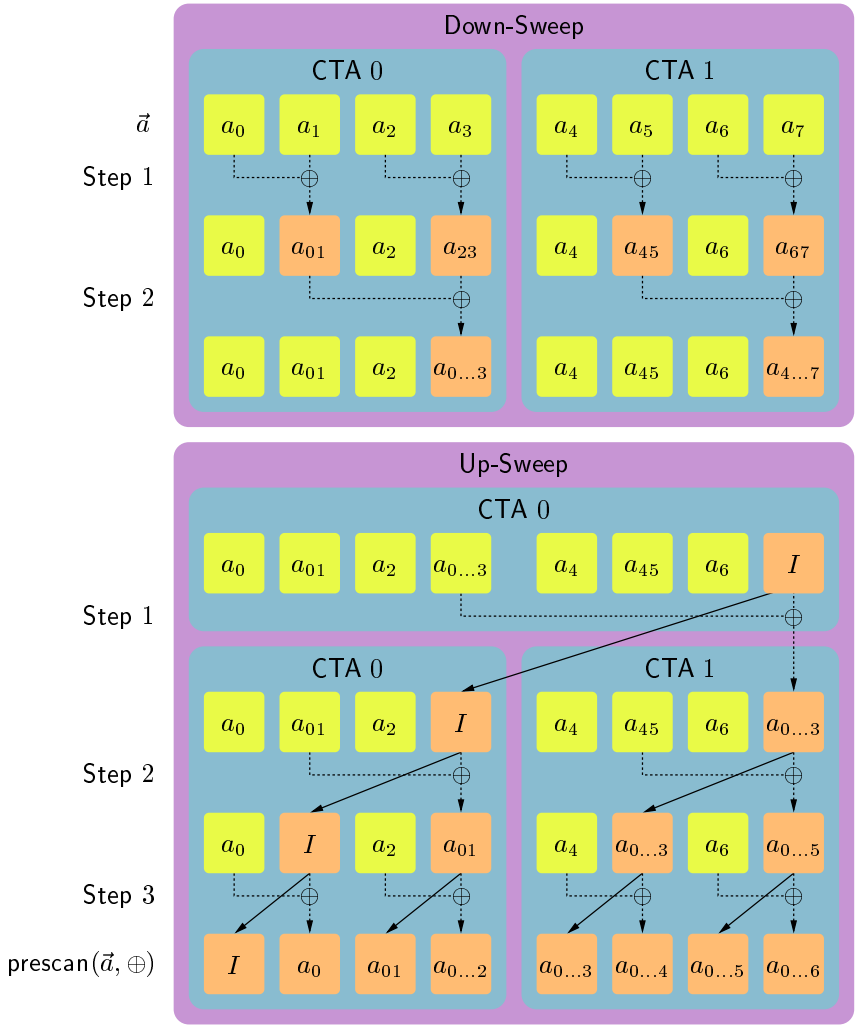
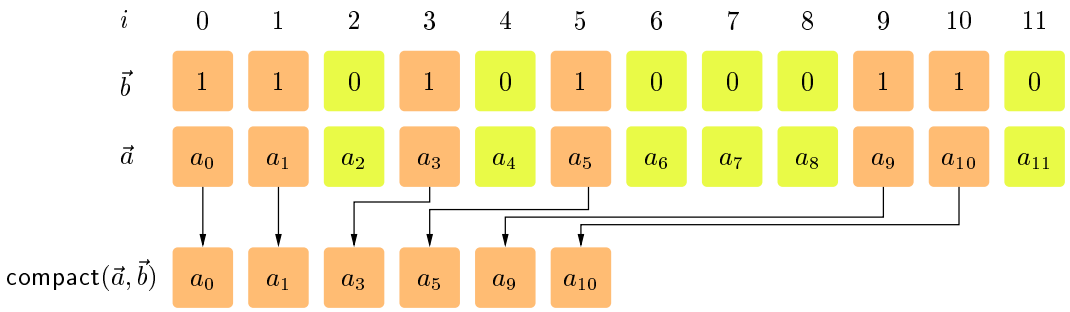


Figure 2.4.: Parallel prescan in CUDA: The computation is decomposed into $\log_2 n - 1$ down-sweep and $\log_2 n$ up-sweep steps, each fully parallelizable. By assigning subsets of operations to CTAs, they can be executed using shared memory and fast synchronization.

computations in $O(\log n)$ steps [Ble90]. A *down-sweep* phase performs a *logarithmic reduction*. Each of $\log_2 n - 1$ steps applies operator $\oplus$ half as many times as the previous step. The last component is replaced with identity I and an *up-sweep* phase follows. $\log_2 n$ steps are taken, doubling the number of copy operations and applications of $\oplus$ with each.

The operations within a step are independent and may run in parallel. Although total complexity remains $O(n)$, work can now be distributed across parallel processing units. At the end of each step, synchronization between the processing units is required to ensure all computations have completed and results are mutually visible. This global synchronization is expensive in CUDA as it occurs only when all threads terminate and a new kernel is launched by the host.

By distributing the work to CTAs as shown in figure 2.4, each can perform part of the down-sweep or up-sweep independently, benefiting from shared memory and fast CTA synchronization [SHZO07]. The amount of work per CTA is limited by its thread count and shared memory size. For large inputs, the same kernel is repeatedly executed with progressively more (up-sweep, pictured) or fewer (down-sweep) CTAs. Within a warp, 32 threads execute the same instruction in lock-step. By rearranging



(a) The components of $\vec{a}$ whose corresponding flags in $\vec{b}$ are set are copied to the output.

(b) A prescan of $\vec{b}$ using the addition operator yields the target indexes for all a_i.

Figure 2.5.: Parallel stream compaction: The index at which a component a_i should be stored in the output vector is computed by a parallel prescan of $\vec{b}$.

operations on the warp level to maximize the number of threads in which instructions do useful work, some redundant computations are introduced but total instruction count is reduced [SHG08].

As is evident by comparing equations 2.23 and 2.24, applying operator $\oplus$ one more time at the end results in a parallel scan. Extension to input sizes that are not powers of two is trivially possible by padding with the identity element I. Another extension is the *segmented scan*. Here, an array $\vec{b}$ of Boolean flags partitions $\vec{a}$ into segments and operator $\oplus$ is applied to each segment independently. Efficient implementations in CUDA follow a similar parallelization pattern [SHZO07, SHG08].

2.2.4.2. Compaction

Given a vector $\vec{b}$ of Boolean flags, *stream compaction* extracts from a vector $\vec{a}$ the components a_i for which $b_i = 1$, preserving their order (see figure 2.5(a)). This is an important operation for job queues, removing terminated jobs so that only active ones are scheduled. To determine the output position at which a_i should be stored, the number of preceding $b_j = 1$ in the input must be counted. As shown in figure 2.5(b), a prescan of $\vec{b}$ with the addition operator provides this information.

A parallel compaction building block is thus obtained by applying a parallel scan to $\vec{b}$ and then copying all a_i with $b_i = 1$ to their target positions in parallel. In CUDA, this can be further improved on [BOA09]. By counting the number of $b_i = 1$ for an entire input range in each warp and applying the prescan only to these totals, prescan input size is significantly reduced. During the copy phase, better coalescing is achieved when the threads of a warp read consecutive a_i and b_i in parallel, compact these into shared memory and then write the result to global memory in parallel again.

2.2.4.3. Split

The *split* operation takes input identical to that of compaction but returns a vector containing all components of $\vec{a}$, rearranged such that those with $b_i = 0$ are followed by those with $b_i = 1$, preserving original order in each group (see figure 2.6). Target indexes can again be computed using a parallel prescan($\vec{b}, +$). With n_i the number of preceding $b_j = 1$ and n' the total number of all $b_i = 1$, the target for a component with $b_i = 0$ is $i - n_i$ and that for a component with $b_j = 1$ is $n - n' + n_i$.

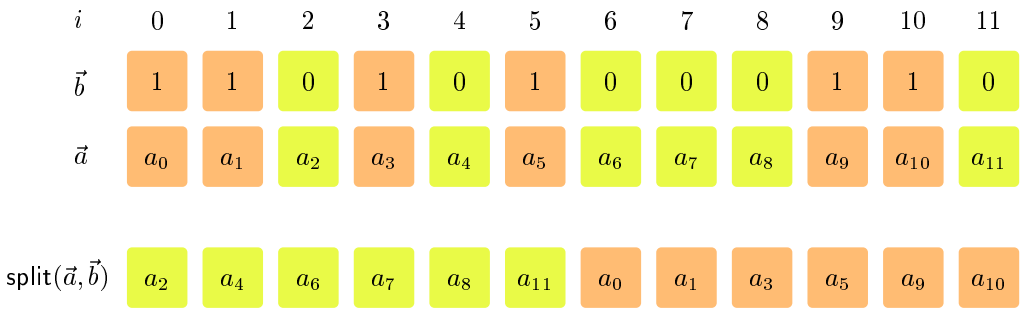


Figure 2.6.: Split: The components of $\vec{a}$ are rearranged in the output such that those with $b_i = 0$ are followed by those with $b_i = 1$, preserving original order in each group.

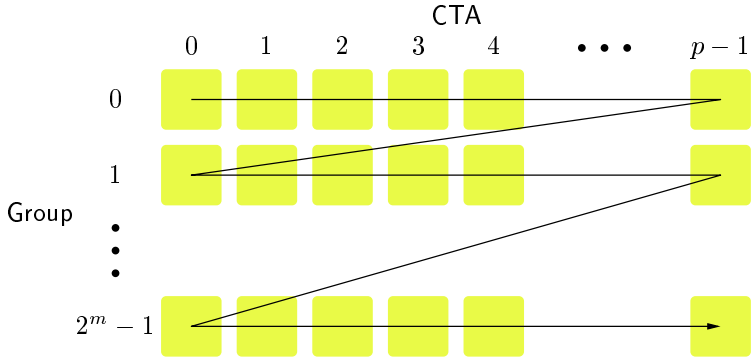


Figure 2.7.: Parallel prescan order for CUDA radix sort: Sizes are added across CTAs, then groups.

2.2.4.4. Radix Sort

A sequence of parallel splits can be used to efficiently sort integer numbers [Ble90]. The first split arranges numbers according to their least significant bit. The next split rearranges them according to their second-least significant bit, preserving the already established order within each group. When sorting k-bit numbers, a total of k splits are required.

The bottleneck of this algorithm is the bandwidth required to transfer numbers to and from global memory for every split. A more efficient implementation is again possible in CUDA [SHG09]. Utilizing fast shared memory, each CTA applies the split operation to a subset of the data. A parallel prescan adds the sizes of the groups produced by all CTAs, revealing where each must be placed in global memory to concatenate them. Efficiency further improves when each CTA performs m split operations in shared memory before concatenating results, producing 2^m groups. Figure 2.7 illustrates the logical order in which the group sizes are then processed by a prescan to ensure concatenation into 2^m groups equivalent to the result of applying m splits to the entire input.

Several libraries implementing these algorithmic building blocks exist [BOA10, HOS+10].

3. Related Work

Physically based rendering aims to calculate the radiance reaching a virtual camera by simulating light transport in a scene. What constitutes a successful rendering algorithm varies with application area. An offline technique may take hours to synthesize a single, high quality image. Interactive algorithms operate on significantly reduced time budgets, must be computationally efficient and might reproduce subsets of physical phenomena only. The following chapter provides a survey of related research with a focus on physically based rendering and its acceleration to interactive frame rates.

3.1. Ray Tracing

Equation 2.9 defines the equilibrium radiance distribution in a scene. Central to the equation is a helper function $r(\vec{x}, \vec{\omega})$ that locates the nearest surface seen from a point $\vec{x}$ in a direction $\vec{\omega}$. This is the *visibility problem*. Its efficient solution is crucial for fast physically based rendering. When a large number of directions are queried from a single point, *rasterization* is a viable solution. An example is the search for the surfaces visible at each screen pixel from a virtual camera. All surfaces are drawn into a screen-sized buffer, keeping track of the depth value z at each pixel. Only when a surface is nearer than that currently stored at the pixel is it drawn and the z value updated [Cat74].

A more flexible and general technique is *ray casting*. Whenever the function r is invoked, a ray equation with parameter $t \geq 0$ is constructed,

$$\vec{z}(t) = \vec{x} + t\vec{\omega}. \tag{3.1}$$

This ray can now be tested for intersection with all surfaces in the scene. Each intersection yields a *hit point* $\vec{z}_{hit} = \vec{z}(t_{hit})$. The nearest surface seen is that with the lowest t_{hit} value.

3.1.1. Recursion

First proposed as a method for solving the visibility problem from a virtual camera [App68], ray casting can also be used in the opposite direction, simulating light transport from its sources. A comprehensive simulation requires rays to be cast recursively as light repeatedly interacts with surfaces and the atmosphere. This extends ray casting to *ray tracing*, both a name for the general concept of recursive casting and a rendering algorithm in itself [Whi80].

The rays emitted by a virtual camera or light source are *primary rays*. These are highly coherent, often sharing a common origin and having similar directions. At interactions with surfaces or the atmosphere, *secondary rays* are spawned. When distinct surfaces or points are designated as light sources, direct illumination is computed by casting *shadow rays*. A shadow ray answers the binary query whether a light source is visible or occluded. Since neither the actual occluding surface nor the distance to it are of interest, an *early out* exists. Whenever *any* surface with $t_{hit} < t_{light}$ is located, the source is occluded and the search terminates. Secondary rays become less coherent with deeper

recursion as their origins and directions spread over the entire scene. Shadow rays toward a point light source are an exception as these have a common origin when followed from the light.

Due to the limited precision of floating point arithmetic, intersection calculations are not perfectly accurate. Small offsets ϵ are used throughout ray tracing to account for this imprecision. Shadow rays, for example, only report a light source as occluded when $t_{hit} + \epsilon < t_{light}$.

3.1.2. Footprints

Recursion extends the concept of an isolated ray to include successors after an interaction. Another conceptual extension is the consideration of neighboring rays forming a *wavefront*. By keeping track of their convergence or divergence, a *footprint* is obtained for each ray, corresponding to the wavefront patch it represents. For rays traced from a light source, this allows flux to be spread over variable areas, accounting for specular reflections [Col94, SFES07], traversal of a participating medium [IZT+07] or the full range of all interactions [Sch09]. For rays emitted by a virtual camera, footprints are useful when choosing a filter size for texture mapping [Ige99], a geometry tessellation level [CLF+03] or the area from which previously stored flux is retrieved [CB04].

3.1.2.1. Neighbor Tracking

Footprints can be calculated by linking the hit points of rays originating as neighbors [Col94]. This is efficient for caustics after a specular reflection [IDN02, UPSK08] but when rays diverge strongly, the hit points can become too distant for a meaningful interpretation. Subdividing divergent patches and discarding those that converge to below a threshold size is one solution [IZT+07]. Tracing two *paraxial neighbors* with each ray [Col95] is another option but triples the number of ray casting operations.

3.1.2.2. Ray Differentials

An alternative approach that avoids the need to link multiple rays arises from differential calculus. In the *ray differentials* framework [Ige99], the recursive path taken by a ray is modeled as a series of functions f_i modifying its position and direction. If the initial position and direction depend on parameters p_1, p_2 and no further parameters are introduced during light transport, this yields

$$(\vec{z}, \vec{\omega})\,(p_1, p_2) = f_n \circ f_{n-1} \circ \ldots \circ f_1\,(p_1, p_2)\,. \tag{3.2}$$

Using the derivatives of this composite function, the distances between a ray and its neighbors can be estimated by first order Taylor approximation. If neighbors are removed in parameter space by Δp_1, Δp_2 on emission, the vectors spanning a footprint at an interaction are

$$\vec{\Delta}_1\,(p_1, p_2) = \vec{z}\,(p_1 + \Delta p_1, p_2) - \vec{z}\,(p_1, p_2) \approx \Delta p_1 \frac{\partial \vec{z}}{\partial p_1}\,(p_1, p_2)\,, \tag{3.3}$$

$$\vec{\Delta}_2\,(p_1, p_2) = \vec{z}\,(p_1, p_2 + \Delta p_2) - \vec{z}\,(p_1, p_2) \approx \Delta p_2 \frac{\partial \vec{z}}{\partial p_2}\,(p_1, p_2)\,. \tag{3.4}$$

To avoid the computational cost of having to differentiate a complex function at every hit point, partial derivatives of its position and direction are tracked with each ray. Whenever a function f_i is applied, the derivatives are updated according to the chain rule. For rays emitted by a virtual camera, p_1, p_2 denote screen coordinates. With $\vec{z}_c$ the camera position and $\vec{u}, \vec{v}, \vec{w}$ the normal, right

and up vectors of its view plane, the ray origins and directions at emission are

$$\vec{z}\,(p_1, p_2) = \vec{z}_c, \qquad \vec{\omega}\,(p_1, p_2) = \widehat{\vec{u} + p_1\,\vec{v} + p_2\,\vec{w}}.$$

Differentiation of these equations yields four partial derivatives, the initial ray differentials

$$\frac{\partial \vec{z}}{\partial p_1}\,(p_1, p_2) = \vec{0}, \qquad \frac{\partial \vec{\omega}}{\partial p_1}\,(p_1, p_2) = \frac{\vec{v} - (\vec{\omega} \cdot \vec{v})\;\vec{\omega}}{\|\vec{u} + p_1\,\vec{v} + p_2\,\vec{w}\|}, \tag{3.5}$$

$$\frac{\partial \vec{z}}{\partial p_2}\,(p_1, p_2) = \vec{0}, \qquad \frac{\partial \vec{\omega}}{\partial p_2}\,(p_1, p_2) = \frac{\vec{w} - (\vec{\omega} \cdot \vec{w})\;\vec{\omega}}{\|\vec{u} + p_1\,\vec{w} + p_2\,\vec{w}\|}. \tag{3.6}$$

3.1.2.3. Propagation

The first operation applied to a ray is propagation through space until the nearest hit point,

$$\vec{z}'\,(p_1, p_2) = \vec{z}\,(p_1, p_2) + t_{hit}\,(p_1, p_2)\;\vec{\omega}\,(p_1, p_2)\,, \qquad \vec{\omega}'\,(p_1, p_2) = \vec{\omega}\,(p_1, p_2)\,.$$

With p either of the two parameters, the partial derivatives are

$$\frac{\partial \vec{z}'}{\partial p}\,(p_1, p_2) = \underbrace{\frac{\partial \vec{z}}{\partial p}\,(p_1, p_2)}_{\text{old differential}} + \underbrace{t_{hit}\,(p_1, p_2)}_{\text{hit distance}}\;\underbrace{\frac{\partial \vec{\omega}}{\partial p}\,(p_1, p_2)}_{\text{old differential}} + \frac{\partial t_{hit}}{\partial p}\,(p_1, p_2)\;\underbrace{\vec{\omega}\,(p_1, p_2)}_{\text{old direction}}\,, \tag{3.7}$$

$$\frac{\partial \vec{\omega}'}{\partial p}\,(p_1, p_2) = \underbrace{\frac{\partial \vec{\omega}}{\partial p}\,(p_1, p_2)}_{\text{old differential}}\,. \tag{3.8}$$

For a planar surface given by $\vec{z} \cdot \vec{n} - d = 0$, the missing partial derivative of t_{hit} is calculated by inserting the ray equation for $\vec{z}$, differentiating and expressing the result in terms of t_{hit}, yielding

$$\frac{\partial t_{hit}}{\partial p}\,(p_1, p_2) = -\frac{\left(\underbrace{\frac{\partial \vec{z}}{\partial p}\,(p_1, p_2)}_{\text{old differential}} + \underbrace{t_{hit}\,(p_1, p_2)}_{\text{hit distance}}\;\underbrace{\frac{\partial \vec{\omega}}{\partial p}\,(p_1, p_2)}_{\text{old differential}}\right) \cdot \vec{n}}{\underbrace{\vec{\omega}\,(p_1, p_2)}_{\text{old direction}} \cdot \vec{n}}. \tag{3.9}$$

This result holds for non-planar surfaces as well [Ige99]. Using equations 3.7 to 3.9, the differentials from equations 3.5 and 3.6 can be updated whenever a ray is propagated through space.

3.1.2.4. Specular Reflection

When a ray encounters a specular reflector, it is mirrored about the surface normal,

$$\vec{z}'\,(p_1, p_2) = \vec{z}\,(p_1, p_2)\,, \qquad \vec{\omega}'\,(p_1, p_2) = \vec{\omega}\,(p_1, p_2) - 2\,(\vec{\omega}\,(p_1, p_2) \cdot \vec{n}\,(p_1, p_2))\;\vec{n}\,(p_1, p_2)\,.$$

As the normal may vary with hit point, it also is a function of p_1, p_2, leading to

$$\frac{\partial \vec{z}'}{\partial p}\,(p_1, p_2) = \underbrace{\frac{\partial \vec{z}}{\partial p}\,(p_1, p_2)}_{\text{old differential}}\,, \tag{3.10}$$

$$\frac{\partial \vec{\omega}'}{\partial p}(p_1,p_2) = \underbrace{\frac{\partial \vec{\omega}}{\partial p}(p_1,p_2)}_{\text{old differential}} - 2\Bigg((\underbrace{\vec{\omega}(p_1,p_2)}_{\text{old direction}} \cdot \vec{n}(p_1,p_2)) \frac{\partial \vec{n}}{\partial p}(p_1,p_2) + \Bigg(\underbrace{\frac{\partial \vec{z}}{\partial p}(p_1,p_2)}_{\text{old differential}} \cdot \vec{n}(p_1,p_2) + \underbrace{\vec{\omega}(p_1,p_2)}_{\text{old direction}} \cdot \frac{\partial \vec{n}}{\partial p}(p_1,p_2) \Bigg) \vec{n}(p_1,p_2) \Bigg). \tag{3.11}$$

Rendering algorithms and graphics APIs typically model all surfaces as *triangle meshes* [SA10]. For planar triangles, the normal derivative is zero and equation 3.11 can be evaluated directly. However, different normals are frequently assigned to the triangle vertices and *interpolated* [Pho75] to better approximate curved surfaces. A point on a triangle is parameterized using *barycentric coordinates* u, v, w as a weighted sum of the vertices $\vec{v}_0$, $\vec{v}_1$, $\vec{v}_2$ such that $\vec{z} = w\,\vec{v}_0 + u\,\vec{v}_1 + v\,\vec{v}_2$ with $u \geq 0$, $v \geq 0$ and $w = 1 - u - v \geq 0$. The interpolated normal at the point then is $\vec{n}(u,v,w) = \widehat{w\,\vec{n}_0 + u\,\vec{n}_1 + v\,\vec{n}_2}$.

Partial derivatives are obtained by expressing the barycentric coordinates in terms of p_1, p_2 and differentiating $\vec{n}(p_1,p_2)$. In the original ray differentials framework, a solution is derived which requires three plane equations to be stored with each triangle that describe its edges. We address the issue of this significant memory overhead in section 5.1.2, showing how the derivative can efficiently be computed without any additional storage.

3.1.2.5. Other Interactions

Refraction can be differentiated analogously to specular reflection [Ige99]. For any other interaction, the influence of additional parameters must be considered. *Path differentials* account for derivatives of equation 3.2 with respect to any number m of parameters [SW01]. Differentials are initialized for p_1, p_2 on emission and more added as interactions with additional parameters occur.

At a hit point, the footprint is then the Minkowski sum of m difference vectors. To calculate these, the distances Δp_i in parameter space between neighboring rays must be determined. When one ray spawns multiple secondary rays, their distances can directly be used as *local deltas*. If each ray is traced independently, its nearest neighbors are unknown. *Global deltas* are proposed for this case. With n_{rays} the total number of rays emitted, the distance to the nearest neighbor is estimated as $\Delta p_i \approx \left(\sqrt[m]{n_{rays}} \right)^{-1}$ for each parameter on a unit scale, presuming a uniform distribution throughout m-dimensional parameter space.

Path differentials require incrementally growing storage and complex calculations. For interactive rendering, inexpensive approximations using fixed storage only are more desirable. We propose and evaluate such methods in chapters 5 and 7. Concurrently to some of our work, a similar technique has been introduced [Sch09]. Only differentials with respect to p_1, p_2 are computed. Additional parameters are accounted for by scaling the footprint at each interaction according to the inverse ray density in the chosen direction, generating larger footprints where fewer rays are sent and vice versa. If only a single specular interaction were to be simulated, the need to store any differentials could be eliminated by calculating a footprint on the fly [WS03].

3.1.3. Surface Intersection

A ray must be tested for intersection with surfaces in the scene to locate its hit point. For the simple example of a sphere with center $\vec{x}_c$ and radius r, an analytic solution exists. Ray equation 3.1 is inserted into $\|\vec{z} - \vec{x}_c\| = r$ and the result solved for t. A quadratic expression emerges as the ray may

enter and leave the sphere again, producing two hit points. In the application to photon mapping, the point at which the ray passes closest to the sphere center $\vec{x}_c$ is also interesting. This can be found by computing $\vec{\delta} = \vec{x}_c - \vec{x}$ and projecting it onto the ray, obtaining $t_{closest} = \vec{\delta} \cdot \vec{\omega}$. The distance from $\vec{z}\,(t_{closest})$ to $\vec{x}_c$ is $d = \|\vec{\delta} - t_{closest}\,\vec{\omega}\|$.

Intersections with other surface types require specialized code paths for each and can be arbitrarily difficult to compute. Modeling all surfaces as triangle meshes avoids this complexity. Only the intersection test between a ray and a triangle is then required.

3.1.3.1. Möller-Trumbore Triangle Intersection

The Möller-Trumbore algorithm [MT97] efficiently tests a ray and a triangle for intersection. If a hit point exists, the corresponding ray parameter t_{hit} and barycentric coordinates u, v are returned. Conceptually, the algorithm uses a transformation matrix M that maps $\triangle(\vec{v}_0, \vec{v}_1, \vec{v}_2)$ to $\triangle(\vec{0}, \vec{i}, \vec{j})$ and the ray direction $\vec{\omega}$ to $\vec{k}$ with $\vec{i}$, $\vec{j}$, $\vec{k}$ the unit coordinate axes. The hit point coordinates are then $(u, v, t)^T = M\,\vec{x}$. These are obtained by Cramer's rule, following algorithm 3.1. Two triangle edges are required. For efficiency, these may be precalculated, storing $\vec{v}_0$, $\vec{e}_1$, $\vec{e}_2$ instead of $\vec{v}_0$, $\vec{v}_1$, $\vec{v}_2$ to represent a triangle. Early outs exist whenever a coordinate is found to lie outside its valid range. In the form presented here, u, v are both calculated first and then jointly tested in line 6. If the hit point lies inside the triangle, its ray parameter t is computed on line 9. The test in line 10 determines whether the hit point is located between a given minimum distance t_{min} and the nearest intersection found so far at t_{hit}. If so, line 13 updates the nearest hit information.

Algorithm 3.1 Möller-Trumbore intersection test for triangle $\triangle(\vec{v}_0, \vec{v}_1, \vec{v}_2)$ and ray $\vec{z}\,(t) = \vec{x} + t\,\vec{\omega}$

1: $(\vec{e}_1, \vec{e}_2) \leftarrow (\vec{v}_1 - \vec{v}_0, \vec{v}_2 - \vec{v}_0)$
2: $\vec{\delta} \leftarrow \vec{x} - \vec{v}_0$
3: $(\vec{p}, \vec{q}) \leftarrow (\vec{\omega} \times \vec{e}_2, \vec{\delta} \times \vec{e}_1)$
4: $det^{-1} \leftarrow (\vec{e}_1 \cdot \vec{p})^{-1}$
5: $(u, v) \leftarrow det^{-1}(\vec{\delta} \cdot \vec{p}, \vec{\omega} \cdot \vec{q})$
6: **if** $u < 0$ **or** $v < 0$ **or** $u + v > 1$ **then**
7: **return false**
8: **end if**
9: $t \leftarrow det^{-1}\,(\vec{e}_2 \cdot \vec{q})$
10: **if** $t < t_{min}$ **or** $t > t_{hit}$ **then**
11: **return false**
12: **end if**
13: $(u_{hit}, v_{hit}, t_{hit}) \leftarrow (u, v, t)$
14: **return true**

3.1.3.2. Wald Triangle Intersection

Applying more precalculation allows for further efficiency gains. The Wald intersection test [Wal04] is based on the observation that hit point barycentric coordinates do not change when a triangle is projected onto a plane. By performing the projection as a precalculation, the intersection problem reduces to two dimensions. The precalculation steps are given by algorithm 3.2.

To minimize numerical error, each triangle is projected onto the coordinate plane on which it has maximal area. The coordinate axes spanning this plane and the axis normal to it are referenced by zero-based indexes i, j, k. $\vec{n}'$ compactly encodes the triangle plane $\vec{z} \cdot \vec{n} = \vec{v}_0 \cdot \vec{n}$, scaled such that

Algorithm 3.2 Precalculation for Wald triangle intersection test

1: $k \leftarrow \arg\max |\vec{n}_i|$
2: $(i, j) \leftarrow (k + 1, k + 2) \mod 3$
3: $\vec{n}' \leftarrow \vec{n}_k^{-1} \ (n_i, n_j, \vec{v}_0 \cdot \vec{n})^T$
4: $\vec{e}'_1 \leftarrow (-e_{1j}, e_{1i}, \det(\vec{e}_{1ji}, \vec{v}_{0ji}))^T \det^{-1}(\vec{e}_{1ij}, \vec{e}_{2ij})$
5: $\vec{e}'_2 \leftarrow (e_{2j}, -e_{2i}, \det(\vec{e}_{2ij}, \vec{v}_{0ij}))^T \det^{-1}(\vec{e}_{1ij}, \vec{e}_{2ij})$

the third component of its normal is one and does not need to be stored. Vectors $\vec{e}'_1$, $\vec{e}'_2$ encode the edges $\vec{e}_1$, $\vec{e}_2$ projected onto the chosen plane, prescaled to reduce divisions. In total, nine floating point values and an integer $k \in \{0, 1, 2\}$ are precalculated per triangle.

Algorithm 3.3 illustrates the intersection test. First, indexes i and j are reconstructed from k. Subsequent steps reference the i-th, j-th and k-th components of $\vec{x}$ and $\vec{\omega}$. Reordering these into vectors $\vec{x}'$ and $\vec{\omega}'$ places them at constant memory locations. An intersection test between the ray and the triangle plane is performed next, enabling an early out if $t < t_{min}$ or $t > t_{hit}$. Lines 7 and 8 compute barycentric hit point coordinates u, v using the two-dimensional projections $\vec{e}'_1$ and $\vec{e}'_2$. If all coordinates are within their valid ranges, line 12 updates the nearest hit information.

Algorithm 3.3 Wald intersection test for a triangle represented by $k, \vec{n}', \vec{e}'_1, \vec{e}'_2$ and ray $\vec{z}(t) = \vec{x} + t\vec{\omega}$

1: $(i, j) \leftarrow (k + 1, k + 2) \mod 3$
2: $(\vec{x}', \vec{\omega}') \leftarrow (\vec{x}, \vec{\omega})_{ijk}$
3: $t \leftarrow \left(\omega'_z + \vec{n}'_{xy} \cdot \vec{\omega}'_{xy}\right)^{-1} \left(n'_z - x'_z - \vec{n}'_{xy} \cdot \vec{x}'_{xy}\right)$
4: **if** $t < t_{min}$ **or** $t > t_{hit}$ **then**
5: **return false**
6: **end if**
7: $\vec{h} \leftarrow \vec{x}'_{xy} + t\vec{\omega}'_{xy}$
8: $(u, v) \leftarrow (\vec{e}'_{2xy} \cdot \vec{h} + e'_{2z}, \vec{e}'_{1xy} \cdot \vec{h} + e'_{1z})$
9: **if** $u < 0$ **or** $v < 0$ **or** $u + v > 1$ **then**
10: **return false**
11: **end if**
12: $(u_{hit}, v_{hit}, t_{hit}) \leftarrow (u, v, t)$
13: **return true**

Relative to algorithm 3.1, the advantages are a lower instruction count and an early out before $\vec{e}'_1$ and $\vec{e}'_2$ have been loaded from potentially slow memory. These come at the cost of more expensive precalculation. On architectures with a *swizzling* operation, line 2 requires no branching and both algorithms have identical branch counts. One difference between their variants presented here is that the Möller-Trumbore test considers only one side of a triangle while the Wald test is double-sided.

3.2. Spatial Indexing

Testing all n surfaces for intersection has complexity $O(n)$ and does not scale to large scenes. This is addressed by organizing surfaces in the nodes of a *spatial index*. Whenever the ray misses a node, any surfaces referenced by it can be safely ignored. In a *hierarchical index*, the nodes form a tree data structure. Surfaces are typically referenced by leaf nodes only. When the ray misses an inner node, the entire hierarchy rooted in it is culled, reducing average case complexity to $O(\log n)$. An actual acceleration is obtained if traversal is fast and eliminates large subsets of data.

In this thesis, spatial indexes over scene surfaces but also over photon interactions are used. Both are jointly referred to as *primitives* here to unify the discussion.

3.2.1. Space Partitioning

One indexing approach is the subdivision of scene space into *disjoint regions* with each leaf node referencing all primitives that intersect it. The simplest space partitioning is a *uniform grid* [FTI86]. A *hierarchical grid* [JW89] improves the indexing of uneven primitive distributions by recursively embedding finer grids where primitive density is high. Both data structures result in large storage overheads for sparsely populated regions.

An *octree* [FTYK83] reduces overheads. Every inner node is evenly divided into eight children, effectively embedding very low resolution grids inside each other. Further acceleration can be achieved by constructing separate octrees for clusters of primitives and grouping these into a hierarchy [RAA+03]. Conceptually, each octree node subdivides its parent using three orthogonal planes. Performing only one such subdivision per node, the *binary space partitioning (BSP) tree* [FKN80] is obtained whose inner nodes consist of a single splitting plane and two child references.

The most popular space partitioning today [WMH+07] is the *kd-tree* [Ben75], a BSP-tree variant in which each splitting plane is coplanar with one of the three coordinate planes. Nodes are highly compact as a splitting plane can be encoded using one floating point value for its distance from the origin and an integer $k \in \{0, 1, 2\}$ for the normal direction. The two bits required for k may be stolen from one of the child references and if sibling nodes are stored in pairs, the second reference dropped. Using 32-bit floating point values and child references, 8 bytes of storage are needed per node. In a *left-balanced* kd-tree, nodes are arranged in heap order to implicitly encode the hierarchy, eliminating the need for any child references [Jen01].

Separate kd-trees may be constructed for different objects. When an object is affinely transformed, its kd-tree can then be retained [Wal04], transforming rays into the local coordinate frame during traversal. Using two splitting planes per node allows more empty space to be cut off [Hav00] at the expense of higher storage requirements. *Quantizing* splitting plane coordinates [HMHB06] reduces storage size but introduces a reconstruction overhead. Marking nodes entirely inside a watertight object as opaque allows for faster detection of occluded shadow rays [DKH09].

3.2.2. Primitive Partitioning

Another approach is the subdivision of primitives into *disjoint sets* with each primitive referenced by exactly one leaf. Since sibling nodes are no longer separated by a plane and may overlap, a *bounding volume* must be stored for each that allows rays to be tested for intersection with it. While more storage is required per node, fewer nodes are sufficient and duplicate references to primitives avoided, yielding a lower total size than that of a kd-tree [GPSS07].

Proposed first are *bounding volume hierarchies (BVHs)* of *oriented bounding boxes (OBBs)* [RW80]. Many other bounding volume types are subsequently suggested [WHG84] with the *slab*, consisting of two parallel planes [KK86], among the simplest. Today, BVHs are virtually synonymous with *binary* hierarchies of *axis-aligned bounding boxes (AABBs)* [WMH+07]. When higher branching factors are used, these follow from collapsing multiple levels of a binary hierarchy [DHK08]. In the following, a binary AABB hierarchy is always meant when the term BVH is used unless specifically noted.

A naïve node representation requires six floating point values to encode bounds as signed distances from the coordinate planes plus two child references in an inner node or a reference to the first primitive and a primitive count in a leaf. Using 32-bit floating point values and references, this leads to 32 bytes per node. Memory requirements can be reduced in several ways. Reordering nodes to implicitly encode the hierarchy eliminates some [Smi98] or all [CSE06] child references. If leaves contain only few primitives, their bounding planes may be omitted, testing all primitives when their parent is visited [FM86]. Replacing the bounds with quantized values allows for a further reduction in node size to 20 [Ter01] or just 12 bytes [Mah05] at the expense of a reconstruction overhead during traversal. Further savings are possible by applying a compression algorithm to the BVH [KMKY09], allowing very large scenes to be processed but incurring a decompression cost.

Hybrid data structures aim to combine the simpler traversal of kd-trees with the lower memory requirements of BVHs. *H-trees* [HHS06], *Bounding interval hierarchies (BIHs)* [WK06] and *B-kd-trees* [WMS06] follow the same idea: Instead of storing the full bounding box of each child, only the bounding planes for a single splitting axis are recorded. The main disadvantage are looser bounds. H-trees improve tightness with interspersed AABB nodes cutting off excessive empty space. Nodes occupy 16 or 32 bytes each. B-kd-tree and BIH record child bounds in the parent node. A B-kd-tree node is 16 bytes in size, bounding each child by two planes with 22 bit precision. BIH nodes occupy 12 bytes, storing one plane per child. The *single slab hierarchy* [EWM08] similarly bounds each node with a single plane, using 8 bytes per node. *Ray-strips* [LYM07] and *ReduceM* [LYTM08] employ tightly fitting BVHs over triangle strips and more lightweight BIHs inside the strips.

We address further potential for removing *redundant* information in section 6.2.

3.2.3. Construction

The spatial index provides acceleration when few of its nodes are visited and many primitives culled. Manual construction of a high quality index [RW80] is only feasible for small, static scenes. In general, an algorithm is required that automatically recursively subdivides the n primitives.

3.2.3.1. SAH

Simple strategies yielding binary hierarchies split each node at its primitive or spatial median [KK86]. *Heuristic* construction promises higher acceleration by choosing the pair of children L, R for each node P that minimizes expected ray tracing cost. With p_P, p_L, p_R the probabilities of nodes being visited by a ray, C_T the traversal cost for an inner node, C_I the cost of a single primitive intersection test and n_P, n_L, n_R the numbers of primitives in each node, the cost metric is [Hav00]

$$C_P(L,R) = p_P C_T + p_L n_L C_I + p_R n_R C_I. \tag{3.12}$$

The method for estimating p_P, p_L, p_R that yields the highest ray tracing acceleration currently known is the *surface area heuristic (SAH)* [MB90]. With S the scene bounding box, the SAH makes three assumptions:

- Ray origins are uniformly distributed in space outside S.
- Ray directions are uniformly distributed on the sphere of directions.
- No ray hits any primitives.

Under these conditions, the probability that a ray intersecting S also intersects a node N is given by the ratio of SA (N) and SA (S), their *surface areas* [KM63],

$$p_N = \frac{\text{SA}\,(N)}{\text{SA}\,(S)}. \tag{3.13}$$

Inserting the above into equation 3.12 gives the cost of each candidate split. When constructing a kd-tree, $O\,(n_P)$ potential splitting planes lie at the starting and ending points of primitive AABBs along the three coordinate axes [Hav00]. For a BVH, $O\,(2^{n_P})$ distributions of primitives to the two children are possible. In practice, primitives are sorted by their centroids along the three axes and the $O\,(n_P)$ partitions of these lists considered only [Wal07]. By sorting and then progressively splitting candidate lists, the construction process reaches optimal $O\,(n \log n)$ complexity [WH06]. Recursion terminates when the expected cost of a leaf is lower than that of any subdivision. This may be detected locally at each node or globally for the entire hierarchy [HB02].

The SAH is able to *shave off* primitives coplanar with the coordinate planes, producing nodes of *zero volume* [WH06]. A number of modifications further improve acceleration. Biasing the heuristic to favor cutting off empty space is beneficial [HKRS02]. When repeated intersection tests with primitives referenced by several nodes are avoided by *mailboxing* [AW87], an adjusted cost metric applies [Hun08]. Tightening AABBs after each step ensures they bound only the parts of primitives not cut off by splitting planes [HB02]. In a BVH, primitives are never cut and large outliers can reduce acceleration. This is alleviated by representing a single primitive as multiple smaller AABBs, built either before [EG07, DK08] or during BVH construction [PGDS09, SFD09].

While the SAH is highly successful, its assumptions are unrealistic. This motivates us to investigate an alternative set of assumptions and the resulting ray tracing acceleration in section 4.1.

3.2.3.2. Faster Construction

In dynamically changing scenes, the spatial index needs to be updated every frame. A BVH can be *refit* by shrinking or expanding node bounds [LAM05, WBS07]. Complete reconstruction is still required to ensure acceleration when cumulative changes degrade BVH quality [LYT06]. For a kd-tree, only reconstruction is possible because sibling nodes must always remain spatially disjoint.

A complete reconstruction is *expensive* as the SAH must be evaluated numerous times and *sequential* since each node depends on its parent. Collecting primitives in small numbers of *bins* leads to reduced cost by considering splits among these only [HKRS02] or using linear [PGSS06] or quadratic [HMS06] interpolation of the SAH between bins. Clustering primitives in a preprocessing step similarly reduces the number of candidate splits [Gar09]. Deeper in the hierarchy, primitive counts are lower and exact SAH evaluation is possible again [HMS06, PGSS06]. If a new spatial index is constructed in each frame, a cost metric biased toward the current virtual camera position may be used [Hav00].

If the top levels are covered in *breadth-first search (BFS)* order first, parallel construction of sub-hierarchies is possible [PGSS06]. At the top, parallelization over primitives is demonstrated for both BVHs [Wal07] and kd-trees [CKL+10]. Simpler primitive [SSK07] or spatial median splits [ZHWG08] can be combined with the SAH deeper in the tree. The exact opposite is also advocated [PL10]. Constructing *all* nodes in BFS order [ZHWG08] maximizes parallelism but has large temporary storage requirements for pending sub-hierarchies. Dynamically switching from BFS processing of the entire hierarchy to a phase that completes unfinished sub-hierarchies [HSZ+10] reduces these. Uniform grids finally offer simple reconstruction as no hierarchical dependencies exist [LD08, KS09].

3.2.3.3. Linear BVH

A particularly fast and highly parallel BVH construction is possible using spatial median splits. Given a set of points in space, their linear positions along a *Morton curve* can be independently computed. The key insight is that sorting by these positions arranges the points in the *depth-first search (DFS)* traversal order of a binary spatial median split hierarchy. This observation enables fast octree construction for point clouds [AGCA08, ZGHG08] and leads to a highly parallel BVH construction algorithm for the CUDA platform [LGS+09].

Regularly subdividing the scene AABB k times along each coordinate axis yields a grid of 2^{3k} cells. The Morton curve traverses this grid in spatially coherent order such that cells close in space are likely to be close on the curve. For a grid cell with k-bit coordinates x, y, z, its Morton curve position is obtained by permuting the bits into $z_0 y_0 x_0 z_1 y_1 x_1 \ldots z_{k-1} y_{k-1} z_{k-1}$ order, beginning with the most significant. This $3k$-bit *Morton code* corresponds to the path from the root of a binary hierarchy to the cell, each bit selecting one of two children. Every node is subdivided at its spatial median along a coordinate axis that cyclically changes with hierarchy level. Combining three bits to indicate one of eight children, the path through an octree is obtained [WvG92].

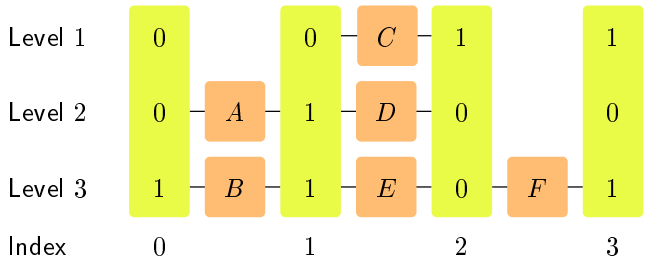
Linear BVH (LBVH) construction begins by creating a *reference list*. A key-value pair is output for each of the n primitives in parallel, containing the Morton code of its centroid and a pointer to the primitive. These references are sorted by Morton code using a parallel radix sort, resulting in an implicit representation of the node hierarchy. The subsequent hierarchy construction is not fully described but using recently published information [PL10], can be largely reconstructed.

For each pair of neighboring primitive references, a *split list* is created. If the most significant bit position at which their Morton codes differ is m, the primitives have common ancestors on levels 0 to m. Splits are output for levels $m+1$ to $3k$ (figure 3.1(a)). A split is as a key-value pair of reference index and hierarchy level. These are generated for all references in parallel, using a parallel prescan of the split counts to concatenate them into a compacted list. This split list is then sorted by hierarchy level, using parallel radix sort again (figure 3.1(b)).

On each level, the interval of all reference indexes $[0, n-1]$ is subdivided into nodes by the splits (figure 3.1(c)). As a parent and its left child start with the same index, left children can be located by processing the splits in levels and keeping track of the most recently constructed node starting at each index. All splits on the same level may be handled in parallel.

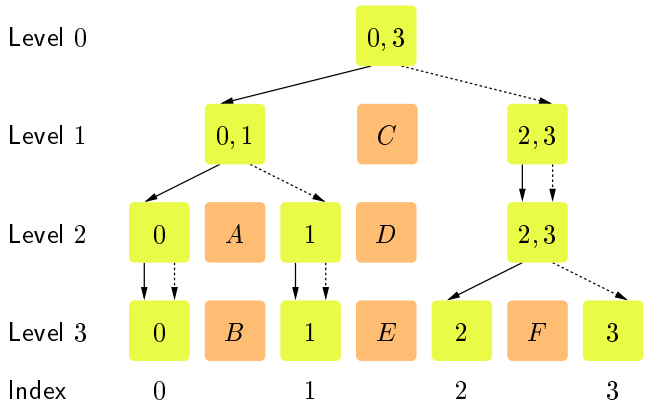
To locate right children, additional steps are taken. A parent node and its right child end with the same index. Sorting all nodes by their last index thus turns all parents and their right children into neighbors, allowing them to be linked in parallel. After this, chains of *singleton* nodes referencing only one child must be removed (figure 3.1(c), levels 1 and 2). This is done by walking up from the leaves in parallel and collapsing singleton nodes with their children. The final step turns the centroid hierarchy into an actual BVH. An AABB is computed for each primitive. Child AABBs are then combined by walking up the tree, processing all nodes on the same level in parallel.

We investigate the use of this algorithm to construct a spatial index over photon interactions in section 6.1 and remove several of its inefficiencies. A recent LBVH variant [PL10] also targets higher efficiency. To exploit spatial coherence, consecutive primitives are clustered by parallel compression whenever the $3l$ most significant bits of their Morton codes match. After a parallel radix sort of the clusters, primitive references are extracted, using a parallel prescan of the cluster sizes to obtain their positions in a compacted list. The references in each cluster are sorted by the remaining $3k - 3l$ bits using only shared memory due to the smaller input sizes.



(a) Four primitive references arranged in the order of their Morton codes 001, 011, 100, 101 and the splits generated between them, labeled with letters.

(b) Splits sorted by hierarchy level.



(c) On each level, the interval $[0, 3]$ is subdivided by the splits into nodes. Left child pointers are constructed with the nodes, right child pointers afterward. Singleton nodes on levels 1 and 2 must be collapsed with their children.

Figure 3.1.: Linear BVH construction: Primitives are sorted in Morton curve order, splits generated between them and traversed to construct a hierarchy of nodes.

Instead of creating and sorting a split list, multiple passes over the primitive references are made, in each pass constructing p levels of nodes. Starting with the root, up to $2^p - 1$ subdivisions are possible in the first pass. All primitive references are compared to their neighbors in parallel and subdivisions collected in a temporary array. Nodes are constructed and the next pass commences, repeating the process for the primitive references in each node on the p-th level. A parallel prescan of all subdivision counts allows new nodes to be output in parallel. Data structures are maintained that link each primitive to the most recent node which contains it, allowing parents and children to be connected immediately. A parallel scan after each pass updates this data.

3.2.4. Traversal

During ray tracing, the spatial index is traversed from its root and any primitives encountered in the leaves are tested for intersection. Should both children of a node be intersected by the ray, traversal branches, requiring a recursion stack or an alternative method of returning to unvisited nodes.

A ray is defined by equation 3.1 and a parameter interval $t \in [t_{min}, t_{max}]$ with $t_{max} \leq \infty$ the ray length and $t_{min} \geq 0$ the minimal permitted hit point distance. Setting $t_{min} = \epsilon > 0$ allows spurious intersections with the surface from which a ray originates to be ignored.

3.2.4.1. kd-Tree

Each kd-tree inner node contains a splitting plane $\vec{z} \cdot \vec{n} - d = 0$ separating its two children with $\vec{n}$ one of the unit coordinate axes. If $\vec{n}$ is the k-th axis, the ray intersects the splitting plane at $t_s = (d - x_k)\ \omega_k^{-1}$. Child nodes are classified as *near* and *far* according to their order along the ray. For $\omega_k > 0$, the left child is near and the right far, vice versa for $\omega_k < 0$.

Three cases are possible. If $t_s > t_{max}$, the ray intersects only the near child and traversal continues with it. $t_s < t_{min}$ is the analog case for the far child. When neither is true, the ray straddles the splitting plane and both children need to be visited. In the near child, the ray has parameter interval $[t_{min}, t_s]$ and in the far, $[t_s, t_{max}]$. Traversal continues with one of the children. The other child and its parameter interval are pushed onto the stack. While children may be visited in any order, starting with the first is more efficient: If a hit is found, the ray is truncated at $t_{hit} < t_s$ and any nodes still on the stack no longer need to be visited.

Upon reaching a leaf, its primitive references are processed. When a hit is found, the remaining primitives in the current leaf must still be tested for intersection as there may be an even nearer hit. Only for a shadow ray does the search immediately terminate since occlusion has been established. If all primitives in a leaf have been tested and no hit was found, the top entry is popped off the stack and traversal continues with it. An empty stack indicates that the ray misses all primitives.

Whenever a parameter interval is popped, its start is identical to the end of the interval considered until then. Rather than pushing the entire interval onto the stack, it is thus sufficient to store only its end. The start is reconstructed by setting $t_{min} \leftarrow t_{max}$ immediately before popping. A stack entry thus holds one floating-point value and one integer reference, a total of 8 bytes.

The properties of floating-point arithmetic [IEE08] allow singularities to be handled without special case treatment. By carefully arranging the conditional statements, $\omega_k = 0$ or a node with zero volume shortening the interval to a single point do not corrupt traversal [Wal04]. In the latter case and for any primitive coincident with the bounds of its node, t_{hit} may lie slightly outside the parameter interval due to imprecision. The interval should therefore be expanded by $\pm\epsilon$ during leaf node traversal.

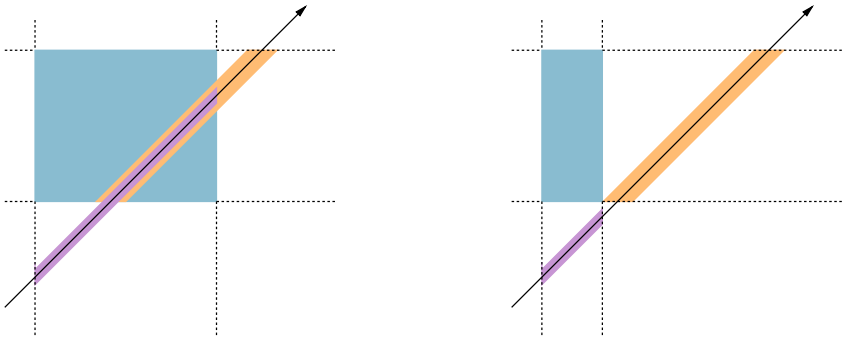
Photon mapping uses a kd-tree over photon interactions [Jen96]. A query consists of a point $\vec{x}$ for which the primitives lying within a distance h_{max} of it are sought. The signed distance to the splitting plane is $\delta = x_k - d$, classifying the children as left near, right far if $\delta < 0$ and left far, right near otherwise. If $|\delta| > h_{max}$, the near child is traversed only. For $|\delta| \leq h_{max}$, both children are visited, pushing the far one onto the stack. Traversal ends when the stack is empty.

3.2.4.2. BVH

A BVH node is traversed using the *slabs test* [KK86]. For each coordinate axis, two node bounding planes enclose a parameter interval $[t_{a,k}, t_{b,k}]$ on the ray. The intersection $[t_a, t_b]$ of these intervals indicates the part of the ray intersecting the node (figure 3.2(a)). When the intervals are disjoint and their intersection is empty, the node is missed (figure 3.2(b)). To account for ray start and end distances, the interval is additionally intersected with $[t_{min}, t_{max}]$.

The test is shown in algorithm 3.4. Vectors $\vec{m}$ and $\vec{M}$ encode front and back bounding planes as signed distances from the coordinate planes. Where the ray direction has negative sign, these must be swapped. Using the minimum and maximum of t_1, t_2 does so without branching (lines 6–7).

As each node bounds only itself, both children must be visited when $[t_a, t_b] \neq \emptyset$. Without explicitly loading the children, their order along the ray is unknown. A simple heuristic is to store the coordinate



(a) An overlap indicates which part of the ray intersects a node.

(b) Disjoint intervals imply a node missed by the ray.

Figure 3.2.: Two-dimensional slabs test: Node bounding planes enclose ray parameter intervals on both axes. The intervals overlap where the ray intersects a node.

Algorithm 3.4 Slabs test for node $(\vec{m}, \vec{M})$ and ray $\vec{z}(t) = \vec{x} + t\vec{\omega}$

$t_a \leftarrow t_{min}$
$t_b \leftarrow t_{max}$
for $k \in \{x, y, z\}$ **do**
  $t_1 \leftarrow (m_k - x_k)\,\omega_k^{-1}$
  $t_2 \leftarrow (M_k - x_k)\,\omega_k^{-1}$
  $t_a \leftarrow \max(t_a, \min(t_1, t_2))$
  $t_b \leftarrow \min(t_b, \max(t_1, t_2))$
end for
if $t_a \leq t_b$ **then**
  visit children
end if

axis on which the node was originally split during construction and use the sign bit of the ray direction for that axis to choose traversal order [Mah05].

When an AABB is stored for each node, the interval $[t_a, t_b]$ can be calculated directly. Only a node reference needs to be pushed onto the stack, occupying 4 bytes. Quantized BVH representations use relative child bounds and require the parent AABB to be pushed as well. If only a subset of bounding planes is stored, the parameter interval is pushed and progressively shortened during traversal, analogously to a kd-tree. The BIH and B-kd-tree store child bounding planes at their parent, obtaining intervals for both children simultaneously. This allows children to be visited in their actual order along the ray and missed nodes to be skipped. By performing the slabs test for pairs of sibling nodes together, the same is possible for BVHs [AL09].

For photon mapping, a BVH over photon interactions is of interest. During node traversal, the position of a query point $\vec{x}$ relative to the six bounding planes is computed (algorithm 3.5). Both children are visited if $\vec{x}$ is found to lie inside the node AABB.

Algorithm 3.5 Photon map test for node $(\vec{m}, \vec{M})$ and point $\vec{x}$

if $m_x < x_x$ **and** $m_y < x_y$ **and** $m_z < x_z$ **and**
  $M_x > x_x$ **and** $M_y > x_y$ **and** $M_z > x_z$ **then**
  visit children
end if

3.2.4.3. Faster Traversal

On old hardware with slow floating point units, determining the relevant planes first can be beneficial. If only the entry point into a BVH node is sought, the three back-facing bounding planes may be ignored [Woo90]. Bounds coincident with those of a parent are similarly uninteresting [ST94].

The challenge on modern SIMD computer architectures is providing each instruction with multiple data items to operate on. Grouping rays into *packets* allows a node or primitive to be tested against these rays in parallel. First proposed for CPUs with 4-wide SIMD [WSBW01], packetization is also applicable to custom hardware [SWS02] and CUDA with 32-wide SIMT [GPSS07]. Its limitation is low SIMD utilization when incoherent secondary rays take different paths and rays must be masked off. Even if all common traversal steps are extracted [Res06], SIMD utilization remains low.

Proposed techniques for improving coherence spawn shadow rays in packets targeting the same light source [BWS03], separate shadow and reflection rays [BEL+07], spawn packets of rays with similar directions [SF07b] or combine these criteria [SCL05, SCL08]. Constructing a five-dimensional tree over origins and directions allows any secondary rays to be *reordered* into coherent packets instead [CHH02]. Directional bins provide a simpler reordering of rays by direction [HHS05]. While reordering improves SIMD utilization, its overheads may not be recouped in practice [MMAM07].

An alternative parallelization approach is to test individual rays against multiple child nodes and primitives at once, using a spatial index whose branching factor matches the SIMD width [EG08, DHK08, WBB08]. Performance is gained for incoherent rays but lost in the coherent case [WBB08]. Another alternative is *breadth first ray tracing* [Han86]. By extracting only the rays visiting a node via parallel compaction [WGBK05, BWB08, GR08, ORM08], any coherence between rays is automatically found. The combination of both concepts leads to an algorithm that adaptively switches between parallelization over rays and nodes [Tsa09]. Queuing rays at predefined points in the spatial index extracts coherence in a similar manner [NFLM07].

When operating on a cluster of independent nodes, work may be distributed between these by assigning each a subset of rays, as proposed for CPUs [WSBW01], custom hardware [SWS02] and GPUs [STK08], or by allocating each to a spatial region [DS84]. Orthogonal to these techniques, multiple intersection tests against the same primitive can be avoided by mailboxing [AW87], recording for each primitive which ray it has been last tested against.

3.2.4.4. GPU Traversal

GPU ray tracing follows the rapid hardware evolution. The first approaches represent rays as pixels and require a quad to be drawn for each primitive to trigger intersection tests. Which primitives to test is determined by the CPU, either using a spatial index [CHH02] or not [PBMH02].

Traversal of a kd-tree on the GPU is added next. As graphics APIs expose no read-write memory suitable for a stack, *stackless* traversal is used. The *kd-restart* algorithm resumes traversal from the root with the ray shortened to begin at the current t_{max} whenever no intersection is found in a leaf. *kd-backtrack* follows *parent links* placed in the tree during construction [FS05]. Adding packetization, a short stack held in registers and pushing the restart point deeper into the tree whenever possible removes most redundant traversals [HSHH07]. Stackless traversal in CUDA is demonstrated with *ropes* pointing to adjacent nodes, threads representing individual rays and warp-sized packets making coherent branching decisions for kd-trees [PGSS07] and BVHs [TMG09].

For a node with zero volume, restarting at the root leads to an infinite loop as t_{max} has the same

value both before and after its traversal. Parent pointers or ropes incur significant storage overheads. We propose an algorithm in section 4.2 that ensures correct kd-tree traversal order using a single register to hold additional information. Similar solutions have recently been applied to kd-trees with parent pointers [HL09] and stackless BVH traversal [Lai10]. Testing the ray against a node yields the decision to visit both, one or none of its children. The first case requires a single bit per hierarchy level to remember whether the first child needs to be visited again when returning via parent pointers or resuming from the root. In the other two cases, no additional storage is needed.

The more flexible CUDA programming model allows stacks for packets of rays to be placed in shared memory [GPSS07]. Letting rays traverse nodes in optimal order *without* packetization further increases performance [BAGJ08, ZHWG08] as threads executing the same instruction automatically fall into lock-step, regardless of whether they are considering the same node or not. A simple kernel that alternates between node traversal and primitive intersection achieves the highest performance by allowing both phases to execute in lock-step [AL09]. *Persistent threads* read rays from a job queue, circumventing the CTA scheduler optimized for uniform workloads. Stacks are located in slower local memory as their total size exceeds shared memory capacity. In section 4.3, we explore the use of now dormant shared memory as explicitly managed caches. Caching the stack top in fast memory has recently similarly been proposed for a hypothetical manycore architecture [AK10].

A combination of breadth first ray tracing, higher spatial index branching factor and packetization is also efficient in CUDA [GL10]. Rays are sorted by hash values into coherent packets. These are traced through a BVH in BFS order, constructing a compact job queue of node and packet pairs after each level using a parallel prescan. Persistent threads are beneficial again. Extending their concept further, rays can be collected at predefined points in the spatial index to generate jobs that traverse parts of the hierarchy together, reducing bandwidth requirements [AK10]. For the general problem of scheduling nonuniform jobs in CUDA, multiple queues with task stealing or sharing are found to perform best [CT08, TPO10].

NVIDIA OptiX [PBD+10] provides a commercial, binary only implementation of the currently fastest ray tracing algorithms for CUDA.

3.3. Monte Carlo Rendering

The radiance reaching a virtual camera through any point in an image is given by equation 2.9. Evaluating this recursive integral over hemispheres of directions naïvely would require an infinite number of rays to be traced for each pixel and is clearly impossible.

Ray tracing [Whi80] decomposes radiance into its direct and indirect components first. Only point light sources are used, reducing direct illumination from an integral to a sum of contributions. A shadow ray is cast for each light and equation 2.13 evaluated if no occlusion occurs. The indirect component is computed for specular interactions only, following at most a single reflection and a single refraction ray per interaction. Rendering algorithms using *Monte Carlo quadrature* improve on ray tracing, efficiently simulating wide ranges of light paths.

3.3.1. Monte Carlo Quadrature

Let I be the integral of a function f over domain X. Monte Carlo quadrature [RK07] estimates the value of I by drawing n random samples x_i from X. With samples distributed according to a

probability density function p, the estimator $\tilde{I}$ is

$$I = \int_X f(x)\, dx = \int_X p(x) \frac{f(x)}{p(x)}\, dx = \mathbb{E}\left[\frac{f(x)}{p(x)}\right] \approx \frac{1}{n}\sum_{i=1}^{n} \frac{f(x_i)}{p(x_i)} =: \tilde{I}. \tag{3.14}$$

Its variance σ^2 is given by

$$\sigma^2 = \frac{1}{n}\int_X \left(\frac{f(x_i)}{p(x_i)} - I\right)^2 p(x_i)\, dx.$$

Applying equation 3.14 to physically based rendering, any integral can be replaced with a weighted sum of n rays cast into randomized directions. This is possible both for rays emitted by a virtual camera and those traced in the opposite direction, from the light sources.

3.3.2. Rendering Algorithms

Distribution ray tracing [CPC84] extends the original ray tracing algorithm to randomly sampled ray directions, simulating diffuse and glossy interactions, area light sources, depth of field and motion blur. *Path tracing* [Kaj86] spawns a single randomized reflection or refraction ray and one shadow ray at each interaction, eliminating branching during recursion.

By tracing rays from a virtual camera, these algorithms *gather* light. Only light paths of the form $LD(S|D)^*E$ can be efficiently simulated this way. At a specular interaction, illumination from a single incident direction is relevant. If the last surface encountered was specular, the probability of finding a light source in the relevant incident direction would be limited by occlusion for area lights and zero for point lights. *Shooting* light from the sources instead allows paths of the form $L(S|D)^*DE$ to be simulated, again ending in a diffuse interaction. *Bidirectional* algorithms have two passes, in the first shooting light and in the second gathering the stored information.

Early techniques store information on surfaces. In *backward ray tracing* [Arv86], each ray carries a fraction of the total light source flux and *splats* it into a texture map on the first diffuse surface seen after a specular interaction. Subsequently, splatting into texture maps with adaptive resolution even without a prior specular interaction is suggested [Hec90], storing global instead of only indirect illumination. The *radiosity* algorithm [GTGB84] calculates indirect illumination by solving a system of linear equations encoding links between surface patches. Only diffuse interactions are considered, calculating irradiance due to emittance on other surfaces. *Bidirectional ray tracing* treats illuminated surfaces as secondary light sources [CF87]. Common to these algorithms are the storage of accumulated directionless quantities and aliasing artifacts due to limited resolution.

Bidirectional path tracing [VG94] connects the endpoints of paths traced in both directions and accounts for all $L(S|D)^*E$ light path types. Its downside is high variance in the Monte Carlo estimates, requiring very large numbers of samples. *Metropolis light transport* [VG97] begins with paths generated by bidirectional path tracing. These undergo repeated mutations by the Metropolis-Hastings algorithm, accepting or rejecting each new path with conditional probability so that their distribution approaches that in an ideal image. All paths have identical contributions and are splat into the image. The difficulty lies in designing mutation strategies that explore the entire space of paths through the scene while providing a high acceptance probability, reducing the computational overhead of generating candidate paths that are subsequently discarded.

3.3.3. Sampling

Although the estimator $\tilde{I}$ in equation 3.14 is unbiased for $n \geq 1$ and any density function p with $f(x) \neq 0 \Rightarrow p(x) > 0$, the choices of n and p have influence on its variance and thus, *image noise*. Increasing n reduces variance but also raises computational cost as more rays need to be traced. Optimizing the sampling strategy reduces variance without requiring additional samples.

3.3.3.1. Importance Sampling

If the integrand has constant sign, $p \propto |f|$ yields zero variance. This optimal choice is unattainable in practice as its definition $p(x) = f(x) I^{-1}$ depends on the unknown integral I. *Importance sampling* approximates the optimum by adjusting p to resemble f based on available information. Following equation 2.9, reflected radiance is an integral of the BRDF f, incident radiance L and an inclination factor $\cos\theta_i$. With L typically unknown until a ray has actually been traced, *BRDF-weighted* sampling using $p \propto f \cos\theta_i$ is common [Whi80]. If *photons* are traced from the lights in a preprocessing pass and their interactions with the scene recorded, L can be approximatively importance-sampled by locating nearby interactions [Jen95]. Analogous information is obtained for the photons by prepending another pass in which *importons* are traced from the virtual camera [PP98]. The theoretical basis for these techniques is an adjoint relationship between light and importance [Chr03].

When not limited to point lights, shadow rays need to be distributed over light source areas. Importance sampling of the radiance emanating from a light source is shown for area lights [KK02] and environment map illumination [ARBJ03]. In both cases, *stratified sampling* is used, dividing the integration domain into *strata* and drawing a predefined number of samples from each. For complex distributions, *resampled importance sampling* [TCE05] improves efficiency by first drawing several candidates from a simpler distribution and then choosing a sample x_i among these to approximate the desired distribution.

3.3.3.2. Quasi-Random Sampling

Avoiding a clumping of random samples in parts of the integration domain further reduces variance. *Jittered sampling* divides X into n equally sized strata and places one sample in each [Coo86]. *Latin hypercube sampling* [MBC79] evenly divides X into n intervals on each coordinate, drawing samples such that every interval contains exactly one. *Quasi-random sampling* guarantees an even distribution by replacing randomization with *deterministic* low-discrepancy sequences of samples. Although a variance reduction is proven for integrands of finite variation only, applications to the rendering equation also benefit, despite infinite variation at surface boundaries [SKP99].

The quasi-random *Halton sequence* [Hal60] consists of elements $x_i = \sum_{k=1}^{j} i_k m^{-k}$ with i_k the k-th least significant bit of i and m a prime number. These j-bit samples regularly fill the interval $[0, 1]$ with increasing density. Multidimensional samples are obtained by combining x_i for different m. The *Sobol' sequence* [Sob67] consists of $x_i = i_1 v_1 \oplus i_2 v_2 \oplus \ldots \oplus i_j v_j$ where $\oplus$ is the exclusive-or operation and the v_k are precalculated direction numbers. These samples fill the range $\left[0, 2^j - 1\right]$ with the desirable properties of jittered and Latin hypercube sampling and lower regularity than a Halton sequence. Multidimensional samples are obtained by combining x_i for different direction numbers. Quasi-random sampling is deterministic and introduces bias in the form of *structured noise*. This is addressed by *randomized scrambling* for both the Halton [Sch08] and Sobol' [KK02] sequences.

3.3.3.3. Adaptive Sampling

With *adaptive sampling*, samples are placed in parts of the integration domain for which convergence has not been reached yet. The established method for primary rays emitted by a virtual camera is to recursively subdivide the image, casting more rays where the difference in radiance between neighbors exceeds a threshold [Whi80]. When tracing in the opposite direction, suggested subdivision criteria are neighboring hit points too distant in space [Hec90] or lying on different surfaces [Col95]. If the samples represent complete paths through the scene, comparing the radiance returned by neighbors is proposed [HJW+08]. For shadow rays, arranging light sources in the order of expected contribution allows sampling to be halted when a radiance threshold is reached [ARBJ03]. An alternative to such empirical criteria is nonextensive Tsallis entropy [XSXZ07]. After estimating the correlation between neighboring samples from a pilot pass, sampling proceeds until entropy falls below a threshold.

Control variates are another variance reduction technique. By approximating I based on nearby samples first and then drawing samples only to adjust this approximation, an estimator with lower variance is obtained [PWP08, PBSP08]. The number of samples is again adaptively decided.

3.4. Photon Mapping

Algorithms storing illumination on surfaces cannot reproduce all light path types. Those connecting paths traced from both directions suffer from high variance. *Photon mapping* [Jen96] provides an alternative to both. In the first pass, photons carrying flux are traced from the light sources and their interactions with surfaces stored in a *spatial* data structure independent of scene geometry.

The second pass traces rays from a virtual camera. At diffuse and glossy surfaces, illumination is reconstructed by *density estimation*. Photon interactions located nearby are assumed to lie on a planar surface with constant BRDF around the query point $\vec{x}$. Their fluxes $\Phi_{i,j}$ are reflected by the BRDF f and divided by the area A from which the interactions have been collected, yielding an estimate of the reflected radiance for query direction $\vec{\omega}$. Combined with reflection rays at specular interactions and shadow rays for direct illumination, all light path types are simulated.

Photon mapping is a biased rendering algorithm: Estimating illumination from an area around the query point leads to blurring; treating the underlying surface as locally flat disregards curvature and discontinuities at silhouettes. Collecting interactions from a smaller area reduces bias but increases variance. An automatic tradeoff is achieved by locating a fixed number k of nearest photon interactions and setting the area A to the cross section of their bounding sphere around $\vec{x}$. Bias is low in regions of dense illumination but allowed to increase where interactions are sparse, countering the higher variance. Photon interactions after LS^+D paths are stored in the *caustic photon map*, all others in the *global photon map*, separating sharp caustics from slowly varying global illumination.

The image is synthesized using a combination of techniques. Shadow and reflection rays handle direct illumination and specular interactions. Optional *shadow photons* traced through surfaces allow shadow rays to be omitted where light source visibility can be interpolated. Density estimation in the caustic photon map provides caustics. For other indirect illumination, *final gather* rays are spawned, performing density estimation in the global photon map at their hit points. This additional level of indirection turns reconstructed global illumination into indirect illumination. An alternative is to store the second and later interactions of each photon only, directly recording indirect illumination and eliminating the need for computationally expensive final gather.

3.4.1. Photon Tracing

Photons are traced identically to rays originating at a virtual camera, using the same spatial index and traversal algorithms. Importance-sampling light source area [Jen95] or area and intensity [LP03] on emission generates photons with equal flux, minimizing variance. If importons have been traced from the virtual camera in a preprocessing pass, the relevance of emission points and directions to the image is estimated from these and also importance-sampled [PP98]. Projecting object bounding volumes onto a texture map around an emission point allows directions to be ignored in which no objects are present [JC95]. Following pilot photons is more efficient [GMSJ03] but carries the risk of missing small objects. For caustic photons, emission toward specular objects only is analogously possible by using projection [Jen96] or pilot photons [GWS04]. Instead of separating global and caustic photons at emission, a photon map can be chosen for each based on its most recent interaction [KW00]. Distributing photon emission in time enables the simulation of motion blur [CJ02].

Photon differentials [SFES07] apply the concept of ray differentials to photon mapping, assigning each photon interaction an individual footprint. We show in section 5.1.1 how to initialize differentials so that the distances between photons in parameter space do not need to be tracked.

An interaction is stored whenever a photon encounters a diffuse or glossy surface. At each surface hit, *Russian roulette* [AK90] decides between onward tracing and termination. Importance-sampling reflectivity ρ, a sample $y \in [0,1]$ is uniformly drawn. If $y < \rho$, a new direction is chosen by BRDF-weighted sampling, combined with the relevance of directions to the image if importons have been traced [PP98]. If $y \geq \rho$, the path ends. This technique decides between specular interaction, diffuse interaction and termination in an unbiased way and allows flux to remain unchanged [Jen96]. When using multiple color bands, Russian roulette is based on the average reflectivity and the flux of a surviving photon must be scaled by $\rho_i \rho_{average}^{-1}$ for the i-th color band. A separate Russian roulette for each color [LP03] is an alternative. In section 5.1.4, we show how to update photon differentials to approximatively take Russian roulette into account.

3.4.1.1. Selective Storage

Nearby importons allow the relevance of a photon interaction to be estimated. Importance-sampling a corresponding storage probability leads to more interactions being recorded where their relevance to the image is high [KW00]. A storage probability can alternatively be derived from the ratio of desired and current interaction density. Desired densities based on importons for global photons and constant for caustic photons are proposed [SW00]. Storing all photon interactions first and then eliminating those too dense achieves a similar effect without requiring importons [HHS05]. Total flux is maintained by scaling the flux of stored interactions by the inverse storage probability [KW00] or distributing that of rejected interactions to others stored nearby [SW00, HHS05].

Superimposing a uniform grid on the scene allows regions of high and low interaction density to be separated [TMS04a]. Interactions are recorded in a low energy global photon map until their number in a cell exceeds a threshold. Subsequent interactions falling into the same cell are directed to a high energy map, forming concentrated clusters that can be approximated as secondary light sources.

3.4.1.2. Selective Tracing

Unless importons have been used in their construction, photon maps are view independent. Photons need to be retraced and photon maps reconstructed only when light sources or scene geometry change.

The cost of doing so is amortized by exploiting *coherence* between frames. Retracing after all changes have completed [PDC+03] is simplest but leads to outdated illumination in intermediate frames. The quasi-random Halton sequence allows photon paths to be sampled in coherent groups. A priority can be computed for each group from its estimated visual importance and the number of pilot photons encountering dynamic objects. This enables progressive updates by retracing the highest priority groups in each frame. Obsolete interactions are removed by following the original paths with negative flux [DBMS02]. More aggressive retracing invalidates any group for which a single pilot photon deviates from its previous path [LC04]. When no changes occur in the scene, photon interactions may be accumulated over multiple frames, exploiting coherence to increase photon map quality [GWS04].

3.4.1.3. Participating Media

The photon mapping algorithm is readily extended to the simulation of light transport in participating media [JC98]. In addition to the caustic and global photon maps, a *volume photon map* is constructed that stores *extinction* events of photons passing through the medium.

A fraction $\sigma_a(t)$ of photons is absorbed and a fraction $\sigma_s(t)$ scattered by the participating medium per unit distance traveled. With $\sigma_e(t) = \sigma_a(t) + \sigma_s(t)$ the extinction coefficient, the total differential change in the number of photons m following a path $\vec{z}(t) = \vec{x} + t\vec{\omega}$ thus is

$$\frac{dm}{dt}(t) = -\sigma_e(t)\, m(t).$$

Under the boundary condition $m(0) = 1$ for a single photon, this differential equation yields

$$m(t) = e^{-\int_0^t \sigma_e(s)\, ds}. \tag{3.15}$$

An individual photon is extinguished at exactly one distance t_e, found by importance-sampling the cumulative distribution function $1 - m(t)$. *Ray marching* locates the extinction distance by advancing through the medium in small steps Δt. The conditional probability that a photon is extinguished within an interval $[t, t+\Delta t]$ after passing through $[0, t]$ before is approximated as

$$\mathrm{P}(t_e \in [t, t+\Delta t] \mid t_e \notin [0,t]) = \frac{m(t) - m(t+\Delta t)}{m(t)} = 1 - e^{-\int_t^{t+\Delta t} \sigma_e(s)\, ds} \approx 1 - e^{-\Delta t \sigma_e(t+\Delta t)}.$$

This probability is sampled by Russian roulette, taking another step or reporting an extinction. For homogeneous media, equation 3.15 simplifies to

$$m(t) = e^{-\sigma_e t}.$$

$1 - e^{-\sigma_e t}$ can be importance-sampled without ray marching by drawing a uniform sample $y \in (0, 1]$ and transforming it to $t_e = -\sigma_e^{-1} \log y$. For inhomogeneous media, using $\Delta t = -\sigma_e^{-1}(t) \log y$ adapts the ray marching step size to local extinction. Adaptation by comparing medium properties at $\vec{z}(t)$ and $\vec{z}(t+\Delta t)$ is another possibility [Jen01]. Curved paths through a medium with inhomogeneous index of refraction are simulated by adjusting the direction after each step [GMSJ03]. Its piecewise constant approximation of the extinction coefficient makes ray marching biased. A provably unbiased alternative [Col68] is to take steps of size $\Delta t = -\sigma_{e,max}^{-1} \log y$ with $\sigma_{e,max}$ an upper bound on the extinction coefficient and decide after each step with probability $1 - \sigma_e(t+\Delta t)\, \sigma_{e,max}^{-1}$ whether to

take another step or report an extinction [WMHL65]. Regardless of sampling scheme, if a surface is reached before extinction occurs, the photon interacts with the surface.

Each extinction is stored in the volume photon map first. Then, Russian roulette decides between absorption and scattering. If the photon is scattered, a new direction is chosen by importance-sampling the phase function. Adding a second Russian roulette that decides about re-emission after absorption and importance-sampling the quantum efficiency function extends the method to inelastic scattering between wavelengths [GMAS05], allowing light transport in complex atmospheric models to be simulated [GSMA08]. In section 7.2, we show how photon differentials can be updated to approximatively take into account extinction, absorption and scattering. Concurrent work [Sch09] is limited to approximating the last two effects only.

3.4.2. Density Estimation

Photon maps allow reflected radiance to be estimated. Originally based on both mathematical transformations and intuition [Jen96], the estimator can be derived in a more rigorous way.

The relationship between reflected radiance L_r leaving a surface at a point $\vec{x}$ in direction $\vec{\omega}$ and that L_i incident from the hemisphere Ω_i above it is given by the rendering equation 2.8 as

$$L_r\left(\vec{x},\vec{\omega}\right) = \int_{\Omega_i} f\left(\vec{x},\vec{\omega}_i,\vec{\omega}\right) L_i\left(\vec{x},\vec{\omega}_i\right) \left|\vec{n}\cdot\vec{\omega}_i\right| \, d\Omega_i.$$

Inserting the definition of radiance from equation 2.2 yields

$$L_r\left(\vec{x},\vec{\omega}\right) = \int_{\Omega_i} f\left(\vec{x},\vec{\omega}_i,\vec{\omega}\right) \frac{d^2\Phi_i}{dA\, d\Omega_i}\left(\vec{x},\vec{\omega}_i\right) \, d\Omega_i.$$

Assuming the surface is planar with constant BRDF, neither Ω_i nor f vary with position $\vec{x}$, allowing differentiation with respect to area A and integration over Ω_i to be exchanged,

$$L_r\left(\vec{x},\vec{\omega}\right) \approx \frac{d}{dA}\left(\int_{\Omega_i} f\left(\vec{\omega}_i,\vec{\omega}\right) \frac{d\Phi_i}{d\Omega_i}\left(\vec{\omega}_i\right) \, d\Omega_i\right)\left(\vec{x}\right).$$

This expresses reflected radiance as the *density* of reflected intensity per unit area, providing the basis for *weighted kernel density estimation* [Par62]. When n samples of the intensity at positions $\vec{x}_j$ are known, reflected radiance at $\vec{x}$ can be estimated as the sum of their values weighted by a *smoothing kernel* K with *bandwidth* h centered around each,

$$L_r\left(\vec{x},\vec{\omega}\right) \approx \sum_{j=1}^{n} \frac{1}{h^2} K\left(\frac{\vec{x}-\vec{x}_j}{h}\right)\left(\int_{\Omega_i} f\left(\vec{\omega}_i,\vec{\omega}\right) \frac{d\Phi_i}{d\Omega_i}\left(\omega_i\right) \, d\Omega_i\right)\left(\vec{x}_j\right).$$

Individual samples are computed from photon interactions. Each interaction expresses the incidence of a flux $\Phi_{i,j}$ at a position $\vec{x}_j$ from a direction $\vec{\omega}_{i,j}$. As the flux is incident from a single direction, the BRDF can be moved outside the integral, leading to

$$L_r\left(\vec{x},\vec{\omega}\right) \approx \sum_{j=1}^{n} \frac{1}{h^2} K\left(\frac{\vec{x}-\vec{x}_j}{h}\right) f\left(\vec{\omega}_{i,j},\vec{\omega}\right)\left(\int_{\Omega_i} \frac{d\Phi_i}{d\Omega_i}\left(\omega_i\right) \, d\Omega_i\right)\left(\vec{x}_j\right),$$

$$L_r(\vec{x},\vec{\omega}) \approx \sum_{j=1}^{n} \frac{1}{h^2} K\left(\frac{\vec{x}-\vec{x}_j}{h}\right) f(\vec{\omega}_{i,j},\vec{\omega})\,\Phi_{i,j}. \tag{3.16}$$

The smoothing kernel must integrate to one [Par62]. Photon mapping adds the requirements of *radial symmetry*, $K(\vec{d}) = K(\|\vec{d}\|)$, and *local support*, $\|\vec{d}\| \geq 1 \Rightarrow K(\vec{d}) = 0$. The latter ensures the kernel is nonzero only within a radius h of query point $\vec{x}$, allowing photon interactions outside it to be ignored. Initially proposed are the uniform and cone kernels [Jen96]. The first is constant, the second decays linearly with distance according to a parameter $c \geq 1$. Quadratic decay provided by the Epanechnikov kernel [Epa69] has superior statistical properties. For $\|\vec{d}\| < 1$, these kernels are

$$K_u(\vec{d}) = \frac{1}{\pi}, \qquad K_c(\vec{d}) = \frac{c-\|\vec{d}\|}{\left(c-\frac{2}{3}\right)\pi}, \qquad K_e(\vec{d}) = \frac{1-\|\vec{d}\|^2}{\frac{1}{2}\pi}.$$

Photon interactions are collected from a sphere with radius h around the query point $\vec{x}$. Any $h > 0$ introduces *proximity bias*, blurring as the illumination estimate at a point is based on information from a region. Other forms of bias arise when the assumption that all interactions lie on the same planar surface is violated. If the surface is curved, its area is underestimated, resulting in *topological bias*. Where the surface ends at a silhouette, its area is overestimated, yielding *boundary bias*. Collecting interactions from within a sphere and assuming they lie on one surface additionally introduces the risk of capturing interactions with other nearby surfaces, leading to *illumination bleeding*.

3.4.2.1. k-th Nearest Neighbor

Using a smaller bandwidth reduces bias but increases variance. Automatic balancing of variance and bias is achieved by locating the k interactions nearest to the query point $\vec{x}$ and setting the bandwidth to $h = \|\vec{x}-\vec{x}_k\|$ [LQ65]. This yields the k-th nearest neighbor density estimator

$$L_r(\vec{x},\vec{\omega}) \approx \frac{1}{\|\vec{x}-\vec{x}_k\|^2} \sum_{j=1}^{k} K\left(\frac{\vec{x}-\vec{x}_j}{\|\vec{x}-\vec{x}_k\|}\right) f(\vec{\omega}_{i,j},\vec{\omega})\,\Phi_{i,j}.$$

Differential checking allows the value of k to also automatically be chosen. For each $\vec{x}$, k is increased until a systematic change in the estimate occurs, indicating an illumination boundary at the current bandwidth [Jen01]. The difficulty of detecting systematic changes in practice is addressed by assuming that an initial estimate based on the k_{min} nearest interactions is unbiased and then performing a binary search in $[k_{min}, k_{max}]$ for the nearest illumination boundary, using the central limit theorem to differentiate between variance and bias [Sch03].

Topological and boundary bias are reduced by explicitly locating nearby surfaces and calculating a separate kernel support area for each interaction [HP01]. Using a spatial index lowers the cost of this method [HP02a]. Constructing separate photon maps for groups of surfaces and collecting interactions only from the photon map corresponding to the surface on which $\vec{x}$ lies counters light leaking [LC03]. A more uniform distribution of interactions can be ensured by applying a diffusion process to their positions, constraining motion to avoid crossing illumination boundaries [SJ09].

Progressive photon mapping [HOJ08] reformulates density estimation such that flux is accumulated at each query point with continuously shrinking bandwidth while photons are being traced. This removes the need to construct and store photon maps. Follow-up work [HJ09] computes density estimates for query areas instead of points, simulating depth of field.

Motion blur is simulated by collecting the photon interactions nearest in space and time. The kernel support becomes the product of a disc in space and an interval in time [CJ02].

Storing *photon rays* instead of the interactions at their ends and collecting all rays that intersect a planar kernel of fixed bandwidth around $\vec{x}$ eliminates boundary bias. Extending rays so that occlusion by the surface containing $\vec{x}$ is ignored reduces topological bias. Query coherence is exploited by maintaining a hierarchy of bounding spheres. During retrieval, the smallest sphere enclosing $\vec{x}$ is located and the hierarchy rooted in it reconstructed, retrieving only the rays relevant within that sphere [LURM02]. A combination of photon rays and k-th nearest neighbor density estimation is obtained by retrieving the k rays passing nearest to $\vec{x}$, using the maximum of the distance to a ray and to its intersection with the tangential plane at $\vec{x}$ as distance metric [HBHS05].

Collecting the k nearest photon interactions or rays is expensive as it requires a set of *candidates* to be retrieved from a region around the query point and a priority queue to be constructed that identifies those nearest [Jen01]. For small k, storing more than k interactions per photon map leaf allows a tight upper bound on $h = \|\vec{x} - \vec{x}_k\|$ to be computed after visiting a single leaf, immediately culling all nodes outside it [VBS99]. Coherence between queries is exploited by finding photon map nodes relevant to $\vec{x}$ and a small region around it. If the next query point falls into the region, the set of nodes is reused without traversal [HBHS05]. Alternatives that eliminate the need to retrieve and classify candidate interactions are bandwidth selection based on path differentials originating at the virtual camera [SW01] and on ray differentials tracked with final gather rays [CLF+03].

3.4.2.2. Histogram

A simpler form of density estimation is *histogramming*. Instead of centering a kernel around each, contributions are collected in predefined bins. Final gather rays importance-sample directions by accumulating the flux of the k nearest photon interactions reflected by the BRDF in directional bins [Jen95]. The relevance of directions to the image is computed analogously from the reflected contributions of the k nearest importons [PP98].

3.4.2.3. Variable Kernels

Selecting a bandwidth per photon interaction instead of per query point leads to *variable kernel density estimation* [BMP77]. The estimator obtained from equation 3.16 by inserting variable bandwidths is

$$L_r(\vec{x}, \vec{\omega}) \approx \sum_{j=1}^{n} \frac{1}{h_j^2} K\left(\frac{\vec{x} - \vec{x}_j}{h_j}\right) f(\vec{\omega}_{i,j}, \vec{\omega})\, \Phi_{i,j}. \tag{3.17}$$

This method computes the contributions of all photon interactions independently while retaining the ability to balance variance and bias by varying the bandwidths h_j. Given an upper bound h on h_j, only photon interactions within a radius h around query point $\vec{x}$ are relevant. These may be collected by traversing a photon map or by splatting a kernel for each.

An early technique [SWH+95] subsequently adopted for splatting on a GPU [SB97] computes a bandwidth per surface from its area and the number of photon interactions. Iterative refinement per interaction is added by estimating variance and bias from the number of splats contributing to each pixel and reducing bandwidths where variance permits [LP03]. Clipping kernel support areas against the surfaces they are splat on further reduces boundary bias. An alternative with better balancing of variance and bias is bandwidth selection by least-squares cross-validation on the interactions [LP02].

Additional boundary bias reduction results from *extended triangles* around surface silhouettes that store interactions without influencing photon paths, tracing beyond interactions in corners and clipping kernel support areas against the plane from which photons are emitted. A simpler option is to base bandwidths on the distances between interactions and the virtual camera [KBW06].

Keeping track of neighboring photons allows splatting of joint kernels with *anisotropic* support regions as spanned by the photon rays [IDN02] or interactions [UPSK08]. Alternatively, a bandwidth may be extracted for each photon from the footprint spanned by neighboring interactions [Wym08]. In a geometrically simple scenario, the footprint can be estimated directly [WD06]. Photon differentials provide more general and robust footprint calculation [SFES07]. Combined with the surface normal, a kernel support region in the form of a *skewed ellipsoid* stretching along illumination boundaries is obtained. Originally applicable to specular interactions only, a subsequent extension [Sch09] approximatively takes diffuse interactions and Russian roulette into account. In chapter 5, we describe our extension that offers alternative solutions for these effects.

Using *inverse photon path probabilities* to compute bandwidths is proposed when splatting photon rays to final gather points. The probabilities are estimated from the spreading of paths during propagation and the sample distributions at surface interactions. A dampened adaptation of kernel support areas to the probabilities avoids excessive bandwidths [HHK+07]. We use these ideas in chapter 5 when extending photon differentials and computing anisotropic kernel support regions for photon interactions. Other applications to bandwidth selection at photon interactions combine the concept with compressing kernels along the normal to reduce light leaking and topological bias [ML09] or bandwidths calculated per surface [YWC+10].

Reverse photon mapping [HHS05] constructs a kd-tree over photon interactions and calculates bandwidths from the interaction densities in its leaves. Another kd-tree is then constructed over final gather points and traversed by the kernels, splatting their contributions. An implementation in custom hardware exists [SF07a]. This reformulation provides an algorithmic improvement during kd-tree traversal as the number of photon interactions is significantly lower than that of final gather points. However, a subsequent analysis [SF07b] reveals that the main bandwidth cost occurs during kd-tree construction, making reverse photon mapping algorithmically slower overall.

When importance-sampling final gather ray directions, variable kernel density estimation is possible by centering a kernel on the hemisphere of directions around each nearby photon interaction [HP02b].

3.4.2.4. Participating Media

Photon mapping assumes radiance is invariant along straight lines. When a participating medium is present in the scene, radiance transfer follows equations 2.18 and 2.19 instead, becoming subject to extinction, in-scattering and medium emission.

Shadow rays are affected by extinction only. With transmittance τ accumulating the effects of extinction along a ray, the radiance returned by a shadow ray for an unoccluded light source is

$$L_i\left(\vec{x}, -\vec{\omega}\right) = \tau\left(0, t_{max}\right) L_e\left(\vec{r}\left(\vec{x}, -\vec{\omega}\right), \vec{\omega}\right).$$

The transmittance of a homogeneous medium $\tau\left(0, t_{max}\right) = e^{-\sigma_e t_{max}}$ is evaluated directly. For an inhomogeneous medium, ray marching is used. Taking s_{max} steps, the medium is approximated as locally homogeneous in each. The s-th step advances from parameter value t_s to t_{s+1} with step size $\Delta t_s = t_{s+1} - t_s$. Given starting point $t_1 = 0$, the transmittance is approximated as

$$\tau\left(0, t_{max}\right) = e^{-\int_0^{t_{max}} \sigma_e(t)\, dt} \approx e^{-\sum_{s=1}^{s_{max}} \Delta t_s \sigma_e(t_s)} = \prod_{s=1}^{s_{max}} e^{-\Delta t_s \sigma_e(t_s)}. \tag{3.18}$$

Any other rays are subject to the full range of effects modeled by equations 2.18 and 2.19,

$$\begin{aligned} L_i(\vec{x}, -\vec{\omega}) = {} & \tau\left(0, t_{max}\right) L_o\left(\vec{r}\left(\vec{x}, -\vec{\omega}\right), \vec{\omega}\right) \\ & + \int_0^{t_{max}} \tau\left(0, t\right) \left(\sigma_s\left(t\right) \int_{\Omega_{4\pi}} p\left(t, \vec{\omega}_i, \vec{\omega}\right) L\left(\vec{z}\left(t\right), \vec{\omega}_i\right)\, d\Omega_i + \epsilon\left(t, \vec{\omega}\right)\right) dt. \end{aligned} \tag{3.19}$$

Ray marching is used again. Approximating the medium as homogeneous and the incident radiance field as constant in every step and using the approximation of transmittance from equation 3.18,

$$\begin{aligned} L_i(\vec{x}, -\vec{\omega}) \approx {} & \prod_{s=1}^{s_{max}} e^{-\Delta t_s \sigma_e(t_s)} L_o\left(\vec{r}\left(\vec{x}, -\vec{\omega}\right), \vec{\omega}\right) \\ & + \sum_{s=1}^{s_{max}} \Delta t_s \prod_{r=1}^{s-1} e^{-\Delta t_r \sigma_e(t_r)} \left(\sigma_s\left(t_s\right) \int_{\Omega_{4\pi}} p\left(t_s, \vec{\omega}_i, \vec{\omega}\right) L\left(\vec{z}\left(t_s\right), \vec{\omega}_i\right)\, d\Omega_i + \epsilon\left(t_s, \vec{\omega}\right)\right). \end{aligned} \tag{3.20}$$

The first term is evaluated by computing L_o at the nearest surface seen using the original photon mapping algorithm, then attenuating it by marching along the ray to accumulate transmittance. For the second term, radiance in-scattered from a sphere of directions $\Omega_{4\pi}$ must be gathered after each ray marching step. The problem is converted to one of density estimation. Section 2.1.4 provides the relationship $\left(\frac{dL}{dt}\left(\vec{z}\left(t\right), \vec{\omega}\right)\right)_{extinction} = -\sigma_e\left(\vec{z}\left(t\right)\right) L\left(\vec{z}\left(t\right), \vec{\omega}\right)$ between radiance and its differential change due to extinction by absorption and out-scattering, allowing the radiance in the integrand to be expressed in terms of extinction on a ray traveling through $\vec{z}\left(t_s\right)$ in direction $\vec{\omega}_i$ with parameter t_i. Combined with the definition of radiance from equation 2.1, an identity is obtained,

$$-\sigma_e\left(t_s\right) L\left(\vec{z}\left(t_s\right), \vec{\omega}_i\right) = \left(\frac{dL}{dt_i}\right)_{extinction}\left(\vec{z}\left(t_s\right), \vec{\omega}_i\right) = \left(\frac{d^3\Phi}{dt_i\, dA_i^{\perp}\, d\Omega_i}\right)_{extinction}\left(\vec{z}\left(t_s\right), \vec{\omega}_i\right) = \left(\frac{d^2\Phi}{dV\, d\Omega_i}\right)_{extinction}\left(\vec{z}\left(t_s\right), \vec{\omega}_i\right). \tag{3.21}$$

After dividing by $-\sigma_e\left(t_s\right)$, this identity is inserted for $L\left(\vec{z}\left(t_s\right), \vec{\omega}_i\right)$ in equation 3.20. Approximation as a locally homogeneous medium with constant radiance field allows differentiation with respect to volume V and integration to be swapped. Coefficients σ_s and σ_e^{-1} further combine into albedo α according to equation 2.20 and the second term in equation 3.20 becomes

$$\sum_{s=1}^{s_{max}} \Delta t_s \prod_{r=1}^{s-1} e^{-\Delta t_r \sigma_e(t_r)} \left(\alpha\left(t_s\right) \frac{d}{dV} \left(\int_{\Omega_{4\pi}} p\left(\vec{\omega}_i, \vec{\omega}\right) \left(-\frac{d\Phi}{d\Omega_i}\left(\vec{\omega}_i\right)\right) d\Omega_i\right)_{extinction}\left(\vec{z}\left(t_s\right)\right) + \epsilon\left(t_s, \vec{\omega}\right)\right).$$

In-scattered radiance is now expressed as the density of in-scattered intensity per unit volume, enabling weighted kernel density estimation. Each of the n extinction events in the volume photon map corresponds to a change in flux by $-\Phi_{i,j}$ on a ray traveling in direction $\vec{\omega}_{i,j}$ at a position $\vec{z}_j$ due to extinction in the participating medium. A sample of in-scattered intensity is obtained from it analogously to section 3.4.2 by exploiting the single incident direction to move the phase function outside the integral, then integrating the flux to $-\Phi_{i,j}$, yielding

$$\approx \sum_{s=1}^{s_{max}} \Delta t_s \prod_{r=1}^{s-1} e^{-\Delta t_r \sigma_e(t_r)} \left(\alpha\left(t_s\right) \sum_{j=1}^{n} \frac{1}{h^3} K\left(\frac{\vec{z}\left(t_s\right) - \vec{z}_j}{h}\right) p\left(\vec{\omega}_{i,j}, \vec{\omega}\right) \Phi_{i,j} + \epsilon\left(t_s, \vec{\omega}\right)\right). \tag{3.22}$$

This equation is evaluated by ray marching and kernel density estimation [JC98]. Starting at the end of the ray, in each ray marching step radiance accumulated so far is attenuated by the next term of equation 3.18, in-scattered radiance is computed by k-th nearest neighbor density estimation and the radiance emitted by the medium is added. Similarly to section 3.4.2.1, the k nearest extinction events are found and a symmetric kernel with bandwidth $h = \|\vec{z}(t_c) - \vec{x}_k\|$ is centered around each. A simple uniform kernel is proposed with $K(\vec{d}) = \frac{3}{4\pi}$ for $\|\vec{d}\| < 1$ and zero otherwise. Only the second and later extinction event are stored for each photon, representing *multiple scattering*. Radiance in-scattered directly from the light sources is computed separately by casting shadow rays from $\vec{z}(t_s)$ toward the lights after every ray marching step.

Ray marching backwards through the medium and updating the accumulated radiance corresponds to a *numerical integration* of the radiative transfer equation 2.17 underlying equation 3.19. Higher order methods increase precision so that Euler integration [GMSJ03] is outperformed by Dormand-Prince [GMAS05]. Extending density estimation to evaluate the contributions of an extinction event to all simulated wavelengths via the quantum efficiency function improves quality when simulating inelastic scattering in a complex atmospheric model [GSMA08].

The *beam radiance estimate* [JZJ08a] is a more efficient reformulation of volumetric photon mapping. A bandwidth is calculated for each extinction event from the estimated distance to its k-th-nearest neighbor. Rays then traverse the photon map and gather in-scattered radiance using the variable kernel method. All extinction events whose kernel is intersected by a ray are collected in a single query, eliminating the need for ray marching other than to accumulate transmittance. Density estimation occurs over the two-dimensional area orthogonal to the ray with a planar kernel centered around each event. Although a mathematical derivation is provided, the resulting physical units are incorrect. We rederive the beam radiance estimate with correct units in chapter 7 and show how to eliminate its k-th nearest neighbor preprocessing pass by using photon differentials.

3.4.3. Photon Map

Interactions are scattered throughout the scene. A photon map organizes them in the nodes of a spatial index, accelerating their retrieval during density estimation.

3.4.3.1. kd-Tree

The spatial index initially proposed [Jen95] is a *left-balanced* kd-tree storing one photon interaction per node. This corresponds to primitive median splits with splitting planes shifted to the right such that the left child contains a complete tree. As the kd-tree has no holes, its nodes can be laid out in heap order to implicitly encode the hierarchy. Separate kd-trees are constructed for caustic and global photons [Jen96]. Importons are indexed by an additional kd-tree [PP98, SW00]. For photons distributed in space and time, a kd-tree subdividing either space or space-time can be used [CJ02].

Storing AABBs at all nodes and placing photon interactions in hash tables at the leaves improves node culling [VBS99]. Partitioning surfaces yields a smaller kd-tree over the interactions with each subset. Automatic division into connected groups of triangles with smoothly varying normals is used to reduce light leaking [LC03]. Manual partitioning into groups of surfaces [LC04] or objects [CB04] ensures that the kd-trees do not exceed memory limits.

Removing the balancing requirement makes child pointers necessary but enables flexible placement of splitting planes. Photon interactions can progressively be inserted as they occur [SW00]. Splitting

at the spatial median accelerates construction while also improving retrieval performance [GWS04]. Reverse photon mapping uses sliding-midpoint subdivision, shifting the spatial median splitting plane to the nearest interaction if one of the child nodes would be empty otherwise. Interactions are stored at leaves only [HHS05] or at all nodes [SF07a].

Heuristic construction analogous to section 3.2.3.1 further accelerates retrieval. At each node P, the pair of children L, R is chosen that minimizes expected retrieval cost. When storing an interaction at every node, the cost C_T of traversing a node and that C_I of testing a primitive are identical. With $C_T = C_I$, this value has no influence on the optimization and equation 3.12 simplifies to

$$C_P(L,R) = p_P + p_L\, n_L + p_R\, n_R. \tag{3.23}$$

The probabilities p_P, p_L, p_R of nodes being visited are estimated by the *voxel volume heuristic (VVH)* [WGS04]. During density estimation, candidate interactions within a maximum distance h_{max} of query point $\vec{x}$ are initially retrieved. After k have been found, the search radius is progressively reduced to the distance of the currently k-th nearest interaction. The probability that a node must be visited is therefore bounded above by and can be approximated as the probability of $\vec{x}$ falling either inside the node or within a distance h_{max} of its bounds. This region is further approximated as the node AABB extended by h_{max} on each side. Assuming a uniform distribution of query points throughout the scene, the probability is proportional to the volume of the region. With S the scene bounding box and $\mathrm{V}_{\pm h_{max}}$ the volume of an AABB extended by h_{max} on every side, the probability that a node N has to be visited is thus approximated as

$$p_N \approx \frac{\mathrm{V}_{\pm h_{max}}(N)}{\mathrm{V}_{\pm h_{max}}(S)}. \tag{3.24}$$

A candidate splitting plane passes through each interaction on every axis. As in section 3.2.3.1, optimal $O(n \log n)$ construction complexity is achieved by progressively splitting sorted candidate lists. Recursion terminates and a leaf is constructed whenever a single primitive is reached. Efficient construction of a kd-tree with approximated VVH in CUDA is demonstrated by modifying a method approximating the SAH [ZHWG08].

We adapt the VVH from kd-trees to BVHs in section 6.1 and show that retrieval performance does not significantly exceed that obtained by applying faster LBVH construction to photon interactions.

3.4.3.2. Alternatives and Extensions

A uniform grid is simpler to construct and traverse than a kd-tree [WKB+02]. However, a study of photon map cache efficiency [SCL05, SCL08] shows it to perform worst among different data structures. For small cache lines, a kd-tree is best and for large cache lines, a kdB-tree [Rob81]. A reordering of final gather points to increase their coherence is also analyzed and found to have higher impact overall than the choice of data structure. Sorting along a spatial Hilbert curve yields the greatest acceleration. Partial reordering [SF07b] stops the sorting process when further overheads are not likely to be recouped.

For photon rays, a kd-tree is constructed by spatial median subdivision with each leaf referencing the rays that intersect it [HBHS05]. To add directional culling, interspersed nodes separate rays facing forward and backward of a surface. Rays for surfaces with similar normals are included in each child, extending the culling to these. Construction is *lazy* with nodes subdivided on query and

collapsed when a memory limit is reached. The bounding sphere hierarchy used to exploit coherence between queries [LURM02] is a similarly lazily constructed data structure.

While k-th nearest neighbor density estimation operates on point data, the variable kernel method uses kernels with known bandwidths. This allows a tight AABB to be computed around each kernel and motivates our investigation of a BVH photon map in chapter 6, replacing space partitioning with primitive partitioning. Since there is no need to calculate separate bandwidths from the k nearest caustic and global interactions, a single photon map is used for both.

3.4.3.3. Participating Media

Extinction events in a participating medium are indexed by an additional left-balanced kd-tree [JC98]. The beam radiance estimate employs this kd-tree as a temporary data structure. For each extinction event, a kernel bandwidth is calculated by retrieving the $m \ll k$ nearest events and estimating the distance to the k-th nearest from these. A BVH is subsequently constructed over the kernels, reusing the hierarchical structure of the kd-tree by adding AABBs to its nodes [JZJ08a].

In chapter 7, we show how to derive bandwidths for extinction events from photon differentials. No temporary kd-tree or approximative k-th nearest neighbor retrieval is required. Instead of being an additional data structure, our BVH replaces the kd-tree.

3.5. Simplification

The cost of physically based rendering is successfully reduced by efficient ray tracing and density estimation. Reordering visible surface points so that those sharing a BRDF and thus having common processing steps are considered together realizes additional coherence benefits [HLJH09]. Simulating light transport remains computationally demanding, however. When interactive frame rates are a priority, the focus may be shifted from physical accuracy to *plausibility*, synthesizing visually pleasing *simplifications* of the most significant optical phenomena. Simplification is not a goal of this thesis but provides the important context of alternative approaches to interactive physically based rendering.

Current GPUs are optimized for rasterization [SA10]. This is equivalent to the tracing of primary rays with a common origin and deterministic, uniform directions [HSHH07]. Combined with shadow maps [Wil78] or shadow volumes [Cro77], rasterization computes direct illumination and provides a baseline for adding simplifications of other effects.

Indirect illumination arriving from diffuse surfaces varies smoothly. This enables the simplification of sampling it more sparsely, either by synthesizing and upsampling a lower resolution image or by permitting higher variance and interpolating between pixels. A *discontinuity buffer* [Kel97b] receiving information about the surface seen at each pixel prevents interpolation across silhouettes and sharp edges. *Interleaved sampling* [KH01] formalizes the concept by tiling an irregular sampling pattern over the image that partitions it into interleaved grids of identically sampled pixels. Interpolation over a neighborhood effectively reconstructs the complete pattern at each pixel. Reordering operations so that the pixels of a grid are processed together yields coherent sampling [SIMP06].

3.5.1. Virtual Point Lights

Point lights are the simplest light source type. By approximating complex illumination with point light sources, its computation is simplified and accelerated.

3.5.1.1. Instant Radiosity

The *instant radiosity* algorithm [Kel97a] simulates indirect illumination by distributing *virtual point lights (VPLs)* on surfaces. A small number of paths are traced from the light sources, placing VPLs at the surface hit points. Diffuse surfaces yield isotropic VPLs, glossy surfaces spot VPLs. Illumination is gathered by blending images lit by the VPLs. Plausible color bleeding results but the small number of VPLs blurs illumination boundaries. When only light sources change, images lit by VPLs can be reused across frames [SSSK04]. The validity of each VPL in the current frame is determined by tracing a ray from the updated light source position to the first interaction on its path. If no occluder is found, the image lit by the VPL is reweighted by its current flux and reused.

For computation on a cluster of CPUs, the original rasterization with shadow volumes is replaced by ray tracing with shadow rays toward the VPLs [WKB+02]. Interleaved sampling reduces the number of shadow rays required. Follow-up work improves coherence by distributing image tiles instead of interleaved pixel grids to the cluster nodes [BWS03] and adds a preprocessing pass to estimate light source importance, assigning each light a corresponding number of VPLs and omitting shadow rays where importance is low [WBS03].

The visual significance of illumination diminishes with *bounce depth*, the number of interactions between light and surface. Concentrating on *single-bounce* indirect illumination is therefore a plausible simplification [TL04]. *Reflective shadow maps (RSMs)* [DS05] accelerate VPL construction in this case. Shadow map texels sampled by a precalculated pattern are directly interpreted as VPLs. A simple shader gathers their illumination without detecting occlusion. Follow-up work uses splatting instead [DS06]. VPLs are placed at a subset of RSM texels by importance-sampling the product of flux, BRDF and relevance to the image, the latter estimated by *ambient occlusion* [Lan02] from nearby surfaces. An ellipsoid bounding the region of significant contribution is rasterized for each VPL and illumination is again computed in a shader that ignores occlusion.

Sampling the scene from the virtual camera to distribute additional VPLs and assigning these *negative* flux that cancels extraneous illumination approximates the effect of VPL occlusion [CS07]. Ambient occlusion in *image space* [SA07] provides a fast approximation of nearby occluding faces. Interpreting these surface samples as VPLs, single-bounce indirect illumination is estimated [RGS09]. False occlusion is reduced by using additional information about surfaces not seen from the virtual camera, obtained by depth peeling [Eve01] or from images synthesized for other virtual camera positions. Taking multiple samples in the same direction counters missed occlusion.

The splatting of VPL illumination is accelerated by iteratively subdividing a single large splat at surface and illumination discontinuities to construct a hierarchy of smaller splats with approximatively uniform contributions. These are rasterized into a multiresolution buffer such that each splat covers only a single pixel, reducing bandwidth cost [NW09]. The results are upsampled and blended. Follow-up work [NSW09] replaces iterative subdivision with a single step in which splats at all resolutions are found. A single quad is rasterized, culling nonexistent splats using the GPU stencil buffer and gathering illumination in a shader. VPLs are distributed by similarly subdividing an RSM at surface discontinuities. In subsequent work [NW10], alternative splat subdivision criteria are suggested that better detect discontinuities.

Distributing VPLs by the Metropolis-Hastings algorithm is an alternative to both tracing paths and sampling RSMs [SIP07]. *Virtual pinhole cameras* are similar in concept to VPLs but approximate caustics due to a single specular interaction instead. The cameras are placed by uniformly sampling specular surfaces. Their contributions are computed in a shader that reflects the query direction and

looks up incident illumination in an environment map. A footprint for the look-up is calculated from the partial derivatives of the reflected direction, similarly to ray differentials [WS03].

3.5.1.2. Clustering

In the *lightcuts* algorithm [WFA+05], all illumination is approximated by a large number of VPLs. A hierarchy of VPL clusters is constructed with a representative VPL chosen for each by importance-sampling flux. At a query point, a *cut* through this *light tree* is iteratively refined, replacing any node whose approximation by the representative VPL introduces error above a perceptual masking threshold with its children. Only the contributions of the representative VPLs in this cut are then evaluated. Precalculating cuts at the corners of image tiles accelerates the process. Follow-up work adds an analogous hierarchy of query points [WABG06]. A single cut through the cross product of light tree and query point tree is found. Motion blur is simulated by distributing VPLs and query points in time, scattering in a homogeneous participating medium by placing them in the medium at distances proportional to its density.

Coherence between the frames of a predefined animation is exploited by clustering both VPLs and frames [HVAPB08]. One image of the scene is rasterized per cluster, illuminated by its representative VPL in the representative frame. The pixels are then reweighted and reprojected to all frames spanned by the cluster. Clustering VPLs into *virtual area lights* allows their visibility to be efficiently estimated by soft shadow mapping [DGR+09].

3.5.2. Photon Mapping

A simplified photon mapping algorithm that focuses on the phenomenon of sharp caustics is obtained by emitting caustic photons only. As instant radiosity approximates smoothly varying indirect illumination, the two algorithms are complementary and their results can be combined [WKB+02]. Further simplification potential exists in approximating photon mapping calculations.

3.5.2.1. Ray and Photon Tracing

Simplified surface representations improve ray tracing and thus also photon mapping efficiency. A *volumetric* representation records which cells of a uniform grid are intersected by surfaces. Reflection and refraction of viewing rays are approximated by mapping slices of the grid onto a mesh and displacing its vertices [IDN02]. *Geometry images* store the vertices of a triangle mesh in texture map color channels [GGH02]. GPU ray tracing is demonstrated with stackless traversal of a BVH constructed by recursively combining four texels into a parent [CHCH06]. Simplifying surfaces by reducing the tessellation level is suggested where ray differentials span a large footprint [CLF+03] or when computing indirect illumination in general [TL04]. A *point based* surface representation accelerates final gather [REG+09]. Each point corresponds to a small disc on a surface. A hierarchy of point clusters with larger discs as representatives is constructed. To gather illumination, a cut through the hierarchy is found and rendered into a micro-buffer such that each disc projects to a single pixel. A warped perspective provides BRDF importance sampling.

For caustic photons that propagate through a small part of the scene only, approximative tracing in image space is possible. After rasterizing diffuse surfaces as seen from the light into a texture map, a specular object is rasterized in the same perspective, each pixel corresponding to the emission of one photon. Specular interactions are computed in a shader and the subsequent hit points found by

iterative search in the texture map of diffuse surfaces [SKP07]. Rasterizing the front and back surfaces of a specular object into separate texture maps allows refraction at its back to also be approximated, either in a single step [Wym05] or iteratively for higher precision [LWL06]. Adaptive emission reduces the number of photons by subdividing the view of the scene from a light until the photons emitted either all miss the specular object or sufficiently converge [WN09].

Another method for image space tracing iteratively searches environment maps of the surrounding diffuse and specular surfaces rasterized from the center of a specular object [SKALP05]. Environment maps around multiple points distributed in the scene extend this technique to interactions with occluded surfaces and to global photon mapping [YWC+10]. One more approach uses depth peeling to obtain information about occluded surfaces, tracing rays by rasterizing lines that cover all pixels with potential hit points and searching for the hits in a shader [KBW06]. A recent survey [SKUP+09] provides an overview of such approximative techniques.

3.5.2.2. Density Estimation

Retrieving an approximate candidate set from a hashed photon map [MM02] yields a simplification of k-th nearest neighbor density estimation. A uniform grid photon map leads to further simplification as its cells can be traversed in distance order, collecting an approximation of the k nearest interactions without constructing a candidate set first [PDC+03]. Ranged search also removes the need to classify candidates, approximating the bandwidth h by *binning* nearby interactions into distance ranges and iteratively subdividing the range that contains the k-th nearest instead [ZHWG08]. When splatting kernels, rasterizing them into a multiresolution buffer at several predefined bandwidths allows k-th nearest neighbor density estimation to be approximated. For each query point, the buffer resolution is read at which approximately k splats contribute [WD06].

The use of a fixed bandwidth [WKB+02] simplifies variable kernel density estimation by forgoing any automatic balancing of variance and bias at different query points. On a GPU, storing interactions in texture maps allows a simple shader to collect those within a fixed bandwidth in image space [LC04] or texture space [CSKSN05]. Splatting kernels into the image [KBW06] or texture maps [SKP07] achieves equivalent results. For variable bandwidths, combining convergent interactions into a single splat and rasterizing larger splats at coarser levels of a multiresolution buffer reduces the memory bandwidth required [Wym08].

3.5.2.3. Photon Map

An approximative alternative to the kd-tree are multiple hash tables with randomized hash bucket boundaries that index blocks of interactions [MM02]. A different insertion order in each table and displacement of the most frequently referenced block on collision reduce the risk of blocks becoming unreferenced. When constructing photon maps on a GPU, early models with limited programmability permit only simple data structures. A uniform grid of fixed capacity cells is proposed [PDC+03]. Should a cell become full, the flux of subsequent interactions falling into it is redistributed to those already stored. Even simpler is a texture map with capacity for a single interaction per texel [CSKSN05]. If multiple photon interactions fall onto the same texel, flux is accumulated or one of the interactions chosen as a representative.

When splatting kernels, no retrieval of nearby photon interactions is required and no spatial index needs to be constructed over these.

3.5.3. Object Space Interpolation

Employed in *object space*, the simplification of sparsely sampling and interpolating slowly varying illumination accelerates its computation at scene surfaces.

3.5.3.1. Caching

A caching scheme *lazily* samples illumination. For each query point $\vec{x}$, interpolation from *cache records* valid at $\vec{x}$ is attempted first. Only if an insufficient number are found is a sample computed and stored in a new cache record. First proposed is *irradiance caching* [WRC88] for diffuse surfaces. Validity regions and interpolation weights are calculated from local surface complexity and changes in normal direction, concentrating records along edges and in corners where illumination discontinuities exist. Estimating *rotational* and *translational gradients* enables higher order interpolation [WH92]. This technique is used in the original photon mapping algorithm [Jen96]. Modifications of the validity increase it in corners [TL04] or reduce it where the query point is closer to the virtual camera than the cache record [SCL08]. Transforming cache records with the dynamic objects they lie on allows approximative reuse across frames [TMS04b]. Irradiance contributions are stored for individual sampling strata and a subset selected by importance-sampling age is updated per frame.

If the BRDF at a query point is not diffuse, the entire incident radiance field is required to compute outgoing illumination. *Radiance caching* [KGPB05] records incident radiance in *spherical* or *hemispherical harmonics* [GKPB04] with 9 harmonic coefficients known to be sufficient for slowly varying illumination [RH01]. Cache records are rotated into the local coordinate frame at $\vec{x}$, removing the need for rotational gradients. Stratification of the samples used to generate records is added in follow-up work [KGBP05] and the precision of gradients improved. When reusing cache records across frames, estimating *temporal* validity and gradients increases interpolation quality [GBP07]. Storing quadtrees of directional contributions at the records allows directions to also lazily be sampled [GK09]. The BRDF at a query point is importance-sampled and the decision whether to interpolate or compute a new sample made for each direction separately.

Splatting cache records is more amenable to a GPU. New records are generated after splatting where accumulated interpolation weights lie below a threshold [GKBP05]. Reducing the validity of existing records where perceptible discontinuities are detected improves quality [KBPŽ06]. Validity is furthermore reduced when gradients are large or local geometric complexity is higher at neighboring records. Reducing validity only across discontinuities preserves interpolation along these [HMS09]. The lightcuts algorithm is used in this work with separate caches for direct and indirect illumination. When simulating light transport in *hair*, a separate cache per strand is suggested [MM06].

Radiance caching for participating media [JDZJ08] separately caches single scattering from lights, from other surfaces and multiple scattering. Illumination is gathered by ray marching, interpolating after each step with exponential weights that approximate attenuation. Validity is computed from relative gradient magnitudes. Follow-up work [JZJ08b] provides translational gradients for irradiance caching at surfaces in a participating medium, taking into account the influence of the medium.

When using control variates to reduce sampling variance, incident radiance and its difference from a prior estimate are cached in directional bins. At a query point, this information is interpolated and new samples are used to improve the resulting estimate only [PBSP08]. In follow-up work [PWP08], the idea is applied to participating media, caching in-scattered radiance.

3.5.3.2. Precalculation

Sampling illumination *a priori* decouples sampling and interpolation but requires *precalculation* points to be selected by another method. Mesh vertices [Gou71], surface patches [GTGB84] and texture map texels [Arv86] are simple choices. Using the vertices of a mesh with independent tessellation enables refinement over consecutive frames. Surfaces selected by importance-sampling the age of and difference between the samples at their vertices are updated by tessellating further or recomputing the samples [TPWG02]. Depending on the BRDF, outgoing radiance toward the virtual camera or irradiance is precalculated. Reuse across frames is proposed for texture maps of irradiance obtained by photon mapping [LC04]. The update order is chosen by estimating accumulated illumination changes since the last update from photon interactions.

Final gather for photon mapping is accelerated by precalculating irradiance at photon interactions and retrieving it at final gather points from the nearest interaction with similar surface normal [Chr99]. No normal comparison is required when groups of surfaces with smoothly varying normals have separate photon maps [LC03]. A hierarchy over the interactions with interpolated irradiance at inner nodes allows retrieval by footprints obtained from ray differentials [CB04]. Precalculating irradiance only for a cut through a hierarchy chosen such that its nodes do not span illumination discontinuities reduces cost [WWZ+09]. Final gather is further performed at the centers of query point clusters only, using these as precalculation points of incident radiance expressed in spherical harmonics.

The latter is also proposed for final gather from radiosity patches [AFO05]. Slowly varying radiance incident from distant surfaces is precalculated at query point clusters. Radiance incident from nearby surfaces is computed per query point, ignoring mutual occlusion and approximating the occlusion of distant surfaces. Classifying each patch individually [SSS02] ensures that irradiance is precalculated for those with slowly varying contributions only. Precalculation points are arranged in a perspective grid covering the view frustum. Instant radiosity is accelerated by omitting shadow rays where VPL visibility precalculated for randomized surface regions is uniform [GWS05].

Precalculating irradiance in directional bins at grid points allows interpolation also for small objects subsequently added to the scene [GSHG98]. A uniform grid is used with finer grids embedded where objects exist. When only the virtual camera changes, incident radiance precalculated in directional bins can be reused [Wal05]. Precalculation points randomly distributed on surfaces are proposed. If light sources may change as well, indirect illumination can efficiently be updated and splat into the image by precalculating a matrix of direct-to-indirect transfer coefficients for a hierarchy of random surface points [LZT+08].

3.5.3.3. Iterative Propagation

When illumination is to be sampled at selected precalculation points, propagating radiance between these only applies the same simplification to light transport. The original radiosity algorithm solves a system of linear equations encoding links between diffuse surface patches [GTGB84]. *Progressive radiosity* [CCWG88] obtains the solution iteratively by propagating the exitance of the patch with the largest remaining flux in each iteration. Adding photons emitted along the links allows paths with specular interactions to also be simulated [GDW00]. Propagation along links between mesh vertices is analogous to that between surface patches [STK08]. Asynchronous updates are suggested, rasterizing the most recent complete solution in each frame. Rasterizing the scene from a patch and computing its contributions to other patches in a shader requires no explicit link calculation [CHL04].

For a uniform grid of precalculation points, rasterizing environment maps around these computes links [NPG03]. Incident radiance expressed in spherical harmonics at the grid points is propagated iteratively. Embedding finer grids around the virtual camera concentrates samples nearby [KD10]. Intensity and a coarse representation of nearby geometry are stored in spherical harmonics. During iterative propagation to neighboring grid cells, the simplified surface representation yields a rough approximation of the fractional occlusion between them.

Outgoing radiance in directional bins at glossy surface patches is iteratively propagated on a GPU by rasterizing the patches in precalculated order for each direction [MKS07]. A reformulation of the rendering equation with *implicit* visibility iteratively propagates incident radiance and *antiradiance* to directional bins at all surfaces on its path. Occlusion is accounted for by the antiradiance [DSDD07].

3.5.4. Participating Media

Several algorithms address light transport in participating media by focusing on specific scenarios. Accumulating emitted and in-scattered radiance by recursive ray marching from the virtual camera is viable for fire and smoke as low albedo leads to shallow recursion [NFJ02]. Focusing on attenuation due to absorption, the effect of a participating medium reduces to occlusion. This is simulated by splatting the medium density into a shadow map [YFSZ06]. A recent representative of the cloud rendering scenario [BNM+08] is based on the concept of most probable paths [PARN04]. For each point on the rasterized cloud surface, a shader iteratively refines an estimate of the *collector area* through which 95% of the radiance reaching this point enters the cloud. Propagation in the cloud is precalculated. Separate collector areas are found for different scattering depths.

3.5.4.1. Single Scattering

A plausible simplification for participating media with low albedo is the simulation of single scattering only, eliminating recursion. Marching through the medium by rasterizing sampling planes orthogonal to the viewing direction accumulates in-scattered radiance for the entire image in parallel [DYN00] with occlusion modeled by a shadow map. Follow-up work [DYN02] uses sub-planes to more densely sample the shadow map and light source intensity. Ray marching with uniform step size in *light space* is proposed to better sample a texture map expressing its intensity distribution [GMF09]. Interleaved sampling reduces computational cost [TU09]. When the medium is inhomogeneous, rasterizing planes orthogonal to the viewing and light source directions in parallel accumulates the attenuation of in-scattered radiance as well [ZC03]. Convolution with a smoothing kernel approximates a blurring of the attenuation due to multiple scattering.

Precalculating in-scattered radiance with gradients at points iteratively distributed in space where discontinuities exist [RZLG08] replaces gathering with splatting into a three-dimensional texture map subsequently rasterized in slices. Precalculation points distributed in image space instead are obtained by adaptively refining a sampling along the rays emanating from a light source [ED10]. Radiance is interpolated at the majority of the precalculation points and from these to the image.

The effect of refraction at participating medium boundaries on incident radiance can be simulated with refracted shadow rays [WZHB09]. Radiance in-scattered to the medium and incident on the surfaces embedded in it is gathered by ray tracing. Refracted shadow rays are computed for a query point by a shader that iteratively refines the rays passing through each triangle forming part of the medium boundary. A spatial index reduces the number of triangles searched.

3.5.4.2. Airlight Integral

Ray marching to gather in-scattered radiance solves the *airlight integral*. An efficient approximation of the integral value for single scattering in an isotropic, homogeneous medium exists as a closed-form expression with precalculated components stored in look-up tables [SRNN05]. A *point spread function* is additionally derived that can be convolved with an environment map to approximate the effect of single scattering on it. Follow-up work extends the method to anisotropic scattering [HHC+06]. A more accurate approximation subsequently devised [PP09b] is then further improved to a closed-form solution without approximation or precalculation [PSP10].

These methods compute in-scattered radiance for an unoccluded ray. To account for occlusion, *shadow volumes*, the bounding planes of occluded scene regions, are rasterized and the integral values for ray segments passing through unoccluded regions accumulated only [BAM06]. A less accurate approximation of the airlight integral is used but may be replaced with the originally proposed expression [WR08]. The shadow volumes can be obtained by extrusion of a shadow map [BSA10]. A classification of scene regions into occluded, unoccluded and uncertain instead follows from object bounding volumes [Nil08]. With these, ray marching is required where occlusion is uncertain.

An extension of the concept to an inhomogeneous medium whose density is modeled as sum of Gaussians exists [ZHG+07]. In-scattered radiance is accumulated by splatting the Gaussians and evaluating the contribution to the integral value for each.

3.5.4.3. Discretization

Using a grid of precalculation points for radiance storage and propagation simplifies computations. An early cloud rendering method [Bli82] employs a two-dimensional grid, simulating single scattering only. Later work [KvH84] uses a three-dimensional grid to precalculate radiance subsequently gathered by ray marching. If the albedo is high, multiple scattering is simulated with radiance stored in spherical harmonics and propagation between grid points via links encoded in a matrix. The *discrete ordinates method* for radiative transfer simulation [Cha50] discretizes directions as well, limiting storage to directional bins at grid points and propagation to their central directions. An application to computer graphics reduces discretization artifacts by propagating exitance from each bin to all bins covering its solid angle [Max94]. A finer subdivision of directional bins during propagation similarly reduces artifacts [Fat09]. Both methods sweep through the medium for iterative propagation.

Precalculation points at randomly distributed particles are an alternative to a grid [SKSU05]. The links between neighboring particles in randomly sampled directions and their radiance are stored in texture maps. A quad is rasterized to iteratively propagate radiance from all particles to their neighbors in a shader. The particles are then splat to the image.

Rasterizing the participating medium in slices, single scattering is simulated by accumulating in-scattered radiance attenuated according to transmittance precalculated at grid points. This accounts for the most probable paths from light to virtual camera [PARN04]. Blurring according to a point spread function approximates the effect of neighboring paths contributing multiple scattering and is efficiently possible by look-up in a coarser version of the precalculation grid. Diffusion theory allows for another approximation of multiple scattering [SKLUT09]. Radiance is decomposed into fluence and vector irradiance. Both are propagated by ray marching from the light source. Assuming concentric layers of the participating medium are homogeneous, the diffusion equations simplify to allow for efficient approximation of multiple scattering then stored in a concentric grid.

For a medium with inhomogeneous index of refraction, propagation occurs by adjusting direction after each ray marching step [IZT+07]. Four neighboring rays are traced together, forming a wavefront patch. Irradiance and wavefront direction are stored at the grid points encountered. The irradiance follows from patch flux and area. With a volumetric surface representation also stored in the grid, gathering by ray marching obtains in-scattered radiance and that reflected by surfaces.

3.5.4.4. Photon Mapping

A number of simplifications target volumetric photon mapping. For density estimation by splatting with a kernel preintegrated along the viewing direction, transmittance to the virtual camera and phase function are approximated by evaluating these at the center of each splat only [BPPP05]. An approximation of *crepuscular rays* is obtained by rasterizing a line along the path through the participating medium taken by each caustic photon [KBW06]. Applying these methods to underwater caustics and crepuscular rays, a kernel bandwidth is chosen according to the distance of a splat from the virtual camera [PP09a]. Blurring of the resulting images counters aliasing artifacts. A more comprehensive approximation of different light paths is obtained by combining the splatting of caustic photons onto surfaces, rasterization of crepuscular rays and an airlight integral technique computing further in-scattered radiance [HDI+10].

Two photon mapping variants for fire and smoke focus on artistic control over physical accuracy. Photons are emitted within the medium, storing their extinction events and surface interactions. In the first method [KIB05], flash photons are added that provide a bright flash around the participating medium. When propagating outside the medium, a flash photon stores flux along its path in a flash photon map until eventual extinction. Density estimation for flash photons is simplified as the flux stored is constant. The second method [Min06] stores extinction events in a three-dimensional grid. Positions are discretized to a sub-grid. Density estimation occurs by collecting all extinction events within a constant bandwidth of a ray.

4. Ray and Photon Tracing

Photon mapping traces photons from the light sources, rays from the virtual camera and shadow rays from the light sources again. Each is an instance of *ray tracing*. Ensuring that this key operation achieves high performance on the CUDA manycore platform is the subject of this chapter.

Ray tracing is inherently scalable by processing rays in parallel. Our efforts thus concentrate on reducing computational cost per ray. With n_T the number of nodes traversed in the spatial index, n_I the number of surfaces tested for intersection and C_T, C_I the costs of these operations, total cost is $n_T\,C_T + n_I\,C_I$. n_I and n_T are minimized by constructing a kd-tree that optimizes expected ray tracing cost. A low C_I results from modeling all surfaces as triangle meshes and employing the efficient Wald intersection test. C_T depends on the efficiency of kd-tree traversal.

Our first contribution is SIROH, a novel heuristic for estimating expected ray tracing cost yielding kd-trees with lower n_T, n_I and higher ray tracing performance than the best currently known heuristic, the SAH, for many scenes. SIROH is the result of a cooperation with Colin Fowler.

We then address traversal. Motivated by earlier GPU work [FS05, HSHH07], we look at stackless kd-tree traversal and propose an extension that allows it to work with nodes of zero volume. We then investigate stack-based traversal, showing its higher performance in CUDA and exploring the use of fast shared memory and registers as explicitly managed caches to further reduce C_T. Our work is motivated by photon mapping but also applies to other rendering algorithms that use ray tracing.

4.1. kd-Tree Construction

The cost of traversing a spatial index node is C_T, that of testing the n_N surfaces referenced by a leaf for intersection, $n_N\,C_I$. The expected ray tracing cost for a given spatial index is the sum of these values over all nodes, weighted by the probabilities of being visited by a ray. A construction process minimizing expected cost is outlined in section 3.2.3.1. Starting with all n surfaces at the root, recursive subdivision of a parent node P into child nodes L, R is applied. With p_P, p_L, p_R their probabilities of being visited by a ray and n_P, n_L, n_R the numbers of primitives they represent, the expected cost of each candidate subdivision is estimated using the metric of equation 3.12,

$$C_P\,(L,R) = p_P\,C_T + p_L\,n_L\,C_I + p_R\,n_R\,C_I. \tag{4.1}$$

Choosing the candidate subdivision that minimizes this expression *greedily* optimizes expected cost, approximating the children as leaves without further subdivision. No traversal cost is counted for the leaves as significant overheads occur only when the path through the spatial index branches at inner nodes. Leaves are either less expensive to traverse or not stored and traversed at all, referencing a list of surfaces directly from their parent instead. Possible criteria for terminating the recursion are n_P falling below a threshold, recursion depth exceeding a threshold or $\min C_P\,(L,R) > p_P\,n_P\,C_I$, indicating that a single leaf has lower expected ray tracing cost than the optimal pair of children.

For BVH and kd-tree, nodes take the shape of AABBs and are subdivided by choosing a splitting plane coplanar with one of the three coordinate axes. In the case of a BVH, every surface is assigned to the child containing its centroid, enlarging the AABB when a surface straddles the splitting plane. For a kd-tree, the surface is assigned to both children instead, leaving child AABBs unchanged.

4.1.1. Geometric Probability

Minimizing equation 4.1 requires a method for computing the probability that a node is visited by a ray. This probability depends on the distributions of ray origins, ray directions and occluding scene surfaces, the first two unknown and the third difficult to express. The SAH simplifies the problem by making three assumptions as shown in section 3.2.3.1. With S the scene AABB, these are:

1. Ray origins are uniformly distributed in space outside S.
2. Ray directions are uniformly distributed on the sphere of directions.
3. No ray hits any surfaces.

The first two assumptions treat the unknown distributions of ray origins and directions as uniform. With the third assumption, occlusion by scene surfaces is eliminated, making all rays infinitely long. Under these assumptions, the probability p_N that a ray intersecting S also visits a node N is given by equation 3.13 as the ratio of their surface areas,

$$p_N = \frac{\mathrm{SA}\,(N)}{\mathrm{SA}\,(S)}. \tag{4.2}$$

Inserting equation 4.2 into equation 4.1 yields a complete cost metric whose greedy minimization provides the highest ray tracing acceleration currently available. However, the question arises whether making more accurate assumptions could produce even better results.

We leave the second and third assumptions about ray directions and occlusion by scene surfaces unchanged. Regarding the first assumption, we make the observation that any reflected or refracted rays *always* originate at surfaces *within* S. For indoor scenes, virtual camera and light sources are also generally located inside the AABB. The actual distribution of ray origins remains unknown and we continue to assume that it is uniform. However, we expect these origins to lie *inside*, not outside S. Our set of assumptions thus is:

1. Ray origins are uniformly distributed in space *inside* S.
2. Ray directions are uniformly distributed on the sphere of directions.
3. No ray hits any surfaces.

Under the revised first assumption, a ray can visit node N by originating inside it or by originating elsewhere in the scene and subsequently entering N. Due to the assumptions of uniformly distributed ray directions and no occlusion, the probability of entering N from a potential ray origin $\vec{x} \in S \setminus N$ is the fraction of directions around $\vec{x}$ in which N is encountered. This is the ratio of the solid angle $\Omega_{\vec{x}}$ subtended by N and the total solid angle 4π (figure 4.1(a)). With ray origins uniformly distributed in S, their probability density function is the reciprocal of the scene volume $\mathrm{V}\,(S)$ so that

$$p_N = \int_N \frac{1}{\mathrm{V}\,(S)}\, d\vec{x} + \int_{S \setminus N} \frac{1}{\mathrm{V}\,(S)} \frac{\Omega_{\vec{x}}}{4\pi}\, d\vec{x} \tag{4.3}$$

$$= \frac{\mathrm{V}(N)}{\mathrm{V}(S)} + \frac{1}{\mathrm{V}(S)} \int_{S \setminus N} \frac{\Omega_{\vec{x}}}{4\pi} \, d\vec{x}. \tag{4.4}$$

The solid angle $\Omega_{\vec{x}}$ is obtained by projecting the sides of N facing $\vec{x}$ onto the unit sphere around it and computing the area covered. Unfortunately, inserting this expression into equation 4.4 leads to an elliptic integral for which no closed-form solution exists.

4.1.2. Numerical Approximation

While no closed-form solution to equation 4.4 is available, numerical approximations can be derived. Cutting the scene space $S \setminus N$ around node N along the planes of its six sides N_j yields twenty-six regions R_i that group potential ray origins $\vec{x}$ by the sets of N_j facing them, such as R_{17} containing all $\vec{x}$ faced by N_4 only and R_{10} those $\vec{x}$ faced by both N_1 and N_4 (figure 4.1(b)). We approximate the solid angle subtended by each set of N_j as uniform throughout the corresponding region R_i and equal to that Ω_i at its center, obtaining the numerical approximation

$$p_N \approx \frac{\mathrm{V}(N)}{\mathrm{V}(S)} + \frac{1}{\mathrm{V}(S)} \sum_{i=1}^{26} \mathrm{V}(R_i) \frac{\Omega_i}{4\pi}. \tag{4.5}$$

To calculate Ω_i, every N_j is decomposed into two triangles first. With vertices $\vec{x}_a$, $\vec{x}_b$, $\vec{x}_c$ expressed in a coordinate system whose origin is the center of R_i, the solid angle subtended by each of the triangles then is [vOS83]

$$\Omega_{\vec{x}_a, \vec{x}_b, \vec{x}_c} = 2 \arctan \frac{\vec{x}_a \cdot (\vec{x}_b \times \vec{x}_c)}{\|\vec{x}_a\| \, \|\vec{x}_b\| \, \|\vec{x}_c\| + \|\vec{x}_a\| \, (\vec{x}_b \cdot \vec{x}_c) + \|\vec{x}_b\| \, (\vec{x}_a \cdot \vec{x}_c) + \|\vec{x}_c\| \, (\vec{x}_a \cdot \vec{x}_b)}.$$

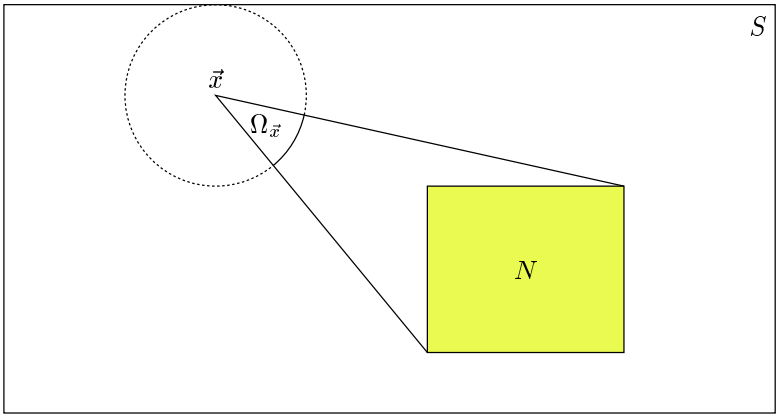
4.1.3. SIROH

Although equation 4.5 can be inserted into equation 4.1, the resulting cost metric is significantly more expensive to evaluate than the SAH. We therefore introduce the *scene-interior ray origin heuristic (SIROH)*, a further approximation with computational cost similar to the SAH.

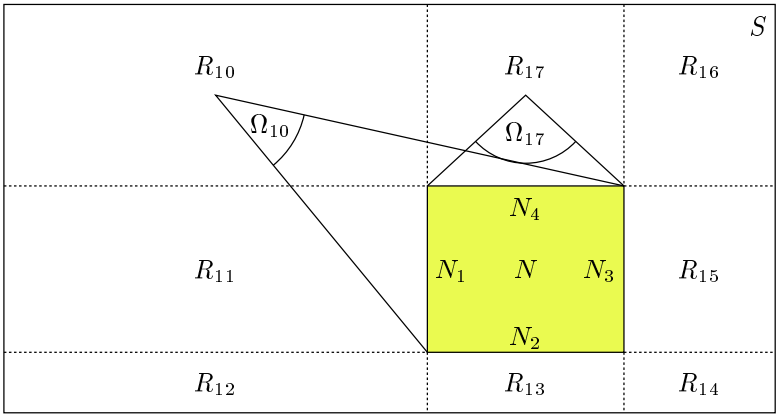
Cutting $S \setminus N$ along the plane of each side N_i *separately* yields six overlapping regions R'_i containing all potential origins $\vec{x}$ faced by one of the N_i (figure 4.1(c)). Points faced by more than one side become part of multiple regions, such as all $\vec{x} \in R_{10}$ now lying in the overlap of regions R'_1 and R'_4. The solid angle subtended by N_i is again approximated as uniform throughout R'_i and equal to that Ω'_i at its center. $\Omega'_i / 4\pi$ is the fraction of directions in which N_i is encountered. The complete bounds of R'_i fully surround its center and are encountered in every direction. $\Omega'_i / 4\pi$ thus is the ratio of the solid angles subtended by N_i and R'_i, defined as the surface areas of their projections onto the unit sphere. To eliminate the need for any trigonometric functions, we omit the projection and obtain

$$\Omega'_i = \frac{\Omega(N_i)}{\Omega(R'_i)} \approx \frac{\mathrm{SA}(N_i)}{\mathrm{SA}(R'_i)}.$$

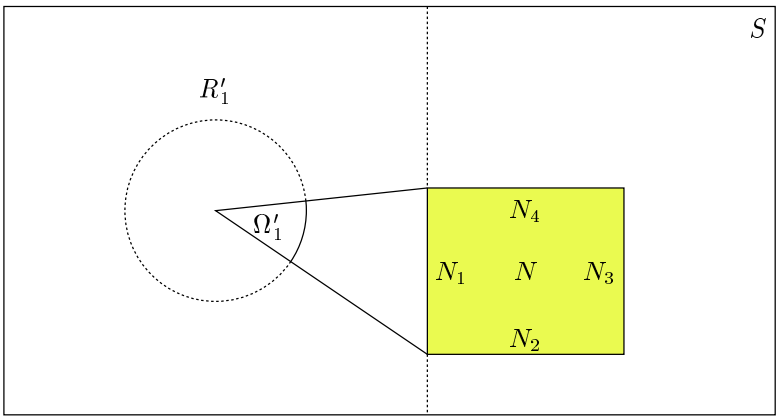
This approximation is justified by its low computational cost and experimental results indicating ray tracing performance comparable to that achieved with the more expensive equation 4.5 for many scenes. Applying the approximation to each of the six regions, the SIROH estimate for p_N is obtained,



(a) For a ray originating at any $\vec{x} \in S \setminus N$, the probability of visiting node N is the fractional solid angle $\Omega_{\vec{x}}/4\pi$ subtended by N at $\vec{x}$.



(b) Cutting the scene space around N along the planes of its six sides N_j yields twenty-six regions R_i, eight of which are visible in the two-dimensional cut. Each R_i groups all $\vec{x} \in S \setminus N$ faced by a particular set of N_j.



(c) Cutting along each side N_i *separately* yields six overlapping regions R'_i, each grouping all $\vec{x} \in S \setminus N$ faced by the corresponding N_i.

Figure 4.1.: Probability p_N that a node N is visited by a ray originating in $S \setminus N$: Cutting the space surrounding node N inside the scene AABB S along the bounding planes of N allows different approximations to be derived.

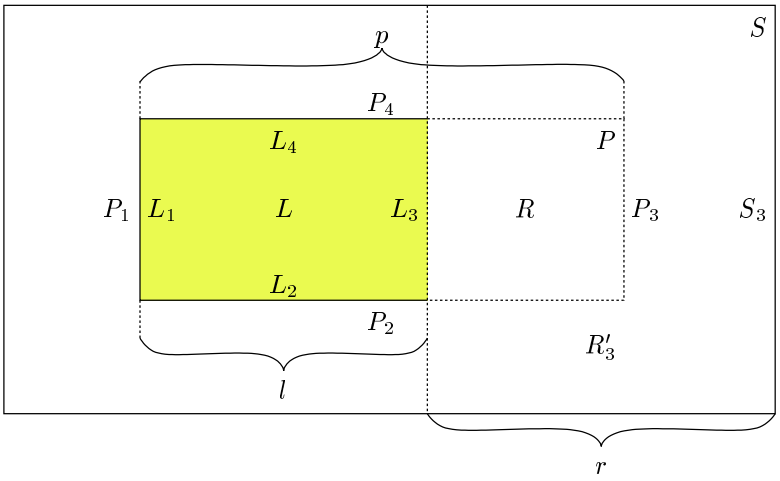


Figure 4.2.: Information reuse in SIROH: Multiple candidate subdivisions of node P into children L, R are evaluated that differ in the values of l and r only, allowing constants to be precalculated that accelerate the evaluation of $p_{L,SIROH}$, $p_{R,SIROH}$.

$$p_N \approx p_{N,SIROH} = \frac{\mathrm{V}(N)}{\mathrm{V}(S)} + \frac{1}{\mathrm{V}(S)} \sum_{i=1}^{6} \mathrm{V}(R'_i) \frac{\mathrm{SA}(N_i)}{\mathrm{SA}(R'_i)} = p_{N,0} + \sum_{i=1}^{6} p_{N,i}. \tag{4.6}$$

SIROH requires only basic arithmetic operations, most of which can furthermore be eliminated by reusing information during recursive kd-tree construction. Figure 4.2 illustrates the subdivision of a parent node P into children L, R. Multiple candidate subdivisions need to be evaluated that differ in the position of the splitting plane and thus the values of l and r only.

Equation 4.6 decomposes into the sum of the probabilities $p_{N,0}$ that a ray originates inside N and $p_{N,i}$ that one enters through a side N_i. For node L, probability $p_{L,1}$ is identical to $p_{P,1}$ and can directly be reused from the parent. $p_{L,0}$, $p_{L,2}$, $p_{L,4}$, $p_{L,5}$, $p_{L,6}$ are proportional to l and can be computed as $p_{L,i} = \frac{l}{p} p_{P,i}$. Only p_{L_3} requires calculation according to equation 4.6. With $\mathrm{C}(S_3)$ the circumference of scene AABB side S_3, the surface area of region R'_3 is $\mathrm{SA}(R'_3) = 2\,\mathrm{SA}(S_3) + r\,\mathrm{C}(S_3)$ while its volume is $\mathrm{V}(R'_3) = r\,\mathrm{SA}(S_3)$. Using these identities, the probability $p_{L,SIROH}$ is transformed into

$$\begin{aligned} p_{L,SIROH} =& p_{P,1} + \frac{l}{p}\left(p_{P,0} + p_{P,2} + p_{P,4} + p_{P,5} + p_{P,6}\right) + \frac{1}{\mathrm{V}(S)} r\,\mathrm{SA}(S_3) \frac{\mathrm{SA}(L_3)}{2\,\mathrm{SA}(S_3) + r\,\mathrm{C}(S_3)} \\ =& \underbrace{p_{P,1}}_{c_1} + l \underbrace{\frac{1}{p}\left(p_{P,0} + p_{P,2} + p_{P,4} + p_{P,5} + p_{P,6}\right)}_{c_2} + \frac{r}{\underbrace{2\frac{\mathrm{V}(S)}{\mathrm{SA}(L_3)}}_{c_3} + r \underbrace{\frac{\mathrm{V}(S)\,\mathrm{C}(S_3)}{\mathrm{SA}(S_3)\,\mathrm{SA}(L_3)}}_{c_4}} \\ =& c_1 + c_2 l + \frac{r}{c_3 + c_4 r} \end{aligned}$$

After computing the constants c_i, $p_{L,SIROH}$ can be evaluated for each candidate splitting plane using three additions, two multiplications and one division. $p_{R,SIROH}$ is calculated analogously.

4.1.4. Results and Discussion

We evaluate the numerical approximation from section 4.1.2 and SIROH against the SAH using the benchmark environments and scenes described in appendix A. All kd-tree construction parameters

Scene	Code Base I			Code Base II	
	SAH	4.1.2	SIROH	SAH	SIROH
Scene 6	382.2	**+7.5%**	**+7.0%**	341.8	**+14.9%**
Sponza	99.6	**+1.9%**	**±0.0%**	215.6	**−0.8%**
Sibenik	101.0	**+2.0%**	**+2.1%**	189.4	**+2.7%**
Fairy	75.9	**+1.3%**	**+2.7%**	144.1	**+3.4%**
Conference	71.9	**+5.1%**	**+4.2%**	171.0	**+0.5%**
Average		**+3.5%**	**+3.2%**		**+4.1%**
Std. Dev.		2.7%	2.6%		6.2%

Table 4.1.: Ray tracing frame rates with the numerical approximation from section 4.1.2 and SIROH for two CUDA benchmark environments, relative to the SAH

are tuned for maximal performance with the SAH and used identically throughout ($C_T = 1$, $C_I = 3$, recursive subdivision while $n_P > 4$ and $\min C_P(L, R) \leq p_P\, n_P\, C_I$). Since the SAH benefits from biasing the cost metric to favor cutting off empty space [HKRS02], the same bias is applied to all heuristics: If either child has a volume greater than 30% that of the parent and contains no surfaces, the cost estimate for this candidate subdivision is reduced by 20%.

Performance is assessed by tracing a combination of primary, reflection and shadow rays in CUDA, using a straightforward stack-based traversal of the kd-tree. Images are synthesized into a buffer, omitting its subsequent visualization as this constant overhead per frame is unrelated to ray tracing. Table 4.1 lists average frame rates for a flight through each scene. The original evaluation uses benchmark code base I of section A.1.1. With a kd-tree constructed using the more expensive numerical approximation, a speedup over the SAH is achieved in all scenes, averaging 3.5%. SIROH attains comparable results, averaging 3.2%. Only in the Sponza Atrium is the SAH frame rate not exceeded. For this scene, the more aggressive approximation of p_N by SIROH appears not to follow its more expensive counterpart with sufficient accuracy.

In order to ensure benefits are not specific to a particular ray tracing implementation, a reevaluation of SIROH using code base II of section A.1.2 is included in table 4.1. This ray tracer has different performance characteristics and significantly higher absolute frame rates. SIROH proves beneficial for it as well with an average speedup over the SAH of 4.1%. In the problematic Sponza Atrium, a slowdown by 0.8% is observed. A further reevaluation using the RT^2 ray tracer [FC07] written by the coauthor of SIROH and static views of the benchmark scenes is presented in table 4.2(a). A speedup averaging 2.9% and a slowdown in the Sponza Atrium are consistent with previous results.

The Radius-CUDA ray tracer [Seg08] renders frame 160 of the Fairy Forest animation with a static virtual camera and an animated point light source. Average frame rates are given in table 4.2(b). When tracing only primary rays, SIROH yields a speedup over the SAH of 2.5%. Shadow rays lead to a slowdown. This is because the light source is located outside the scene, violating the assumptions underlying SIROH. After moving the light source inside the scene, a speedup is obtained again.

For further analysis, the number n_T of inner node traversals and n_I of triangle intersection tests in the original evaluation are listed in table 4.3. As algorithm 3.3 has an early out after intersecting the triangle plane, intersections with planes and barycentric coordinate calculations are counted separately. Both the more expensive numerical approximation and SIROH reduce the numbers of traversals and intersections in most cases, closely following each other. Only in the Sponza Atrium does SIROH lead to significantly more plane intersections, corresponding to the slowdowns seen.

Scene	SAH	SIROH
Scene 6	115.9	+**8.5**%
Sponza	64.5	−**3.9**%
Sibenik	63.1	+**4.2**%
Fairy	21.6	+**0.4**%
Conference	46.4	+**5.1**%
Average		+**2.9**%
Std. Dev.		4.8%

(a) RT^2 ray tracer [FC07]

Setting	SAH	SIROH
Unshaded	43.5	+**2.4**%
Unshadowed	43.3	+**2.5**%
Light outside	15.1	−**3.8**%
Light inside	14.0	+**1.7**%

(b) Radius-CUDA ray tracer [Seg08]

Table 4.2.: Ray tracing frame rates with SIROH for two other ray tracers, relative to the SAH

Scene	Traversals			Plane Intersections			Triangle Intersections		
	SAH	4.1.2	SIROH	SAH	4.1.2	SIROH	SAH	4.1.2	SIROH
Scene 6	9.68	−24.9%	−25.2%	7.59	−15.3%	−14.3%	1.10	−7.3%	−3.0%
Sponza	30.83	−0.5%	−4.3%	7.95	+1.1%	+25.5%	3.06	−12.3%	−13.7%
Sibenik	40.44	−5.8%	−5.6%	5.06	−11.0%	−10.5%	2.04	−7.2%	−4.0%
Fairy	41.30	+5.5%	−1.1%	8.09	−5.7%	−3.9%	2.35	−2.8%	−6.5%
Conference	23.56	−13.0%	−18.6%	12.99	−21.4%	−17.4%	2.29	−6.1%	+1.4%
Average		−7.7%	−11.0%		−10.4%	−4.1%		−7.1%	−5.2%
Std. Dev.		11.8%	10.4%		8.7%	17.3%		3.4%	5.6%

Table 4.3.: Inner node traversals, triangle plane intersections and barycentric coordinate calculations with the numerical approximation from section 4.1.2 and SIROH, relative to the SAH

Scene	CPU I (s)		CPU II (s)		Inner Nodes		Surface References	
	SAH	SIROH	SAH	SIROH	SAH	SIROH	SAH	SIROH
Scene 6	0.01	+7.4%	0.003	+11.4%	2801	+1.5%	4038	+1.6%
Sponza	0.91	+5.8%	0.644	+6.2%	207425	−1.0%	346767	−0.5%
Sibenik	0.95	+2.3%	0.667	−11.5%	192667	−0.3%	317533	−0.3%
Fairy	3.41	+13.0%	2.239	+27.7%	788603	+1.2%	1587203	+1.2%
Conference	3.25	+7.6%	3.232	+4.5%	611297	±0.0%	2155935	−0.7%
Average		+7.2%		+7.7%		+0.3%		+0.3%
Std. Dev.		3.9%		14.1%		1.0%		1.1%

Table 4.4.: Construction timings and kd-tree statistics with SIROH, relative to the SAH

Construction of a kd-tree with the more expensive numerical approximation is on the order of minutes. No exact numbers are given as our implementation is naïve and serves to assess ray tracing performance only. Even with aggressive optimizations, it would remain slow due to the trigonometric functions required. Results for SIROH are presented in table 4.4. Construction is timed on an Intel Pentium D 965 (CPU I) and an Intel Core2 Quad Q9450 (CPU II). In both cases, SIROH construction is competitive with the SAH, taking only 7.2% respectively 7.7% longer on average. Also given are kd-tree statistics, showing that the trees obtained with SIROH and the SAH are of very similar size, differing in the number of inner nodes and surface references at leaves by only 0.3% on average.

In summary, the numerical approximation from section 4.1.2 shows that the assumption of ray origins distributed uniformly *inside* the scene can improve ray tracing performance over the SAH for

all benchmarked scenes, albeit at the cost of more expensive kd-tree construction. SIROH is successful in achieving similar speedups for many scenes with construction cost comparable to the SAH. As with any heuristic, cases can be found where a slowdown results. The Sponza Atrium proves problematic for SIROH. The large increase in plane intersections indicates that many surfaces referenced by the leaves may in fact lie outside or only skim these. Tightening surface AABBs after each subdivision would ensure that only the parts not cut off by previous splitting planes are considered [HB02] but this would also increase construction cost. We use SIROH in its current form.

4.2. Stackless kd-Tree Traversal

The kd-tree is traversed from its root for each ray $\vec{z}(t) = \vec{x} + t\vec{\omega}$, $t \in [t_{min}, t_{max}]$ as described in section 3.2.4.1. At every inner node visited, the children are classified as *near* and *far* according to their order along the ray and the parameter value t_s is computed for which the ray intersects the splitting plane. If $t_s > t_{max}$, only the near child is intersected by the ray and traversal continues with it. $t_s < t_{min}$ is the analogous case for the far child. When $t_{min} \leq t_s \leq t_{max}$, traversal *branches*, continuing with the near child and the parameter interval $[t_{min}, t_s]$ for which the ray intersects it first and returning to the far child with parameter interval $[t_s, t_{max}]$ later.

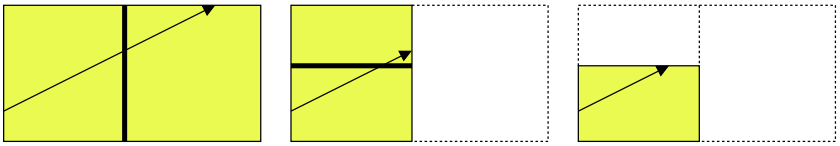
Upon reaching a leaf, all surfaces referenced by it are tested for intersection. Traversing the near child of every branching point first ensures that leaves are visited in their order along the ray. If a hit is found in the current leaf, traversal thus terminates as no nearer hit can exist in any remaining leaves. Otherwise, traversal continues with the far child of the most recent branching point.

Pushing the far children of branching points onto a *stack* is simplest but not always possible or efficient. GPUs accessed through graphics APIs expose no suitable read-write memory. Fast shared memory is available in CUDA but limited to 16 kB per SM. Placing stacks in it requires packetization, limited occupancy or the construction of shallow trees, all detrimental to performance. Local memory is larger but slow. We therefore investigate *stackless* traversal as introduced in section 3.2.4.4. Adding parent pointers to the kd-tree enables returns to branching points by upward traversal but incurs storage overheads. Stackless traversal of an *unmodified* kd-tree is achieved by shifting the start of the ray so that it begins beyond the end of the parameter interval $[t_a, t_b]$ at the current leaf and restarting from the root. The result is an identical traversal path until the most recent branching point. Here, the near child is now missed and the far child traversed instead [FS05].

4.2.1. Zero Volume Nodes

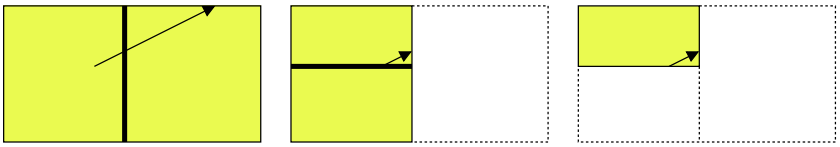
Care must be taken when shifting the start of the ray to ensure correct traversal order. Changing the parameter interval from $[t_{min}, t_{max}]$ to $[t_b, t_{max}]$ is not sufficient as the previous leaf is still intersected for $[t_b, t_b]$, causing it to be visited after every restart in an infinite loop. Using open intervals prevents infinite looping, as illustrated by the stackless traversal of an example kd-tree in figure 4.3(a)–(c). However, nodes with *zero volume* are missed. This is shown in figure 4.3(d). The splitting plane at the root yields a parameter interval $(t_{min}, t_{s,P})$ in the left child. This child in turn has a splitting plane coplanar with its left bound so that $t_{s,L} = t_{min}$ and the parameter interval in the left child is $(t_{min}, t_{min}) = \emptyset$, causing traversal to continue with the right child only.

A node with zero volume is not a defect but a desirable feature. It shaves off surfaces coplanar with the bounds of its parent, separating them from the geometry inside and allowing both to be



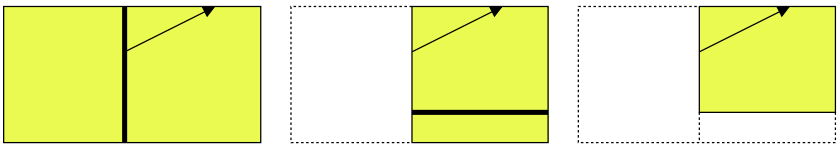
(t_{min}, t_{max}) $(t_{min}, t_{s,P})$ $(t_{min}, t_{s,L})$

(a) Traversal with $t \in (t_{min}, t_{max})$ visits the leaf intersected for $(t_{min}, t_{s,L})$.



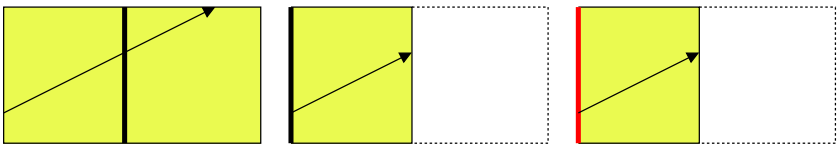
$(t_{s,L}, t_{max})$ $(t_{s,L}, t_{s,P})$ $(t_{s,L}, t_{s,P})$

(b) Restart with $t \in (t_{s,L}, t_{max})$ visits the leaf intersected for $(t_{s,L}, t_{s,P})$.



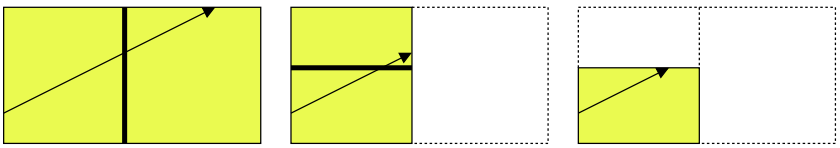
$(t_{s,P}, t_{max})$ $(t_{s,P}, t_{max})$ $(t_{s,P}, t_{max})$

(c) Restart with $t \in (t_{s,P}, t_{max})$ visits the leaf intersected for $(t_{s,P}, t_{max})$.



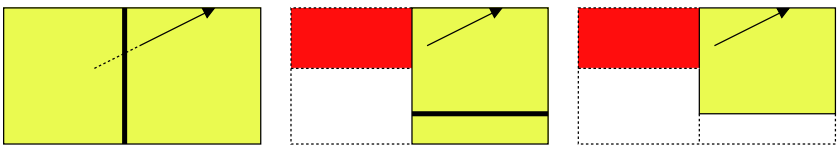
(t_{min}, t_{max}) $(t_{min}, t_{s,P})$ $(t_{min}, t_{s,P})$

(d) Traversal with $t \in (t_{min}, t_{max})$ *misses* the red leaf intersected for $[t_{min}, t_{min}]$.



$[t_{min}, t_{max}]$ $[t_{min}, t_{s,P}]$ $[t_{min}, t_{s,L}]$

(e) Traversal with $t \in [t_{min}, t_{max}]$ visits the leaf intersected for $[t_{min}, t_{s,L}]$.



$[t_{s,L} + \epsilon, t_{max}]$ $[t_{s,L} + \epsilon, t_{max}]$ $[t_{s,L} + \epsilon, t_{max}]$

(f) Restart with $t \in [t_{s,L} + \epsilon, t_{max}]$ *misses* the red leaf intersected for $[t_{s,L}, t_{s,P}]$.

Figure 4.3.: Stackless kd-tree traversal by restarting at the root with shifted ray start: The algorithm is successful when all nodes have nonzero volume. Otherwise, nodes are missed.

indexed optimally [WH06]. Using closed intervals but shifting the start of the ray by an additional small offset $\epsilon > 0$ prevents infinite looping while allowing nodes with zero volume to be visited. However, correct traversal cannot be guaranteed. If ϵ is too small, numerical imprecision can still lead to an infinite loop. If it is too large, nodes within $(t_b, t_b + \epsilon)$ are missed. This is illustrated by a repeat of the traversal from figure 4.3(a)–(c). The initial traversal path from the root is unchanged, visiting the correct leaf (figure 4.3(e)). After restarting with an additional offset ϵ, the path deviates (figure 4.3(f)), missing the leaf previously visited in figure 4.3(b) and skipping to that of figure 4.3(c) instead. The problem is compounded by the presence of multiple nodes coplanar with each other. No adjustment of the parameter interval can guarantee correct traversal in this general case.

4.2.2. Traversal Algorithm

Instead of provoking a change in traversal path by modifying the parameter interval, we propose to store a flag for each branching point on the path that indicates whether the near or far child should be traversed next. Traversal can then be restarted with the original parameter interval $[t_{min}, t_{max}]$ after toggling the flag for the most recent branching point. As branching may occur on every level of the tree, one bit of storage space is required per tree level. We use a single 32-bit $flags$ register, allowing for stackless traversal of trees up to 32 levels deep.

The flags are initialized to zero. When a branching point is reached during traversal from the root, the flag for the current level is tested, continuing with the near child if the flag is zero and with the far child otherwise. Upon reaching a leaf, the flag at the most recent branching point must be toggled to one before restarting traversal. In order to efficiently do so, flags at the levels *without* branching are set to one during traversal. The simple addition of a one at the level above the leaf then *ripples* through the flags register, toggling the flags for levels without branching back to zero until the most recent branching point is reached and its zero flag is toggled to one.

Algorithm 4.1 illustrates the complete process. Traversal begins by setting the parameter interval to $[t_{min}, t_{max}]$, *node* to the root and initializing a helper register $flag$ with the flag bit for the current depth in the tree which is the most significant bit for the root.

The loop in lines 7 to 24 traverses inner nodes by descending to a child and updating the current parameter interval $[t_a, t_b]$. If $t_s < t_a$ or $t_s > t_b$, only one child is intersected. The current *node* is replaced by this child in line 10 or 13 and the $flags$ bit for the current level is set by bitwise conjunction with the helper register $flag$.

If both children are intersected, the $flags$ bit for the current level is tested in line 15. When the bit is not set, lines 19–20 replace *node* with the near child and the parameter interval with $[t_a, t_s]$. Lines 16–17 analogously descend to the far child with $[t_s, t_b]$ if the bit is set. The final step in line 23 shifts the $flag$ register to the right so that it contains the flag bit for the next level.

When a leaf is reached, line 25 adds the flag bit for its parent to $flags$. This addition toggles the bits above it up to and including the first zero by means of elementary binary arithmetic. After visiting the leaf, traversal restarts at the root by looping back to line 3. The termination criterion $flags = 0$ on line 27 is met if the last addition cleared the flags for all levels, indicating that no branching point remains whose right child needs to be traversed. If a hit point is found while visiting the leaf in line 26, $flags$ is also set to zero, causing an early termination of the traversal.

The stackless traversal of an example kd-tree including a zero volume node is shown in figure 4.4(a) and (b). During the initial traversal from the root, $flags$ is zero. Both children are intersected at the

Algorithm 4.1 Stackless kd-tree traversal ($\ll$, $\vee$, $\wedge$ are bitwise shift, conjunction and disjunction)

```
1:  flags ← 0
2:  repeat
3:    t_a ← t_min
4:    t_b ← t_max
5:    node ← 0
6:    flag ← 1 ≪ 31
7:    while node is inner do
8:      if t_s < t_a then
9:        flags ← flags ∨ flag
10:       node ← far
11:     else if t > t_b then
12:       flags ← flag ∨ flag
13:       node ← near
14:     else
15:       if flags ∧ flag then
16:         t_a ← t_s
17:         node ← far
18:       else
19:         t_b ← t_s
20:         node ← near
21:       end if
22:     end if
23:     flag ← flag ≫ 1
24:   end while
25:   flags ← flags + (flag ≪ 1)
26:   visit leaf
27: until flags = 0
```

root node P, causing traversal of the near child, L. Both children are intersected here again and the near leaf A is visited with $flags$ still zero. Adding the flag bit for the parent of A yields $flags = 01_2$ and traversal restarts. The flag bit at the root is still zero and near child L is traversed again. Here, the set flag bit changes the traversal path, visiting the far leaf B. The flag bit for its parent is added again, resulting in $flags = 01_2 + 01_2 = 10_2$ and traversal restarts. The flag bit for the root is now set and its far child R is traversed. The ray misses its splitting plane and intersects the far leaf only. The flag bit for the second tree level is thus set, resulting in $flags = 11_2$, before continuing to the far leaf D. Adding the flag bit for its parent overflows the $flags$ register to zero, terminating traversal.

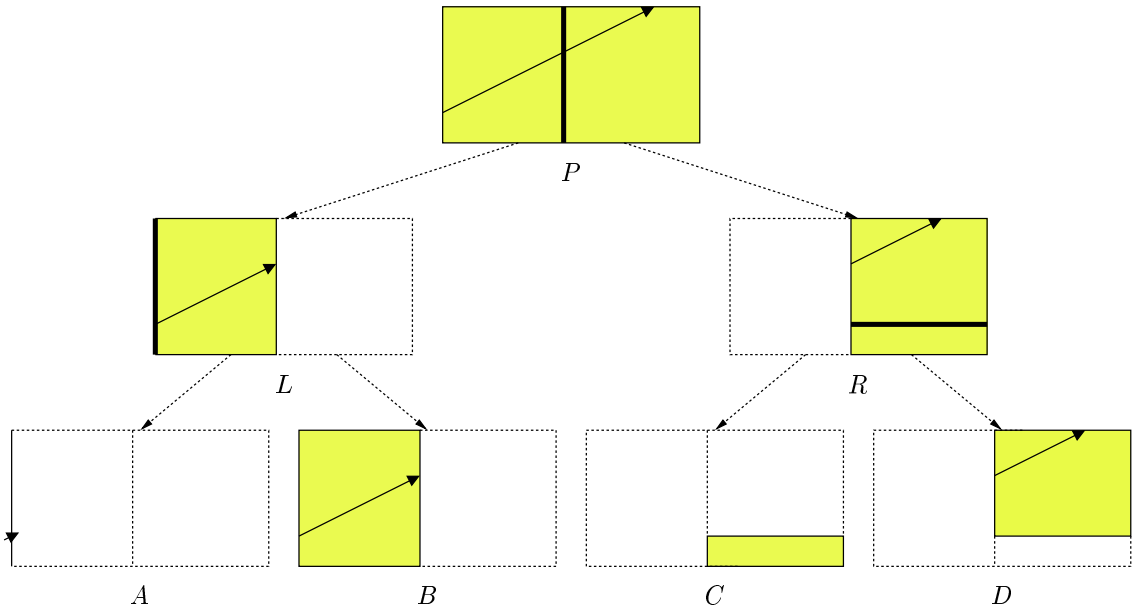
4.2.3. Extensions

Restarting from the root leads to redundancy as the same nodes are traversed again to reach the most recent branching point. Two extensions reducing such redundant traversals are *push-down* and *short-stack* [HSHH07]. Both are readily integrated into our traversal algorithm.

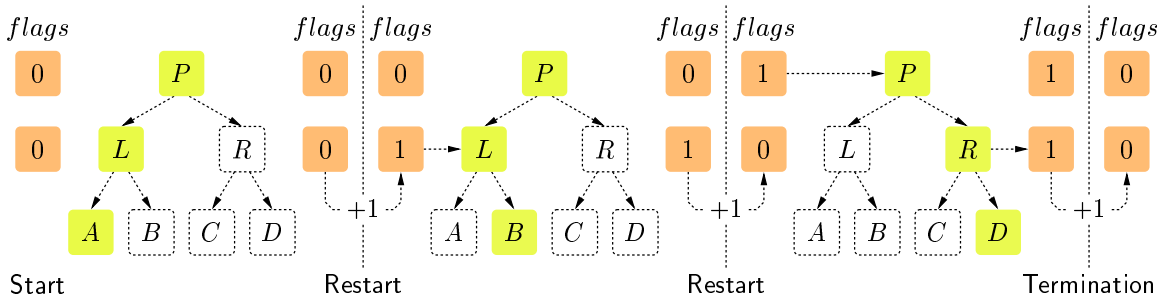
Push-down is based on the observation that the traversal path changes only at branching points. As the path from the root to the first branching point cannot change, the restart point may be *pushed down* to it. After the traversal of the near child at this branching point completes, subsequent restarts always lead to the traversal of its far child, allowing for further push-down. In general, the restart point can be pushed down after every restart to the first branching point whose children still both need to be traversed. The addition of push-down is illustrated in algorithm 4.2 using blue color.

Algorithm 4.2 Stackless kd-tree traversal with push-down (blue) and short-stack (red) extensions ($\ll$, $\vee$, $\wedge$ are bitwise shift, conjunction and disjunction)

```
1:  stack.front ← 0
2:  stack.size ← 0
3:  restart ← 0
4:  restart.flag ← 1 ≪ 31
5:  flags ← 0
6:  repeat
7:    if stack.size > 0 then
8:      t_a ← t_b
9:      pop (node, flag, t_b)
10:   else
11:     t_a ← t_min
12:     t_b ← t_max
13:     node ← restart
14:     flag ← restart.flag
15:     push ← true
16:   end if
17:   while node is inner do
18:     if t < t_a then
19:       flags ← flags ∨ flag
20:       node ← far
21:     else if t > t_b then
22:       flags ← flags ∨ flag
23:       node ← near
24:     else
25:       if flags ∧ flag then
26:         t_a ← t
27:         node ← far
28:       else
29:         if push then
30:           t_min ← t_a
31:           t_max ← t_b
32:           restart ← node
33:           restart.flag ← flag
34:           push ← false
35:         end if
36:         push (far, flag ≫ 1, t_b)
37:         t_b ← t
38:         node ← near
39:       end if
40:     end if
41:     flag ← flag ≫ 1
42:   end while
43:   flags ← flags + (flag ≪ 1)
44:   visit leaf
45: until flags = 0
```



(a) The ray given should visit the leaves of this kd-tree in the order A, B, D.



(b) A *flags* register controls traversal decisions at branching points during traversal from the root. By toggling the flag for the most recent branching point before each restart, leaves are visited in correct order.

Figure 4.4.: Stackless kd-tree traversal by restarting at the root with flags register: The flags guarantee correct traversal order despite the presence of zero volume node A.

The *restart* node initially is the root, $restart.flag$ contains its flag bit and the parameter interval $[t_{min}, t_{max}]$ spans the entire ray. Each traversal begins by resetting *node*, *flag* and $[t_a, t_b]$ to these values in lines 11–14. A *push* flag is additionally set in line 15. Traversal then follows the original algorithm. When the first branching point is reached at which both children still need to be traversed, the restart point is pushed down to it in lines 30–33 by recording the current *node*, *flag* and the parameter interval $[t_a, t_b]$ for which the ray intersects the node. The *push* flag is then cleared to avoid further push-down until the next restart.

Where push-down makes restarts more efficient, short-stack aims to eliminate them. If available, fast memory is used as a limited capacity stack onto which the far children of branching points are pushed during traversal. Should the stack become full, each additional push displaces the oldest entry. After visiting a leaf, traversal continues with the node popped off the top of the stack. Due to the limited capacity, branching points may exist whose far child still needs to be traversed but whose stack entry has been displaced. To ensure these are not missed, a restart is performed when the stack is empty. The short-stack extension is shown in algorithm 4.2 using red color.

Scene	Stack-based	Stackless			
			push-down	short-stack	push-down short-stack
Scene 6	7.31	+35.7%	+21.1%	±0.0%	±0.0%
Sponza	29.81	+136.7%	+97.1%	+0.3%	+0.2%
Sibenik	37.63	+151.5%	+107.2%	+0.8%	+0.8%
Fairy	37.43	+182.7%	+98.2%	+0.6%	+0.6%
Conference	19.57	+108.1%	+59.4%	+0.3%	+0.3%
Average		+122.9%	+76.6%	+0.4%	+0.4%
Std. Dev.		55.7%	36.0%	0.3%	0.3%

Table 4.5.: Inner node traversals per ray with stackless traversal, relative to a stack-based traversal

Line 36 pushes the far child onto the stack whenever traversal reaches a branching point and continues with its near child. Popping in line 9 then replaces the restart in lines 11–15 if at least one entry is present on the stack. The start of the parameter interval t_a is equal to its end t_b in the leaf just visited, allowing t_a to be set by line 8 without storing it on the stack.

We implement the stack as a round-robin buffer. To push an entry, it is stored at the buffer position $(stack.front + stack.size) \mod stack.capacity$. If $stack.size < stack.capacity$, the stack is not full yet and $stack.size$ is incremented. Otherwise, $stack.front$ is incremented. To pop the top, $stack.size$ is decremented and the entry at position $(stack.front + stack.size) \mod stack.capacity$ read.

4.2.4. Results and Discussion

Our stackless traversal algorithm is evaluated using the benchmark environment and scenes from appendix A. The more efficient code base II of section A.1.2 is employed, tracing up to four rays per pixel as described in the appendix. The kd-tree for each scene is constructed using the SIROH heuristic with the same parameters as in section 4.1.4. Ray tracing and radiance computation are implemented as a single CUDA kernel using 54 registers, allowing 256 threads to be resident per SM. For the short-stack extension, four stack entries per thread are placed in shared memory. Each entry consists of three 32-bit values for a total size of 48 bytes per short-stack or 12 kB per SM.

Stack-based and stackless traversal visit leaves in the same order, differing only in the processing of inner nodes. The former requires stack operations in local memory, the latter redundant traversals. Table 4.5 lists the average number n_T of inner nodes traversed per ray during a flight through each scene. Compared to stack-based traversal, restarting at the root leads to an average of 122.9% additional inner node traversals. The push-down extension reduces this figure to 76.6%. Short-stack, whether combined with push-down or not, eliminates almost all redundancy, requiring only 0.4% more inner nodes to be traversed than during a stack-based traversal.

By removing the need for a stack in local memory and introducing only minimal redundancy, stackless traversal with the short-stack extension requires fewer global memory accesses overall than a stack-based traversal. Despite this, its performance is *worse*, as evidenced by the average frame rates for a flight through each scene given in table 4.6. Compared to a straightforward stack-based traversal implementation, our stackless algorithm without any extensions yields 29.1% lower frame rates on average. The push-down extension reduces this slowdown to 22.0%, short-stack to 7.3% and the combination of both extensions to 10.1%.

Scene	Stack-based	Stackless			
			push-down	short-stack	push-down short-stack
Scene 6	392.6	−5.3%	−2.6%	−1.3%	−2.5%
Sponza	213.7	−37.1%	−30.6%	−8.7%	−12.2%
Sibenik	194.6	−43.5%	−35.8%	−10.2%	−13.8%
Fairy	149.0	−41.6%	−29.8%	−11.6%	−14.9%
Conference	172.0	−18.1%	−11.4%	−4.6%	−7.2%
Average		−29.1%	−22.0%	−7.3%	−10.1%
Std. Dev.		16.7%	14.3%	4.2%	5.2%

Table 4.6.: Ray tracing frame rates with stackless traversal, relative to a stack-based traversal

Scene	Stack-based	Stackless			
			push-down	short-stack	push-down short-stack
Scene 6	674.2	+18.4%	+14.6%	+13.6%	+16.8%
Sponza	2249.9	+75.1%	+61.5%	+24.0%	+29.8%
Sibenik	2294.9	+96.0%	+74.8%	+27.4%	+34.0%
Fairy	3129.6	+90.5%	+59.9%	+28.8%	+35.3%
Conference	2246.2	+42.0%	+29.6%	+16.5%	+20.3%
Average		+64.4%	+48.1%	+22.1%	+27.2%
Std. Dev.		33.2%	25.0%	6.7%	8.3%

Table 4.7.: Instructions per ray with stackless traversal, relative to a stack-based traversal

The CUDA documentation [NVI10b] and optimization guidelines [NVI10a] identify global memory accesses and computational cost due to warp divergence or expensive arithmetic as the bottlenecks that should be addressed during algorithm design. As noted in section 3.2.4.4, ray tracing performance is maximized by not explicitly controlling warp divergence, tracing rays independently and letting threads executing the same instruction automatically fall into lock-step. Stackless traversal introduces no expensive arithmetic and, when extended by short-stack, reduces the number of global memory accesses. Its lower performance must thus be due to an effect not covered by the documentation.

We find that focusing on global memory accesses, warp divergence and the use of SFUs or DFUs for expensive arithmetic operations overlooks the possibility that kernels may become *compute bound* by simple instructions executing on the SPs. Table 4.7 lists the average number of instructions required to trace a ray. The lower performance of stack-based traversal with short-stack is explained by it being compute bound and requiring 22.1% more instructions on average than stack-based traversal. Combining push-down and short-stack yields an almost identical number of global memory accesses but requires additional operations to update the restart point, resulting in an instruction count 27.2% higher than stack-based traversal and a corresponding reduction in frame rates. For the other stackless traversal variants, the effects of further redundant traversals and higher instruction counts are compounded, resulting in the observed slowdowns.

In summary, we propose a stackless algorithm that guarantees correct traversal of arbitrary kd-trees. With the short-stack extension, this algorithm more closely follows the CUDA optimization guidelines than stack-based traversal but nevertheless experiences a slowdown. We find the source

of its lower performance in an effect whose importance is overlooked by the CUDA documentation: In addition to the documented potential bottlenecks, a kernel becoming compute bound by simple instructions proves to be a realistic possibility that should be taken into account.

Two other stackless traversal algorithms also use one bit of storage space per spatial index hierarchy level. kd-jump [HL09] achieves stackless operation not by restarting but by upward traversal. Using an implicit kd-tree representation [WFM+05] in which the index of a parent node can be computed from that of a child, the flag bits indicate how many levels of upward traversal are required. This work is significantly different from ours, targeting volumetric surface representations, using spatial median splits only and resulting in an overall different algorithm.

Closer to our work is a solution concurrently proposed for stackless BVH traversal [Lai10]. Its *restart trail* serves the same purpose as our *flags* register and is updated by the same bitwise operations. Differences exist in other parts of the algorithm. With a BVH, restarts can occur both at leaves and inner nodes. Traversal cannot terminate immediately after finding a hit as other leaves may contain nearer hit points. When using the short-stack extension, the ray parameter interval is not stored on the stack as it can be reconstructed by clipping the ray against the AABB of the popped node. The authors also do not store the value of *flag* on the stack, reconstructing it from the lowest flag bit set after popping instead. A sentinel bit on the restart trail signals the end of traversal while our algorithm overflows the *flags* register to zero for this purpose. The addition of push-down is not explored. Overall, the authors find similar numbers of redundant traversals. They also note a slowdown relative to stack-based traversal for ray tracing on the CUDA platform but do not quantify or analyze it in detail.

4.3. Stack-Based kd-Tree Traversal

Having established that straightforward stack-based kd-tree traversal yields faster ray tracing than its stackless variants, we adopt this approach and investigate opportunities for accelerating it further. Section 3.2.4.4 shows that the best practice of grouping rays into packets on SIMD architectures does not transfer to the SIMT manycore processing model underlying CUDA, requiring different acceleration techniques to be devised. With section 4.2.4 illustrating the risk of ray tracing becoming compute bound, these should be simple and computationally inexpensive.

As described in section 2.2.3.3, the next generation of CUDA hardware accelerates global memory accesses with caches located in additional shared memory. This motivates us to explore the use of shared memory, otherwise dormant during stack-based traversal, and registers as caches. Transparent caching requires hardware support not present on the GTX 280 but *explicitly managed* caches can be realized. One possibility is to preload data to which multiple accesses are expected into shared memory. This technique is also suggested by the CUDA optimization guidelines [NVI10a]. Another option is to store temporary data in shared memory or registers, accessing global memory only when temporary storage is exceeded. Since these caches must be managed entirely in kernel code, strategies with minimal management overhead are required.

4.3.1. Node Caching

The first caching target are *kd-tree nodes*. Whenever a node is traversed by multiple rays, placing it in a cache eliminates global memory accesses. Loading kd-tree nodes through the hardware texturing

units provides transparent caching and is used by all our ray tracing kernels. However, following section 2.2.3.1, shared memory is significantly faster than texture caches and preloading nodes into it promises further acceleration. Ensuring that rays traverse the scene coherently so that relevant parts of the kd-tree can be preloaded would require explicit synchronization. Since CUDA performs best when rays are traced independently, such synchronization is undesirable. We therefore propose to preload the part of the kd-tree that each ray is *guaranteed* to traverse instead: its top. To simplify preloading, the top of the kd-tree is padded into a complete binary tree by filling any holes due to nonexistent nodes with zero bytes. No such padding is applied to the uncached deeper levels of the kd-tree, thus introducing no significant storage overhead.

Applying the concept of persistent threads [AL09], a single CTA is resident on each SM, tracing rays by processing jobs from a queue. The cost of preloading nodes is thus amortized in several ways. First, each node needs to be preloaded only once per CTA as its threads access the same shared memory. Second, nodes preloaded at CTA launch remain in shared memory and are used for all rays processed by the CTA via the job queue. Third, all CTAs preload the same nodes. The texturing unit through which these are loaded therefore only needs to transfer each node from global memory once, serving the requests by multiple CTAs from its cache.

Every thread of a CTA preloads an equal number of nodes. The threads then synchronize to ensure their writes are mutually visible and proceed to trace rays independently, accessing the cache without requiring further synchronization.

4.3.1.1. Node Cache I

In section 3.2.1, the minimal size of a kd-tree node is established as 8 bytes, consisting of a 32-bit floating point value encoding the distance of its splitting plane from the origin, two bits expressing the normal direction of the plane and a 30-bit index referencing a pair of child nodes. While the 16 kB of shared memory per SM could theoretically hold the top 11 levels of the kd-tree with $2^{11} - 1$ nodes, some of the memory is used for internal purposes by CUDA. We therefore cache the top 10 kd-tree levels, containing $2^{10} - 1$ nodes and occupying 8 kB of shared memory. Each of the 256 threads in a CTA preloads 4 nodes.

During kd-tree traversal, a node is loaded from shared memory if its index is less than 2^{10} and from global memory via the texturing unit otherwise.

4.3.1.2. Node Cache II

Stack-based kd-tree traversal requires the node index and end of parameter interval to be pushed onto the stack for the far child whenever traversal branches. During BVH traversal, only the node index has to be pushed as the parameter interval can be reconstructed after popping by clipping the ray against the node AABB. This serves as an inspiration for an alternative cache variant which additionally stores the AABB of each cached kd-tree node. An AABB consists of six 32-bit floating point values, increasing the storage requirements per cached node to 32 bytes. The top 8 levels of the kd-tree are cached, containing $2^8 - 1$ nodes and occupying 8 kB of shared memory. Each thread preloads a single node and its AABB.

The stack used during kd-tree traversal is split into two parts as detailed in the next section. For nodes with an index lower than 2^8, only the node index is pushed onto the stack. For all other nodes, the index and the end of the parameter interval must be stored.

4.3.2. Stack Caching

Pushing onto the stack and popping its top require local memory accesses. Local memory is a dedicated region of global memory not subject to any form of caching on the GTX 280. *Stack entries* are therefore our second target for explicitly managed caching, modifying traversal to store part of the stack in shared memory or registers.

4.3.2.1. Stack Cache I

The short-stack extension for stackless traversal described in section 4.2.3 places a limited capacity stack in fast memory. Applying this idea to stack-based traversal, a short-stack in fast memory can be used to cache the most recent stack entries. Since shared memory is to serve as a kd-tree node cache, we place the short-stack in registers. A single stack entry is cached in the registers $head.node$ and $head.t_b$, initialized with $head.node \leftarrow 0$ to mark the cache as empty. The push operation is illustrated in algorithm 4.3. If $head.node > 0$, an entry is currently cached and must be flushed by pushing it onto the local memory stack. After flushing or when the cache is currently empty, $node$ and t_b are stored in the register pair.

Algorithm 4.3 Push operation with stack cache I

```
1: if head.node > 0 then
2:     push_local (head.node, head.t_b)
3: end if
4: (head.node, head.t_b) ← (node, t_b)
```

Algorithm 4.4 shows the pop operation. The start of the ray parameter interval t_a after popping is equal to the previous end of the interval t_b and is set in line 1. If the cache is currently empty, $node$ and t_b are then popped from local memory. When the cache does contain an entry, $node$ and t_b are obtained from it and the cache is marked as empty by setting $head.node \leftarrow 0$.

Algorithm 4.4 Pop operation with stack cache I

```
1: t_a ← t_b
2: if head.node = 0 then
3:     pop_local (node, t_b)
4: else
5:     (node, t_b) ← (head.node, head.t_b)
6:     head.node ← 0
7: end if
```

4.3.2.2. Stack Cache II

In section 4.3.1.2, a cache for kd-tree nodes is proposed that also stores their AABBs. When pushing a cached node onto the stack, only its index needs to be stored as the ray parameter interval can be reconstructed from the AABB. The stack is split into two parts. AABBs are cached for the top 8 levels of the kd-tree. Since only one branching point can lie on the traversal path per kd-tree level, at most eight of these nodes are simultaneously present on the stack. Each has an index in the range $0 \leq node \leq 2^8 - 1$, requiring eight bits of stack space per entry and 64 bits in total. The stack for nodes on the top 8 levels of the kd-tree can therefore compactly be stored in a single 64-bit register. Whenever a $node$ is pushed onto it, the register is shifted left by eight bits and $node$ is written to

its least significant bits. To pop the stack top, all but its eight least significant bits are masked off, obtaining *node*. The parameter interval $[t_a, t_b]$ is then computed by clipping the ray against its AABB and the register is shifted right by eight bits. For nodes lying deeper in the kd-tree, *node* and t_b are stored in a conventional stack located in local memory.

Since the CUDA platform is a 32-bit architecture, our actual implementation splits the stack over two 32-bit registers, top_1 and top_2. Both are initialized to zero. Algorithm 4.5 illustrates the push operation. If $node < 2^8$, it is stored in a register. The first four entries are placed in top_1, subsequent entries in top_2. The test in line 2 relies on every *node* pushed onto the stack being nonzero. This is the case as only far children are pushed during traversal, never the root with $node = 0$. When the *node* lies deeper in the kd-tree, line 8 pushes *node* and t_b onto the local memory stack.

Algorithm 4.5 Push operation with stack cache II ($\ll$, $\vee$ are bitwise shift and conjunction)

```
1: if node < 2^8 then
2:    if top_1 < 2^24 then
3:       top_1 ← (top_1 ≪ 8) ∨ node
4:    else
5:       top_2 ← (top_2 ≪ 8) ∨ node
6:    end if
7: else
8:    push_local (node, t_b)
9: end if
```

The pop operation follows algorithm 4.6. Nodes are pushed onto the stack in their order along a traversal path from the root and must be returned in reverse order, beginning with the node deepest in the tree. The local memory stack storing nodes beyond the top 8 levels of the kd-tree is therefore consulted first. If $stack.size > 0$, at least one entry is present on it and the stack top is popped in lines 2–3. When the local memory stack is empty, registers top_1 and top_2 are tested in line 4. If these are also empty, traversal terminates. Otherwise, the least significant eight bits are copied into *node*, returning entries from top_2 first to obtain the reverse of the order in which they are pushed. Shifting the register right by eight bits then removes the popped entry. Line 12 computes the parameter interval $[t_a, t_b]$ by clipping the ray against the *node* AABB obtained from the kd-tree node cache. Only the end of the parameter interval has to be clipped. Its start can be obtained as in line 2 by setting $t_a \leftarrow t_b$ before updating t_b.

Algorithm 4.6 Pop operation with stack cache II ($\gg$, $\wedge$ are bitwise shift and disjunction)

```
1: if stack.size > 0 then
2:    t_a ← t_b
3:    pop_local (node, t_b)
4: else if top_1 > 0 or top_2 > 0 then
5:    if top_2 > 0 then
6:       node ← top_2 ∧ 255
7:       top_2 ← top_2 ≫ 8
8:    else
9:       node ← top_1 ∧ 255
10:      top_1 ← top_1 ≫ 8
11:   end if
12:   [t_a, t_b] ← [t_b, t_max] ⋂ AABB (node)
13: end if
```

4.3.2.3. Stack Cache III

A third modification caches not far children pushed onto the stack but leaves reached during traversal, postponing their visits. Upon reaching a leaf, *node* and t_a are stored in registers $leaf.node$ and $leaf.t_a$ and traversal continues by popping the top of the stack. Only when a *second* leaf is reached are the postponed and current leaves visited with parameter intervals $[leaf.t_a, t_a]$ and $[t_a, t_b]$. If no hit is found in either, traversal continues by clearing the cache and popping the stack top.

Postponing leads to *speculative* inner node traversal as the ray may actually terminate at a hit in the leaf, making further traversal unnecessary. The motivation is to extract more common instructions and increase parallel processing unit utilization, reducing overall computational cost. If one thread must traverse m inner nodes to reach a leaf and n to reach another while the remainder of the warp needs n and m traversals, the entire warp executes $\max(m, n)$ traversals before each leaf visit with some of the execution units masked off after the first $\min(m, n)$. By postponing, only $m+n$ traversals are executed to reach both leaves with all threads operating in lock-step.

In its original form [AL09], speculative inner node traversal stops when all threads of a warp have postponed a leaf. This allows postponed leaves to be visited sooner and avoids traversal until each thread reaches a second leaf, increasing processing unit utilization when threads require different numbers of traversals in total and some must be masked off. However, the need for threads within a warp to communicate is introduced. While *warp voting* is inexpensive in CUDA, the entire warp must execute the voting instruction. When ray tracing is used as part of a more complex rendering kernel, only a subset of the threads may be tracing rays at the same time, thus following different code paths within each warp and precluding the use of voting. This is exemplified by our implementation with primary, reflection and shadow rays in a single kernel.

4.3.3. Results and Discussion

Bar the last, all caching methods target global memory accesses, the key bottleneck of the CUDA platform. As our findings from section 4.2.4 caution that even a small increase in computational cost may negate benefits, we evaluate the impact on memory accesses, instruction counts and resulting frame rates for different combinations of the proposed caching methods. The benchmark environment and settings are identical to those used in section 4.2.4. A kd-tree is constructed for each scene from section A.2.1 with the SIROH heuristic and traversed by the CUDA ray tracing kernel of code base II from section A.1.2. The *baseline* is traversal with a stack in local memory and no explicitly managed caches. Results are given as averages over a flight through each scene.

Table 4.8(a) shows the number of inner nodes loaded via the texturing unit per ray. Stack caching has no influence on it (column 3) with the exception of stack cache III which loads 13.2% more nodes on average due to speculative traversal (column 4). The node caches are very successful at eliminating loads via the texturing unit by delivering nodes from shared memory. Relative to the baseline, node cache II reduces the number of loads by an average of 60.7% (column 6). Node cache I covers a larger part of the kd-tree and removes 72.3% of the loads (columns 2, 5). With speculative traversal, the average number of loads lies 63.6% below the baseline (column 7).

Accesses to the local memory stack are counted in table 4.8(b). Since node caching does not affect these, several columns show identical results. With stack cache I, an average 71.1% of the stack operations (push and pop) no longer access local memory. This shows that most nodes pushed onto the stack are popped off again before further branching and a cache with capacity for a single entry is

	1	2	3	4	5	6	7
Node Cache	Baseline	I	None	None	I	II	I
Stack Cache		None	I	III	I	II	III
Scene 6	7.31	−96.4%	±0.0%	+16.0%	−96.4%	−93.2%	−94.3%
Sponza	29.81	−63.8%	±0.0%	+13.6%	−63.8%	−49.9%	−52.0%
Sibenik	37.63	−60.9%	±0.0%	+11.1%	−60.9%	−43.8%	−49.1%
Fairy	37.43	−62.6%	±0.0%	+13.0%	−62.6%	−52.5%	−53.4%
Conference	19.57	−77.7%	±0.0%	+12.5%	−77.7%	−64.3%	−68.9%
Average		−72.3%	±0.0%	+13.2%	−72.3%	−60.7%	−63.6%
Std. Dev.		15.0%	0.0%	1.8%	15.0%	19.6%	18.8%

(a) Inner nodes loaded via the texturing unit per ray

	1	2	3	4	5	6	7
Node Cache	Baseline	I	None	None	I	II	I
Stack Cache		None	I	III	I	II	III
Scene 6	2.01	±0.0%	−88.7%	+2.2%	−88.7%	−88.1%	+2.2%
Sponza	12.31	±0.0%	−63.7%	+8.9%	−63.7%	−48.1%	+8.9%
Sibenik	14.46	±0.0%	−60.4%	+5.0%	−60.4%	−37.1%	+5.0%
Fairy	16.96	±0.0%	−68.6%	+4.1%	−68.6%	−48.7%	+4.1%
Conference	6.69	±0.0%	−74.1%	+5.8%	−74.1%	−59.7%	+5.8%
Average		±0.0%	−71.1%	+5.2%	−71.1%	−56.4%	+5.2%
Std. Dev.		0.0%	11.1%	2.5%	11.1%	19.4%	2.5%

(b) Stack entries accessed in local memory per ray

	1	2	3	4	5	6	7
Node Cache	Baseline	I	None	None	I	II	I
Stack Cache		None	I	III	I	II	III
Scene 6	97.2%	−0.3%	+0.4%	+0.1%	−0.3%	−0.4%	−0.4%
Sponza	90.8%	−1.6%	+0.4%	−0.4%	−1.4%	−1.6%	−2.1%
Sibenik	90.1%	−3.1%	+0.3%	+0.1%	−2.6%	−2.3%	−2.5%
Fairy	86.3%	−4.1%	−0.1%	−1.4%	−4.0%	−4.1%	−5.6%
Conference	90.7%	−2.0%	−0.3%	−1.2%	−1.7%	−2.1%	−3.4%
Average	91.0%	−2.2%	+0.1%	−0.6%	−2.0%	−2.1%	−2.8%
Std. Dev.	3.9%	1.5%	0.3%	0.7%	1.4%	1.3%	1.9%

(c) Texture cache hit ratios

	1	2	3	4	5	6	7
Node Cache	Baseline	I	None	None	I	II	I
Stack Cache		None	I	III	I	II	III
Scene 6	74.51	−75.6%	−19.1%	+13.0%	−94.7%	−92.1%	−73.5%
Sponza	336.95	−45.2%	−18.6%	+12.2%	−63.8%	−49.4%	−34.2%
Sibenik	416.72	−44.0%	−16.8%	+9.4%	−60.8%	−41.9%	−34.1%
Fairy	435.06	−43.1%	−21.4%	+10.2%	−64.5%	−51.3%	−35.5%
Conference	210.03	−57.9%	−18.9%	+10.8%	−76.8%	−63.2%	−49.9%
Average		−53.2%	−19.0%	+11.1%	−72.1%	−59.6%	−45.4%
Std. Dev.		13.9%	1.7%	1.5%	14.1%	19.7%	17.1%

(d) Global memory bytes accessed per ray in (a) and (b) with a simplified memory subsystem

Table 4.8.: Global memory accesses during inner node traversal with explicitly managed caches, relative to baseline without

	1	2	3	4	5	6	7
Node Cache	Baseline	I	None	None	I	II	I
Stack Cache		None	I	III	I	II	III
Scene 6	674.2	+7.7%	+2.4%	+11.8%	+8.7%	+17.1%	+18.8%
Sponza	2249.9	+12.8%	+4.1%	+44.2%	+14.7%	+23.3%	+58.3%
Sibenik	2294.9	+15.2%	+4.6%	+39.6%	+17.2%	+26.1%	+55.7%
Fairy	3129.6	+13.5%	+4.6%	+91.5%	+15.9%	+25.8%	+110.7%
Conference	2246.2	+8.5%	+2.5%	+53.6%	+9.3%	+17.3%	+63.4%
Average		+11.6%	+3.6%	+48.1%	+13.2%	+21.9%	+61.4%
Std. Dev.		3.3%	1.1%	28.8%	3.9%	4.5%	32.7%

Table 4.9.: Instructions per ray with explicitly managed caches, relative to baseline without

sufficient to serve the majority of operations (columns 3, 5). Stack cache II moves the stack for the top 8 levels of the kd-tree into registers. A considerable reduction in accesses to the local memory stack is again achieved, averaging 56.4% (column 6). With stack cache III, speculative traversal causes the average number of accesses to increase by 5.2% instead (columns 4, 7).

Table 4.8(c) lists texture cache hit ratios. These are affected by nodes loaded via the texturing unit and in turn affect the performance of such loads. While hit ratios are available for the first TPC only, thus varying between experiments as rays are nondeterministically assigned to processing units, the conclusion can be drawn that they do not significantly change with stack or node caching. As described in section 2.2.3.1, the remainder of the complex CUDA memory subsystem is largely undocumented. The actual performance impact of changes in memory accesses is thus difficult to predict, depending on which nodes are cached and which local stack accesses are coalesced.

A *qualitative* assessment of the impact is possible with a simplified memory subsystem model that serves every access independently, avoiding the unpredictable influences of caching and coalescing. With inner nodes and stack entries occupying 8 bytes each, tables 4.8(a) and (b) combine to 4.8(d). Node cache I is seen to significantly reduce the global memory bandwidth required (column 2). Stack cache I results in a smaller reduction (column 3), both of which add when the caches are combined (column 5). Stack and node cache II combine to a somewhat smaller reduction (column 6). The speculative traversal of stack cache III increases bandwidth requirements (column 4) but a reduction below the baseline results when combined with node cache I (column 7). Populating node caches requires only 240 kB of bandwidth per kernel as 30 CTAs are launched and preload 8 kB each.

Computational cost is illustrated by table 4.9 as the number of instructions required to trace a ray. Stack cache I adds 3.6% (column 3), node cache I 11.6% (column 2) instructions to the baseline on average. Combining both approximately sums their overheads (column 5). Stack and node cache II together add an average of 21.9% instructions due to the more complex stack representation and ray clipping against node AABBs (column 6). Stack cache III proves to have the *opposite* of its intended effect. Each instruction executed by a warp is counted for all its threads, whether participating or masked off. Extracting common instructions should thus decrease the measured instruction count. Instead, the overheads of additional traversals and cache management dominate, leading to an average increase by 48.1% (column 4) or 61.4% if node cache I is also used (column 7).

The resulting frame rates are presented in table 4.10. Stack cache III requires more global memory accesses and instructions, leading to a slowdown averaging 31.3% (column 4). Adding node cache I reduces global memory accesses but increases instruction count further, making the kernel compute

	1	2	3	4	5	6	7
Node Cache	Baseline	I	None	None	I	II	I
Stack Cache		None	I	III	I	II	III
Scene 6	392.6	+2.4%	+3.8%	−6.2%	+6.1%	+2.5%	−3.2%
Sponza	213.7	−0.3%	+1.3%	−29.2%	+2.3%	−4.3%	−29.1%
Sibenik	194.6	−1.0%	+1.2%	−27.1%	+1.8%	−6.0%	−27.2%
Fairy	149.0	−1.5%	+1.5%	−51.7%	+1.0%	−5.9%	−52.5%
Conference	172.0	−0.1%	+0.4%	−42.1%	+0.8%	−2.9%	−42.0%
Average		−0.1%	+1.7%	−31.3%	+2.4%	−3.3%	−30.8%
Std. Dev.		1.5%	1.3%	17.2%	2.2%	3.5%	18.5%

Table 4.10.: Ray tracing frame rates with explicitly managed caches, relative to baseline without

bound with a slowdown of 30.8% on average (column 7). The combination of stack and node cache II also appears compute bound. Despite a considerable reduction in global memory accesses, the higher instruction count leads to a slowdown for all but the simplest scene, averaging 3.3% (column 6).

Stack cache I confirms the motivation behind our work. Despite a small increase in instruction count, the reduction in global memory accesses with this method proves beneficial and yields a speedup averaging 1.7% (column 3). Node cache I offers larger memory access reduction at higher computational cost, making the kernel compute bound and leading to a slowdown for all but the simplest scene (column 2). Both methods combined provide an average speedup of 2.4% (column 5). This appears to contradict the conclusion that node cache I alone makes ray tracing compute bound as instruction count and performance are both higher when the caches are combined. The explanation lies in the complexity of the CUDA platform. Actual performance is difficult to predict not only for the memory subsystem but also for the processing units. Changes in the number of instructions serve as an indication but benchmarks are required to measure actual performance.

Recent work [AK10] evaluates explicitly managed caches for ray tracing on a hypothetical manycore architecture. Rays are enqueued at predefined points in the spatial index, coherently traversing sets of nodes preloaded into shared memory. This approach requires a reorganization of the entire ray tracing process and an intricate job scheduler. Stack caching inspired by short-stack is also proposed. Four entries are held in registers, storing only node indexes as a BVH is used. The authors report significant bandwidth reductions. However, computational cost, identified as an important bottleneck in our results, is not considered.

In summary, we find that explicitly managed caches in shared memory and registers can further improve the performance of ray tracing with stack-based kd-tree traversal in CUDA. The complexity of the CUDA platform and lack of documentation make it impossible to predict actual performance as memory access and execution patterns change, emphasizing the need for benchmark evaluation. Although the average speedup of our best caching method is only 2.4%, the baseline already is a high performance ray tracer for which large gains are unlikely. We also find that speculative traversal, yielding a speedup with warp voting [AL09], leads to slowdowns when warp voting is not possible.

5. Density Estimation

Photon mapping links photons traced from the light sources and rays traced from the virtual camera by *weighted kernel density estimation*. Radiance reflected at query point $\vec{x}$ into query direction $\vec{\omega}$ is estimated as the density of reflected intensity per unit area. A sample of the intensity can be obtained at every nearby photon interaction by reflecting its flux $\Phi_{i,j}$ from the incident direction $\vec{\omega}_{i,j}$ via the BRDF f. The density per area is estimated by centering a smoothing kernel K with bandwidth h around each interaction and summing their weighted contributions.

Using k-th nearest neighbor density estimation as originally proposed adapts the bandwidth to the local interaction density, automatically balancing variance and bias. However, this approach requires candidate interactions to be retrieved and placed in a priority queue to identify those k nearest $\vec{x}$. On the CUDA platform, the priority queues exceed available shared memory and must be placed in local memory, leading to a large overhead of slow memory accesses.

Variable kernel density estimation avoids this inefficiency by assigning each photon interaction an individual bandwidth h_j and summing the independent contributions of those whose kernel support regions overlap $\vec{x}$. The challenge lies in determining bandwidths adaptive to the local illumination. Existing techniques as listed in section 3.4.2.3 either reintroduce the need to locate nearby interactions when computing h_j or adapt to a subset of the factors affecting illumination only.

Photon differentials adapt to the influences of sampling at emission, propagation through space and specular interactions. In this chapter, we extend the adaptation to diffuse reflections and Russian roulette while reducing storage and bandwidth requirements. We then describe how an anisotropic kernel support region is derived and the kernel efficiently evaluated for it. An adaptation dampening is finally introduced that approximates the influence of illumination arriving via different paths.

5.1. Photon Differentials

The photon differentials approach uses the concept of ray differentials. Each photon path is modeled as a series of functions modifying position and direction. *Differentials*, the partial derivatives of the composite function, are tracked with every photon. These allow the positions of neighboring photons to be estimated by first order Taylor approximation, spanning a footprint at each interaction that defines an anisotropic kernel support region adaptive to the local interaction density.

Equations for initializing and updating differentials are provided in section 3.1.2.2. These assume that the path is a function of two parameters p_1, p_2 sampled at emission. Propagation and specular interactions conform to this model. Interactions with other surface types or Russian roulette do not as each of these introduces additional parameters. While path differentials do provide an extension to higher parameter counts, they incur the overheads of growing storage and complex calculations. For interactive rendering, less expensive solutions with fixed storage are desirable. We show how to approximatively account for diffuse reflection and Russian roulette while tracking differentials with respect to two parameters only for every photon.

5.1.1. Initialization

Differential initialization is described in section 3.1.2.2 for deterministic ray emission from a virtual camera only. Stochastic photon emission from a light source therefore requires a rederivation of the initial differentials and the distances in parameter space between neighbors. With $\vec{z}_l$ the position of a point light, R a matrix expressing its rotation and $p_1 \in [0, \pi]$, $p_2 \in [0, 2\pi)$ the inclination and azimuth in its local coordinate frame, the initial position and direction of a photon can be parameterized as

$$\vec{z}\,(p_1, p_2) = \vec{z}_l, \qquad \vec{\omega}\,(p_1, p_2) = R \begin{pmatrix} \cos p_2 \sin p_1 \\ \sin p_2 \sin p_1 \\ \cos p_1 \end{pmatrix}. \tag{5.1}$$

Initial photon differentials are obtained by differentiating these equations,

$$\frac{\partial \vec{z}}{\partial p_1}\,(p_1, p_2) = \vec{0}, \qquad \frac{\partial \vec{\omega}}{\partial p_1}\,(p_1, p_2) = R \begin{pmatrix} \cos p_2 \cos p_1 \\ \sin p_2 \cos p_1 \\ -\sin p_1 \end{pmatrix}, \tag{5.2}$$

$$\frac{\partial \vec{z}}{\partial p_2}\,(p_1, p_2) = \vec{0}, \qquad \frac{\partial \vec{\omega}}{\partial p_2}\,(p_1, p_2) = R \begin{pmatrix} -\sin p_2 \sin p_1 \\ \cos p_2 \sin p_1 \\ 0 \end{pmatrix}. \tag{5.3}$$

The distances Δp_1, Δp_2 in parameter space between a photon and its neighbors depend on the sampling of directions. As noted in section 3.4.1, variance is minimized when all photons have equal flux. This is ensured at emission by importance-sampling light source intensity. In the simplest case of an isotropic source, flux is emitted evenly into the sphere of directions. This readily generalizes to a spotlight emitting flux evenly into a cone of directions around the zenith of its local coordinate frame with apex angle $2\alpha \in [0, 2\pi]$. The solid angle into which flux is emitted becomes the area on the unit sphere corresponding to $p_1 \in [0, \alpha]$, $p_2 \in [0, 2\pi)$. Due to the distortion in the mapping from spherical coordinates to sphere surface, the area element is $\sin p_1 \, dp_2 \, dp_1$ and the solid angle

$$\Omega_\alpha = \int_0^\alpha \int_0^{2\pi} \sin p_1 \, dp_2 \, dp_1 = 2\pi \left(1 - \cos \alpha\right).$$

Even flux distribution corresponds to a constant intensity $I\,(\vec{\omega}) = \frac{\Phi}{2\pi(1-\cos\alpha)}$ throughout this solid angle. As a function of p_1, p_2, it is subject to the distorted mapping so that $I\,(p_1, p_2) = \frac{\Phi \sin p_1}{2\pi(1-\cos\alpha)}$. The probability density function importance-sampling intensity thus is

$$p\,(p_1, p_2) = \begin{cases} \dfrac{\sin p_1}{2\pi\,(1 - \cos\alpha)} & \text{if } p_1 \leq \alpha, \\ 0 & \text{else.} \end{cases}$$

Computing the marginal probability density functions shows that $p\,(p_1, p_2)$ is separable and p_1, p_2 may be sampled independently,

$$p\,(p_1) = \int_0^{2\pi} \frac{\sin p_1}{2\pi\,(1 - \cos\alpha)}\, dp_2 = \frac{\sin p_1}{1 - \cos\alpha}, \tag{5.4}$$

$$p\,(p_2) = \int_0^{\alpha} \frac{\sin p_1}{2\pi\,(1 - \cos\alpha)}\, dp_1 = \frac{1}{2\pi}. \tag{5.5}$$

Parameters distributed according to equations 5.4 and 5.5 are obtained by transforming uniformly distributed samples $y_1 \in [\cos\alpha, 1]$, $y_2 \in [0, 2\pi)$ via their inverted cumulative distribution functions to $p_1 = \arccos y_1$, $p_2 = y_2$, as noted for spheres [Sea96] and recently for cones of directions [Ken07].

With stochastic sampling, the actual neighboring photons are unknown until all have been emitted. Additionally, each neighbor is likely to differ in p_1 and p_2. This is addressed by setting Δp_1, Δp_2 to the *expected* distances between a photon and neighbors differing in p_1 or p_2 only. Using first-order Taylor approximation, the directions of these are estimated as $\vec{\omega}(p_1, p_2) + \Delta p_1 \frac{\partial \vec{\omega}}{\partial p_1}(p_1, p_2)$ and $\vec{\omega}(p_1, p_2) + \Delta p_2 \frac{\partial \vec{\omega}}{\partial p_2}(p_1, p_2)$. With uniformly distributed photon directions, the expected lengths of the difference vectors between these and $\vec{\omega}(p_1, p_2)$ are equal. As n photons are emitted into a solid angle of $2\pi(1 - \cos\alpha)$, they are furthermore expected to span a solid angle of $\frac{2\pi(1-\cos\alpha)}{n}$ so that

$$\mathbb{E}\left[\left\|\Delta p_1 \frac{\partial \vec{\omega}}{\partial p_1}(p_1, p_2)\right\|\right] = \mathbb{E}\left[\left\|\Delta p_2 \frac{\partial \vec{\omega}}{\partial p_2}(p_1, p_2)\right\|\right] = \sqrt{\frac{2\pi(1 - \cos\alpha)}{n}}. \tag{5.6}$$

The lengths of the initial differentials in equations 5.2 and 5.3 are

$$\left\|\frac{\partial \vec{\omega}}{\partial p_1}(p_1, p_2)\right\| = 1, \qquad \left\|\frac{\partial \vec{\omega}}{\partial p_2}(p_1, p_2)\right\| = \sin p_1. \tag{5.7}$$

Combining equations 5.6 and 5.7, the expected distances between neighboring photons are

$$\Delta p_1 \approx \mathbb{E}[\Delta p_1] = \sqrt{\frac{2\pi(1 - \cos\alpha)}{n}}, \qquad \Delta p_2 \approx \mathbb{E}[\Delta p_2] = \sqrt{\frac{2\pi(1 - \cos\alpha)}{n}} \frac{1}{\sin p_1}.$$

In total, four differentials and two distances in parameter space are tracked with each photon, allowing its footprint spanning vectors to be computed according to equations 3.3 and 3.4 as

$$\vec{\Delta}_1(p_1, p_2) \approx \Delta p_1 \frac{\partial \vec{z}}{\partial p_1}(p_1, p_2), \qquad \vec{\Delta}_2(p_1, p_2) \approx \Delta p_2 \frac{\partial \vec{z}}{\partial p_2}(p_1, p_2). \tag{5.8}$$

We observe that the equations from section 3.1.2.2 for updating differentials preserve a scaling applied to the differentials with respect to a parameter p. The corresponding distance in parameter space remains constant and therefore also preserves any scaling. If the initial differentials with respect to p_1, p_2 are scaled by c_1, c_2 and the distances in parameter space by c_1^{-1}, c_2^{-1}, the footprint spanning vectors in equation 5.8 thus remain unchanged. This can be exploited to reduce storage requirements. At emission, initial differentials with respect to p_1 are scaled by $\sqrt{2^{-1}(1 - \cos\alpha)}$ and those with respect to p_2 by $\sqrt{2^{-1}(1 - \cos\alpha)}\sin^{-1} p_1$, yielding

$$\frac{\partial \vec{z}}{\partial p_1}(p_1, p_2) = \vec{0}, \qquad \frac{\partial \vec{\omega}}{\partial p_1}(p_1, p_2) = \sqrt{\frac{1}{2}(1 - \cos\alpha)}\, R \begin{pmatrix} \cos p_2 \cos p_1 \\ \sin p_2 \cos p_1 \\ -\sin p_1 \end{pmatrix}, \tag{5.9}$$

$$\frac{\partial \vec{z}}{\partial p_2}(p_1, p_2) = \vec{0}, \qquad \frac{\partial \vec{\omega}}{\partial p_2}(p_1, p_2) = \sqrt{\frac{1}{2}(1 - \cos\alpha)}\, R \begin{pmatrix} -\sin p_2 \\ \cos p_2 \\ 0 \end{pmatrix}. \tag{5.10}$$

The distances in parameter space are scaled by the inverses of these factors and become

$$\Delta p_1 = \Delta p_2 = \sqrt{\frac{4\pi}{n}}. \tag{5.11}$$

Photon differentials initialized according to equations 5.9 and 5.10 must be tracked with each photon and updated as it is traced through the scene. The distances in parameter space, however, are now replaced by a global constant. Not having to track these with each photon reduces the storage and bandwidth requirements during photon tracing.

5.1.2. Specular Reflection

Equations for updating differentials upon specular reflection are provided in section 3.1.2.4. These require the partial derivatives of the surface normal $\vec{n}\,(p_1, p_2)$ to be computed. A method is proposed in the original ray differentials framework [Ige99] which requires three planes to be stored for each triangle that describe its edges. Storage overhead and a corresponding bandwidth cost are incurred. We derive an alternative solution that allows the partial derivatives to be computed without the need for any additional storage or bandwidth.

According to section 3.1.2.4, the normal is interpolated as $\vec{n}\,(u, v, w) = \widehat{w\,\vec{n}_0 + u\,\vec{n}_1 + v\,\vec{n}_2}$ where u, v, w are the barycentric coordinates of hit point $\vec{z}\,(p_1, p_2)$. Since $w = 1 - u - v$, a coordinate pair $\vec{\lambda} = (u, v)$ is sufficient, allowing the normal to be interpolated as $\vec{n}(\vec{\lambda}) = \widehat{\vec{n}_0 + u\,\vec{n}'_1 + v\,\vec{n}'_2}$ from vectors $\vec{n}_0$, $\vec{n}'_1 = \vec{n}_1 - \vec{n}_0$, $\vec{n}'_2 = \vec{n}_2 - \vec{n}_0$ stored for each triangle. The partial derivative of the normal with respect to either parameter p thus decomposes by the chain rule into

$$\frac{\partial \vec{n}}{\partial p}\,(p_1, p_2) = \frac{\partial \vec{n}}{\partial p}(\vec{\lambda}\,(\vec{z}\,(p_1, p_2))) = \frac{\partial \vec{n}}{\partial \vec{\lambda}}(\vec{\lambda}\,(\vec{z}\,(p_1, p_2)))\,\frac{\partial \vec{\lambda}}{\partial \vec{z}}\,(\vec{z}\,(p_1, p_2))\;\frac{\partial \vec{z}}{\partial p}\,(p_1, p_2)\,. \tag{5.12}$$

The normal $\vec{n}(\vec{\lambda})$ is the product of the unnormalized interpolation result $\vec{n}'(\vec{\lambda}) = \vec{n}_0 + u\,\vec{n}'_1 + v\,\vec{n}'_2$ and the inverse of its length, $\|\vec{n}'(\vec{\lambda})\|^{-1}$. Differentiating this product with respect to the components of $\vec{\lambda}$ yields the first term of equation 5.12,

$$\frac{\partial \vec{n}}{\partial \vec{\lambda}}(\vec{\lambda}) = \left(\frac{\vec{n}'_1 - (\vec{n}'_1 \cdot \vec{n}(\vec{\lambda}))\vec{n}(\vec{\lambda})}{\|\vec{n}'(\vec{\lambda})\|}, \frac{\vec{n}'_2 - (\vec{n}'_2 \cdot \vec{n}(\vec{\lambda}))\vec{n}(\vec{\lambda})}{\|\vec{n}'(\vec{\lambda})\|}\right). \tag{5.13}$$

Equation 5.13 is a combination of intermediary results computed during normal interpolation that requires no additional storage, bandwidth or expensive calculations. To obtain the second term of equation 5.12, the relationship between the barycentric $\vec{\lambda}$ and Cartesian $\vec{z}$ hit point coordinates must be analyzed. For the efficient Wald intersection test of section 3.1.3.2, this follows from algorithms 3.2 and 3.3. The relevant components of these are

$$k = \arg\max |\vec{n}_i|\,, \qquad \text{algorithm 3.2, line 1} \tag{5.14}$$

$$(i, j) = (k + 1, k + 2) \mod 3, \qquad \text{algorithm 3.2, line 2} \tag{5.15}$$

$$\vec{e}'_1 = (-e_{1j}, e_{1i}, \det(\vec{e}_{1ji}, \vec{v}_{0ji}))^T \det^{-1}(\vec{e}_{1ij}, \vec{e}_{2ij})\,, \qquad \text{algorithm 3.2, line 4} \tag{5.16}$$

$$\vec{e}'_2 = (e_{2j}, -e_{2i}, \det(\vec{e}_{2ij}, \vec{v}_{0ij}))^T \det^{-1}(\vec{e}_{1ij}, \vec{e}_{2ij})\,, \qquad \text{algorithm 3.2, line 5} \tag{5.17}$$

$$(\vec{x}', \vec{\omega}') = (\vec{x}, \vec{\omega})_{ijk}\,, \qquad \text{algorithm 3.3, line 2} \tag{5.18}$$

$$\vec{h} = \vec{x}'_{xy} + t\,\vec{\omega}'_{xy}, \qquad \text{algorithm 3.3, line 7} \tag{5.19}$$

$$(u, v) = (\vec{e}'_{2xy} \cdot \vec{h} + e'_{2z}, \vec{e}'_{1xy} \cdot \vec{h} + e'_{1z}). \qquad \text{algorithm 3.3, line 8} \tag{5.20}$$

In equations 5.14 and 5.15, i, j, k are defined as a permutation of coordinate axis indexes $(0, 1, 2)$. Equations 5.18 and 5.19 thus extract two of the three Cartesian hit point coordinates $\vec{z} = \vec{x} + t\,\vec{\omega}$

such that $\vec{h} = \vec{z}_{ij}$ and equation 5.20 uses the precalculated values from equations 5.16 and 5.17 to compute the barycentric hit point coordinates as

$$\vec{\lambda}\left(\vec{z}\right) = \begin{pmatrix} \vec{e}'_{2xy} \cdot \vec{z}_{ij} + \vec{e}'_{2z} \\ \vec{e}'_{1xy} \cdot \vec{z}_{ij} + \vec{e}'_{1z} \end{pmatrix}.$$

Taking the derivatives with respect to the components of $\vec{z}$ in their permuted i, j, k order yields

$$\frac{\partial \vec{\lambda}}{\partial \vec{z}_{ijk}}\left(\vec{z}\right) = \begin{pmatrix} e'_{2x} & e'_{2y} & 0 \\ e'_{1x} & e'_{1y} & 0 \end{pmatrix}.$$

A derivative with respect to $\vec{z}$ is obtained by undoing the permutation,

$$\frac{\partial \vec{\lambda}}{\partial \vec{z}}\left(\vec{z}\right) = \begin{cases} \begin{pmatrix} 0 & e'_{2x} & e'_{2y} \\ 0 & e'_{1x} & e'_{1y} \end{pmatrix} & \text{if } k = 0, \\ \begin{pmatrix} e'_{2y} & 0 & e'_{2x} \\ e'_{1y} & 0 & e'_{1x} \end{pmatrix} & \text{if } k = 1, \\ \begin{pmatrix} e'_{2x} & e'_{2y} & 0 \\ e'_{1x} & e'_{1y} & 0 \end{pmatrix} & \text{if } k = 2. \end{cases} \tag{5.21}$$

Equation 5.21 provides the second term of equation 5.12 by permuting data loaded during hit point computation, requiring no additional storage, bandwidth or calculations. The final term needed to evaluate equation 5.12 is the current differential $\frac{\partial \vec{z}}{\partial p}\left(p_1, p_2\right)$. By multiplying equations 5.13, 5.21 and this differential, the partial derivative $\frac{\partial \vec{n}}{\partial p}\left(p_1, p_2\right)$ of the surface normal is thus efficiently obtained. To update differentials during specular reflection, partial derivatives with respect to both parameters are required. Since equations 5.13 and 5.21 are independent of p, their product can be reused, reducing the computation for the second parameter further.

5.1.3. Diffuse Reflection

Upon encountering a diffuse surface, flux is redistributed into the entire hemisphere above it, regardless of incident direction. The photon path thus becomes the function of two *additional* parameters p'_1, p'_2 sampled to choose a new photon direction. With $R\left(p_1, p_2\right)$ a matrix expressing the surface orientation at hit point $\vec{z}\left(p_1, p_2\right)$ and $p'_1 \in \left[0, \frac{\pi}{2}\right]$, $p'_2 \in \left[0, 2\pi\right)$ the inclination and azimuth in the local coordinate frame, the photon position and direction after diffuse reflection are

$$\vec{z}'\left(p_1, p_2, p'_1, p'_2\right) = \vec{z}\left(p_1, p_2\right), \qquad \vec{\omega}\left(p_1, p_2, p'_1, p'_2\right) = R\left(p_1, p_2\right) \begin{pmatrix} \cos p'_2 \sin p'_1 \\ \sin p'_2 \sin p'_1 \\ \cos p'_1 \end{pmatrix}.$$

We avoid the overhead of having to track more differentials with each diffuse reflection by treating it as an *absorption* and a *reemission*. The photon is thus considered to originate at the hit point on the diffuse surface with parameters p'_1, p'_2 only so that

$$\vec{z}'\left(p'_1, p'_2\right) = \vec{z}\left(p_1, p_2\right), \qquad \vec{\omega}\left(p'_1, p'_2\right) = R\left(p_1, p_2\right) \begin{pmatrix} \cos p'_2 \sin p'_1 \\ \sin p'_2 \sin p'_1 \\ \cos p'_1 \end{pmatrix}.$$

Differentials identical to equations 5.2 and 5.3 with p'_1, p'_2 substituted for p_1, p_2 result. However, the distances $\Delta p'_1$, $\Delta p'_2$ in parameter space must be rederived due to a different sampling of photon directions. A diffuse surface redirects *radiance* evenly into the hemisphere of directions. Intensity is proportional to radiance projected onto the surface so that $I(\vec{\omega}') \propto |\vec{\omega}' \cdot \vec{n}|$. In spherical coordinates, $|\vec{\omega}' \cdot \vec{n}| = \cos p'_1$ and with the distorted mapping to the hemisphere, $I(p'_1, p'_2) \propto \cos p'_1 \sin p'_1$. The probability density function importance-sampling intensity thus is

$$p(p'_1, p'_2) = \frac{\cos p'_1 \sin p'_1}{\pi}.$$

Marginal probability density functions show that p'_1, p'_2 may be sampled independently,

$$p(p'_1) = \int_0^{2\pi} \frac{\cos p'_1 \sin p'_1}{\pi} \, dp'_2 = 2 \cos p'_1 \sin p'_1, \tag{5.22}$$

$$p(p'_2) = \int_0^{\frac{\pi}{2}} \frac{\cos p'_1 \sin p'_1}{\pi} \, dp'_1 = \frac{1}{2\pi}. \tag{5.23}$$

The cumulative distribution functions are $F(p'_2) = (2\pi)^{-1} p'_2$ and

$$F(p'_1) = \int_0^{p'_1} 2 \cos p \sin p \, dp = 1 - \cos^2 p'_1.$$

Parameters distributed according to equations 5.22 and 5.23 are obtained by drawing uniformly distributed samples $y'_1 \in [0, 1]$, $y'_2 \in [0, 2\pi)$ and transforming these into $p'_1 = \arccos \sqrt{1 - y'_1}$, $p'_2 = y'_2$ via the inverted cumulative distribution functions.

Since each photon is traced independently, no neighbors at the same remission point exist. This is addressed by using *global deltas* [SW01] as described in section 3.1.2.5. With n photons traced in total, the distances to neighbors are computed by assuming all sample the same parameter space. As in section 5.1.1, $\Delta p'_1$, $\Delta p'_2$ are additionally set to expected values due to stochastic sampling. The difference vectors between the photon direction and those of its neighbors are again estimated using first-order Taylor approximation as $\Delta p'_1 \frac{\partial \vec{\omega}'}{\partial p'_1}(p'_1, p'_2)$ and $\Delta p'_2 \frac{\partial \vec{\omega}'}{\partial p'_2}(p'_1, p'_2)$. With the hemisphere of directions sampled proportional to $I(\vec{\omega}') \propto |\vec{\omega}' \cdot \vec{n}| = \cos p'_1$, the expected solid angle spanned by neighboring photon directions is $c \cos^{-1} p'_1$, proportional to the inverse. All n photons together are expected to cover the entire hemisphere from which the proportionality constant c follows,

$$\mathbb{E}\left[n \frac{c}{\cos p'_1}\right] = 2\pi \Leftrightarrow c = \frac{\pi}{n}.$$

Approximating the distribution of photon directions as locally uniform, the expected lengths of both difference vectors are equal so that

$$\mathbb{E}\left[\left\|\Delta p'_1 \frac{\partial \vec{\omega}'}{\partial p'_1}(p'_1, p'_2)\right\|\right] = \mathbb{E}\left[\left\|\Delta p'_2 \frac{\partial \vec{\omega}'}{\partial p'_2}(p'_1, p'_2)\right\|\right] = \sqrt{\frac{\pi}{n \cos p'_1}}. \tag{5.24}$$

Since the differentials are identical to equations 5.2 and 5.3, their lengths follow equation 5.7. Combining these with equation 5.24, the expected distances between neighboring photons are

$$\Delta p'_1 \approx \mathbb{E}[\Delta p'_1] = \sqrt{\frac{\pi}{n \cos p'_1}}, \qquad \Delta p'_2 \approx \mathbb{E}[\Delta p'_2] = \sqrt{\frac{\pi}{n \cos p'_1}} \frac{1}{\sin p'_1}.$$

The need to track distances in parameter space can be eliminated analogously to section 5.1.1. Differentials are scaled by $\sqrt{(4\cos p_1')^{-1}}$ respectively $\sqrt{(4\cos p_1')^{-1}}\sin^{-1} p_1'$ to obtain

$$\frac{\partial \vec{z}''}{\partial p_1'}(p_1', p_2') = \vec{0}, \qquad \frac{\partial \vec{\omega}'}{\partial p_1'}(p_1', p_2') = \sqrt{\frac{1}{4\cos p_1'}} R(p_1, p_2) \begin{pmatrix} \cos p_2' \cos p_1' \\ \sin p_2' \cos p_1' \\ -\sin p_1 \end{pmatrix}, \tag{5.25}$$

$$\frac{\partial \vec{z}''}{\partial p_2'}(p_1', p_2') = \vec{0}, \qquad \frac{\partial \vec{\omega}'}{\partial p_2'}(p_1', p_2') = \sqrt{\frac{1}{4\cos p_1'}} R(p_1, p_2) \begin{pmatrix} -\sin p_2' \\ \cos p_2' \\ 0 \end{pmatrix}. \tag{5.26}$$

Scaled by the inverses of these factors, the distances in parameter space become identical to those of equation 5.11, allowing the same global constant to be used for all photons,

$$\Delta p_1' = \Delta p_2' = \sqrt{\frac{4\pi}{n}}. \tag{5.27}$$

Treating diffuse reflection as an absorption and reemission from the hit point with new parameters has the disadvantage that any prior footprint adaptation is discarded. To approximatively preserve it, we derive an offset $t_v(p_1, p_2)$ from this footprint. The photon is then reemitted with differentials as given by equations 5.25 and 5.26 not at $\vec{z}(p_1, p_2)$ but at

$$\vec{z}_v'(p_1', p_2') = \vec{z}(p_1, p_2) - t_v(p_1, p_2)\, \vec{\omega}'(p_1', p_2').$$

Having traveled a distance $t_v(p_1, p_2)$ since reemission, the photon reaches the actual hit point again at $\vec{z}''(p_1', p_2') = \vec{z}_v'(p_1', p_2') + t_v(p_1, p_2)\, \vec{\omega}(p_1', p_2') = \vec{z}(p_1, p_2)$ with its differentials now

$$\frac{\partial \vec{z}''}{\partial p_1'}(p_1', p_2') = t_v(p_1, p_2) \frac{\partial \vec{\omega}'}{\partial p_1'}(p_1', p_2'), \qquad \frac{\partial \vec{\omega}''}{\partial p_1'}(p_1', p_2') = \frac{\partial \vec{\omega}'}{\partial p_1'}(p_1', p_2'), \tag{5.28}$$

$$\frac{\partial \vec{z}''}{\partial p_2'}(p_1', p_2') = t_v(p_1, p_2) \frac{\partial \vec{\omega}'}{\partial p_2'}(p_1', p_2'), \qquad \frac{\partial \vec{\omega}''}{\partial p_2'}(p_1', p_2') = \frac{\partial \vec{\omega}'}{\partial p_2'}(p_1', p_2'). \tag{5.29}$$

Offsetting thus does not affect the direction $\vec{\omega}'$ or its differentials but leads to the photon having a nonzero footprint at $\vec{z}(p_1, p_2)$ after reflection. The footprint area is

$$A'(p_1', p_2') = \left\| \Delta p_1' \frac{\partial \vec{z}''}{\partial p_1'}(p_1', p_2') \times \Delta p_2' \frac{\partial \vec{z}''}{\partial p_2'}(p_1', p_2') \right\|.$$

Inserting $\Delta p_1'$, $\Delta p_2'$ from equation 5.27, differentials from equations 5.28, 5.29 and 5.25, 5.26 yields

$$A'(p_1', p_2') = t_v^2(p_1, p_2) \frac{\pi}{n \cos p_1'}. \tag{5.30}$$

The footprint area before reflection is

$$A(p_1, p_2) = \left\| \Delta p_1 \frac{\partial \vec{z}}{\partial p_1}(p_1, p_2) \times \Delta p_2 \frac{\partial \vec{z}}{\partial p_2}(p_1, p_2) \right\| = \frac{4\pi}{n} \left\| \frac{\partial \vec{z}}{\partial p_1}(p_1, p_2) \times \frac{\partial \vec{z}}{\partial p_2}(p_1, p_2) \right\|.$$

With directions sampled proportional to $I(\vec{\omega}') \propto \cos p_1'$, the expected value of the footprint area after reflection follows from equation 5.30 as $\mathbb{E}[A'(p_1', p_2')] = t_v^2(p_1, p_2)\frac{2\pi}{n}$. Adaptation before reflection is approximatively preserved by computing the offset $t_v(p_1, p_2)$ so that $\mathbb{E}[A'(p_1', p_2')] = A(p_1, p_2)$,

$$\begin{aligned}
&\mathbb{E}\left[A'\left(p_1', p_2'\right)\right] = A\left(p_1, p_2\right) \\
\Leftrightarrow\ & t_v^2\left(p_1, p_2\right)\frac{2\pi}{n} = \frac{4\pi}{n}\left\|\frac{\partial \vec{z}}{\partial p_1}\left(p_1, p_2\right) \times \frac{\partial \vec{z}}{\partial p_2}\left(p_1, p_2\right)\right\| \\
\Rightarrow\ & t_v\left(p_1, p_2\right) = \sqrt{2\left\|\frac{\partial \vec{z}}{\partial p_1}\left(p_1, p_2\right) \times \frac{\partial \vec{z}}{\partial p_2}\left(p_1, p_2\right)\right\|}.
\end{aligned}$$

In summary, when a photon hits a diffuse surface, an offset $t_v\left(p_1, p_2\right)$ is calculated from the cross product of its current position differentials. The photon is then absorbed and reemitted with a new direction chosen by sampling parameters p_1', p_2' and differentials as given in equations 5.25 and 5.26. Footprint area, representing any prior scaling and adaptation of the differentials, is approximatively preserved by offsetting the reemission point. Since offsetting serves the purpose of modifying differentials only, it is sufficient to *virtually* offset the reemission point, continuing to trace from the actual hit point but with the differentials from equations 5.28 and 5.29. The parameter count and distances between neighboring photons for these remain unchanged. No distinction between parameters before and after diffuse reflection thus needs to be made, allowing symbols p_1, p_2 to always be used for the current parameters of a photon.

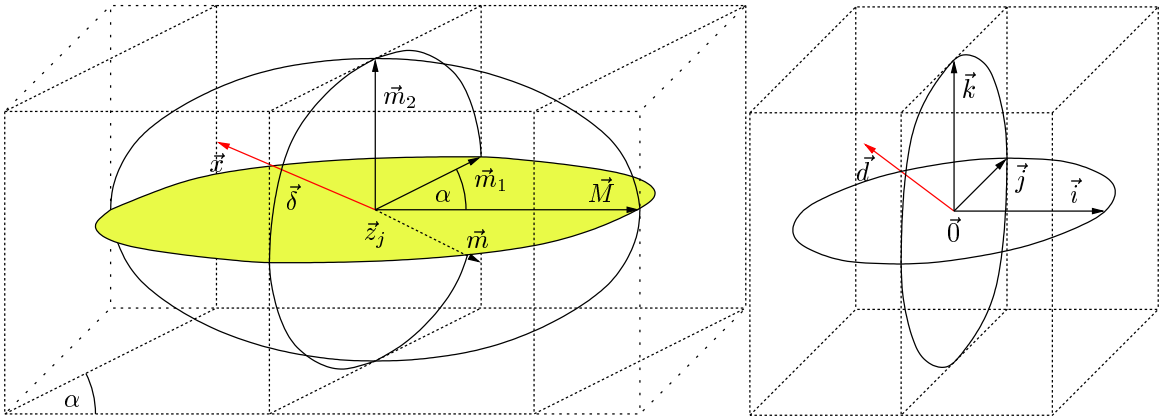
5.1.4. Russian Roulette

At each surface hit, Russian roulette samples *another* parameter to decide between reflection and absorption. As first described in the context of path differentials [SW01], the need for additional differentials can be avoided by assuming other photons encounter surfaces with equal reflectivity ρ. Russian roulette then reduces the number of photons by factor ρ while leaving their distribution in parameter space unchanged. It can thus approximatively be accounted for by increasing the expected distances in parameter space for each photon undergoing Russian roulette.

Differentials are scaled during emission from a light source and reemission from a diffuse surface so that the expected distances in parameter space become equal to the same global constant for all photons. Irrespective of the actual sample distributions, Russian roulette therefore increases both expected distances for a photon by factor $\rho^{-\frac{1}{2}}$. As noted in section 5.1.1, increasing the lengths of the differentials by $\rho^{-\frac{1}{2}}$ instead has the same effect while leaving the distances in parameter space globally constant and avoiding the need to track them with each photon.

5.2. Bandwidth Selection

Variable kernel density estimation for photon mapping follows equation 3.17. The flux $\Phi_{i,j}$ of a photon interaction at $\vec{z}_j = \vec{z}\left(p_1, p_2\right)$ is weighted for each query point $\vec{x}$ by the BRDF f, a radially symmetric smoothing kernel K with local support and the Jacobian determinant of the transformation to its local coordinate frame. Although the kernel is planar, a three-dimensional support region is used. This enables contribution to nearby points also on curved surfaces since for each $\vec{x}$, a cut through the support region containing $\vec{x}$ and $\vec{z}_j$ can serve as kernel support area. Selecting a single bandwidth h_j is simplest, leading to a spherical kernel support region around $\vec{z}_j$ with radius h_j and circular cuts through it. However, a more complex shape could also be chosen.



(a) The area of any cut through the kernel support region is approximated by that containing the photon footprint (yellow).

(b) The kernel is evaluated after transforming the region to a unit sphere.

Figure 5.1.: Kernel support region defined by photon differentials: Photon interaction position $\vec{z}_j$ and semiprincipal axes $\vec{M}$, $\vec{m}_1$, $\vec{m}_2$ yield a skewed ellipsoid.

5.2.1. Anisotropic Kernel Support Region

Photon differentials yield footprint spanning vectors adaptive to the local interaction density at $\vec{z}_j$. An anisotropic kernel support region aligned with these is therefore desirable, providing an automatic tradeoff between variance and bias for two distinct directions. We scale the two spanning vectors by a manually chosen spread factor s, obtaining the vectors

$$\vec{M} = \arg\max \left(\| s\, \vec{\Delta}'_1 (p_1, p_2) \|, \| s\, \vec{\Delta}'_2 (p_1, p_2) \| \right),$$
$$\vec{m}_1 = \arg\min \left(\| s\, \vec{\Delta}'_1 (p_1, p_2) \|, \| s\, \vec{\Delta}'_2 (p_1, p_2) \| \right).$$

These serve as the semimajor and semiminor axes of an ellipse, skewed if the angle between them is $\alpha \neq \frac{\pi}{2}$. Adding another semiprincipal axis $\vec{m}_2$ of the same length as $\vec{m}_1$ aligned with the surface normal $\vec{n}_j = \vec{n}(p_1, p_2)$, a *skewed ellipsoid* adaptive to the local interaction density at $\vec{z}_j$ is obtained (figure 5.1(a)). $\vec{m}_1$ is reoriented so that $\vec{M}$, $\vec{m}_1$, $\vec{m}_2$ form a right-handed coordinate system.

With this anisotropic kernel support region around $\vec{z}_j$, the contribution at any $\vec{x}$ is weighted by transforming $\vec{x}$ from a cut containing $\vec{x}$ and $\vec{z}_j$ to the local coordinate frame of the kernel, evaluating K and computing the Jacobian determinant. If the transformation is affine, the Jacobian determinant is the ratio of the cut area to that of the unit circle on which the kernel is defined. Following the original work on photon differentials [SFES07], we approximate the area for any orientation by that containing the photon footprint (figure 5.1(a), yellow), $A = \pi \| \vec{M} \times \vec{m}_1 \|$. This allows the Jacobian determinant to be accounted for by storing flux weighted by it, $\Phi'_{i,j} = \pi A^{-1} \Phi_{i,j}$.

The skewed ellipsoid can be affinely transformed to a unit sphere at the origin, mapping each $\vec{x}$ to a point $\vec{d}$ (figure 5.1(b)). Since K is radially symmetric, its value then depends on $\|\vec{d}\|$ only, regardless of cut orientation. With $\vec{\delta} = \vec{x} - \vec{z}_j$, the transformation is $\vec{d} = M\, \vec{\delta}$. Its inverse maps the three unit coordinate axes to the semiprincipal axes of the skewed ellipsoid so that $M^{-1} = (\vec{M}, \vec{m}_1, \vec{m}_2)$ and

$$M = \langle \vec{M}, \vec{m}_1, \vec{m}_2 \rangle^{-1} (\vec{m}_1 \times \vec{m}_2, \vec{m}_2 \times \vec{M}, \vec{M} \times \vec{m}_1)^T.$$

In the original photon differentials technique, storing M or reconstructing it from the footprint spanning vectors and surface normal during density estimation is proposed. In both cases, twelve floating point values must be loaded to obtain $\vec{z}_j$ and M. We show how $\|\vec{d}\|$ can be computed from $\vec{x}$ using nine values, reducing storage and bandwidth requirements. The definition expands to

$$\|\vec{d}\|^2 = \|M\,\vec{\delta}\|^2 = \langle \vec{M}, \vec{m}_1, \vec{m}_2 \rangle^{-2} \left(\langle \vec{m}_1, \vec{m}_2, \vec{\delta} \rangle^2 + \langle \vec{m}_2, \vec{M}, \vec{\delta} \rangle^2 + \langle \vec{M}, \vec{m}_1, \vec{\delta} \rangle^2 \right).$$

Dependencies on $\vec{m}_1$ and $\vec{m}_2$ can be removed by exploiting the facts that $\vec{m}_1^{-2} = \vec{m}_2^{-2}$ and the two are orthogonal so that $c_1^2\,\vec{m}_1^2 + c_2^2\,\vec{m}_2^2 = (c_1\,\vec{m}_1 + c_2\,\vec{m}_2)^2$. With $\vec{m} = \vec{m}_1 \times \vec{m}_2$,

$$\begin{aligned}
\|\vec{d}\|^2 =& (\vec{m}\cdot\vec{M})^{-2} \left((\vec{m}\cdot\vec{\delta})^2 + \vec{m}_1^{-2}\,\vec{m}_1^2\,\langle \vec{m}_2, \vec{M}, \vec{\delta} \rangle^2 + \vec{m}_2^{-2}\,\vec{m}_2^2\,\langle \vec{M}, \vec{m}_1, \vec{\delta} \rangle^2 \right) \\
=& (\vec{m}\cdot\vec{\delta})^2 (\vec{m}\cdot\vec{M})^{-2} + \vec{m}_1^{-2} (\vec{m}\cdot\vec{M})^{-2} \left(\vec{m}_1\,\langle \vec{m}_2, \vec{M}, \vec{\delta} \rangle + \vec{m}_2\,\langle \vec{M}, \vec{m}_1, \vec{\delta} \rangle \right)^2 \\
=& (\vec{m}\cdot\vec{\delta})^2 (\vec{m}\cdot\vec{M})^{-2} + \vec{m}_1^{-2} (\vec{m}\cdot\vec{M})^{-2} \left(\vec{m}_1\,\langle \vec{m}_2, \vec{M}, \vec{\delta} \rangle - \vec{m}_2\,\langle \vec{m}_1, \vec{M}, \vec{\delta} \rangle \right)^2 \\
=& (\vec{m}\cdot\vec{\delta})^2 (\vec{m}\cdot\vec{M})^{-2} + \vec{m}_1^{-2} (\vec{m}\cdot\vec{M})^{-2} \left((\vec{M}\times\vec{\delta}) \times (\vec{m}_1 \times \vec{m}_2) \right)^2 \\
=& (\vec{m}\cdot\vec{\delta})^2 (\vec{m}\cdot\vec{M})^{-2} + \vec{m}_1^{-2} (\vec{m}\cdot\vec{M})^{-2} \left(\vec{m} \times (\vec{M}\times\vec{\delta}) \right)^2 \\
=& (\vec{m}\cdot\vec{\delta})^2 (\vec{m}\cdot\vec{M})^{-2} + \vec{m}_1^{-2} (\vec{m}\cdot\vec{M})^{-2} \left(\vec{M}\,(\vec{m}\cdot\vec{\delta}) - \vec{\delta}\,(\vec{m}\cdot\vec{M}) \right)^2 \\
=& (\vec{m}\cdot\vec{\delta})^2 (\vec{m}\cdot\vec{M})^{-2} + \vec{m}_1^{-2} \left((\vec{m}\cdot\vec{\delta}) (\vec{m}\cdot\vec{M})^{-1}\,\vec{M} - \vec{\delta} \right)^2.
\end{aligned}$$

With $\vec{m}' = \|\vec{m}_1\|\,\hat{\vec{m}}$, the identity $\vec{m}'^2 = \vec{m}_1^2$ holds and

$$\|\vec{d}\|^2 = (\vec{m}'\cdot\vec{\delta})^2 (\vec{m}'\cdot\vec{M})^{-2} + \vec{m}'^{-2} \left((\vec{m}\cdot\vec{\delta}) (\vec{m}'\cdot\vec{M})^{-1}\,\vec{M} - \vec{\delta} \right)^2.$$

To evaluate this expression for a query point $\vec{x}$, only the nine floating point values $\vec{z}_j$, $\vec{m}'$, $\vec{M}$ are needed. If $\|\vec{d}\|^2 < 1$, the query point lies within the kernel support region and the weight $K(\|\vec{d}\|)$ is computed. Contributions to back-facing surfaces are avoided by comparing the normals at $\vec{x}$ and $\vec{z}_j$. No additional storage or bandwidth is required for $\vec{n}_j$ as $\vec{n}_j = \widehat{\vec{m}' \times \vec{M}}$.

All evaluation may be omitted when $\vec{x}$ lies outside an AABB of the kernel support region. This is defined by six bounding planes, each spanned by a pair of unit coordinate axes $\vec{u}_i$, $\vec{v}_i$ and offset from $\vec{z}_j$ by c_i along the normal $\vec{w}_i = \vec{u}_i \times \vec{v}_i$. Planes tangential to the kernel support region are the tightest. These can be found by affinely transforming the plane equations with the skewed ellipsoid. For each, position vector $\vec{p}_i = c_i\,M\,\vec{w}_i$ and spanning vectors $M\,\vec{u}_i$, $M\,\vec{v}_i$ result, yielding the normal

$$\vec{w}_i' = (M\,\vec{u}_i) \times (M\,\vec{v}_i) = \det M\; M^{-T}\;(\vec{u}_i \times \vec{v}_i) = \det M\; M^{-T}\,\vec{w}_i.$$

The plane is tangential to the unit sphere at the origin when its distance from the origin is one,

$$\left| \hat{\vec{w}}_i' \cdot \vec{p}_i \right| = \left| \frac{\det M\; M^{-T}\,\vec{w}_i}{\| \det M\; M^{-T}\,\vec{w}_i \|} \cdot (c_i\,M\,\vec{w}_i) \right| = \left| c_i\,(M^{-T}\,\vec{w}_i) \cdot (M\,\vec{w}_i) \right| \|M^{-T}\,\vec{w}_i\|^{-1} \stackrel{!}{=} 1. \tag{5.31}$$

Rewriting the dot product reduces it to

$$(M^{-T}\,\vec{w}_i) \cdot (M\,\vec{w}_i) = (\vec{w}_i^T M^{-1})^T \cdot (M\,\vec{w}_i) = (\vec{w}_i^T M^{-1})\;(M\,\vec{w}_i) = \vec{w}_i^T M^{-1} M\,\vec{w}_i = \vec{w}_i^2 = 1. \tag{5.32}$$

Inserting equation 5.32 into equation 5.31 yields

$$|c_i| \, \|M^{-T} \, \vec{w}_i\|^{-1} = 1 \Leftrightarrow c_i = \pm \|M^{-T} \, \vec{w}_i\|^{-1}.$$

The tightest bounding planes having a unit coordinate axis $\vec{w}_i$ as normal are thus offset from $\vec{z}_j$ by $\pm\|M^{-T} \, \vec{w}_i\|^{-1}$, leading to the AABB

$$\left(\vec{z}_j - \left(\|M^{-T} \, \vec{i}\|, \|M^{-T} \, \vec{j}\|, \|M^{-T} \, \vec{k}\| \right)^T, \vec{z}_j + \left(\|M^{-T} \, \vec{i}\|, \|M^{-T} \, \vec{j}\|, \|M^{-T} \, \vec{k}\| \right)^T \right).$$

5.2.2. Dampened Adaptation

Using photon differentials, a kernel support region adaptive to the local interaction density of photons neighboring in parameter space and thus following similar paths is obtained. This provides the desirable separation of interaction densities for differently focused or colored illumination arriving from different parts of the scene. However, when *similar* illumination reaches a surface via different paths, interaction densities could be combined to reduce kernel support regions.

Finding the interactions of different paths that contribute similar illumination is computationally expensive. The need to locate nearby interactions after photon tracing would be reintroduced and a classification of these as either similar or not required. We propose an inexpensive heuristic for interactive rendering instead. As the kernel support region grows, the probability increases that different paths lead to similar interactions inside it, indicating that the region should be reduced. The overall result is approximated by *dampening* the adaptation. We are motivated by a method that adapts kernel support areas to inverse photon path probabilities [HHK+07], using an exponent $e < 1$ to dampen the adaptation. With p the parameter for which the longer position differential results at $\vec{z}_j$, its length is scaled so that

$$\frac{\partial \vec{z}'}{\partial p}(p_1, p_2) = \left\| \frac{\partial \vec{z}}{\partial p}(p_1, p_2) \right\|^{\frac{1}{4}} \widehat{\frac{\partial \vec{z}}{\partial p}}(p_1, p_2).$$

The differential for the other parameter p' is scaled by the same factor, resulting in

$$\frac{\partial \vec{z}'}{\partial p'}(p_1, p_2) = \frac{\left\| \frac{\partial \vec{z}}{\partial p'}(p_1, p_2) \right\|}{\left\| \frac{\partial \vec{z}}{\partial p}(p_1, p_2) \right\|} \widehat{\frac{\partial \vec{z}}{\partial p'}}(p_1, p_2).$$

Construction of the kernel support region then follows section 5.2.1. By scaling both differentials with the same factor, the footprint shape and therefore also that of the support region are unaffected by the adaptation dampening. Only the ranges of their sizes are compressed. To guarantee that no excessively large kernel support regions result despite the damping, the lengths of all skewed ellipsoid semiprincipal axes are further limited to h_{max}.

Another potential difficulty in practice are degenerate interactions whose footprint parallelogram has a height vanishing against the base. We address this issue by discarding a photon interaction if the footprint height is less than a small threshold fraction f of its base,

$$\left\| \frac{\partial \vec{z}}{\partial p'}(p_1, p_2) \right\| |\sin \alpha| < f \left\| \frac{\partial \vec{z}}{\partial p}(p_1, p_2) \right\|. \tag{5.33}$$

Scene	Photon Tracing					Density Estimation	
	Emissions	Interactions	Discarded	Spread s	Clamped	k_{VK}	k_{kNN}
Scene 6	131072	163286	0.15%	20	0.00%	337.6	301
Sponza	262144	300339	0.15%	40	2.94%	199.1	155
Sibenik	524288	417649	0.16%	55	7.19%	192.2	201
Fairy	131072	9478	0.86%	25	2.83%	36.0	75
Conference	131072	83003	0.16%	25	0.22%	256.9	201
Passage	131072	69201	0.17%	15	0.00%	459.6	451
Ring	131072	75876	0.46%	10	0.00%	186.6	51
Average			0.30%		1.88%		
Std. Dev.			0.27%		2.70%		

Table 5.1.: Photon tracing and density estimation statistics

5.2.3. Results and Discussion

We evaluate the proposed methods on the seven benchmark scenes from section A.2. All parameters are adjusted to balance computational cost and image quality. The number of photons emitted is set to 2^{18} for the Sponza Atrium, 2^{19} for the Sibenik Cathedral due to their large sizes and 2^{17} for all other scenes. Photons are traced from an isotropic point light source until their third surface interaction. Since the first interaction represents direct illumination efficiently accounted for by shadow rays, only the second and third interactions are stored, corresponding to two-bounce indirect illumination. All sampling is controlled by pregenerated true random numbers.

Statistics are provided in table 5.1. The decision whether to store a photon interaction is made according to equation 5.33. Using threshold $f = \frac{1}{20}$, on average 0.30% of the interactions are deemed to have degenerate footprints and are discarded. For those remaining, kernel support regions are computed. We find that a spread factor s following the scene size best balances variance and bias. To eliminate excessively large support regions with detrimental impact on performance, all semiprincipal axis lengths are limited to $h_{max} = 1$. This clamping affects 1.88% of the stored interactions on average. The anisotropy of the kernel support regions is illustrated in figure 5.2, showing that the major $\vec{M}$ and minor $\vec{m}_1$, $\vec{m}_2$ semiprincipal axis lengths follow different distributions.

Figure 5.3 illustrates rendering results for an example view of each scene. All images are rendered using the benchmark environment from section A.1 with up to four rays traced per pixel. Photon mapping additionally performs density estimation at the primary and reflection ray hit points. The average number of photon interactions contributing per query point $\vec{x}$ for a flight through each scene is listed as k_{VK} in table 5.1. We find that k-th nearest neighbor density estimation achieves comparable image quality when the $k_{kNN} \approx k_{VK}$ nearest interactions are used.

In the Fairy Forest, illumination is primarily direct with most photons escaping before their second interaction. For all other scenes, the indirect illumination computed by photon mapping is a significant addition to the rendering result. Previously shadowed walls and ceilings become illuminated in the Sponza Atrium. The same is true of the ceiling in Scene 6. Color bleeding effects are seen in many scenes, from the blue floor onto the walls and ceiling in Scene 6, between the stone walls emphasizing their color in the Sibenik Cathedral and from the brown table top onto the ceiling in the Conference Room. The most pronounced color bleeding occurs for the Passage as indirect illumination reaches the walls of a previously shadowed white room after reflection on those of a red room, giving these a red tint. While anisotropic kernel support regions can better follow illumination boundaries, k-th

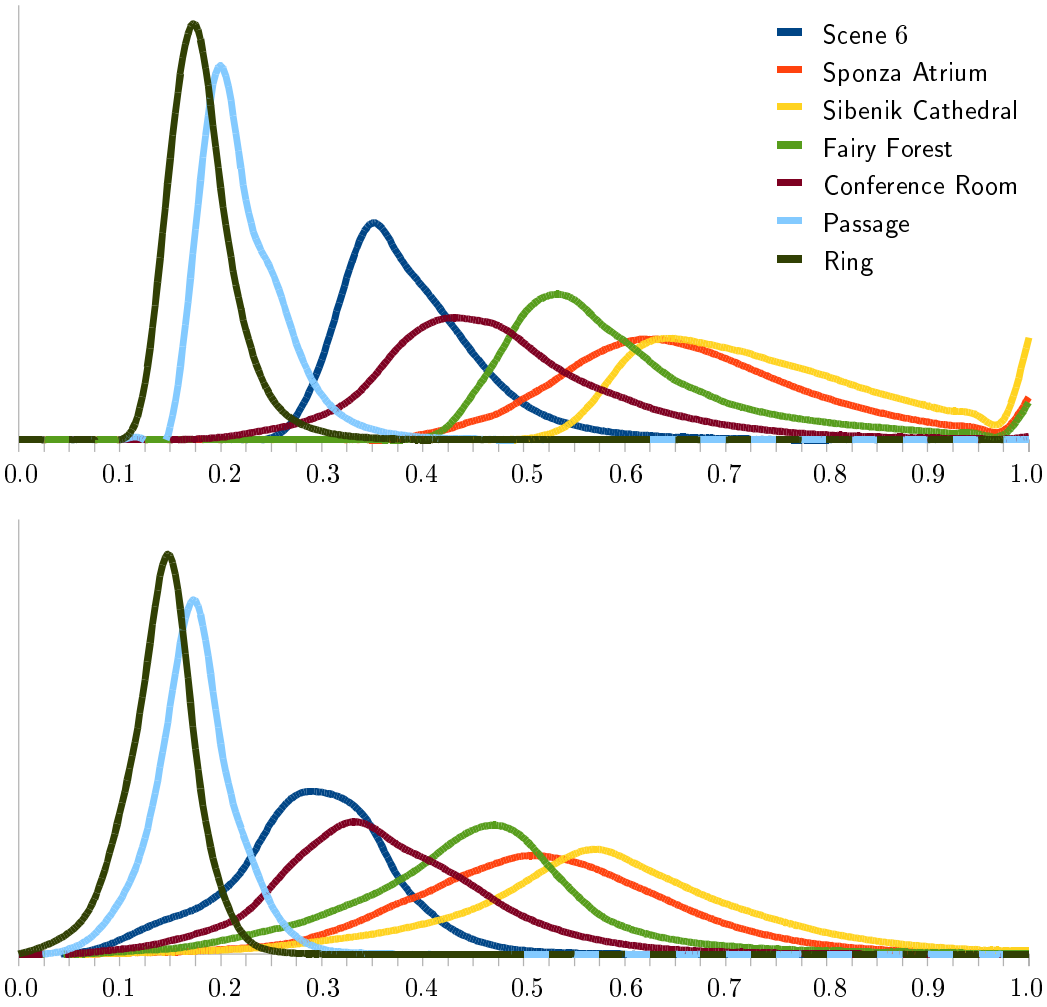
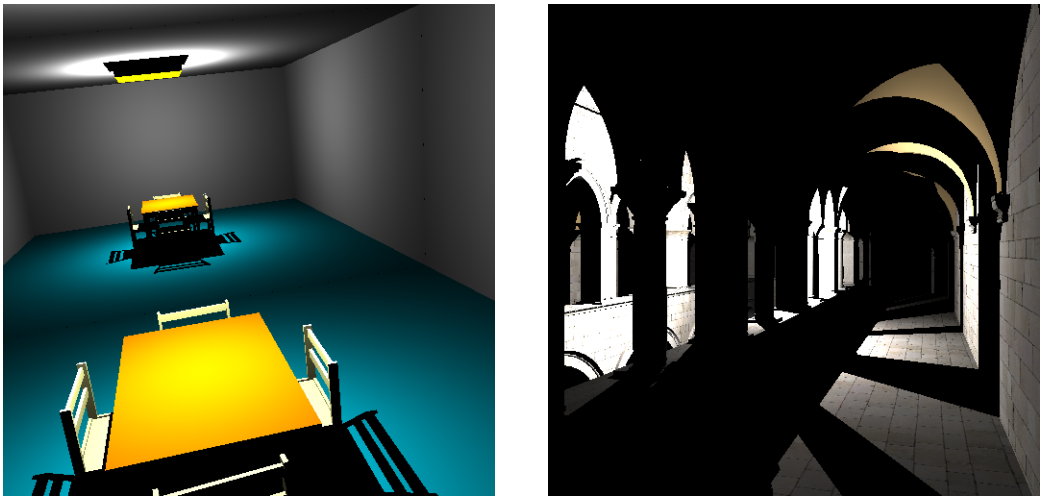


Figure 5.2.: Distribution of major ($\|\vec{M}\|$, top) and minor ($\|\vec{m}_1\| = \|\vec{m}_2\|$, bottom) skewed ellipsoid semiprincipal axis lengths.

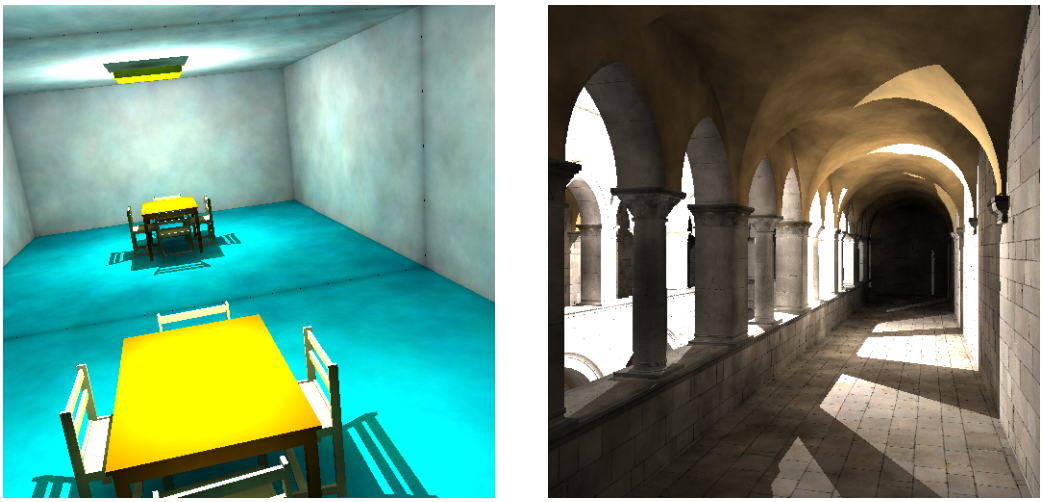
nearest neighbor density estimation reduces the risk of outliers with excessively large or small kernel bandwidths. The overall effect is comparable image quality.

Disparate image quality is observed for the Ring. Photon differentials yield separate adaptation to the local interaction density for photons following different paths, resulting in a sharp caustic and smooth diffuse illumination on the same surface. The anisotropic kernel support regions furthermore align with the illumination boundary along the edge of the caustic. As the caustic is curved, support regions become tangential to it, leading to a streaking artifact. Using the Epanechnikov smoothing kernel instead of a simpler uniform kernel reduces the visual impact and improves image quality. With k-th nearest neighbor density estimation, kernel support regions are spherical and thus unable to follow the sharp illumination boundary. Computing a single bandwidth from all nearby interactions also does not distinguish between diffuse and specular illumination. Separate global and caustic photon maps would address the latter issue but also incur additional overheads.

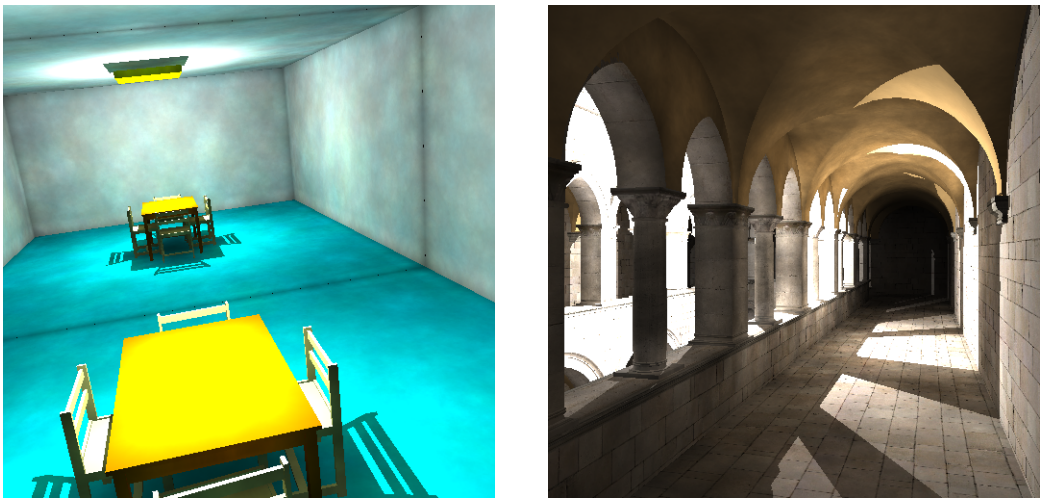
In summary, the methods for computing anisotropic kernel support regions proposed in this chapter lead to image quality matching or exceeding that of k-th nearest neighbor density estimation, successfully eliminating the need for this costly operation.



(a) Ray tracing



(b) Variable kernel density estimation



(c) k-th nearest neighbor density estimation

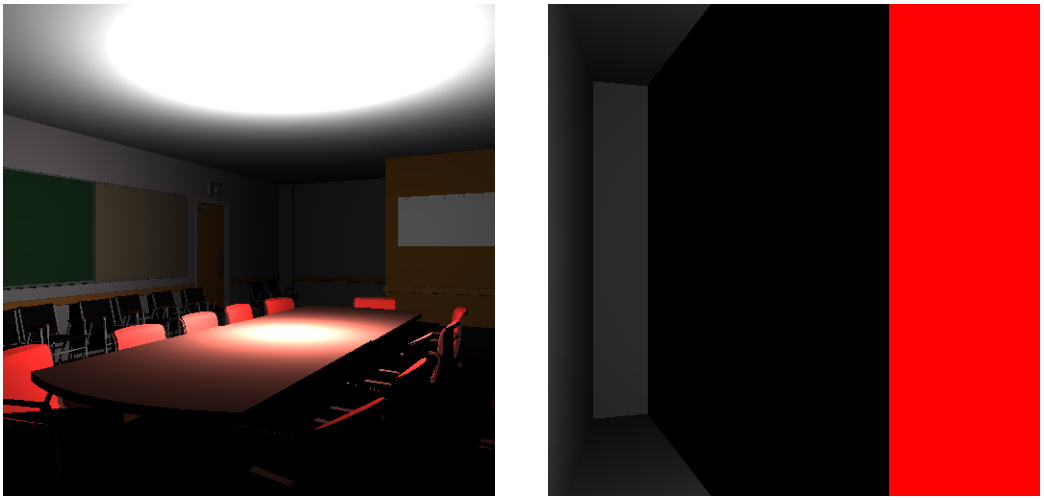
Figure 5.3.: Images rendered by ray tracing, photon mapping with variable kernel density estimation and with k-th nearest neighbor density estimation: Scene 6, Sponza Atrium

(d) Ray tracing

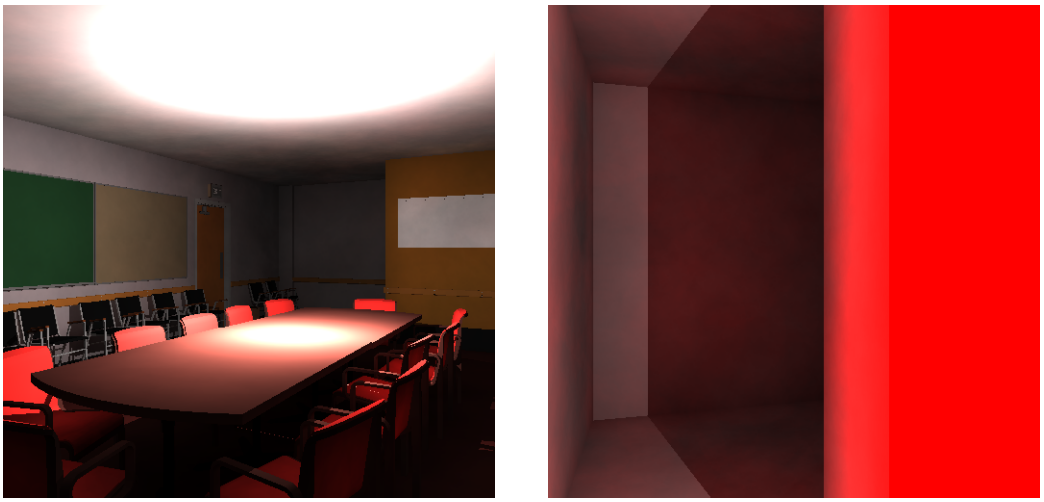
(e) Variable kernel density estimation

(f) k-th nearest neighbor density estimation

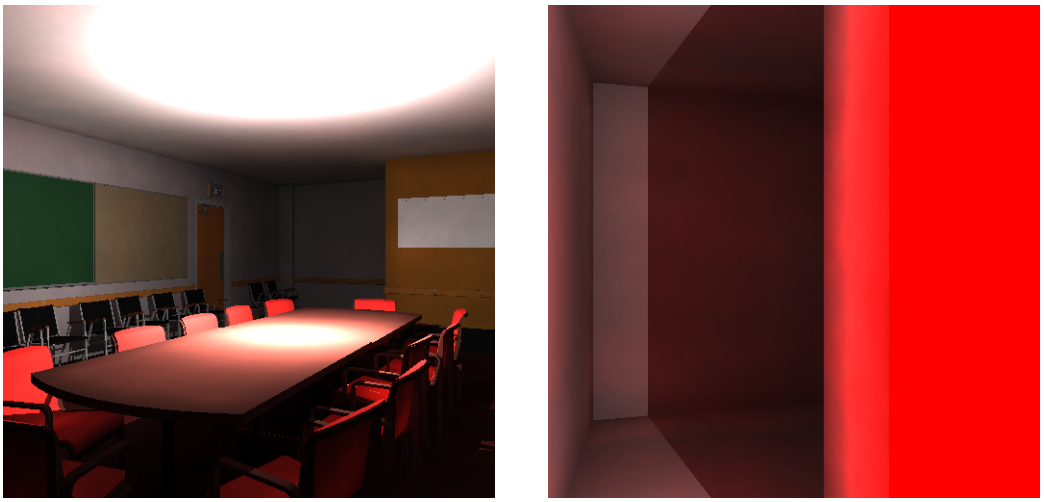
Figure 5.3.: Images rendered by ray tracing, photon mapping with variable kernel density estimation and with k-th nearest neighbor density estimation: Sibenik Cathedral and Fairy Forest



(g) Ray tracing



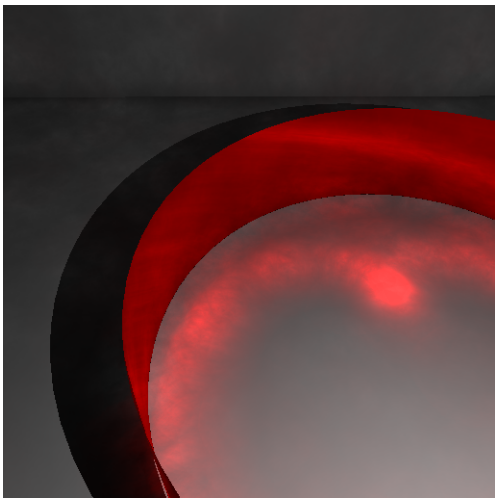
(h) Variable kernel density estimation



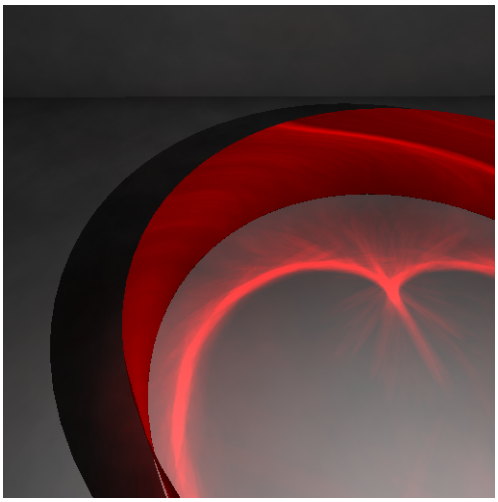
(i) k-th nearest neighbor density estimation

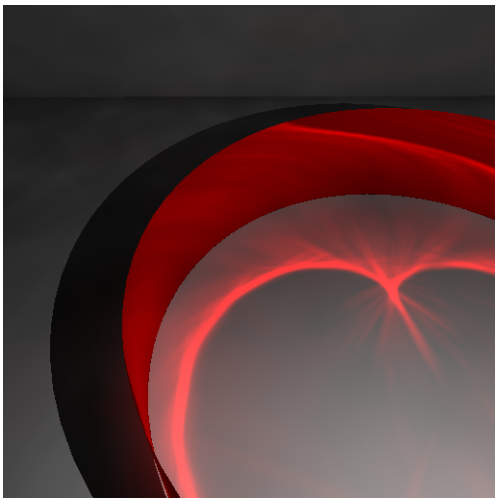
Figure 5.3.: Images rendered by ray tracing, photon mapping with variable kernel density estimation and with k-th nearest neighbor density estimation: Conference Room and Passage

(j) Ray tracing



(k) k-th nearest neighbor density estimation





(l) Variable kernel density estimation, uniform kernel (left), Epanechnikov kernel (right)

Figure 5.3.: Images rendered by ray tracing, photon mapping with variable kernel density estimation and with k-th nearest neighbor density estimation: Ring

6. Photon Map

Density estimation requires that all photon interactions contributing at a query point $\vec{x}$ be retrieved. This is accelerated by constructing a *photon map*, a spatial index over the interactions. Analogously to ray tracing, a hierarchical index reduces average case complexity from $O(n)$ to $O(\log n)$.

Use of a kd-tree is proposed in the original photon mapping algorithm. Based on space partitioning, this subdivides the scene into disjoint regions and requires that each interaction be referenced by those regions it intersects. In the case of k-th nearest neighbor density estimation, interactions represent point data with *zero* spatial extent. Each is therefore referenced exactly once. For variable kernel density estimation, interactions are assigned kernel support regions and have *nonzero* spatial extent. This leads to an unpredictable number of references per interaction. Memory management overhead is incurred as storage requirements incrementally grow while the photon map is being constructed. During density estimation, multiple references to the same interaction may be encountered and care must be taken to count its contribution only once.

We avoid these issues by replacing the kd-tree with a BVH. Based on primitive partitioning, a BVH references each interaction from a single leaf. Storage requirements are reduced and construction is simplified. Multiple references to the same interaction are never encountered during density estimation, allowing contributions to be accumulated without further precautions.

Our first contribution is an adaptation of the voxel volume heuristic from a kd-tree to a BVH photon map. Heuristic construction provides high retrieval acceleration but it is computationally expensive. Since dynamically changing illumination requires a new photon map in each frame, we investigate the use of faster LBVH construction instead. Our contributions are the removal of several inefficiencies in this algorithm and the addition of a new termination criterion. We benchmark density estimation performance with photon maps constructed using the VVH and LBVH approaches, showing that the acceleration provided by the former does not significantly exceed that by the latter. We then address photon map storage, introducing a novel compact BVH representation that reduces storage requirements and the bandwidth used during density estimation.

Having addressed all constituent components, we demonstrate that the resulting photon mapping algorithm achieves interactive frame rates on the CUDA manycore platform for all benchmark scenes with dynamic illumination.

6.1. BVH Construction

The performance of variable kernel density estimation depends on the number n_T of nodes traversed in the photon map and the number n_I of interactions retrieved from it. Adapting the voxel volume heuristic allows a BVH photon map to be constructed that minimizes n_T and n_I. LBVH construction uses spatial median splits, leading to higher n_T and n_I. Comparing density estimation performance with both methods benchmarks LBVH construction against a high standard and allows its suitability for photon mapping to be determined.

6.1.1. Voxel Volume Heuristic

Heuristic photon map construction begins with all n interactions at the root, repeatedly subdividing a parent node P into the pair of children L, R that minimizes expected density estimation cost. The method originally proposed constructs a kd-tree referencing a single interaction from each node, removing the need to consider the costs of traversal and retrieval separately.

For each candidate subdivision, cost is greedily approximated by assuming that if a child is visited, so is the entire hierarchy rooted in it, regardless of actual further subdivision. With p_P, p_L, p_R the probabilities that P, L, R are visited and n_P, n_L, n_R the numbers of interactions they represent, this yields the cost metric of equation 3.23,

$$C_P\left(L,R\right) = p_P + p_L\, n_L + p_R\, n_R. \tag{6.1}$$

When k-th nearest neighbor density estimation is used, the probability p_N that a node N is visited can be approximated by the voxel volume heuristic from section 3.4.3.1. Given query point $\vec{x}$, candidate interactions are initially retrieved from within a distance h_{max} of it. After the first k are found, this distance is progressively reduced. The probability of visiting a node N is thus bounded above by and can be approximated as the probability that $\vec{x}$ falls either inside N or within a distance h_{max} of its bounds. This region is further approximated as the node AABB with each bounding plane shifted outward by h_{max} and a volume of $\mathrm{V}_{\pm h_{max}}\left(N\right)$.

All query points are known to lie on surfaces within the scene AABB S. Under the simplifying assumption that surfaces are uniformly distributed throughout scene space, so are the query points. The probability p_N can thus be approximated using equation 3.24 as

$$p_N \approx \frac{\mathrm{V}_{\pm h_{max}}\left(N\right)}{\mathrm{V}_{\pm h_{max}}\left(S\right)}.$$

With a BVH photon map, the cost metric of equation 6.1 changes. A variable number of interactions is now referenced from each leaf, leading to separate costs C_T of traversing a node and C_I for retrieving an interaction. Our BVH representation stores child nodes in pairs. When visiting their parent, the traversal cost $2\,C_T$ for both children is therefore incurred, leading to the cost metric

$$C_P\left(L,R\right) = p_P\, 2\, C_T + p_L\, n_L\, C_I + p_R\, n_R\, C_I. \tag{6.2}$$

The computation of probability p_N is affected by the change from k-th nearest neighbor to variable kernel density estimation. All interactions whose kernel support region contains query point $\vec{x}$ must now be retrieved. Thus, precisely the nodes containing $\vec{x}$ are visited. Retaining the assumption of query points uniformly distributed throughout the scene, p_N is then given by the ratio of volumes

$$p_N = \frac{\mathrm{V}\left(N\right)}{\mathrm{V}\left(S\right)}. \tag{6.3}$$

Equations 6.2 and 6.3 yield an adaptation of the voxel volume heuristic to a BVH photon map. The construction process itself is identical to that of a BVH over scene surfaces, achieving $O\left(n \log n\right)$ complexity by sorting and then progressively splitting lists of candidate splitting planes as described in section 3.2.3.1. The same termination criteria as in section 4.1 may be used, stopping subdivision when n_P falls below a threshold, recursion depth exceeds a threshold or $\min C_P\left(L,R\right) > p_P\, n_P\, C_I$.

6.1.2. Linear BVH

The LBVH construction described in section 3.2.3.3 is a faster alternative to heuristic construction. It is based on the observation that sorting primitives by their linear positions along a Morton curve traversing the scene AABB S arranges them in the DFS traversal order of a binary spatial median split hierarchy. Inner nodes are then constructed in parallel for each level of the hierarchy wherever the traversal paths from the root to two neighboring primitives differ. After removing singleton nodes referencing a single child, a BVH is obtained. While the technique is proposed for BVH construction over scene surfaces, it can identically be applied to photon interactions.

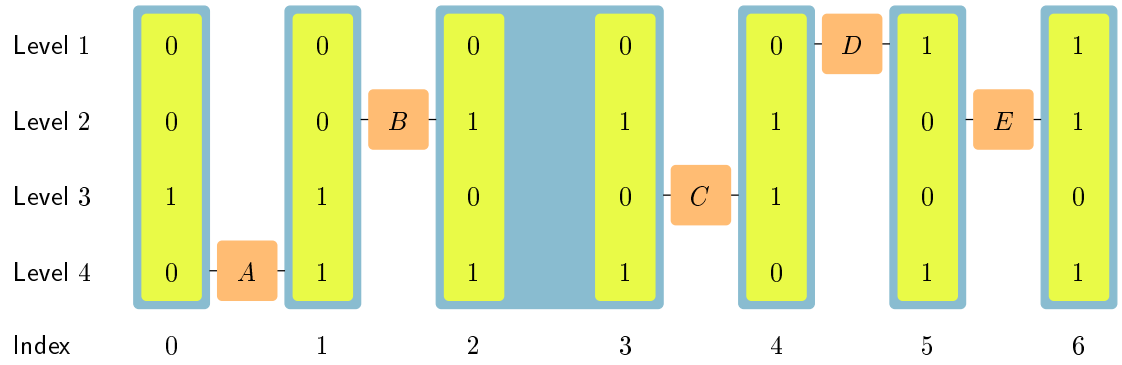
When traversing a BVH, paths from the root to the leaves *split* at inner nodes. Considered in the opposite direction, paths from the leaves to the root *merge* at the same points. We exploit this duality to improve efficiency, avoiding the construction of singleton nodes or the need for additional steps to compute node AABBs as required in the original algorithm.

The linear position of a primitive along the Morton curve is obtained by permuting the three k-bit coordinates of its centroid within the scene AABB. This yields a $3k$-bit *Morton code* corresponding to the path from the root to the primitive, each bit indicating whether the left or right child is traversed. The root represents the interval $[0, n-1]$ of all primitives. With each hierarchy level, this interval is recursively subdivided as the paths for groups of primitives deviate. A dual process thus begins with subintervals representing groups of primitives having identical Morton codes and progressively merges these into larger intervals while constructing the corresponding BVH nodes.

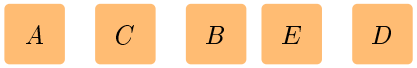
As in the original algorithm, references are generated for all primitives in parallel, each containing a pointer to a primitive and its Morton code. These are arranged in Morton curve order by parallel radix sort. Intervals of neighboring references with identical Morton codes are merged first. This can efficiently be implemented as a parallel segmented scan. For all other neighboring references, the most significant bit position at which their Morton codes differ indicates the hierarchy level at which the intervals containing these should be merged. *Merge points* are generated in parallel, each containing a reference index and the level at which it should be merged with its right neighbor (figure 6.1(a)). These are then arranged by hierarchy level using another parallel radix sort (figure 6.1(b)).

Merge points on the same hierarchy level can all be processed in parallel. For each, the bounds of its two subintervals must be found to determine those of the merged interval. This information can efficiently be tracked during the construction process. An array holding interval sizes is initialized to $S(i) = 1$ for all n references first. When merging an interval $[j, k]$ of references with identical Morton codes, the interval sizes at its first and last index are updated to $S(j) = S(k) = k - j + 1$. To process a merge point with reference index i, the intervals ending at i and starting at $i+1$ are merged. Their sizes are read from $S(i)$ and $S(i+1)$, yielding the subinterval bounds $[i - S(i) + 1, i]$ and $[i+1, i+S(i+1)]$. The size of the merged interval is then stored at its first and last indexes, $i - S(i) + 1$ and $i + S(i+1)$ (figure 6.1(c)). With this arrangement, interval sizes are read and written at two array positions per merge point, allowing it to be processed in $O(1)$.

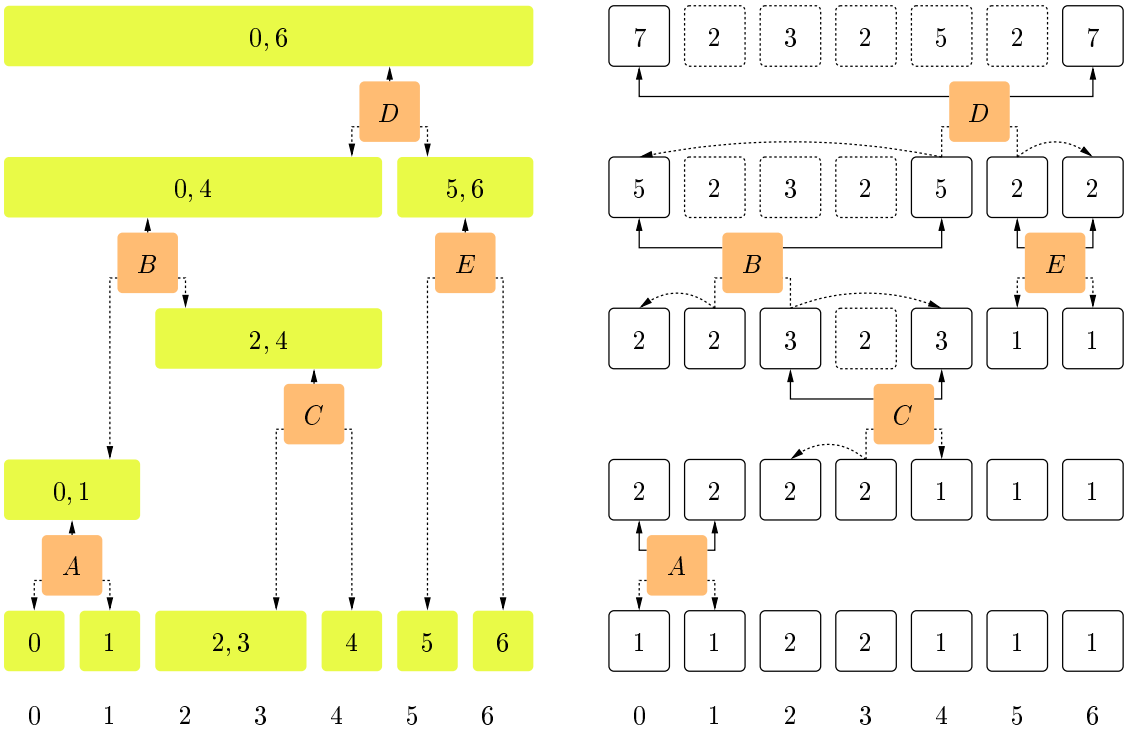
Information about the AABBs and nodes corresponding to the intervals can be tracked in the same way, enabling node construction in $O(1)$. The information is held in two additional arrays, initialized for the i-th reference by setting $N(i) = 0$ and $AABB(i)$ to the AABB of its primitive. When merging an interval $[j, k]$ of references with identical Morton codes, $AABB(j)$ is updated to the union of their AABBs. During the subsequent processing of a merge point with reference index i, the AABBs of both subintervals are read from $AABB(i - S(i) + 1)$ and $AABB(i+1)$. Their union is then stored at the first index of the merged interval, $i - S(i) + 1$.



(a) Seven primitive references arranged in the order of their Morton codes 0010, 0011, 0101, 0101, 0110, 1001, 1101 (yellow), the six intervals with identical Morton codes they form (blue) and the merge points generated between these (orange).



(b) Merge points sorted by hierarchy level.



(c) Each merge point concatenates subintervals of $[0, 6]$ into nodes. As intervals are concatenated on the left, information about their sizes is updated on the right.

Figure 6.1.: Efficient linear BVH construction: Primitives are sorted in Morton curve order, merge points generated between the subintervals having identical Morton codes and processed to construct a hierarchy of BVH nodes.

Pointers to the BVH nodes corresponding to these subintervals are read from $N\left(i - S\left(i\right) + 1\right)$ and $N\left(i + 1\right)$. An inner node with these as children is constructed and a pointer to it stored at index $i - S\left(i\right) + 1$. Since the array is initialized to zero, a null pointer may be read for either subinterval. This indicates that no corresponding node yet exists and causes a leaf to be constructed before the inner node for the merged interval. If both subintervals yield null pointers, two leaves are constructed. Alternatively, the two subintervals may be merged while leaving $N\left(i - S\left(i\right) + 1\right) = 0$. In this way, a leaf is subsequently constructed for a larger interval higher in the hierarchy. During recursive subdivision, the choice of when to construct a leaf is made by a *termination criterion*. Our dual construction process allows the same choice, thus enabling the use of termination criteria such as those listed in the previous section.

6.1.3. Termination Criterion

In addition to existing termination criteria, a minimal leaf size threshold n_{auto} specific to BVH photon map construction can be derived. Recursive subdivision is then terminated and a leaf constructed when $n_P \leq n_{auto}$ at a node. For the dual construction by merging intervals, an inner node is first constructed when $S\left(i\right) + S\left(i + 1\right) > n_{auto}$ at a merge point.

The simplifying assumption is made that kernel support areas and query points $\vec{x}$ are uniformly distributed over the scene surfaces. The expected number of interactions contributing at any $\vec{x}$ is then given by the ratio of the total kernel support areas to the total surface area. Following section 5.2.1, the area of any cut through an anisotropic kernel support region is approximated as $\pi \left\| \vec{M} \times \vec{m}_1 \right\|$. This value is computed while constructing each support region and can efficiently be accumulated for all n_i photon interactions by a parallel prescan. The area of the scene surfaces is precalculated during spatial index construction by summing the areas $A\left(T_i\right)$ of all n_t triangles. With c a proportionality constant, a leaf size threshold is then set to

$$n_{auto} = c \frac{\sum_{i=0}^{n_i - 1} \pi \left\| \vec{M}_i \times \vec{m}_{i,1} \right\|}{\sum_{i=0}^{n_t - 1} A\left(T_i\right)}.$$

Using this threshold, large leaves are constructed when many interactions are expected to contribute per query point, allowing these to be retrieved with few inner node traversals. If few contributions are expected, leaves become small, reducing the retrieval of extraneous interactions.

The proportionality constant is empirically determined as $c = 8$. To avoid excessively small leaves, n_{auto} is furthermore clamped so that $n_{auto} \geq 5$. During VVH construction, the criterion $\min C_P\left(L, R\right) > p_P\, n_P\, C_I$ and a maximal recursion depth of 64 are additionally used.

6.1.4. Results and Discussion

We use the seven benchmark scenes from section A.2 for evaluation. Photon interactions and kernel support regions follow section 5.2.3. To cover a wider range of image quality settings, two further benchmark scenarios are added, the first Scene 6 with 2^{18} photons emitted and spread $s = 28$, the second the Conference Room with 2^{19} photons emitted and spread $s = 50$.

VVH construction is implemented as a serial process with empirically determined $C_T = 0$, $C_I = 1$. LBVH construction uses a series of CUDA kernels and 30-bit Morton codes. All rendering occurs in a single CUDA kernel following benchmark code base I from section A.1.1. Up to four rays are traced per pixel as specified in section A.1 with variable kernel density estimation at primary and reflection

	FPS										n_{auto}
Leaf Size	1	2	4	8	16	32	64	128	n_{auto}		
Scene 6	2.60	3.27	3.49	3.71	3.74	3.68	3.55	3.28	3.74	−0.1%	21
	1.14	1.34	1.49	1.58	1.65	1.68	1.66	1.61	1.67	−0.4%	40
Sponza	5.70	6.54	6.98	7.05	6.75	6.26	5.68	4.95	7.13	+1.1%	5
Sibenik	4.99	5.70	6.06	6.08	5.88	5.52	5.00	4.42	6.12	+0.7%	5
Fairy	31.54	32.50	33.33	33.85	33.23	31.62	28.71	25.17	33.58	−0.8%	5
Conference	5.06	6.02	6.11	6.11	5.88	5.47	5.01	4.50	6.11	±0.0%	8
	0.79	0.93	1.04	1.10	1.13	1.11	1.06	1.00	1.11	−1.8%	31
Passage	4.78	5.17	5.39	5.58	5.69	5.72	5.54	5.28	5.67	−0.8%	40
Ring	8.88	9.01	9.25	9.41	9.28	8.75	8.19	7.49	9.40	−0.1%	7
Average										−0.3%	
Std. Dev.										0.9%	

(a) VVH construction

	FPS										n_{auto}
Leaf Size	1	2	4	8	16	32	64	128	n_{auto}		
Scene 6	2.56	2.91	3.23	3.50	3.56	3.55	3.35	3.14	3.59	+0.9%	21
	1.05	1.19	1.34	1.45	1.53	1.58	1.59	1.57	1.58	−0.8%	40
Sponza	5.46	6.08	6.38	6.47	6.21	5.78	5.26	4.59	6.53	+0.9%	5
Sibenik	4.23	4.61	5.25	5.37	5.28	5.01	4.74	4.21	5.34	−0.5%	5
Fairy	29.88	30.45	31.06	31.30	30.61	28.36	23.58	19.72	31.11	−0.6%	5
Conference	4.93	5.43	5.63	5.64	5.40	4.95	4.65	4.20	5.64	±0.0%	8
	0.73	0.81	0.90	0.96	1.00	1.00	0.97	0.92	1.00	−0.5%	31
Passage	4.46	4.73	4.95	5.19	5.28	5.21	5.01	4.52	5.22	−1.0%	40
Ring	7.79	7.87	8.09	8.18	8.37	7.90	7.61	6.73	8.20	−2.0%	7
Average										−0.4%	
Std. Dev.										0.9%	

(b) LBVH construction

Table 6.1.: Rendering frame rates with different minimal photon map leaf size thresholds

ray hit points. Results are averaged over a flight through each scene. Since it is unrelated to density estimation, the overhead of visualizing results by copying them to the screen is omitted.

The termination criterion is evaluated first. Table 6.1 lists the average frame rates obtained with different minimal photon map leaf size thresholds. The optimal threshold can be seen to depend on both scene and photon interactions. Using the proposed termination criterion, a threshold n_{auto} close to the optimum is computed in all cases. Frame rates with n_{auto} are only 0.3% (VVH) respectively 0.4% (LBVH) lower on average than those achieved by selecting between eight different thresholds through expensive benchmarking. For dynamically changing illumination, n_{auto} has the further advantage of adjusting to the photon interactions in each individual frame.

Statistics for VVH and LBVH construction with n_{auto} are given in table 6.2. The photon maps obtained using the two approaches are of very similar size, differing in the number of nodes by an average of only 1.8%. VVH construction is on the order of seconds and therefore not suitable for an interactive setting. LBVH construction efficiently utilizes the CUDA manycore platform, taking from 2.67 ms for the 8911 photon interactions in the Fairy Forest to 26.02 ms for the 413316 interactions in the Sibenik Cathedral. Density estimation performance is compared in table 6.3. In

Scene	Interactions	Nodes		Construction (ms)
		VVH	LBVH	
Scene 6	160298	23313	−5.8%	10.68
	319997	24839	−7.0%	19.27
Sponza	292551	166635	+2.8%	18.13
Sibenik	413316	236247	+3.3%	26.02
Fairy	8911	5345	+5.2%	2.67
Conference	97911	36557	−1.4%	7.31
	390223	39411	−4.3%	24.53
Passage	70256	5477	−7.7%	6.06
Ring	74101	31081	−1.1%	6.28
Average			−1.8%	
Std. Dev.			4.7%	

Table 6.2.: Photon map node counts, relative to the VVH, and construction timings with LBVH

Scene	Interactions	n_T		n_I		FPS	
		VVH	LBVH	VVH	LBVH	VVH	LBVH
Scene 6	160298	288.4	+4.3%	1163.8	+8.9%	3.74	−3.9%
	319997	334.7	+2.7%	2636.1	+9.1%	1.67	−5.8%
Sponza	292551	401.4	+14.1%	386.4	+8.7%	7.13	−8.4%
Sibenik	413316	461.9	+13.3%	466.5	+8.6%	6.12	−12.8%
Fairy	8911	114.2	+29.5%	89.9	+6.1%	33.58	−7.4%
Conference	97911	392.6	+13.5%	596.3	+10.3%	6.11	−7.7%
	390223	587.1	+9.7%	3494.4	+12.3%	1.11	−10.2%
Passage	70256	211.3	+11.8%	1683.6	+7.2%	5.67	−7.9%
Ring	74101	369.9	+21.0%	481.9	+13.1%	9.40	−12.7%
Average			+13.3%		+9.4%		−8.5%
Std. Dev.			8.1%		2.2%		3.0%

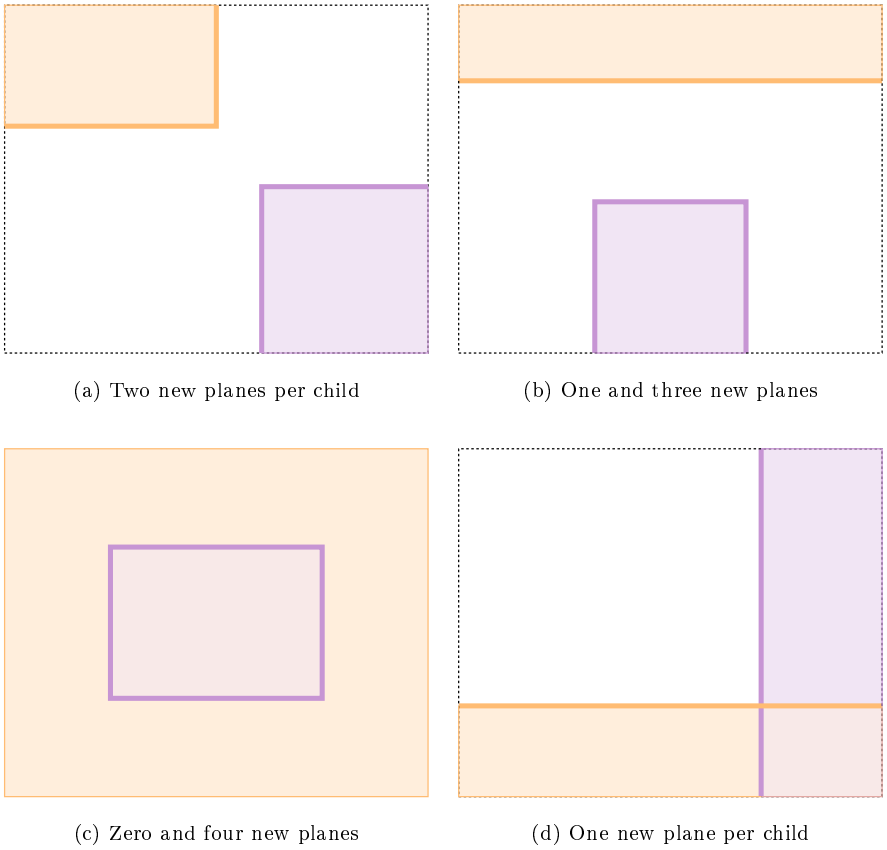
Table 6.3.: Photon map node traversals, interaction retrievals per query point and rendering frame rates with LBVH construction, relative to the VVH

line with predictions, LBVH photon map construction leads to an average of 13.3% more inner nodes being traversed and 9.4% more photon interactions being retrieved per query point. The impact on rendering frame rates is a reduction by 8.5% on average.

In summary, LBVH construction leads to a moderate slowdown during rendering because of higher n_T, n_I. In return, photon map construction is accelerated from several seconds to less than 30 ms. This provides a tradeoff between construction cost and retrieval acceleration amenable to photon mapping with dynamically changing illumination.

6.2. BVH Storage

A BVH is defined by its node bounds, child references at the inner nodes and primitive references at the leaves. Following section 3.2.2, the need to store child references can be eliminated by rearranging nodes to implicitly encode the hierarchy. Storage requirements for the bounds may be reduced by quantizing them at all nodes or omitting those at the leaves. However, by discarding information,



(a) Two new planes per child

(b) One and three new planes

(c) Zero and four new planes

(d) One new plane per child

Figure 6.2.: BVH bounding plane inheritance: In two dimensions, a pair of sibling nodes (orange, purple) has a total eight bounding planes. Only up to four of these are new (bold), the remainder being inherited from the parent (dashed).

these techniques lower the bounding tightness. We show how to eliminate *redundant* information only, reducing storage requirements while leaving bounding tightness unaffected.

6.2.1. Compact Representation

Each BVH node has six bounding planes, corresponding to its minimum and maximum extents along the three coordinate axes. Whenever a plane is coincident with that of its parent, explicit storage is redundant and a single bit marking the plane as *inherited* sufficient instead. Per coordinate axis, a node may inherit the minimum bounding plane, maximum bounding plane, both or neither. The number of new bounding planes that are *not* inherited and must still explicitly be stored thus varies unpredictably from zero to six for each node.

Given a *pair of sibling nodes*, however, whenever one node introduces a new bounding plane, the other must inherit the corresponding plane from the parent. If this were not the case, either would the second node extend outside the parent or both would be entirely interior to it, the parent thus not providing the tightest possible AABB around its children. Neither is allowed in a BVH. Two-dimensional examples are provided in figure 6.2. Each pair of sibling nodes has eight bounding planes of which only up to four are new. In the extension to three dimensions, a sibling node pair has twelve bounding planes with at most six of these new.

A complete BVH representation requires six floating point values per node that express its bounds as signed distances from the coordinate planes. Unless the nodes form a complete binary tree implicitly encoding the hierarchy, at least one reference to a pair of children is also needed. With 32-bit floating point values and references, 28 bytes of storage are used per node.

Omitting inherited bounding planes allows for a more compact BVH representation. Sibling nodes are stored in pairs. Per pair, six 32-bit floating point values ($\vec{m}$, $\vec{M}$) encode the new minimum and maximum bounding planes for the three coordinate axes. Two 3-bit masks ($\vec{l}$, $\vec{L}$) assign these to either the left or right node. A pair of 28-bit indexes holds the child references for the two nodes and a final 2 bits distinguish inner nodes from leaves. In case of a leaf, the 28-bit index references its first primitive. No primitive count is stored, using a flag bit on the last primitive to mark the end of a leaf instead. If both nodes inherit a bounding plane, the number of new planes is further reduced. To maintain constant storage size per pair and avoid the need for special case handling, the parent plane is replicated and assigned to the left node in this case.

In total, 32 bytes are used per pair of sibling nodes, reducing BVH storage requirements by 42.9%. As only redundant information is eliminated, bounding tightness is maintained. The construction process is unaffected and no additional cost is incurred in its course.

6.2.2. Ray Tracing Traversal

Although introduced here in the context of a photon map, the compact BVH representation is valid for other spatial indexes as well. For reference, its application to a BVH over scene surfaces used in ray tracing acceleration is described first. Node traversal with a complete BVH representation follows the slabs test from algorithm 3.4. For each coordinate axis, the parameter interval $[t_{a,k}, t_{b,k}]$ enclosed on the ray by two bounding planes is computed. The intersection of these three intervals indicates the part of the ray intersecting the node. When using the compact BVH representation, inherited planes are not readily available. The slabs test must thus be modified, accounting for this and also simultaneously computing intervals $[t_{a,l}, t_{b,l}]$, $[t_{a,r}, t_{b,r}]$ for two sibling nodes.

The compact BVH representation stores only a subset of the node bounding planes. As described in section 3.2.4.2, this requires that when pushing onto the stack, the current parameter interval $[t_a, t_b]$ also be placed on it. Traversal of a sibling node pair then follows algorithm 6.1. Parameter intervals $[t_{a,l}, t_{b,l}]$, $[t_{a,r}, t_{b,r}]$ for both nodes are initialized to the parent interval $[t_a, t_b]$. If this was popped off the stack, the intervals are further clamped against the current ray end distance t_{max}.

For each of the three coordinate axes, the interval bounds t_1, t_2 are computed from the two new bounding planes as in the original slabs test, then distributed to the two nodes by lines 7–8. When $l_k = 1$, the minimum bounding plane belongs to the left node and $t_{1,l}$ is set to t_1. The corresponding bound $t_{1,r}$ for the right node is set to $-t_3$. If the plane belongs to the right node, t_1 and $-t_3$ are exchanged. The interval bound t_2 is analogously distributed to either $t_{2,l}$ or $t_{2,r}$, using t_3 for the other node. $[t_{a,l}, t_{b,l}]$, $[t_{a,r}, t_{b,r}]$ are then updated by intersection with the resulting intervals. Calculated in line 6, t_3 has the effect of producing an unbounded interval where an inherited bounding plane would have been. Inherited planes are thus effectively ignored. This is permissible as the parent interval $[t_a, t_b]$ already is bounded by any inherited planes.

The resulting parameter intervals are assessed from line 14 onward, visiting the child pair of a node if its parameter interval is not empty. Should the children of both be visited, the nodes are classified as near and far according to the order of their entry points $t_{a,l}$, $t_{a,r}$ along the ray first. Traversal

Algorithm 6.1 Slabs test for sibling nodes encoded as $\vec{m}$, $\vec{M}$, $\vec{l}$, $\vec{L}$ and ray $\vec{z}(t) = \vec{x} + t\vec{\omega}$ with parent interval $t \in [t_a, t_b]$

1: $t_{a,l}, t_{a,r} \leftarrow t_a$
2: $t_{b,l}, t_{b,r} \leftarrow \min(t_b, t_{max})$
3: **for** $k \in \{x, y, z\}$ **do**
4: $\quad t_1 \leftarrow (m_k - x_k)\omega_k^{-1}$
5: $\quad t_2 \leftarrow (M_k - x_k)\omega_k^{-1}$
6: $\quad t_3 \leftarrow \infty\omega_k^{-1}$
7: $\quad (t_{1,l}, t_{1,r}) \leftarrow$ **if** $l_k = 1$ **then** $(t_1, -t_3)$ **else** $(-t_3, t_1)$
8: $\quad (t_{2,l}, t_{2,r}) \leftarrow$ **if** $L_k = 1$ **then** (t_2, t_3) **else** (t_3, t_2)
9: $\quad t_{a,l} \leftarrow \max(t_{a,l}, \min(t_{1,l}, t_{2,l}))$
10: $\quad t_{b,l} \leftarrow \min(t_{b,l}, \max(t_{1,l}, t_{2,l}))$
11: $\quad t_{a,r} \leftarrow \max(t_{a,r}, \min(t_{1,r}, t_{2,r}))$
12: $\quad t_{b,r} \leftarrow \min(t_{b,r}, \max(t_{1,r}, t_{2,r}))$
13: **end for**
14: **if** $t_{a,l} \leq t_{b,l}$ **and** $t_{a,r} \leq t_{b,r}$ **then**
15: $\quad$ visit children of both nodes
16: **else if** $t_{a,l} \leq t_{b,l}$ **then**
17: $\quad$ visit children of left node
18: **else if** $t_{a,r} \leq t_{b,r}$ **then**
19: $\quad$ visit children of right node
20: **end if**

then continues with the child pair of the near node while that of the far node is pushed onto the stack along with its parameter interval.

After popping a node pair and a parameter interval $[t_a, t_b]$ off the stack, a check whether $t_a > t_{max}$ is performed. If so, a hit nearer than the entry point has already been found, allowing the traversal of the node pair to be omitted. The conditional statements in lines 7–8 can be expressed using the ternary operator and executed without branching on hardware supporting predicated instructions.

6.2.3. Photon Mapping Traversal

For a photon map, node traversal with a complete BVH representation follows algorithm 3.5. The query point $\vec{x}$ is tested against the six bounding planes of the node, determining whether it is located inside or outside the half-space bounded by each. If $\vec{x}$ is on the outside of any bounding plane, the node is missed. Otherwise, both of its children are visited.

When using the compact BVH representation, a pair of sibling nodes is traversed by algorithm 6.2. The query point is known to lie inside all inherited bounding planes. If it were not to, the current node pair would not have been reached. Only new planes must thus be considered. A bitmask $miss$ is constructed, indicating for which of the six new bounding planes $\vec{x}$ is located on the outside. Cast to an integer, the result of testing $\vec{x}$ against a plane is 0 or 1. The bitmask can thus be obtained by shifting the six results into place, requiring no branching. By concatenating $\vec{l}$ and $\vec{L}$, a corresponding bitmask $left$ is then assembled that expresses which node each bounding plane belongs to.

If $miss$ contains only zeroes, neither of the nodes is missed and the child pairs of both must be visited. Otherwise, the query point lies outside at least one node. The bitwise disjunction of $left$ and $miss$ indicates the bounding planes of the left node that $\vec{x}$ lies outside. If none of the bits are set, the node is not missed and its child pair is visited. The analogous test for the right node uses the bitwise complement of $left$.

Algorithm 6.2 Photon map test for sibling nodes encoded as $\vec{m}$, $\vec{M}$, $\vec{l}$, $\vec{L}$ and point $\vec{x}$ ($\ll$, $\vee$, $\wedge$, $\neg$ are bitwise shift, conjunction, disjunction and negation)

$miss \leftarrow ((m_x \geq x_x) \ll 5) \vee ((m_y \geq x_y) \ll 4) \vee ((m_z \geq x_z) \ll 3) \vee$
$((M_x \leq x_x) \ll 2) \vee ((M_y \leq x_y) \ll 1) \vee (M_z \leq x_z)$
$left \leftarrow (\vec{l} \ll 3) \vee \vec{L}$
if not $miss$ **then**
    visit children of both nodes
else if not $left \wedge miss$ **then**
    visit children of left node
else if not $\neg left \wedge miss$ **then**
    visit children of right node
end if

When both child pairs are to be visited, traversal continues with one while the other is pushed onto the stack. Since no early out exists, the simple choice of always pushing the left child pair and traversing the right first can be made. Contrary to ray tracing traversal, the compact BVH representation does not require any additional information to be stored on the stack.

6.2.4. Results and Discussion

The compact representation of the BVH photon map is evaluated on the seven benchmark scenes from section A.2. Photon interactions and kernel support regions follow section 5.2.3, LBVH photon map construction and CUDA rendering kernel section 6.1.4. The baseline is a complete representation of the BVH photon map, using 32 bytes per node to encode its bounding planes and child or primitive references as described in section 3.2.2. With the compact BVH representation, 32 bytes are sufficient for a sibling node pair, thus reducing photon map storage requirements by 50%.

Statistics for a flight through each scene are provided in table 6.4. The number and order of nodes traversed is identical for both photon map representations. Using the baseline, whenever the query point $\vec{x}$ lies inside a node, traversal continues with one of its children while the other is pushed onto the stack. With the compact BVH representation, a sibling node pair is traversed together. Pushing onto the stack is only necessary if $\vec{x}$ lies inside *both* nodes. This reduces the number of stack operations by an average of 72.0%. With 64 respectively 32 bytes used to store a node pair and 4 bytes for a stack entry, the total global memory bandwidth required per query point during photon map traversal can be estimated using the simplified memory subsystem introduced in section 4.3.3. The compact BVH representation yields a reduction in bandwidth by 54.0%. The number of instructions executed per query point is reduced by 7.4% on average. As a result, the rendering kernel attains a speedup of 15.8%. The overhead of copying results to the screen for visualization is omitted.

We also evaluate the application of the compact BVH representation to ray tracing acceleration using the five benchmark scenes from section A.2.1. A BVH over the scene surfaces is constructed offline for each scene, employing the SAH with parameters tuned for maximal performance ($C_T = 1$, $C_I = 3$, recursive subdivision while $n_P > 5$ and $\min C_P(L, R) \leq p_P\, n_P\, C_I$). Large primitives are represented by multiple smaller AABBs [EG07]. The baseline is provided by a complete BVH representation recently proposed for ray tracing acceleration in CUDA [AL09] with sibling nodes stored in pairs. For each pair, 56 bytes are used to encode its twelve bounding planes and two child or primitive references. The compact BVH representation needs 32 bytes per pair, thus reducing storage requirements for BVH nodes by 42.9%.

Scene	n_T	Stack Accesses		Byte Accesses		Instructions		FPS	
		B	C	B	C	B	C	B	C
Scene 6	300.7	300.7	−68.5%	10826	−52.1%	228263	−5.0%	3.28	+8.0%
Sponza	457.8	457.8	−72.7%	16482	−52.5%	113632	−9.4%	4.99	+28.7%
Sibenik	523.4	523.4	−72.6%	18844	−52.5%	136347	−9.0%	4.11	+27.8%
Fairy	147.9	147.9	−77.3%	5323	−53.0%	16732	−8.9%	28.31	+11.9%
Conference	445.6	445.6	−71.4%	16042	−71.4%	152826	−7.2%	4.88	+14.9%
Passage	236.3	236.3	−68.4%	8505	−52.0%	196525	−5.2%	4.65	+6.2%
Ring	447.7	447.7	−72.5%	16119	−52.5%	89435	−7.5%	6.49	+14.7%
Ring Ep.	447.7	447.7	−72.5%	16119	−52.5%	90809	−7.4%	6.45	+14.2%
Average			−72.0%		−54.0%		−7.4%		+15.8%
Std. Dev.			2.8%		7.4%		1.6%		8.3%

Table 6.4.: Global memory accesses during node traversal, instructions per query point and rendering frame rates with compact BVH photon map (C), relative to complete BVH (B)

Our original evaluation uses an older variant of benchmark code base II from section A.1.2. Images are rendered at 1024×1024 resolution, tracing one primary ray and, if a surface is hit, one shadow ray per pixel. With the compact BVH representation, the parameter interval $[t_a, t_b]$ is stored on the stack. This allows the traversal of a node pair to be omitted when $t_a > t_{max}$, indicating that a nearer hit than its entry point has already been found. Storing the entry point t_a on the stack enables the same check for primary ray traversal in the baseline, yielding a slight speedup. The size of a stack entry becomes 8 bytes for primary rays, 4 bytes for shadow rays in the baseline and 12 bytes for all rays with the compact BVH representation.

The average number of nodes traversed, stack operations performed and bytes accessed in global memory for these per ray are listed in table 6.5(a) for a flight through each scene. With the compact BVH representation, global memory bandwidth requirements are reduced by an average of 33.8%. Although the computationally more expensive traversal algorithm increases the average number of instructions executed by 20.8%, the rendering kernel proves memory bound, benefiting from the lower bandwidth cost and attaining a speedup of 0.8% on average.

A reevaluation for the current code base II is presented in table 6.5(b). Images are rendered at 512×512 resolution with up to four rays per pixel as specified in section A.1. The average number of nodes traversed per ray is listed for the baseline. Only node pairs are pushed onto the stack for it as storing entry points yields a slowdown. This reduces the entry size from 8 to 4 bytes but makes the check whether $t_a > t_{max}$ impossible. Using the compact BVH representation, the check is retained and 2.39% fewer nodes are traversed on average. The number of stack accesses is identical for both representations. With the simplified memory model, the compact BVH representation has 31.3% lower global memory bandwidth requirements on average.

The rendering kernel uses 61 registers in the baseline and 67 for the compact BVH representation. An increase over the threshold of 64 registers reduces occupancy. To avoid it, we force the register count to 64, spilling some information into slow local memory instead. Combined with the execution of 35.5% more instructions on average than the baseline, this reverses the previously observed speedup, leading to an average slowdown by 7.5% instead.

In summary, the compact BVH representation successfully reduces storage and global memory bandwidth requirements. Applied to the photon map, traversal becomes more efficient and a speedup

Scene	Nodes	Accesses		Byte Accesses		Instructions		FPS	
		Node	Stack	B	C	B	C	B	C
Scene 6	582	17.41	4.39	534.6	−30.3%	1386.4	+16.1%	88.0	+0.1%
Sponza	46942	50.36	10.09	1488.2	−35.4%	3065.9	+22.3%	43.7	+0.4%
Sibenik	56870	67.50	13.14	1999.9	−35.0%	3942.6	+22.8%	33.9	+1.9%
Fairy	97086	49.14	8.73	1459.9	−34.7%	3166.0	+21.1%	42.2	+0.5%
Conference	149352	41.85	8.09	1251.4	−33.8%	2618.7	+21.7%	49.8	+1.3%
Average					−33.8%		+20.8%		+0.8%
Std. Dev.					2.1%		2.7%		0.8%

(a) Older code base, 1024 × 1024 resolution, primary and shadow rays

Scene	Nodes	Accesses		Byte Accesses		Instructions		FPS	
		Node	Stack	B	C	B	C	B	C
Scene 6	582	16.99	4.04	508.2	−27.5%	1213.2	+36.4%	242.5	−9.1%
Sponza	46942	53.10	10.10	1567.5	−32.9%	3242.0	+33.6%	163.1	−6.2%
Sibenik	56870	70.66	13.28	2084.8	−32.1%	4276.5	+37.1%	111.9	−6.5%
Fairy	97086	51.83	8.91	1522.4	−32.7%	3573.5	+34.1%	145.4	−9.2%
Conference	149352	43.45	8.25	1282.7	−31.3%	2843.7	+35.4%	172.4	−6.6%
Average					−31.3%		+35.3%		−7.5%
Std. Dev.					2.2%		1.4%		1.5%

(b) Current code base, 512 × 512 resolution, primary, reflection and shadow rays

Table 6.5.: Global memory accesses during node traversal, instructions per ray and frame rates with compact BVH (C), relative to complete BVH (B)

over a complete BVH representation is achieved. While a similar reduction in storage and global memory bandwidth is possible for ray tracing, the more complex traversal algorithm may lead to a slowdown instead, depending on the specific benchmark environment used.

6.3. Combined Results

By combining the techniques from chapters 4 to 6, a complete photon mapping algorithm is obtained that efficiently operates on the CUDA manycore platform. A kd-tree is constructed over the scene surfaces first, using SIROH as described in section 4.1. Photons are then traced in CUDA, tracking differentials with each by following section 5.1 and computing anisotropic kernel support regions at their hit points as described in section 5.2. A BVH photon map is constructed over these in CUDA using the efficient LBVH algorithm from section 6.1.2. To synthesize an image, the CUDA kernel from code base I in section A.1.2 is used. Tracing up to four rays per pixel and invoking density estimation at the primary and reflection ray hit points, global illumination is rendered into a buffer. This is finally visualized by rasterizing a quad with the buffer as a texture in OpenGL.

The image quality obtained is demonstrated in section 5.2.3. Table 6.6 shows that with the same settings, interactive frame rates are possible when illumination dynamically changes and all photons are retraced in each frame. Results are given as averages over a flight through each scene as before. The light source is now also animated, however, dynamically changing illumination. Frame rates are measured for the complete rendering algorithm, including visualization by OpenGL. Interactive rendering with at least three frames per second results for all benchmark scenes from section A.2.

Scene	Interactions	k	Time (ms)			FPS
			Trace	LBVH	Render	
Scene 6	163038	337.6	10.53	10.75	264.77	3.50
Sponza	299895	199.1	55.95	18.29	201.51	3.63
Sibenik	416996	192.2	102.47	26.24	197.08	3.07
Fairy	9478	36.0	14.93	2.63	34.27	19.30
Conference	82867	256.9	29.14	6.38	228.89	3.78
Passage	69083	459.6	3.98	5.84	208.63	4.58
Ring	75525	186.6	7.29	6.46	156.45	5.88
Ring Ep.	75525	186.6	7.29	6.46	158.11	5.82

Table 6.6.: Photon interactions, contributions per query, times per frame for photon tracing, LBVH photon map construction, rendering and frame rates with dynamic illumination

A breakdown of the time per frame into the components photon tracing, photon map construction and rendering shows that the last step is the most expensive for all scenes. This is in line with the fact that the rendering kernel performs the majority of operations, tracing rays, traversing the BVH photon map and accumulating the contributions of the retrieved photon interactions.

7. Participating Media

The photon mapping algorithm extends to *volumetric photon mapping* for simulating light transport in participating media. Following section 3.4.1.3, each photon is propagated until it either encounters a surface or is extinguished by a particle suspended in the atmosphere. An *extinction event* is stored in the latter case and Russian roulette used to decide between absorption or scattering into a new direction. During image synthesis, radiance in-scattered to a ray passing through the medium is then reconstructed from nearby extinction events by density estimation as described in section 3.4.2.4. A volume photon map accelerates extinction event retrieval.

In this chapter, the efficient photon mapping components introduced in the preceding three chapters are adapted to volumetric photon mapping. Our first contribution is a rederivation of the beam radiance estimate. This reformulation of volumetric photon mapping replaces ray marching plus density estimation after each step with a more efficient single density estimation per ray. However, its derivation suffers from incorrect physical units. We show that the equations proposed are in fact correct and only their relationship to photon flux requires adjustment.

The beam radiance estimate employs variable kernel density estimation. A kernel bandwidth h_j is obtained for each extinction event by locating the k nearest events in a preprocessing pass. We extend photon differentials to adapt to the influence of the participating medium instead, eliminating the use of any offline preprocessing. Due to the high computational cost of volumetric photon mapping, no anisotropic kernel support region but a single bandwidth h_j is computed from the differentials. The adaptation dampening that approximatively accounts for the effect of illumination arriving via different paths is adapted for volumetric photon mapping.

We then focus on the rendering component. By splitting it into multiple CUDA kernels, a different arrangement of computations in each step is possible. While known to be detrimental to ray tracing, we show that *packetization* is beneficial for density estimation. Different methods of arranging rays into packets are explored. Combining all components, we demonstrate interactive frame rates with volumetric photon mapping and dynamically changing illumination.

7.1. Beam Radiance Estimate

Radiance arriving at a point $\vec{x}$ from a direction $\vec{\omega}$ in the presence of a participating medium is given by equation 3.19,

$$\begin{aligned} L_i(\vec{x},-\vec{\omega}) &= \tau\left(0,t_{max}\right) L_o\left(\vec{r}\left(\vec{x},-\vec{\omega}\right),\vec{\omega}\right) \\ &\quad + \int_0^{t_{max}} \tau\left(0,t\right)\left(\sigma_s\left(t\right)\int_{\Omega_{4\pi}} p\left(t,\vec{\omega}_i,\vec{\omega}\right) L\left(\vec{z}\left(t\right),\vec{\omega}_i\right)\, d\Omega_i + \epsilon\left(t,\vec{\omega}\right)\right)\, dt \\ &= \tau\left(0,t_{max}\right) L_o\left(\vec{r}\left(\vec{x},-\vec{\omega}\right),\vec{\omega}\right) \\ &\quad + \int_0^{t_{max}} \tau\left(0,t\right)\sigma_s\left(t\right)\int_{\Omega_{4\pi}} p\left(t,\vec{\omega}_i,\vec{\omega}\right) L\left(\vec{z}\left(t\right),\vec{\omega}_i\right)\, d\Omega_i\, dt + \int_0^{t_{max}} \tau\left(0,t\right)\epsilon\left(t,\vec{\omega}\right)\, dt. \end{aligned}$$

For the first term, the radiance L_o emanating from the nearest surface seen is computed using the original photon mapping algorithm and attenuated by the transmittance of the medium. Following section 3.4.2.4, transmittance is calculated directly in a homogeneous medium and accumulated by ray marching otherwise. The last term is the radiance ϵ emitted by the medium, accumulated and attenuated by ray marching. Volumetric photon mapping is concerned with the central term, in-scattered radiance,

$$\int_0^{t_{max}} \tau\left(0,t\right)\sigma_s\left(t\right)\int_{\Omega_{4\pi}} p\left(t,\vec{\omega}_i,\vec{\omega}\right)L\left(\vec{z}\left(t\right),\vec{\omega}_i\right)\,d\Omega_i\,dt.$$

Inserting the relationship between radiance and the differential change in flux from equation 3.21,

$$=\int_0^{t_{max}} \tau\left(0,t\right)\alpha\left(t\right)\int_{\Omega_{4\pi}} p\left(t,\vec{\omega}_i,\vec{\omega}\right)\left(-\frac{d^2\Phi}{dV\,d\Omega_i}\right)_{extinction}\left(\vec{z}\left(t\right),\vec{\omega}_i\right)d\Omega_i\,dt.$$

Decomposing the differential volume into $dV = dt\,dA^{\perp}$, the medium is approximated as homogeneous and the incident radiance field as constant for any plane orthogonal to the ray $\vec{z}\left(t\right)=\vec{x}+t\vec{\omega}$. Differentiation with respect to $dA^{\perp}$ and integration can then be swapped, yielding

$$\approx \frac{d}{dA^{\perp}}\left(\int_0^{t_{max}} \tau\left(0,t\right)\alpha\left(t\right)\int_{\Omega_{4\pi}} p\left(t,\vec{\omega}_i,\vec{\omega}\right)\left(-\frac{d^2\Phi}{dt\,d\Omega_i}\right)_{extinction}\left(\vec{z}\left(t\right),\vec{\omega}_i\right)\,d\Omega_i\,dt\right)\left(\vec{x}\right).$$

In-scattered radiance is now expressed as the density of in-scattered intensity per unit area in the tangential plane at the ray origin, enabling weighted kernel density estimation. Given n intensity samples for parallel rays with origins $\vec{x}_j$ in the tangential plane, the estimator is

$$\approx \sum_{j=1}^{n}\frac{1}{h^2}K\left(\frac{\vec{x}_j-\vec{x}}{h}\right)\left(\int_0^{t_{max}} \tau\left(0,t\right)\alpha\left(t\right)\int_{\Omega_{4\pi}} p\left(t,\vec{\omega}_i,\vec{\omega}\right)\left(-\frac{d^2\Phi}{dt\,d\Omega_i}\right)_{extinction}\left(\vec{z}_j\left(t\right),\vec{\omega}_i\right)\,d\Omega_i\,dt\right)\left(\vec{x}_j\right).$$

Each of the n extinction events in the volume photon map corresponds to a change in flux by $-\Phi_{i,j}$ at a position $\vec{z}_j$ for a ray traveling in direction $\vec{\omega}_{i,j}$ due to extinction in the participating medium. Parameterizing the position as one along a ray with direction $\vec{\omega}$ and an origin $\vec{x}_j$ in the tangential plane, $\vec{z}_j=\vec{z}_j\left(t_j\right)=\vec{x}_j+t_j\vec{\omega}$, a sample of reflected intensity is obtained analogously to section 3.4.2 by exploiting the single incident direction and single incident ray parameter to move the transmittance, absorption coefficient and phase functions outside the integrals,

$$=\sum_{j=1}^{n}\frac{1}{h^2}K\left(\frac{\vec{x}_j-\vec{x}}{h}\right)\tau\left(0,t_j\right)\alpha\left(t_j\right)p\left(t_j,\vec{\omega}_{i,j},\vec{\omega}\right)\left(\int_0^{t_{max}}\int_{\Omega_{4\pi}}\left(-\frac{d^2\Phi}{dt\,d\Omega_i}\right)_{extinction}\left(t,\vec{\omega}_i\right)\,d\Omega_i\,dt\right)\left(\vec{x}_j\right).$$

The extinguished flux integrates to $-\Phi_{i,j}$ so that

$$=\sum_{j=1}^{n}\frac{1}{h^2}K\left(\frac{\vec{x}_j-\vec{x}}{h}\right)\tau\left(0,t_j\right)\alpha\left(t_j\right)p\left(t_j,\vec{\omega}_{i,j},\vec{\omega}\right)\Phi_{i,j}. \tag{7.1}$$

Expressed in the same notation, the original beam radiance estimate [JZJ08a] is

$$\sum_{j=1}^{n}\frac{1}{h^2}K\left(\frac{\vec{x}_j-\vec{x}}{h}\right)\tau\left(0,t_j\right)\sigma_s\left(t_j\right)p\left(t_j,\vec{\omega}_{i,j},\vec{\omega}\right)\Phi_{i,j}. \tag{7.2}$$

Comparing equations 7.1 and 7.2, it is apparent that what is referred to as *photon power* $\Phi_{i,j}$ with unit W in the original derivation of the beam radiance estimate is in fact the quotient of flux $\Phi_{i,j}$ and extinction coefficient $\sigma_e(t_j)$ with unit Wm. After applying this correction, equation 7.2 becomes identical to equation 7.1. Variable kernel density estimation is enabled by assigning each extinction event an individual kernel bandwidth h_j. In-scattered radiance is then estimated as

$$\sum_{j=1}^{n} \frac{1}{h_j^2} K\left(\frac{\vec{x}_j - \vec{x}}{h_j}\right) \tau(0, t_j)\, \alpha(t_j)\, p(t_j, \vec{\omega}_{i,j}, \vec{\omega})\, \Phi_{i,j}. \tag{7.3}$$

Only the length of $\vec{x}_j - \vec{x}$ is relevant for a radially symmetric kernel. This is the distance between the projection of an extinction event position $\vec{z}_j$ onto the tangential plane at the ray origin $\vec{x}$ and the ray origin itself. Equivalently, it is the shortest distance at which the ray passes $\vec{z}_j$. All extinction events must thus be retrieved and their contributions evaluated whose kernel support region is intersected by the ray, the ray thus passing within a distance less than h_j of $\vec{z}_j$. The ray parameter for which this closest distance results is equal to the distance t_j between $\vec{z}_j$ and the tangential plane. Following section 3.1.3, it is given by $t_j = (\vec{z}_j - \vec{x}) \cdot \vec{\omega}$.

7.2. Photon Differentials

Use of equation 7.3 to estimate in-scattered radiance requires that a bandwidth h_j be chosen at each extinction event. The method from chapter 5 based on photon differentials can be extended for this purpose. Initialization of the photon differentials at emission from a light source and their update upon interaction with a surface follow section 5.1. Additional influences that should be accounted for are photon extinction during propagation, sampling of photon directions when scattering occurs in the medium and Russian roulette deciding between absorption and scattering.

7.2.1. Propagation

A photon is propagated through a participating medium until it either encounters a surface or is extinguished. The distance to the nearest surface hit point $t_{hit}(p_1, p_2)$ can be found by ray tracing. To determine the distance to an extinction event, its cumulative distribution function $F(t_e)$ must be importance-sampled, introducing an additional parameter p_3. A parameterization of the extinction distance is $t_e(p_3) = F^{-1}(p_3)$ with uniformly distributed $p_3 \in [0, 1)$.

The cumulative distribution function is derived in section 3.4.1.3 as $F(t_e) = 1 - \tau(0, t_e)$ with $\tau(0, t_e) = e^{-\int_0^{t_e} \sigma_e(s)\, ds}$ the transmittance of the medium. For a homogeneous medium, inverting the cumulative distribution function allows a uniformly distributed sample $y \in [0, 1)$ to be transformed into $t_e = -\sigma_e^{-1} \log(1 - y)$. In the case of an inhomogeneous medium, no closed-form solution exists but an efficient and unbiased sampling method is provided.

7.2.1.1. Until Surface Interaction

With probability $1 - F(t_{hit}(p_1, p_2)) = \tau(0, t_{hit}(p_1, p_2))$, the distances satisfy $t_{hit}(p_1, p_2) \leq t_e(p_3)$ so that the photon encounters a surface at distance $t_{hit}(p_1, p_2)$ before becoming extinguished. Its position and differentials are updated as described in section 3.1.2.3. Analogously to the handling of Russian roulette in section 5.1.4, the assumption is made that other photons reach a surface

with equal probability. The effect of extinction in the participating medium then matches that of Russian roulette, reducing the number of photons to a fraction $\tau\left(0, t_{hit}\left(p_1, p_2\right)\right)$ without affecting their distribution. It can thus be handled in the same way, scaling differentials with $\tau^{-\frac{1}{2}}\left(0, t_{hit}\left(p_1, p_2\right)\right)$ and avoiding the need to track differentials with respect to an additional parameter. After propagation, the photon interacts with the surface as described in chapter 5.

7.2.1.2. Until Extinction Event

If $t_{hit}\left(p_1, p_2\right) > t_e\left(p_3\right)$, the photon is extinguished in the participating medium after propagation by distance $t_e\left(p_3\right)$. Its position and direction thus become

$$\vec{z}'\left(p_1, p_2, p_3\right) = \vec{z}\left(p_1, p_2\right) + t_e\left(p_3\right)\, \vec{\omega}\left(p_1, p_2\right), \qquad \vec{\omega}'\left(p_1, p_2, p_3\right) = \vec{\omega}\left(p_1, p_2\right).$$

Differentiation with respect to the now three parameters yields six differentials,

$$\frac{\partial \vec{z}'}{\partial p_1}\left(p_1, p_2, p_3\right) = \underbrace{\frac{\partial \vec{z}}{\partial p_1}\left(p_1, p_2\right)}_{\text{old differential}} + \underbrace{t_e\left(p_3\right)}_{\substack{\text{extinction}\\ \text{distance}}} \underbrace{\frac{\partial \vec{\omega}}{\partial p_1}\left(p_1, p_2\right)}_{\text{old differential}}, \qquad \frac{\partial \vec{\omega}'}{\partial p_1}\left(p_1, p_2, p_3\right) = \underbrace{\frac{\partial \vec{\omega}}{\partial p_1}\left(p_1, p_2\right)}_{\text{old differential}}, \tag{7.4}$$

$$\frac{\partial \vec{z}'}{\partial p_2}\left(p_1, p_2, p_3\right) = \underbrace{\frac{\partial \vec{z}}{\partial p_2}\left(p_1, p_2\right)}_{\text{old differential}} + \underbrace{t_e\left(p_3\right)}_{\substack{\text{extinction}\\ \text{distance}}} \underbrace{\frac{\partial \vec{\omega}}{\partial p_2}\left(p_1, p_2\right)}_{\text{old differential}}, \qquad \frac{\partial \vec{\omega}'}{\partial p_2}\left(p_1, p_2, p_3\right) = \underbrace{\frac{\partial \vec{\omega}}{\partial p_2}\left(p_1, p_2\right)}_{\text{old differential}}, \tag{7.5}$$

$$\frac{\partial \vec{z}'}{\partial p_3}\left(p_1, p_2, p_3\right) = \frac{\partial t_e}{\partial p_3}\left(p_3\right)\, \underbrace{\vec{\omega}\left(p_1, p_2\right)}_{\text{old direction}}, \qquad \frac{\partial \vec{\omega}'}{\partial p_3}\left(p_1, p_2, p_3\right) = \vec{0}. \tag{7.6}$$

Only the derivative of the extinction distance $t_e\left(p_3\right)$ with respect to p_3 is unknown. To compute it, the inverse $p_3\left(t_e\right) = F\left(t_e\right)$ of $t_e\left(p_3\right) = F^{-1}\left(p_3\right)$ is differentiated first

$$\frac{\partial p_3}{\partial t_e}\left(t_e\right) = \frac{\partial\left(1 - \tau\left(0, t\right)\right)}{\partial t}\left(t_e\right) = \frac{\partial(1 - e^{-\int_0^t \sigma_e(s)\, ds})}{\partial t}\left(t_e\right) = \sigma_e\left(t_e\right) e^{-\int_0^{t_e} \sigma_e(s)\, ds} = \sigma_e\left(t_e\right) \tau\left(0, t_e\right).$$

The reciprocal yields the required derivative of $t_e\left(p_3\right)$,

$$\frac{\partial t_e}{\partial p_3}\left(p_3\right) = \frac{1}{\sigma_e\left(t_e\left(p_3\right)\right) \tau\left(0, t_e\left(p_3\right)\right)}. \tag{7.7}$$

Inserting equation 7.7 into equation 7.6 allows photon differentials to be updated after propagation. Following section 5.1, the differentials with respect to p_1, p_2 are scaled such that the expected distances between the photon and its neighbors in two-dimensional parameter space are $\Delta = \sqrt{4\pi n^{-1}}$ for both. Scaling them by the additional factor $\sqrt{4\pi}$ then leads to expected distances $\Delta' = \sqrt{n^{-1}}$. As described in section 3.1.2.5, this corresponds to a uniform distribution of n photons throughout two-dimensional parameter space with unit scale for each parameter. Since p_3 is also uniformly sampled on a unit scale, p_1, p_2, p_3 together uniformly sample a three-dimensional unit parameter space. The expected distance for each then becomes $\Delta'' = \sqrt[3]{n^{-1}}$. With this, three footprint spanning vectors can be computed by first order Taylor approximation,

$$\vec{\Delta}_1\left(p_1, p_2, p_3\right) = \sqrt[3]{\frac{1}{n}} \sqrt{4\pi} \frac{\partial \vec{z}'}{\partial p_1}\left(p_1, p_2, p_3\right), \tag{7.8}$$

$$\vec{\Delta}_2\left(p_1, p_2, p_3\right) = \sqrt[3]{\frac{1}{n}}\sqrt{4\pi}\frac{\partial \vec{z}''}{\partial p_2}\left(p_1, p_2, p_3\right), \tag{7.9}$$

$$\vec{\Delta}_3\left(p_1, p_2, p_3\right) = \sqrt[3]{\frac{1}{n}}\frac{\partial \vec{z}''}{\partial p_3}\left(p_1, p_2, p_3\right). \tag{7.10}$$

These three vectors span a parallelepiped, yielding a three-dimensional footprint. An undesirable effect is a growth in computational cost and storage space due to the two additional differentials with respect to p_3. However, six photon differentials are a temporary state. The footprint is processed to compute a single kernel bandwidth h_j at the extinction event. If the photon is scattered, the number of parameters and thus differentials is reduced as described in the following section.

7.2.2. Scattering

When a photon is scattered after extinction, two additional parameters p'_1, p'_2 must be sampled to choose a new direction. As with diffuse reflection in section 5.1.3, we treat scattering as an absorption and a reemission with parameters p'_1, p'_2 only. The three parameters p_1, p_2, p_3 after propagation are thus replaced with p'_1, p'_2, reducing their number to two again.

Scattering follows a phase function p modeling the redirection of *radiance* into the sphere of directions. Since the scattering occurs at a point in the participating medium, radiance is not projected onto any surface, making *intensity* and radiance proportional. A new photon direction is thus chosen by importance-sampling the phase function. For an isotropic medium, this is given by equation 2.21,

$$p\left(\cos\theta\right) = \frac{1}{4\pi}. \tag{7.11}$$

Anisotropic scattering is modeled by the Schlick phase function from equation 2.22 with $k \in (-1, 1)$,

$$p\left(\cos\theta\right) = \frac{1-k^2}{4\pi\left(1-k\cos\theta\right)^2}. \tag{7.12}$$

Equation 7.12 simplifies to equation 7.11 for $k = 0$. Only anisotropic scattering is thus discussed further, including isotropic scattering as a special case. The phase angle θ is the inclination of the new photon direction in a local coordinate frame with $\vec{\omega}\left(p_1, p_2, p_3\right)$ as zenith. Since $\cos\theta$ is strictly monotonic for $\theta \in [0, \pi]$, a change of variable is possible to

$$p\left(\theta\right) = p\left(\cos\theta\right)\left|\frac{d\cos\theta}{d\theta}\right| = \frac{\left(1-k^2\right)\sin\theta}{4\pi\left(1-k\cos\theta\right)^2}. \tag{7.13}$$

Using $p'_1 \in [0, \pi]$, $p'_2 \in [0, 2\pi)$ to denote inclination and azimuth in this local coordinate frame and a rotation matrix $R\left(p_1, p_2, p_3\right)$ to encode its orientation, the photon is considered to originate at the scattering point with parameters p'_1, p'_2 only so that

$$\vec{z}''\left(p'_1, p'_2\right) = \vec{z}\left(p_1, p_2, p_3\right), \qquad \vec{\omega}'\left(p'_1, p'_2\right) = R\left(p_1, p_2, p_3\right)\begin{pmatrix}\cos p'_2 \sin p'_1\\ \sin p'_2 \sin p'_1\\ \cos p'_1\end{pmatrix}. \tag{7.14}$$

This parameterization using spherical coordinates is equal to that employed during emission from a point light source. The corresponding differentials are therefore identical to equations 5.2 and 5.3 with parameters p'_1, p'_2 substituted for p_1, p_2. To importance-sample equation 7.13 and determine a

new photon direction, the marginal probability density functions are computed for the two parameters first, yielding

$$p\left(p_1'\right) = \int_0^{2\pi} \frac{\left(1-k^2\right)\sin p_1'}{4\pi\left(1-k\cos p_1'\right)^2}\, dp_2' = \frac{\left(1-k^2\right)\sin p_1'}{2\left(1-k\cos p_1'\right)^2}, \tag{7.15}$$

$$p\left(p_2'\right) = \int_0^{\pi} \frac{\left(1-k^2\right)\sin p_1'}{4\pi\left(1-k\cos p_1'\right)^2}\, dp_1' = \frac{1}{2\pi}. \tag{7.16}$$

Integration leads to the cumulative distribution functions

$$F\left(p_1'\right) = \int_0^{p_1'} \frac{\left(1-k^2\right)\sin p}{2\left(1-k\cos p\right)^2}\, dp = \frac{k^2-1}{2k\left(1-k\cos p_1'\right)} - \frac{k^2-1}{2k\left(1-k\right)},$$

$$F\left(p_2'\right) = \int_0^{p_2'} \frac{1}{2\pi}\, dp = \frac{1}{2\pi} p_2'.$$

Parameters distributed according to equations 7.15 and 7.16 are obtained by transforming uniformly distributed samples $y_1' \in [0,2]$, $y_2' \in [0,2\pi)$ via the inverted cumulative distribution functions into $p_1' = \arccos \frac{y_1'+k-1}{k y_1' - k + 1}$, $p_2' = y_2'$. The expected distances $\Delta p_1'$, $\Delta p_2'$ to neighboring photons are derived by following the same steps as for diffuse reflection in section 5.1.3. With n photons traced in total, the distances to neighbors are computed by assuming all sample the same parameter space. Difference vectors between the photon direction and those of its neighbors are estimated using first-order Taylor approximation as $\Delta p_1' \frac{\partial \vec{\omega}'}{\partial p_1'}\left(p_1', p_2'\right)$ and $\Delta p_2' \frac{\partial \vec{\omega}'}{\partial p_2'}\left(p_1', p_2'\right)$. The expected solid angle spanned by neighboring photon directions is proportional to the inverse of $p\left(\vec{\omega}'\right)$. With spherical coordinates sampled according to the probability density function in equation 7.13 and their distorted mapping to the sphere of directions, the expected solid angle thus is

$$\frac{c}{p\left(\vec{\omega}'\right)} = \frac{c}{p\left(p_1', p_2'\right)\frac{1}{\sin p_1'}} = c\frac{4\pi\left(1-k\cos p_1'\right)^2}{1-k^2}.$$

The proportionality constant c follows from the expectation that all n photons together cover the entire sphere of directions,

$$\mathbb{E}\left[nc\frac{4\pi\left(1-k\cos p_1'\right)^2}{1-k^2}\right] = 4\pi \Leftrightarrow c = \frac{1}{n}.$$

Approximating the distribution of photon directions as locally uniform, the expected lengths of both difference vectors are equal so that

$$\mathbb{E}\left[\left\|\Delta p_1' \frac{\partial \vec{\omega}'}{\partial p_1'}\left(p_1', p_2'\right)\right\|\right] = \mathbb{E}\left[\left\|\Delta p_2' \frac{\partial \vec{\omega}'}{\partial p_2'}\left(p_1', p_2'\right)\right\|\right] = \sqrt{\frac{4\pi\left(1-k\cos p_1'\right)^2}{n\left(1-k^2\right)}}. \tag{7.17}$$

Since differentials after reemission at the scattering point are identical to equations 5.2 and 5.3, their lengths follow equation 5.7. Combining these with equation 7.17, the expected distances in parameter space between neighboring photons are

$$\Delta p_1' \approx \mathbb{E}\left[\Delta p_1'\right] = \sqrt{\frac{4\pi\left(1-k\cos p_1'\right)^2}{n\left(1-k^2\right)}}, \qquad \Delta p_2' \approx \mathbb{E}\left[\Delta p_2'\right] = \sqrt{\frac{4\pi\left(1-k\cos p_1'\right)^2}{n\left(1-k^2\right)}}\frac{1}{\sin p_1'}.$$

The need to track these with each photon is eliminated analogously to emission and diffuse reflection. The differentials are scaled by $\frac{|1-k\cos p_1'|}{\sqrt{1-k^2}}$ respectively $\frac{|1-k\cos p_1'|}{\sqrt{1-k^2}}\sin^{-1} p_1'$, obtaining

$$\frac{\partial \vec{z}''}{\partial p_1'}\left(p_1', p_2'\right) = \vec{0}, \qquad \frac{\partial \vec{\omega}'}{\partial p_1'}\left(p_1', p_2'\right) = \frac{|1-k\cos p_1'|}{\sqrt{1-k^2}} R\left(p_1, p_2, p_3\right) \begin{pmatrix} \cos p_2' \cos p_1' \\ \sin p_2' \cos p_1' \\ -\sin p_1 \end{pmatrix}, \tag{7.18}$$

$$\frac{\partial \vec{z}''}{\partial p_2'}\left(p_1', p_2'\right) = \vec{0}, \qquad \frac{\partial \vec{\omega}'}{\partial p_2'}\left(p_1', p_2'\right) = \frac{|1-k\cos p_1'|}{\sqrt{1-k^2}} R\left(p_1, p_2, p_3\right) \begin{pmatrix} -\sin p_2' \\ \cos p_2' \\ 0 \end{pmatrix}. \tag{7.19}$$

Scaled by the inverses of these factors, the distances in parameter space become identical to those of equation 5.11, allowing the same global constant to be used for all photons,

$$\Delta p_1' = \Delta p_2' = \sqrt{\frac{4\pi}{n}}.$$

Treating scattering as an absorption and a reemission has the disadvantage of discarding any prior footprint adaptation. To approximatively preserve it, the reemission point is virtually offset to $\vec{z}'\left(p_1', p_2'\right) - t_v\left(p_1, p_2, p_3\right)\vec{\omega}'\left(p_1', p_2'\right)$, analogously to section 5.1.3. Since the offsetting is virtual, the photon always reaches the actual scattering point at $\vec{z}''\left(p_1', p_2'\right) = \vec{z}\left(p_1, p_2, p_3\right)$ without being affected by the participating medium. Its direction $\vec{\omega}''\left(p_1', p_2'\right) = \vec{\omega}'\left(p_1', p_2'\right)$ is unchanged, the differentials are

$$\frac{\partial \vec{z}''}{\partial p_1'}\left(p_1', p_2'\right) = t_v\left(p_1, p_2, p_3\right)\frac{\partial \vec{\omega}'}{\partial p_1'}\left(p_1', p_2'\right), \qquad \frac{\partial \vec{\omega}''}{\partial p_1'}\left(p_1', p_2'\right) = \frac{\partial \vec{\omega}'}{\partial p_1'}\left(p_1', p_2'\right), \tag{7.20}$$

$$\frac{\partial \vec{z}''}{\partial p_2'}\left(p_1', p_2'\right) = t_v\left(p_1, p_2, p_3\right)\frac{\partial \vec{\omega}'}{\partial p_2'}\left(p_1', p_2'\right), \qquad \frac{\partial \vec{\omega}''}{\partial p_2'}\left(p_1', p_2'\right) = \frac{\partial \vec{\omega}'}{\partial p_2'}\left(p_1', p_2'\right). \tag{7.21}$$

Parameters p_1, p_2, p_3 lead to a three-dimensional footprint before scattering. The adaptation is approximatively preserved by maintaining its volume. Since scattering yields parameters p_1', p_2' only, the assumption of a subsequent propagation by *zero distance* is made to obtain a three-dimensional footprint after scattering also. This corresponds to $p_3' = 0$ since $t_e\left(0\right) = 0$. The photon position $\vec{z}'''\left(p_1', p_2', 0\right) = \vec{z}\left(p_1, p_2, p_3\right)$ and direction $\vec{\omega}'''\left(p_1', p_2'\right) = \vec{\omega}'\left(p_1', p_2'\right)$ remain unchanged. Inserting the zero propagation distance $t_e\left(0\right) = 0$ into equations 7.4 and 7.5 shows that the position differentials with respect to parameters p_1', p_2' also do not change,

$$\frac{\partial \vec{z}'''}{\partial p_1'}\left(p_1', p_2', 0\right) = \frac{\partial \vec{z}''}{\partial p_1'}\left(p_1', p_2'\right) + t_e\left(0\right)\frac{\partial \vec{\omega}''}{\partial p_1'}\left(p_1', p_2'\right) = \frac{\partial \vec{z}''}{\partial p_1'}\left(p_1', p_2'\right), \tag{7.22}$$

$$\frac{\partial \vec{z}'''}{\partial p_2'}\left(p_1', p_2', 0\right) = \frac{\partial \vec{z}''}{\partial p_2'}\left(p_1', p_2'\right) + t_e\left(0\right)\frac{\partial \vec{\omega}''}{\partial p_2'}\left(p_1', p_2'\right) = \frac{\partial \vec{z}''}{\partial p_2'}\left(p_1', p_2'\right). \tag{7.23}$$

An additional position differential with respect to parameter p_3' is given by equation 7.7. Inserting equation 7.6 and noting that $\tau\left(0, 0\right) = 1$, the differential is

$$\frac{\partial \vec{z}'''}{\partial p_3'}\left(p_1', p_2', 0\right) = \frac{\partial t_e}{\partial p_3'}\left(0\right)\vec{\omega}''\left(p_1', p_2'\right) = \frac{1}{\sigma_e\left(t_e\left(0\right)\right)\tau\left(0, t_e\left(0\right)\right)}\vec{\omega}''\left(p_1', p_2'\right) = \frac{1}{\sigma_e\left(0\right)}\vec{\omega}''\left(p_1', p_2'\right). \tag{7.24}$$

The three corresponding footprint spanning vectors are computed by inserting the differentials from equations 7.22 to 7.24 into equations 7.8 to 7.10, yielding a footprint volume after scattering of

$$V'\left(p'_1, p'_2\right) = \left\| \left\langle \vec{\Delta}'_1\left(p'_1, p'_2, 0\right), \vec{\Delta}'_2\left(p'_1, p'_2, 0\right), \vec{\Delta}'_2\left(p'_1, p'_2, 0\right) \right\rangle \right\|$$
$$= \frac{4\pi}{n\sigma_e\left(0\right)} \left\| \left\langle \frac{\partial \vec{z}''}{\partial p'_1}\left(p'_1, p'_2\right), \frac{\partial \vec{z}''}{\partial p'_2}\left(p'_1, p'_2\right), \vec{\omega}''\left(p'_1, p'_2\right) \right\rangle \right\|.$$

Inserting equations 7.20, 7.21 and $\vec{\omega}''\left(p'_1, p'_2\right) = \vec{\omega}'\left(p'_1, p'_2\right)$, this becomes

$$V'\left(p'_1, p'_2\right) = \frac{4\pi}{n\sigma_e\left(0\right)} t_v^2\left(p_1, p_2, p_3\right) \left\| \left\langle \frac{\partial \vec{\omega}'}{\partial p'_1}\left(p'_1, p'_2\right), \frac{\partial \vec{\omega}'}{\partial p'_2}\left(p'_1, p'_2\right), \vec{\omega}'\left(p'_1, p'_2\right) \right\rangle \right\|.$$

With the photon direction from equation 7.14 and its differentials from equations 7.18 and 7.19, the expression evaluates to

$$V'\left(p'_1, p'_2\right) = \frac{4\pi}{n\sigma_e\left(0\right)} t_v^2\left(p_1, p_2, p_3\right) \frac{\left(1 - k\cos p'_1\right)^2}{1 - k^2}. \tag{7.25}$$

The footprint volume before scattering follows from equations 7.8 to 7.10 as

$$V\left(p_1, p_2, p_3\right) = \left\| \left\langle \vec{\Delta}_1\left(p_1, p_2, p_3\right), \vec{\Delta}_2\left(p_1, p_2, p_3\right), \vec{\Delta}_2\left(p_1, p_2, p_3\right) \right\rangle \right\|$$
$$= \frac{4\pi}{n} \left\| \left\langle \frac{\partial \vec{z}}{\partial p_1}\left(p_1, p_2, p_3\right), \frac{\partial \vec{z}}{\partial p_2}\left(p_1, p_2, p_3\right), \frac{\partial \vec{z}}{\partial p_3}\left(p_1, p_2, p_3\right) \right\rangle \right\|. \tag{7.26}$$

Prior adaptation is approximatively preserved by ensuring that the expected value of equation 7.25 is identical to equation 7.26. For photon directions sampled according to equations 7.15 and 7.16, this expected value is

$$\mathbb{E}\left[V'\left(p'_1, p'_2\right)\right] = \frac{4\pi}{n\sigma_e\left(0\right)} t_v^2\left(p_1, p_2, p_3\right).$$

The offset required to preserve adaptation follows as

$$\begin{aligned}
&\mathbb{E}\left[V'\left(p'_1, p'_2\right)\right] &&= V\left(p_1, p_2, p_3\right) \\
\Leftrightarrow\ &\frac{4\pi}{n\sigma_e\left(0\right)} t_v^2\left(p_1, p_2, p_3\right) &&= \frac{4\pi}{n} \left\| \left\langle \frac{\partial \vec{z}}{\partial p_1}\left(p_1, p_2, p_3\right), \frac{\partial \vec{z}}{\partial p_2}\left(p_1, p_2, p_3\right), \frac{\partial \vec{z}}{\partial p_3}\left(p_1, p_2, p_3\right) \right\rangle \right\| \\
\Rightarrow\ &t_v\left(p_1, p_2, p_3\right) &&= \sqrt{\sigma_e\left(0\right) \left\| \left\langle \frac{\partial \vec{z}}{\partial p_1}\left(p_1, p_2, p_3\right), \frac{\partial \vec{z}}{\partial p_2}\left(p_1, p_2, p_3\right), \frac{\partial \vec{z}}{\partial p_3}\left(p_1, p_2, p_3\right) \right\rangle \right\|}.
\end{aligned}$$

Virtual offsetting is handled analogously to diffuse reflection by computing $t_v\left(p_1, p_2, p_3\right)$ and then initializing the differentials after scattering according to equations 7.20 and 7.21.

7.2.3. Russian Roulette

An additional use of Russian roulette in volumetric photon mapping is the decision between absorption and scattering after extinction. The probability of a photon being scattered into a new direction is given by the medium albedo $\alpha\left(t_e\right)$ at the extinction point. Assuming that other photons are scattered with equal probability, the same argumentation as for Russian roulette at a surface in section 5.1.4 holds. Russian roulette then reduces the number of photons by a fraction $\alpha\left(t_e\right)$ without affecting their distribution. It is thus accounted for by increasing the lengths of the photon differentials with respect to the three parameters by factor $\alpha^{-\frac{1}{3}}\left(t_e\right)$ each upon scattering.

7.3. Bandwidth Selection

The beam radiance estimate from equation 7.3 uses variable kernel density estimation to reconstruct in-scattered radiance. This requires that a kernel support region be chosen for each extinction event.

7.3.1. Isotropic Kernel Support Region

Extinction is preceded by propagation to the extinction point. Following section 7.2.1.2, the photon thus has three parameters and a three-dimensional footprint. The footprint spanning vectors given by equations 7.4 to 7.6 adapt to the influence of scene surfaces and participating medium on the distances between neighboring extinction events. A kernel support region aligned with these is thus desirable. Due to the high computational cost of volumetric photon mapping, we propose that instead of an anisotropic kernel support as in section 5.2.1, a simple single bandwidth h_j be used. The bandwidth is computed so that the volume of the resulting spherical kernel support region around $\vec{z}_j = \vec{z}\left(p_1, p_2, p_3\right)$ follows that of the footprint from equation 7.26,

$$V\left(p_1, p_2, p_3\right) = \frac{4\pi}{n} \left\| \left\langle \frac{\partial \vec{z}}{\partial p_1}\left(p_1, p_2, p_3\right), \frac{\partial \vec{z}}{\partial p_2}\left(p_1, p_2, p_3\right), \frac{\partial \vec{z}}{\partial p_3}\left(p_1, p_2, p_3\right) \right\rangle \right\|. \tag{7.27}$$

The differential with respect to p_3 can be separated into its constant and variable components. We model the participating medium as made up of a single substance with only the density varying spatially in the inhomogeneous case. The extinction coefficient at any point then is the product of an upper bound σ_e and the local medium density, $\sigma_e\left(\vec{z}\right) = \rho\left(\vec{z}\right)\sigma_e$. Inserting equation 7.7 and this identity into equation 7.6, the differential becomes

$$\frac{\partial \vec{z}}{\partial p_3}\left(p_1, p_2, p_3\right) = \frac{1}{\sigma_e\left(t_e\left(p_3\right)\right)\tau\left(0, t_e\left(p_3\right)\right)}\vec{\omega}\left(p_1, p_2\right) = \frac{1}{\sigma_e}\frac{\vec{\omega}\left(p_1, p_2\right)}{\rho\left(t_e\left(p_3\right)\right)\tau\left(0, t_e\left(p_3\right)\right)}.$$

After inserting this into equation 7.27, all constants can be collected outside the triple product,

$$V\left(p_1, p_2, p_3\right) = \frac{4\pi}{n\sigma_e} \left\| \left\langle \frac{\partial \vec{z}}{\partial p_1}\left(p_1, p_2, p_3\right), \frac{\partial \vec{z}}{\partial p_2}\left(p_1, p_2, p_3\right), \frac{\vec{\omega}\left(p_1, p_2\right)}{\rho\left(t_e\left(p_3\right)\right)\tau\left(0, t_e\left(p_3\right)\right)} \right\rangle \right\|.$$

The bandwidth h_j is the cubic root of this volume, scaled by a manually chosen spread factor s,

$$h_j = s \sqrt[3]{\frac{4\pi}{n\sigma_e} \left\| \left\langle \frac{\partial \vec{z}}{\partial p_1}\left(p_1, p_2, p_3\right), \frac{\partial \vec{z}}{\partial p_2}\left(p_1, p_2, p_3\right), \frac{\vec{\omega}\left(p_1, p_2\right)}{\rho\left(t_e\left(p_3\right)\right)\tau\left(0, t_e\left(p_3\right)\right)} \right\rangle \right\|}. \tag{7.28}$$

7.3.2. Dampened Adaptation

Adaptation dampening is proposed in section 5.2.2 to approximatively account for the influence of illumination arriving via different paths on the local interaction density. The same considerations apply to the extinction event density in volumetric photon mapping. An efficient dampening method is provided by the *log1pf* function in CUDA, implementing the operation $x' = \log\left(1 + x\right)$. Scaling the result by a constant $d > 0$ and x by its inverse, the amount of dampening can be controlled. This dampening is applied to the variable component of equation 7.28, resulting in the kernel bandwidth

$$h'_j = s\, d \sqrt[3]{\frac{4\pi}{n\sigma_e}} \log\left(1 + \frac{1}{d}\sqrt[3]{\left\| \left\langle \frac{\partial \vec{z}}{\partial p_1}\left(p_1, p_2, p_3\right), \frac{\partial \vec{z}}{\partial p_1}\left(p_1, p_2, p_3\right), \frac{\vec{\omega}\left(p_1, p_2\right)}{\rho\left(t_e\left(p_3\right)\right)\tau\left(0, t_e\left(p_3\right)\right)} \right\rangle \right\|}\right).$$

7.3.3. Spectral Considerations

Reflectivity ρ at a surface interaction and albedo α at an extinction event are importance-sampled using Russian roulette. As described in section 3.4.1, the probability of onward tracing is an average across all color bands. This requires that the photon flux in the i-th color band be scaled by the ratio of the probability for this band and the average, $\rho_i \rho_{average}^{-1}$ respectively $\alpha_i \alpha_{average}^{-1}$.

When importance-sampling the extinction distance in section 7.2.1, we use the maximum of the extinction coefficients from all color bands, samples thus following the band for which the participating medium is densest. The probability that the photon reaches a surface in section 7.2.1.1 then is $e^{-\int_0^{t_{hit}(p_1,p_2)} \sigma_{e,i}(s)\,ds}$ for the i-th color band and $e^{-\int_0^{t_{hit}(p_1,p_2)} \sigma_{e,max}(s)\,ds}$ for the actual photon. Flux must again be scaled by their ratio. For a homogeneous medium, this follows directly as

$$e^{-\int_0^{t_{hit}(p_1,p_2)} \sigma_{e,i}(s) - \sigma_{e,max}(s)\,ds} = e^{(\sigma_{e,max} - \sigma_{e,i}) t_{hit}(p_1,p_2)}. \tag{7.29}$$

In the case of an inhomogeneous medium, ray marching is used analogously to equation 3.18. When only the density is spatially varying, the required ratio becomes

$$e^{-\int_0^{t_{hit}(p_1,p_2)} \sigma_{e,i}(s) - \sigma_{e,max}(s)\,ds} \approx e^{-\sum_{s=1}^{s_{max}} \Delta t_s \rho(t_s)(\sigma_{e,i} - \sigma_{e,max})} = e^{(\sigma_{e,max} - \sigma_{e,i}) \sum_{s=1}^{s_{max}} \Delta t_s \rho(t_s)}. \tag{7.30}$$

For a photon propagated until an extinction event in section 7.2.1.2, the distance t_e (p_3) is sampled according to the probability density function $p(t_e) = \sigma_{e,max} e^{-\int_0^t \sigma_{e,max}(s)\,ds}$. The corresponding function for the i-th color band is $p(t_e) = \sigma_{e,i} e^{-\int_0^t \sigma_{e,i}(s)\,ds}$. The ratio of these expressions is identical to equations 7.29 and 7.30 with only an additional factor $\sigma_{e,i} \sigma_{e,max}^{-1}$.

7.3.4. Results and Discussion

We use nine combinations of scene and participating medium as benchmark scenarios. Ring, Sponza Atrium and Sibenik Cathedral are chosen as a representative subset of the scenes from section A.2. The Ring is smallest and simplest. In the variant employed, the ring is lifted off the ground plane, allowing a *volume caustic* to form. The level of detail is slightly increased, yielding 522 triangles. Virtual camera and light source position are adjusted to visualize the caustic. The Sponza Atrium represents moderate size and complexity. Demonstrating the possibility of a bounded participating medium, this scene is only partially filled and the virtual camera position adjusted to better visualize the medium. The Sibenik Cathedral serves as an example of a large scene.

Three scenarios are evaluated per scene. As shown in table 7.1, these illustrate the influence of different parameters. For the Ring, the light source is varied from an isotropic point light to a spot with apex angle $\alpha = 0.5$ first. Participating medium properties are then adjusted, changing from neutral to colored by reducing the scattering in the green and blue color bands. The participating medium in the Sponza Atrium exhibits lower albedo. Absorption in the red color band is reduced to color it first and a spatially varying density then encoded in a three-dimensional texture map to add inhomogeneity. The absorption and scattering coefficients listed are averages, showing a match with the previous homogeneous medium. Anisotropy is demonstrated in the Sibenik Cathedral, varying from $k = 0.5$ for forward scattering to $k = -0.5$ for backward scattering.

All sampling decisions are controlled by a quasi-random Sobol' sequence, minimizing variance. The number of photons emitted is 2^{20} for the first two scenes and 2^{18} for the last due to higher computational cost experienced with it. In-scattered radiance is reconstructed by the beam radiance

Scene	Omni-directional	σ_a			σ_s			Homo-geneous	k
		R	G	B	R	G	B		
Ring	Yes	0.01	0.01	0.01	0.50	0.50	0.50	Yes	0.0
	No	0.01	0.01	0.01	0.50	0.50	0.50	Yes	0.0
	No	0.01	0.01	0.01	0.50	0.01	0.01	Yes	0.0
Sponza	Yes	0.30	0.30	0.30	0.10	0.10	0.10	Yes	0.0
	Yes	0.05	0.30	0.30	0.10	0.10	0.10	Yes	0.0
	Yes	0.05	0.30	0.30	0.10	0.10	0.10	No	0.0
Sibenik	Yes	0.05	0.05	0.05	0.15	0.15	0.15	Yes	0.5
	Yes	0.05	0.05	0.05	0.15	0.15	0.15	Yes	0.0
	Yes	0.05	0.05	0.05	0.15	0.15	0.15	Yes	−0.5

Table 7.1.: Light source type, absorption and scattering coefficients, homogeneity and anisotropy of the participating medium for nine benchmark scenarios

Scene	Bounce 1	Bounce 0, 1	Bounce 1, 2
Ring	2.5	2.5	1.5
	10.0	10.0	2.5
	10.0	10.0	2.5
Sponza	6.0	3.0	3.0
	6.0	3.0	3.0
	10.0	5.0	5.0
Sibenik	7.0	4.0	3.0
	5.0	3.0	2.5
	5.0	3.0	2.5

Table 7.2.: Adaptation dampening for different bounce depths and benchmark scenarios

estimate. Baseline kernel bandwidths are given by the distances from the extinction events to their 85th-nearest neighbors. Bandwidths based on photon differentials are computed with spread $s = 3$ and dampening factor d tuned so that image quality and bandwidth distribution are comparable to the baseline. The number of times a photon is reflected or scattered before an extinction event is measured by its bounce depth. Table 7.2 lists the dampening factors d obtained when events are stored at different bounce depths. A smaller d leads to stronger dampening.

Bounce depth 0 is single-scattered radiance. If only this is simulated, all in-scattered illumination arrives from the light source directly and no adaptation dampening is necessary. As extinction events for more and higher bounce depths are stored, the probability of different paths reaching the same region increases and the amount of dampening required to match the baseline grows. For the Ring, a spot light leads to less overlap between different paths and thus less dampening. The Sponza Atrium is larger and has a participating medium with lower albedo. Fewer paths thus overlap and less dampening is needed under the same illumination conditions. This effect is even more apparent with an inhomogeneous medium. The Sibenik Cathedral is larger but has higher albedo, leading to a similar amount of dampening. When the medium is strongly forward scattering, different paths are less likely to overlap, thus needing less dampening than with isotropic or backward scattering.

All further evaluation is based on bounce depths 0 and 1, simulating single and multiple scattering. Photon tracing and density estimation statistics are provided in table 7.3. Extinction events are identical for the baseline and photon differentials. The maximal bandwidth h_{max} is set for each scene so that only excessive bandwidths are removed. On average, 0.10% of the extinction events are

Scene	Photon Tracing					Density Estimation	
	Emissions	Extinctions	h_{max}	Clamped$_B$	Clamped$_{PD}$	k_B	k_{PD}
Ring	1048576	816605	1	0.00%	0.00%	766.87	+37.95%
	1048576	894416	1	0.00%	0.00%	645.28	+18.56%
	1048576	818157	1	0.00%	0.00%	599.85	+11.59%
Sponza	1048576	255811	2	0.11%	0.00%	1124.68	−15.21%
	1048576	269959	2	0.11%	0.00%	1145.64	−12.23%
	1048576	258533	2	0.12%	0.02%	728.14	+22.30%
Sibenik	262144	355852	3	0.19%	0.00%	1298.92	+3.97%
	262144	362540	3	0.18%	0.00%	1290.03	−7.75%
	262144	369746	3	0.18%	0.00%	1287.44	−3.51%
Average				0.10%	0.00%		+6.19%
Std.Dev.				0.08%	0.01%		17.82%

Table 7.3.: Photon tracing and density estimation statistics with kernel bandwidths obtained by the baseline (B) and photon differentials (PD) methods

affected by clamping in the baseline. With photon differentials, clamping occurs in a single benchmark scenario only, affecting 0.02% of the events.

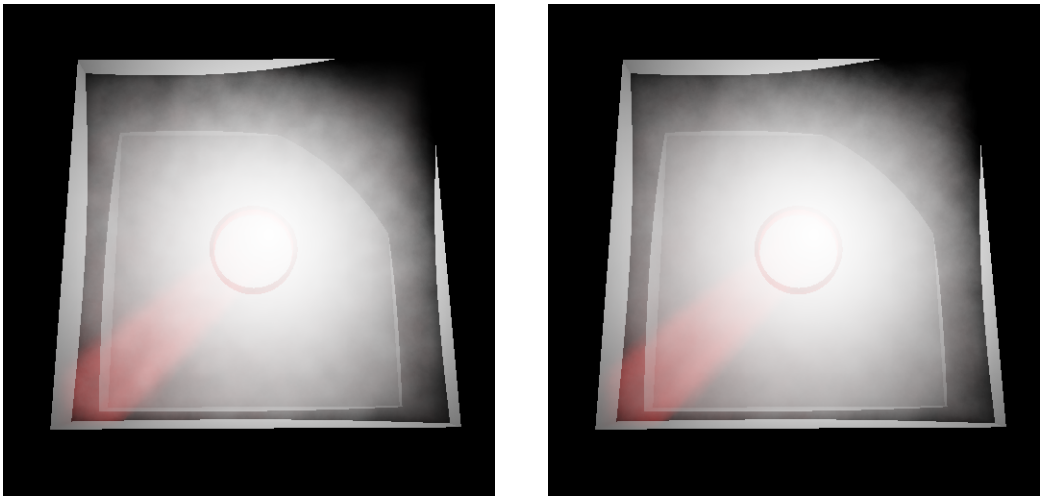
Images are rendered using the benchmark environment from section A.1 with up to four rays traced per pixel. Radiance is attenuated by the medium transmittance for each ray and the radiance in-scattered to primary and reflection rays added by the beam radiance estimate. No surface interactions are stored, focusing on volumetric photon mapping only. A simple tone mapping operator allows a higher dynamic range to be reproduced by transforming radiance L reaching the virtual camera to $L' = \frac{L}{L+1W\,m^{-2}\,sr^{-1}}$. Figure 7.1 presents the rendering results for an example view of each benchmark scenario. As shown in table 7.3, the number of extinction events contributing per ray during a flight through a scenario is similar to the baseline, differing from it by 6.19% on average.

A volume caustic arises in figure 7.1(a) and (b) due to illumination focusing by the red specular ring. The same caustic is present in figure 7.1(c) albeit less prominent due to the red color of the participating medium. Caustic sharpness is improved over the baseline. For the Sponza Atrium, a reduction in variance relative to the baseline is observed in figure 7.1(d)–(f). The same is true of the Sibenik Cathedral with overall variance increasing as scattering shifts from primarily forward in figure 7.1(g) to primarily backward in figure 7.1(i).

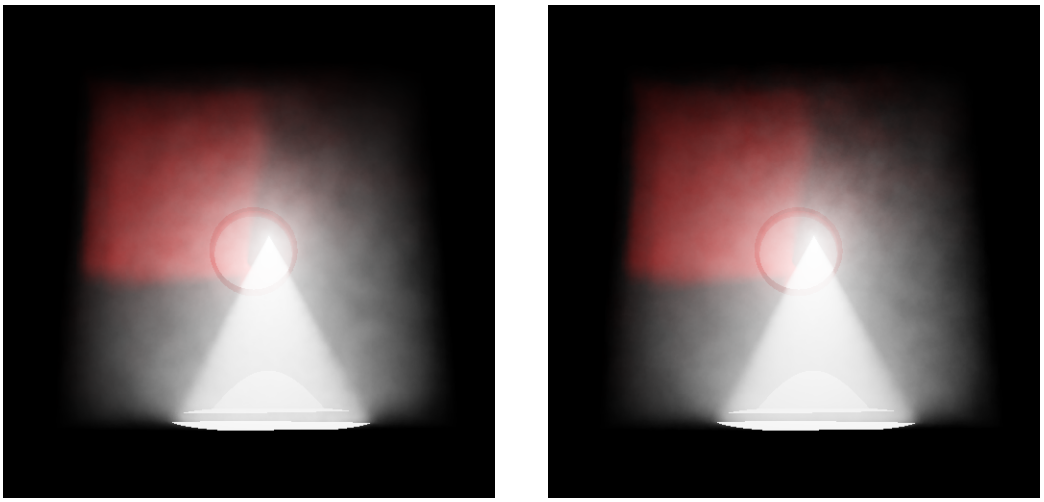
Work concurrent to ours [Sch09] also extends photon differentials to volumetric photon mapping. The rendering of a volume caustic with anisotropic kernel support regions is demonstrated. However, the influences of extinction and illumination arriving via different paths are not accounted for. Since no analysis of kernel bandwidths or performance is provided, a direct comparison is not possible.

7.4. Stream Processing

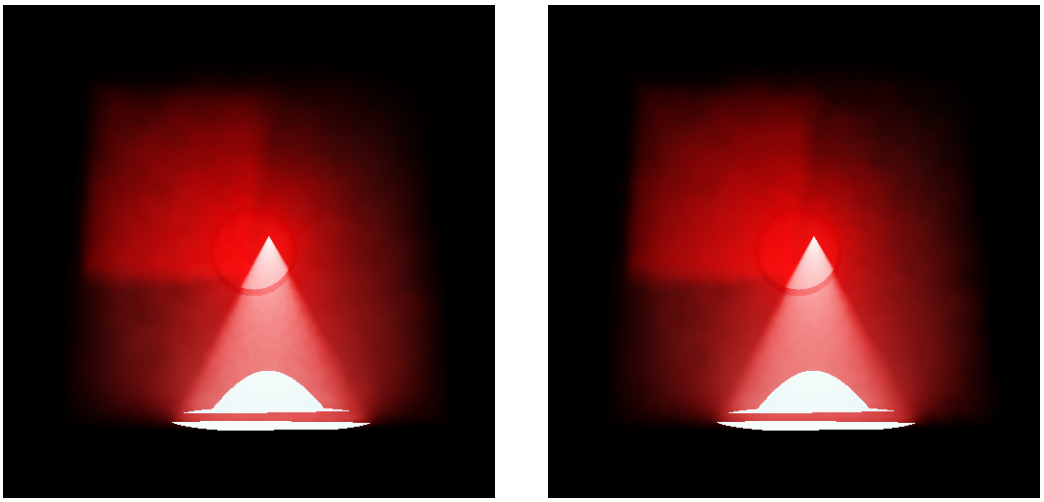
The CUDA programming model exposes a GPU as a general purpose stream processing device. Each kernel transforms an input data stream into one or more output streams. Other kernels may be applied to these output streams, chaining their execution. The data in an input stream thus corresponds to *jobs* processed by the threads of a kernel. This execution model is made explicit with persistent threads [AL09], reading a block of jobs from a queue for each warp, processing these and reading the next block until the queue is exhausted.



(a) Ring, omnidirectional light, neutral participating medium

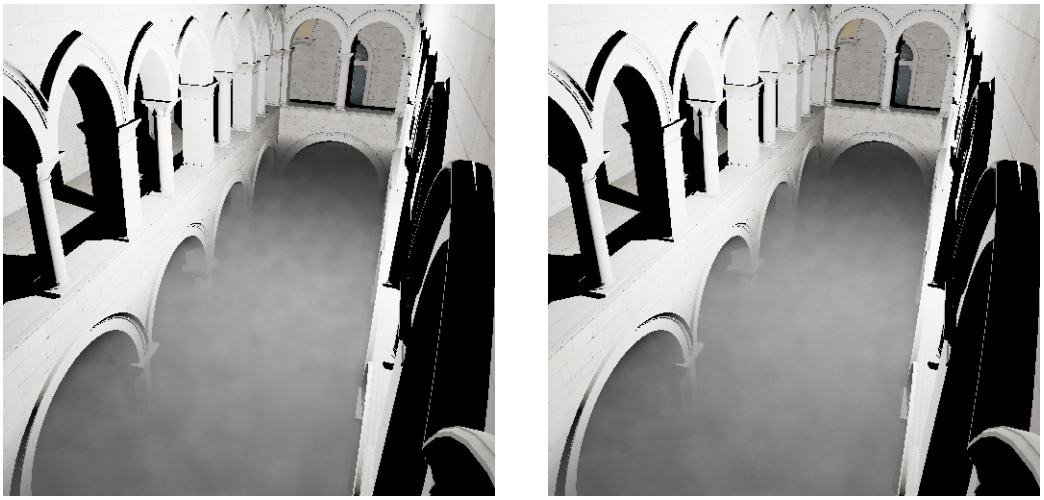


(b) Ring, spot light, neutral participating medium

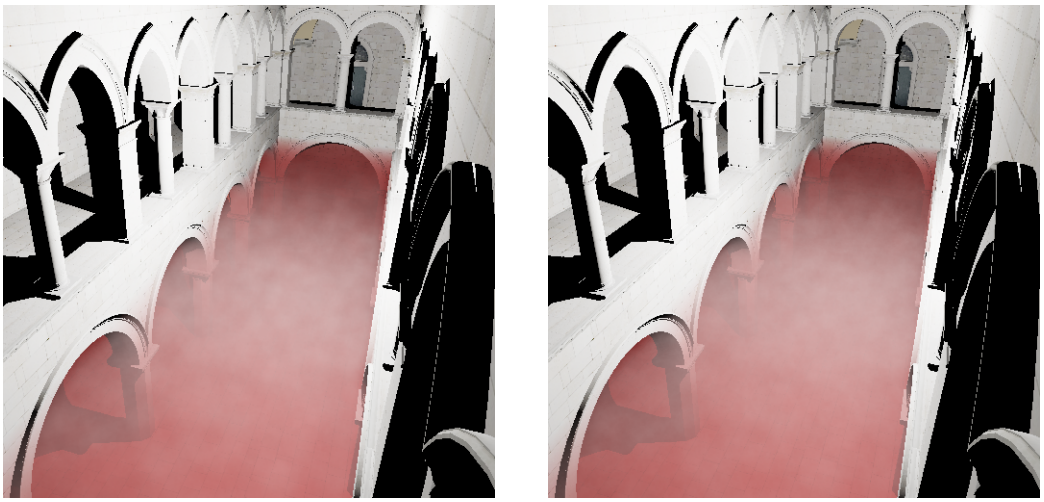


(c) Ring, spot light, colored participating medium

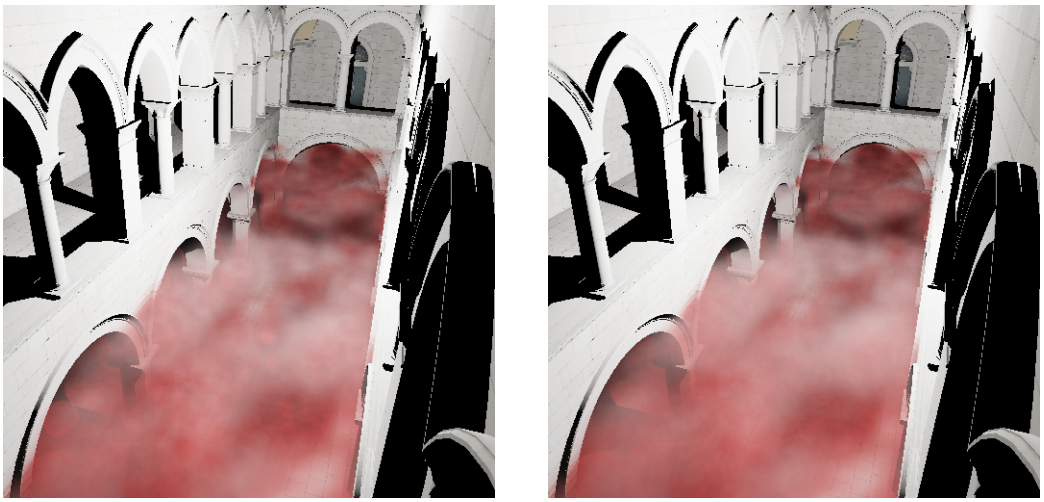
Figure 7.1.: Images rendered by volumetric photon mapping: Kernel bandwidths derived from k-th nearest neighbors on the left, from photon differentials on the right.



(d) Sponza Atrium, homogeneous, neutral participating medium

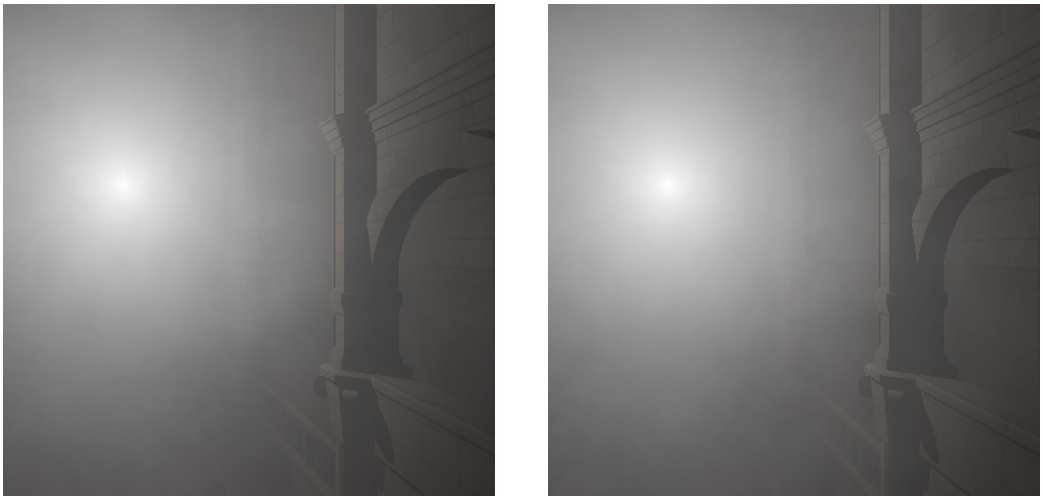


(e) Sponza Atrium, homogeneous, colored participating medium

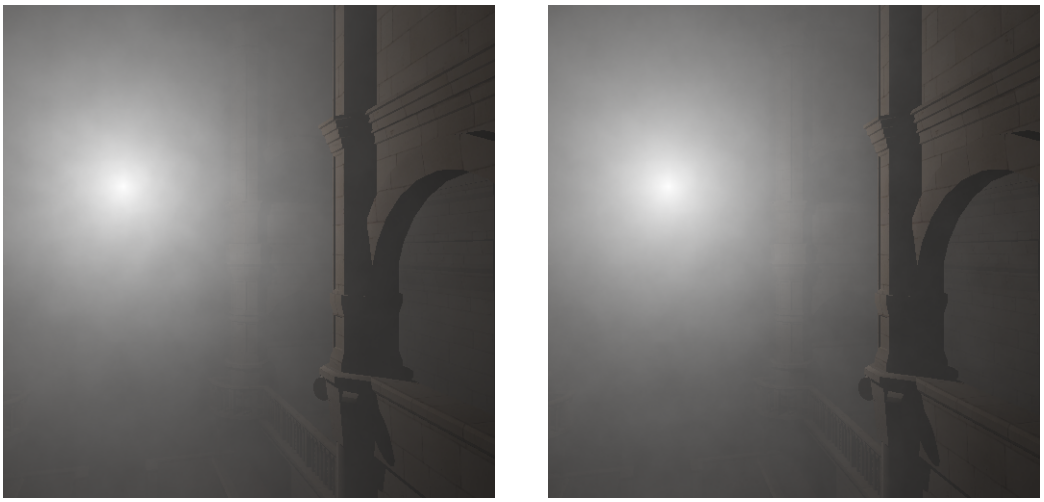


(f) Sponza Atrium, inhomogeneous, colored participating medium

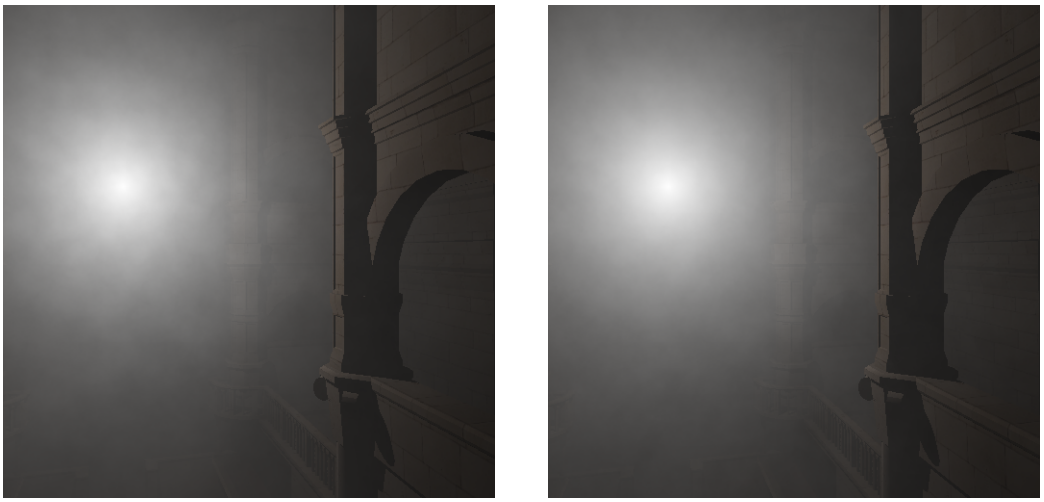
Figure 7.1.: Images rendered by volumetric photon mapping: Kernel bandwidths derived from k-th nearest neighbors on the left, from photon differentials on the right.



(g) Sibenik Cathedral, backward scattering participating medium



(h) Sibenik Cathedral, isotropically scattering participating medium



(i) Sibenik Cathedral, forward scattering participating medium

Figure 7.1.: Images rendered by volumetric photon mapping: Kernel bandwidths derived from k-th nearest neighbors on the left, from photon differentials on the right.

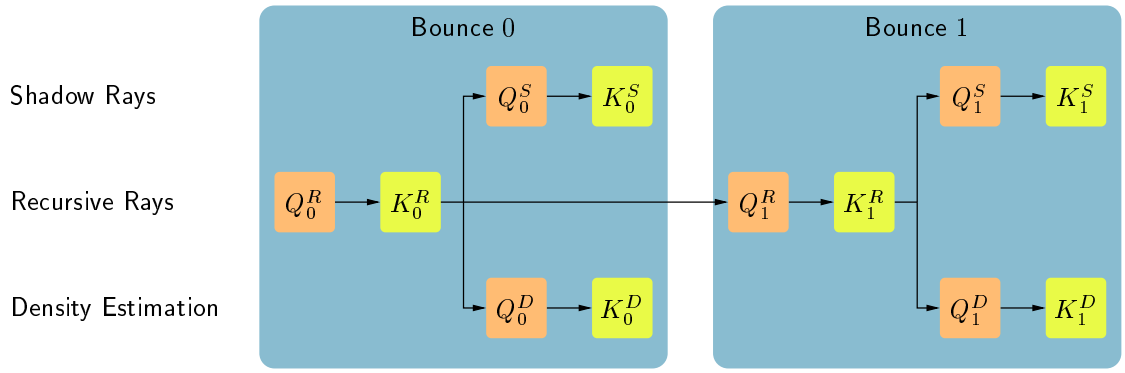


Figure 7.2.: Rendering by stream processing: Kernels (K, yellow) processes jobs from corresponding queues (Q, orange). Recursive rays (R), shadow rays (S) an density estimation (D) are handled by separate kernels.

Benchmark code base II from section A.1.2 decomposes rendering into six CUDA kernels. Their arrangement and that of the corresponding job queues is illustrated in figure 7.2. Kernel $K_{R,0}$ traces primary rays from job queue $Q_{R,0}$. For each ray passing through the participating medium, a density estimation job is output into queue $Q_{D,0}$. If a surface is hit, a shadow ray job is placed in $Q_{S,0}$ and if the surface has a specular BRDF component, additionally a reflection ray job in $Q_{R,1}$. The queues are processed by the corresponding kernels $K_{D,0}$, $K_{S,0}$, $K_{R,1}$. The last of these traces reflection rays, outputting additional density estimation and shadow ray jobs into queues $Q_{D,1}$, $Q_{S,1}$ to be processed by kernels $K_{D,1}$, $K_{S,1}$. A job is specified by its ray parameters, attenuation accumulated along the path and the image pixel it refers to. Radiance contributions by all kernels are accumulated in each image pixel and then tone-mapped by a simple postprocessing kernel, generating an image.

This decomposition incurs the overheads of multiple kernel launches and job queue management. Its advantages are smaller kernels with lower register counts and the possibility to vary the arrangement of jobs and computations between these.

7.4.1. Job Queuing

One primary ray is traced for each of the $512 \times 512 = 262144$ image pixels. No explicit job queue is required for these. The first thread in a warp uses an atomic instruction to increment a global counter, assigning a block of 32 consecutive primary ray indexes to the threads of the warp. Rays are traced and the next block is assigned to the threads until the counter reaches 262144. All other queues are explicitly populated by kernels $K_{R,0}$, $K_{R,1}$, placing from 0 to 262144 jobs in each.

7.4.1.1. Queue Type I

The simplest queue type uses a one-to-one mapping between jobs and primary rays. Jobs due to the primary ray with index i are output at position i in other queues as well. Where no job exists, a flag marking the queue position as empty is set instead. This queue type has the least management overhead. To process jobs, blocks of consecutive indexes are assigned to threads as during primary ray tracing. Where no job exists, the thread is masked off as inactive. High parallel processing unit utilization thus cannot be guaranteed.

7.4.1.2. Queue Type II

A second queue type improves processing unit utilization. Job queues are populated identically but then preprocessed by stream compaction to extract existent jobs only. The downsides of this approach are a compaction overhead and a potential loss in coherence as jobs corresponding to pixels from more distant parts of the image are assigned to the threads of a warp.

7.4.1.3. Queue Type III

The third queue type eliminates the compaction overhead. Each warp maintains a local queue with capacity for 7×32 jobs. Threads output jobs at consecutive positions in this queue, using an efficient shared memory atomic to increment a local counter. The first thread in a warp flushes its local to a global queue when less than 32 available positions remain. The global job queue thus consists of large compacted blocks. When processing jobs, the first thread of a warp increments a global counter to obtain an entire block. This is handled by the warp looping, processing 32 jobs at a time. Only for the last iteration may not all threads be active, requiring a subset to be masked off. Blocks are processed until the queue is exhausted.

7.4.2. Packetization

Computations may be arranged in each kernel either so that threads process jobs independently or act as a packet. Ray tracing, as noted in section 3.2.4.4, performs best if rays are traced independently. We show that the opposite is true for the beam radiance estimate. Packetization is achieved by a single *master* thread per warp making BVH photon map traversal decisions and maintaining a traversal stack in shared memory. The child nodes that must be visited for at least one of the threads in the packet are efficiently determined by warp voting.

For queue types II and III, the first thread in a warp is always active and acts as the master. With queue type I, an unpredictable subset of the threads is masked off. A master is therefore elected. All active threads concurrently write their IDs to the same shared memory location. The thread whose ID persists after the write becomes the master.

7.4.3. Results and Discussion

We use flights through the nine benchmark scenarios from section 7.3.4 for evaluation. A BVH photon map is constructed over the extinction events by the LBVH algorithm from section 6.1.2. Since the compact BVH representation of section 6.2 does not provide a significant performance benefit when queried by a ray, a complete BVH representation with sibling nodes stored in pairs is used instead. A minimal leaf size threshold of 16 is empirically determined for all scenarios.

The choice of job queue type and that between independent threads or packets lead to different memory accesses and execution patterns. Table 7.4 illustrates resulting computational cost, measured as the average number of instructions required by kernels K_0^D, K_1^D to perform density estimation per ray. Queue type I with independent threads serves as the baseline. The other queue types add 2.9% and 2.0% instructions to the baseline on average. This indicates that any improvements due to higher utilization of the parallel processing units are negated by lower coherence.

In CUDA, threads executing the same instruction automatically fall into lock-step. When part of a warp traverses photon map nodes and another visits extinction events, however, the threads

Scene	Extinctions	Independent			Packet		
		Queue I	Queue II	Queue III	Queue I	Queue II	Queue III
Ring	816605	437786	+8.6%	+5.7%	−62.7%	−55.0%	−60.0%
	894416	313319	+5.9%	+4.0%	−61.7%	−55.1%	−58.8%
	818157	278936	+4.9%	+3.1%	−61.7%	−54.8%	−59.0%
Sponza	255811	419273	+2.3%	+1.3%	−63.2%	−60.4%	−61.7%
	269959	440520	+2.3%	+1.6%	−63.6%	−60.7%	−62.4%
	258533	504986	+2.2%	+0.9%	−66.3%	−63.8%	−65.0%
Sibenik	355852	595010	−0.3%	+0.5%	−64.1%	−64.0%	−63.8%
	362540	419524	+0.5%	+0.3%	−60.8%	−60.5%	−60.6%
	369746	503910	−0.4%	+0.3%	−64.3%	−64.2%	−64.1%
Average			+2.9%	+2.0%	−63.1%	−59.8%	−61.7%
Std. Dev.			3.0%	1.9%	1.7%	4.0%	2.3%

Table 7.4.: Density estimation instructions per ray with different job queue types and packetization

follow two different code paths that must be serialized. This increases the total instruction count. Packetization ensures lock-step operation by traversing each node and visiting each extinction event relevant to any of the threads in a warp with the entire warp in parallel. This proves to significantly lower the instruction count for density estimation. The least number of instructions is needed with queue type I and packetization, lying 63.1% below the baseline on average.

Traversal stacks are located in local memory for independent threads and in shared memory for packets. Any other changes in memory access patterns are difficult to measure. With packetization, an entire warp accesses the same node or extinction event. For independent threads, accesses are not coordinated but are served via the texturing unit caches. Table 7.5 shows the total time required for density estimation per frame. Except for queue type III, the differences between the columns follow the differences in instruction count. Density estimation is thus compute bound, making memory accesses a secondary influence whose detailed analysis is not essential.

Queue type I performs best. With queue type II, a slight slowdown results. Queue type III leads to a larger slowdown. Since the instruction count is not higher, this slowdown is due to a more complex change in the execution pattern or a shift in memory accesses. Packetization, in line with reducing instruction counts, significantly improves timings. The highest performance is achieved for all benchmark scenarios with queue type I and packetization.

7.5. Combined Results

The techniques introduced in this chapter extend the efficient photon mapping algorithm for the CUDA manycore platform from section 6.3 to volumetric photon mapping. Table 7.6 shows that interactive frame rates are achieved for the benchmark scenarios from section 7.3.4 with dynamically changing illumination and all photons retraced in each frame. Results are given as averages over a flight through each scene with an animated light source. Frame rates are measured for the complete algorithm. The overhead of copying images to the screen is included in the ray tracing time. Photon tracing and density estimation are the most expensive components for all scenes. Despite a significantly higher number of contributions per query than in section 6.3, density estimation performance is comparable due to decomposition into multiple kernels and the use of packetization.

Scene	Extinctions	Independent (ms)			Packet (ms)		
		Queue I	Queue II	Queue III	Queue I	Queue II	Queue III
Ring	816605	1055.2	+3.1%	+24.1%	−74.7%	−71.3%	−67.4%
	894416	704.0	+4.2%	+23.0%	−74.5%	−68.7%	−66.7%
	818157	599.3	+4.1%	+24.5%	−72.8%	−67.0%	−64.7%
Sponza	255811	321.1	+2.0%	+34.7%	−68.4%	−65.8%	−61.3%
	269959	336.0	+1.9%	+36.6%	−68.6%	−66.0%	−61.8%
	258533	499.5	+2.7%	+38.2%	−77.1%	−75.5%	−71.5%
Sibenik	355852	1156.0	+0.5%	+22.6%	−70.9%	−70.3%	−69.9%
	362540	817.7	+0.3%	+19.6%	−68.0%	−67.5%	−67.1%
	369746	1009.9	+0.3%	+21.7%	−71.9%	−71.4%	−71.1%
Average			+2.1%	+27.2%	−74.7%	−69.3%	−66.8%
Std. Dev.			1.5%	7.1%	3.2%	3.2%	3.7%

Table 7.5.: Density estimation times per frame with different job queue types and packetization

Scene	Extinctions	k	Time (ms)				FPS
			Trace	LBVH	RT	DE	
Ring	892776	1156.3	49.83	20.37	10.44	263.25	2.91
	1012995	903.5	49.60	23.04	9.96	203.52	3.50
	922823	791.6	46.89	21.17	9.96	182.69	3.84
Sponza	310189	1091.1	109.03	8.27	11.74	119.31	4.03
	327471	1152.6	110.22	8.67	11.74	123.93	3.93
	311908	1010.3	169.94	8.23	12.12	132.55	3.10
Sibenik	351797	1305.7	26.31	9.76	11.96	322.11	2.70
	358297	1151.8	26.25	9.63	11.94	249.37	3.36
	365337	1194.7	26.50	10.64	11.89	269.88	3.14

Table 7.6.: Photon extinction events, contributions per query, times per frame for photon tracing, LBVH photon map construction, ray tracing (RT), density estimation (DE) and frame rates with dynamic illumination

8. Conclusions and Future Work

This final chapter of the thesis provides a summary discussion of its findings and an outlook at directions for future research.

8.1. Conclusions

Photon mapping is a state of the art algorithm for physically based global illumination rendering. It simulates the full range $L(S|D)^*E$ of paths from light source to virtual camera in a conceptually simple and elegant manner. However, fast and efficient operation is difficult to achieve. Our investigation provides efficient techniques for each component of the algorithm. These combine to enable photon mapping with dynamically changing illumination at interactive frame rates on current consumer hardware. The extension to volumetric photon mapping for participating media is also addressed and interactive frame rates are once again achieved.

The presentation of our work is preceded by an introduction of the required background knowledge and a review of related work in the field. Rendering algorithms targeting plausible, visually pleasing images need not be concerned with the details of light transport. Physically based rendering on the other hand aims to follow physical principles as accurately as possible under the given memory and time constraints. Chapter 2 therefore provides a description of the physics behind light transport. Physical terms and quantities are used consistently throughout this thesis, ensuring that the link between the rendering algorithm and the principles of light transport is maintained.

Manycore computing is also introduced in the chapter. As this is an emerging trend in computer architecture, new hardware designs based on this paradigm are constantly being developed while those in existence evolve further. The choice of CUDA is motivated by the fact that it is the first manycore platform to reach wide availability as it is implemented on inexpensive commodity NVIDIA GPU hardware. Details specific to CUDA are described but the manycore paradigm is general and all algorithmic building blocks presented are universal so that parallel algorithms constructed from these are widely applicable to emerging and future platforms.

A systematic investigation of the three major components making up the photon mapping algorithm (ray and photon tracing, density estimation, photon map construction and traversal) in chapters 4 to 6 leads to techniques for all of these that ensure efficient operation. The results combine into an efficient rendering algorithm that operates at interactive frame rates. Our findings are also useful on their own, however, providing improvements to generic components employed by other rendering algorithms as well. Insights into opaque aspects of the CUDA platform are additionally gained.

SIROH is introduced in chapter 4. This heuristic yields spatial indexes for ray tracing acceleration with higher traversal performance than the current state of the art, the SAH. Rather than modifying the existing heuristic, we change the underlying assumptions and derive a new heuristic from these. The speedup observed validates this approach, showing that although the SAH produces high quality spatial indexes, revisiting the fundamental assumptions it is based on can lead to even better results.

The investigation of stackless traversal in the same chapter illustrates the development of GPU hardware and programming models. Stack-based traversal is not possible on older GPUs accessed via graphics APIs due to lack of suitable read-write memory. With a current GPU and CUDA, not only is stack-based traversal enabled but also high performance achieved. Our results for explicitly managed caches lead to the important observation that when optimizing memory accesses, CUDA kernels may become compute bound and ultimately less efficient. This is an unexpected finding as the CUDA documentation and optimization guidelines are highly focused on memory accesses and specific expensive operations only. The result of this investigation thus provides a valuable insight for algorithm design on the CUDA platform in general.

Chapter 5 focuses on density estimation. We show that variable kernel density estimation is a viable alternative to the k-th nearest neighbor density estimation originally used in photon mapping. An extension of the photon differentials framework from emission, propagation and specular interactions to diffuse reflections and Russian roulette proves successful in choosing anisotropic kernel support regions adaptive to the local photon interaction density. Handling diffuse reflection as an absorption and a reemission yields constant storage requirements for the differentials during photon tracing.

Separate adaptation results for illumination arriving via different paths. This can be detrimental or beneficial. When the illumination from different directions is similar, an adaptation to the joint interaction density would yield smaller kernel support regions with less bias and higher performance. However, the required information about the global distribution of photon paths can fundamentally never be derived from what is tracked with an individual photon. The adaptation dampening we propose reduces excessive bandwidths but is only a heuristic approximation. It nevertheless leads to image quality on par with k-th nearest neighbor density estimation.

When the illumination from different directions does vary, separate adaptation is highly desirable. A sharp caustic on a diffusely illuminated surface is reproduced without the need to explicitly divide photons into caustic and global. This example also illustrates a limitation of the technique. Kernel support regions are adaptive to the illumination conditions at the interaction positions only. If the illumination undergoes complex changes, the support regions may not follow it accurately. Streaking artifacts result for a curved caustic as kernels align with its tangent. More flexible kernel support regions than the skewed ellipsoids used could address the issue but would require more storage space and complex computations.

Photon map construction, storage and traversal are investigated in chapter 6. Replacing the kd-tree originally used with a BVH, variable kernel density estimation is simplified as no photon interaction can be retrieved more than once. The use of highly parallel LBVH construction is also enabled. As initially published, this algorithm is not fully described. Additional details recently provided allow it to be largely reconstructed and show its inefficiencies. We derive alternatives to the inefficient steps. Although our focus is on the construction of a photon map, our improvements to the algorithm are general and applicable anywhere LBVH construction is used. The novel compact BVH representation proposed in this chapter can similarly be applied to a BVH in any context. However, while storage and bandwidth requirements are always reduced, performance may be lower, depending on application. This is in line with our observation that CUDA kernels may become compute bound when optimizing memory accesses and experience a slowdown due to additional instructions required.

An evaluation of the complete photon mapping algorithm at the end of chapter 6 demonstrates interactive performance for a range of benchmark scenes with dynamically changing illumination. The breakdown into individual components identifies density estimation as the main bottleneck that

dominates timings. This motivates an investigation of further improvements to it in chapter 7 when extending the algorithm to volumetric photon mapping. A rederivation of the beam radiance estimate provides the basis for efficient variable kernel density estimation. Kernel bandwidths are computed by a further extension of the photon differentials framework to extinction and scattering. Our choice of isotropic kernel support regions is owed to the higher number of contributions per density estimation, countering the increased computational cost.

We find that density estimation is significantly accelerated by decomposing rendering into multiple CUDA kernels and using packetization for this step. This result again demonstrates the opaqueness of the CUDA platform and the need for benchmarking to find the approach that performs best. The complete algorithm achieves interactive frame rates for a range of benchmark scenarios. Image quality is comparable to that of the original beam radiance estimate method with an offline preprocessing step in which bandwidths are determined by k-th nearest neighbor searches.

In summary, we show that by addressing the efficiency of its constituent components, photon mapping at interactive frame rates is possible on the CUDA manycore platform. As CUDA is only one early example of the emerging trend toward manycore computing, our techniques can be expected to provide benefits on future computer architectures as well. Where results do not transfer directly, a starting point for further investigation is given.

8.2. Future Work

Our work is based on an NVIDIA GTX 280 GPU. The next generation of NVIDIA hardware with higher computational power and more memory is now reaching wide availability. Without changes to the algorithm, higher frame rates can be expected. Increasing the number of photons emitted would exploit the additional resources to improve image quality instead. Further gains are likely possible by adjusting the algorithm to changed performance characteristics. Due to the lack of detailed documentation, a reevaluation of the choices made during algorithm design by benchmarking is required to identify the adjustments necessary for efficient operation.

With the emergence of manycore computing as a general trend in computer architecture, competing platforms offering similar performance to CUDA are becoming available. OpenCL, a vendor neutral API, is especially promising as it introduces a wider standardization. Our work provides a starting point for an investigation of interactive photon mapping on this and any other manycore platform. Benchmarking can again identify which choices made for the CUDA platform transfer directly and where new algorithms are required. Should a sufficient increase in computational power become available, the addition of final gather would significantly enhance image quality, bringing the algorithm in line with production quality offline rendering.

Photon differentials adapt kernel bandwidths to the interaction densities of photons arriving via different paths separately. k-th nearest neighbor density estimation uses the joint distribution of all paths. A hybrid of both techniques would further enhance image quality by considering the paths contributing similar illumination. Photon differentials themselves can be further improved. We reduce storage requirements and computational costs by approximating several influences on the distribution of photon paths. If more resources are available, more accurate solutions should be investigated. The scene representation we use is limited to a single point light source, diffuse and specular surfaces. Adding support for more complex light sources and BRDFs by extending photon differentials to these is another interesting direction for future work.

An improvement orthogonal to the light transport simulation is support for dynamically changing scene geometry. Techniques for efficiently reconstructing the spatial index over the scene surfaces per frame exist. By combining these with the rendering algorithm from this thesis, a solution can be derived that allows every aspect of the scene to change interactively. This has applications from games through artistic use to architectural design.

Our work on SIROH shows that ray tracing, a constituent component of photon mapping but also of many other physically based rendering algorithms, can be accelerated by revisiting established best practices and modifying the underlying assumptions. SIROH can likely be outperformed by further improving the assumptions made during spatial index construction and deriving heuristics that more accurately model expected ray tracing cost.

A. Benchmarks

All algorithms are evaluated on the NVIDIA CUDA manycore platform.

A.1. Benchmark Environment

The CUDA device used is an NVIDIA GTX 280. Summarizing from section 2.2.3, this GPU has 240 parallel processing units grouped into 30 SMs that execute warps of threads in 32-wide SIMT. 1 GB global memory with 141.7 GB/s bandwidth and 16 kB fast shared memory per SM are available. Taking into account all processing units, including the SFUs, DFUs and address calculations by texturing units, theoretical peak performance is 933 GFLOP/s.

Scene and spatial index are preloaded into global memory by a host computer. This offline step is implemented in single threaded C++ code for ease of maintenance. The host runs Arch Linux on an Intel Pentium D 965 3.73 GHz initially and on an Intel Core2 Quad Q9450 2.66 GHz after hardware failure. Triangle meshes are read either from a Wavefront OBJ file [Ali95b] with MTL files [Ali95a] specifying surface properties or as raw lists of triangles and properties. The OBJ and MTL formats are popular and simple but lack physical units. Scaling factors are therefore separately specified for each scene that ensure physically plausible surface sizes and BRDF parameters.

Algorithms are evaluated within a physically based renderer executing on the device. As the host is responsible for launching kernels and initiating memory transfers only, its performance is not critical and single threaded C++ code is used again. Synthesized images are visualized on the same GPU by transferring them to an OpenGL texture map via a shared buffer and rasterizing a textured quad.

Images are rendered at 512×512 resolution, for each pixel tracing a ray emitted by the virtual camera, a reflection ray if the surface seen is specular and shadow rays from a point light source toward the hit points of both. Photon mapping adds a photon tracing pass, photon map construction on the device and density estimation at every diffuse surface seen during image synthesis. Volumetric photon mapping performs density estimation for each viewing ray.

A.1.1. Code Base I

A thread is spawned for each ray and photon emitted, relying on the GPU scheduler for successive execution. Photon tracing is iterative with one kernel invoked per recursion depth. Ray tracing from the virtual camera and density estimation occur in a single kernel. The parallel algorithm building blocks described in section 2.2.4 are manually implemented whenever required.

A.1.2. Code Base II

Several performance improvements are realized in a reimplementation of the original benchmark environment. Applying the concept of persistent threads [AL09], photons and rays are traced by spawning only as many threads as can be resident at one time and processing jobs from a queue. Ray

tracing from the virtual camera and density estimation are decomposed into separate kernels, enabling packetization for density estimation as proposed in section 7.4. Algorithmic building blocks are replaced with optimized implementations provided by the CUDPP [HOS+10] and chag::pp [BOA10] libraries. Since scenes and hardware are unchanged, all performance gains are due to more efficient use of the manycore architecture.

A.2. Benchmark Scenes

Five scenes are commonly used for evaluation, complemented by two additional scenes for photon mapping. Figures A.1 and A.2 provide ray traced example views. Performance for different parts of a scene is assessed by averaging results over a flight through it. Starting with the example view on the right, the virtual camera advances by 10 cm and turns right by 0.25° after each frame. The light source optionally moves 3 cm along each coordinate axis. 50 frames are used for the common scenes and 10 frames for the smaller photon mapping scenes.

A.2.1. Common Scenes

The scenes are selected to cover a wide range of rendering scenarios. Their properties are summarized in table A.1. Scene size affects the number of photons required to simulate indirect illumination. The number of triangles is a measure of surface complexity, affecting ray tracing cost. Texture maps occupy global memory, reducing the amount available to the rendering algorithm. As CUDA requires the number of texture maps to be specified at compile time, all are stacked into a single three-dimensional texture map. The scenes are:

Scene 6 Courtesy of Peter Shirley. This simple scene serves to assess correct scene traversal and maximal frame rates. One wall is a specular mirror.

Sponza Atrium Courtesy of Marko Dabrovic. A classic benchmark for physically based rendering, the scene has moderate complexity with texture maps approximating further details, typical of interactive scenarios. Different copies of the scene exist, all exhibiting some form of corruption. The variant used here is manually corrected by Colin Fowler.

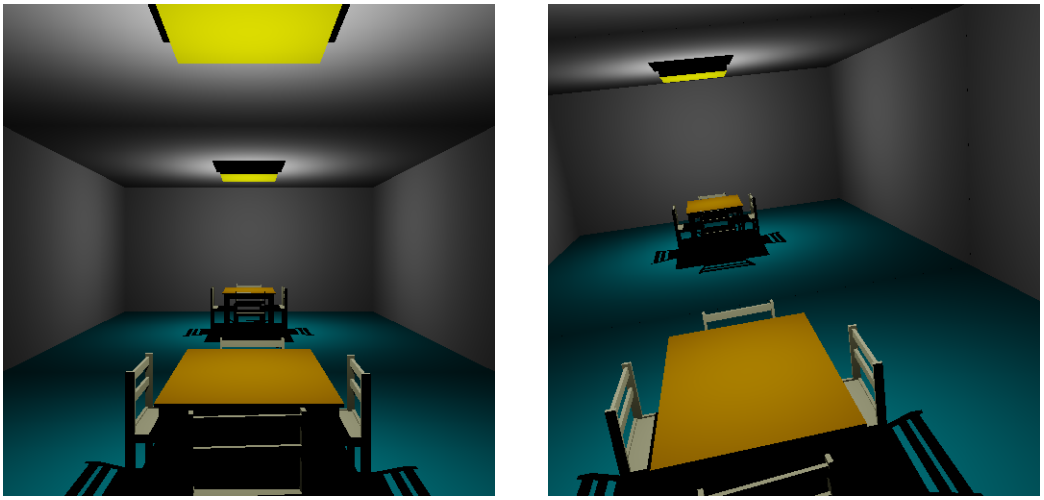
Sibenik Cathedral Courtesy of Marko Dabrovic. The large size of the cathedral requires a high number of photons to be traced when simulating indirect illumination. Glass surfaces and the floor blend a specular and a diffuse BRDF.

Fairy Forest Courtesy of Ingo Wald. This static scene is the first frame of an animation in which a faerie dances around a glade. The number of triangles is more than twice that of the previous scenes and large high resolution texture maps occupy half the available global memory.

Conference Room Courtesy of Greg Ward. A classic ray tracing benchmark, the scene is originally an MGF file [War96] containing triangle meshes and smooth surfaces. Different copies of the scene with smooth surfaces tessellated into triangle meshes exist, varying by tessellation level and the corruption introduced. A manually corrected variant with fine tessellation is used.

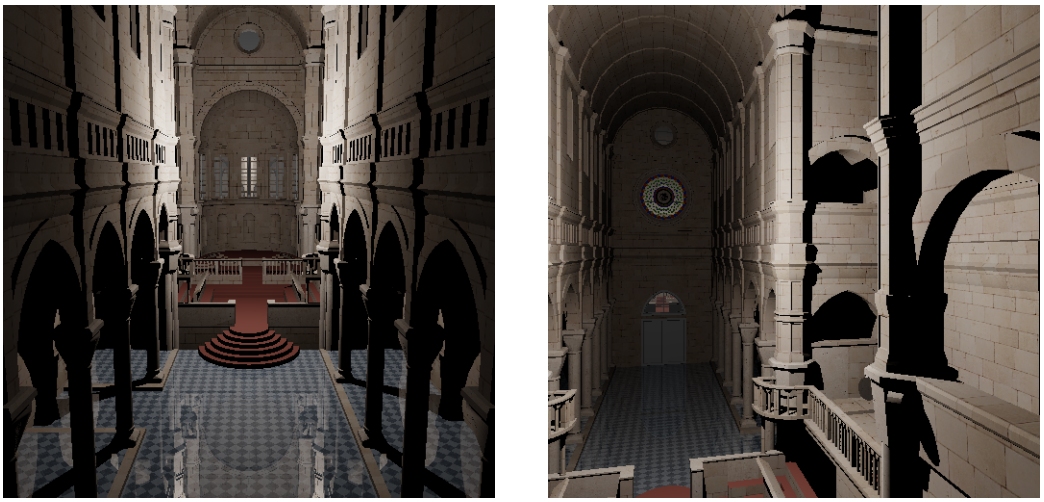
A.2.2. Photon Mapping Scenes

Additional scenes are used to illustrate prominent indirect illumination effects missing from ray traced images such as those of figure A.2 and added by photon mapping:



(a) Scene 6

(b) Sponza Atrium

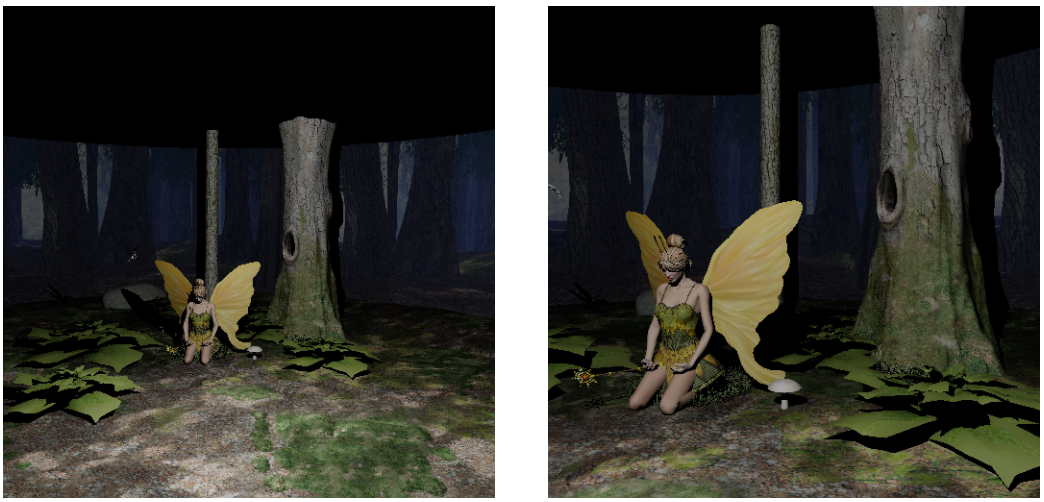


(c) Sibenik Cathedral

Figure A.1.: Ray traced views of the benchmark scenes for algorithm evaluation: Overview on the left, first frame of a flight through the scene on the right.

Scene	Size (m)			Triangles	Texture Maps (MB)
	L	W	H		
Scene 6	6.0	6.0	3.0	804	0
Sponza	34.8	15.6	16.6	76107	64
Sibenik	126.0	80.0	32.6	76643	16
Fairy	12.5	12.5	3.2	174117	512
Conference	11.3	7.2	2.7	282755	0
Passage	2.0	1.0	1.0	30	0
Ring	4.0	4.0	2.0	138	0

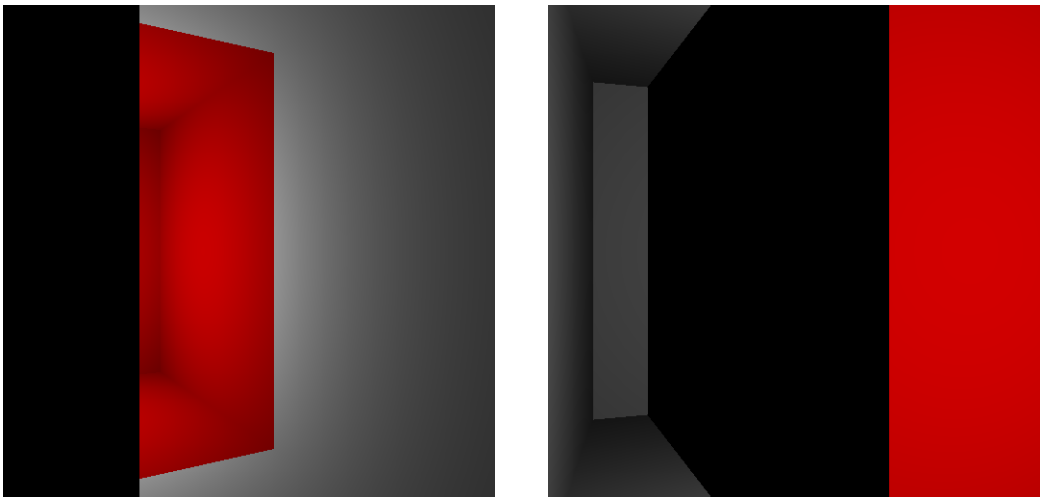
Table A.1.: Properties of the benchmark scenes used for algorithm evaluation



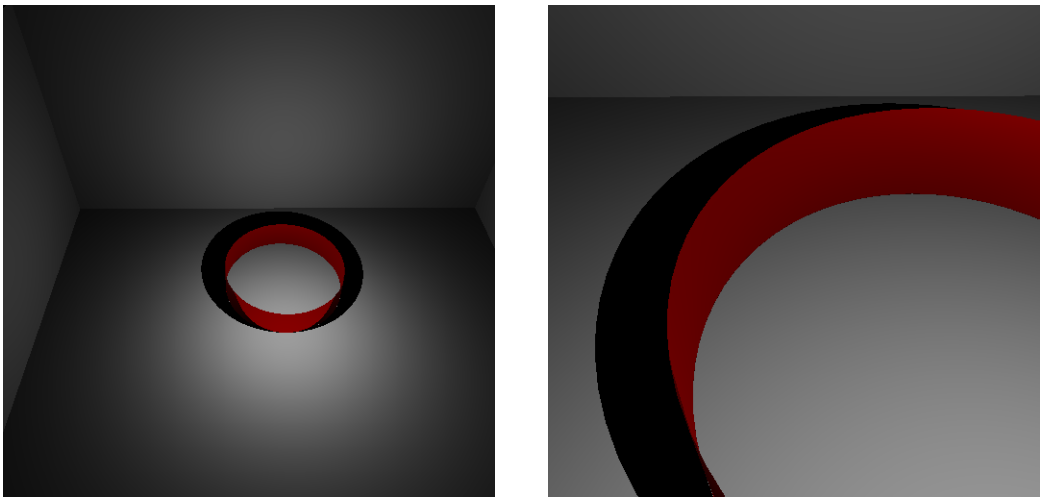
(d) Fairy Forest

(e) Conference Room

Figure A.1.: Ray traced views of the benchmark scenes for algorithm evaluation: Overview on the left, first frame of a flight through the scene on the right.



(a) Passage



(b) Ring

Figure A.2.: Ray traced views of the additional benchmark scenes for photon mapping evaluation: Overview on the left, first frame of a flight through the scene on the right.

Passage This scene exhibits strong color bleeding. Two rooms, one painted red and the other white, are connected by a narrow passage. The red room contains a white light source. Illumination in the other room is primarily indirect, arriving via reflections on the red walls.

Ring The specular ring focuses light onto a diffuse surface, causing a cardioid caustic.

Bibliography

[Adv09] Advanced Micro Devices Incorporated. *Technical Overview ATI Stream Computing*, 2009.

[AFO05] O. Arikan, D.A. Forsyth, and J.F. O'Brien. Fast and detailed approximate global illumination by irradiance decomposition. In *ACM SIGGRAPH 2005*, pages 1108–1114, 2005.

[AGCA08] P. Ajmera, R. Goradia, S. Chandran, and S. Aluru. Fast, parallel, GPU-based construction of space filling curves and octrees. In *i3D 2008 Posters*, page 10, 2008.

[AK90] J. Arvo and D. Kirk. Particle transport and image synthesis. In *ACM SIGGRAPH 1990*, pages 63–66, 1990.

[AK10] T. Aila and T. Karras. Architecture considerations for tracing incoherent rays. In *HPG 2010*, pages 113–122, 2010.

[AL09] T. Aila and S. Laine. Understanding the efficiency of ray traversal on GPUs. In *HPG 2009*, pages 145–149, 2009.

[Ali95a] Alias|Wavefront. *Advanced Visualizer File Formats*, 4.2 edition, 1995.

[Ali95b] Alias|Wavefront. *Advanced Visualizer User's Guide*, 4.2 edition, 1995.

[Amd67] G.M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Spring Joint Computer Conference 1967*, pages 483–485, 1967.

[App68] A. Appel. Some techniques for shading machine renderings of solids. In *AFIPS Spring Joint Computer Conference 1968*, pages 37–45, 1968.

[ARBJ03] S. Agarwal, R. Ramamoorthi, S. Belongie, and H.W. Jensen. Structured importance sampling of environment maps. In *ACM SIGGRAPH 2003*, pages 605–612, 2003.

[Arv86] J. Arvo. Backward ray tracing. In *ACM SIGGRAPH 1986 Courses*, pages 12:259–12:263, 1986.

[AW87] J. Amanatides and A. Woo. A fast voxel traversal algorithm for ray tracing. In *Eurographics 1987*, pages 3–10, 1987.

[BAGJ08] B.C. Budge, J.C. Anderson, C. Garth, and K.I. Joy. A straightforward CUDA implementation for interactive ray-tracing. In *RT 2008 Posters*, page 178, 2008.

[BAM06] V. Biri, D. Arquès, and S. Michelin. Real time rendering of atmospheric scattering and volumetric shadows. In *WSCG 2006*, pages 65–72, 2006.

[BEL+07] S. Boulos, D. Edwards, J.D. Lacewell, J. Kniss, J. Kautz, P. Shirley, and I. Wald. Packet-based whitted and distribution ray tracing. In *Graphics Interface 2007*, pages 177–184, 2007.

[Ben75] J.L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[BFH+04] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. In *ACM SIGGRAPH 2004*, pages 777–786, 2004.

[Ble90] G.E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, Carnegie Mellon University, Pittsburgh, PA, USA, 1990.

[Bli82] J.F. Blinn. Light reflection functions for simulation of clouds and dusty surfaces. In *ACM SIGGRAPH 1982*, pages 21–29, 1982.

[BLS93] P. Blasi, B. Le Saëc, and C. Schlick. A rendering algorithm for discrete volume density objects. In *Eurographics 1993*, pages 21–29, 1993.

[BMP77] L. Breiman, W. Meisel, and E. Purcell. Variable kernel estimates of multivariate densities. *Technometrics*, 19(2):135–144, 1977.

[BNM+08] A. Bouthors, F. Neyret, N. Max, E. Bruneton, and C. Crassin. Interactive multiple anisotropic scattering in clouds. In *i3D 2008*, pages 173–182, 2008.

[BOA09] M. Billeter, O. Olsson, and U. Assarsson. Efficient stream compaction on wide SIMD many-core architectures. In *HPG 2009*, pages 159–166, 2009.

[BOA10] M. Billeter, O. Olsson, and U. Assarsson. chag::pp. http://www.cse.chalmers.se/~billeter/pub/pp/, 2010.

[BPPP05] A. Boudet, P. Pitot, D. Pratmarty, and M. Paulin. Photon splatting for participating media. In *GRAPHITE 2005*, pages 197–204, 2005.

[BSA10] M. Billeter, E. Sintorn, and U. Assarsson. Real time volumetric shadows using polygonal light volumes. In *HPG 2010*, pages 39–45, 2010.

[BW99] M. Born and E. Wolf. *Principles of Optics: Electromagnetic Theory of Propagation, Interference and Diffraction of Light*. Cambridge University Press, seventh edition, 1999.

[BWB08] S. Boulos, I. Wald, and C. Benthin. Adaptive ray packet reordering. In *RT 2008*, pages 131–138, 2008.

[BWS03] C. Benthin, I. Wald, and P. Slusallek. A scalable approach to interactive global illumination. In *Eurographics 2003*, pages 621–630, 2003.

[Cat74] E.E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, University of Utah, Salt Lake City, UT, USA, 1974.

[CB04] P.H. Christensen and D. Batali. An irradiance atlas for global illumination in complex production scenes. In *EGSR 2004*, pages 133–141, 2004.

[CCWG88] M.F. Cohen, S.E. Chen, J.R. Wallace, and D.P. Greenberg. A progressive refinement approach to fast radiosity image generation. In *ACM SIGGRAPH 1988*, pages 75–84, 1988.

[CDP09] S. Collange, D. Defour, and D. Parello. Barra, a modular functional GPU simulator for GPGPU. Technical Report hal-00359342, Centre pour la Communication Scientifique Directe, Villeurbanne, France, 2009.

[CF87] S. Chattopadhyay and A. Fujimoto. Bi-directional ray tracing. In *CGI 1987*, pages 335–343, 1987.

[Cha50] S. Chandrasekhar. *Radiative Transfer*. Clarendon Press, 1950.

[CHCH06] N.A. Carr, J. Hoberock, K. Crane, and J.C. Hart. Fast GPU ray tracing of dynamic meshes using geometry images. In *Graphics Interface 2006*, pages 203–209, 2006.

[CHH02] N.A. Carr, J.D. Hall, and J.C. Hart. The ray engine. In *Graphics Hardware 2002*, pages 37–46, 2002.

[CHL04] G. Coombe, M.J. Harris, and A. Lastra. Radiosity on graphics hardware. In *Graphics Interface 2004*, pages 161–168, 2004.

[Chr99] P.H. Christensen. Faster photon map global illumination. *Graphics Tools*, 4(3):1–10, 1999.

[Chr03] P.H. Christensen. Adjoints and importance in rendering: An overview. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):329–340, 2003.

[CJ02] M. Cammarano and H.W. Jensen. Time dependent photon mapping. In *EGWR 2002*, pages 135–144, 2002.

[CKL+10] B. Choi, R. Komuravelli, V. Lu, H. Sung, R.L. Bocchino, S.V. Adve, and J.C. Hart. Parallel SAH k-d tree construction. In *HPG 2010*, pages 77–86, 2010.

[CLF+03] P.H. Christensen, D.M. Laur, J. Fong, W.L. Wooten, and D. Batali. Ray differentials and multiresolution geometry caching for distribution ray tracing in complex scenes. In *Eurographics 2003*, pages 543–552, 2003.

[Col68] W.A. Coleman. Mathematical verification of a certain Monte Carlo sampling technique and applications of the technique to radiation transport problems. *Nuclear Science and Engineering*, 32:76–81, 1968.

[Col94] S. Collins. Adaptive splatting for specular to diffuse light transport. In *EGWR 1994*, pages 119–135, 1994.

[Col95] S. Collins. Reconstruction of illumination from area luminaires. In *EGWR 1995*, pages 274–283, 1995.

[Coo86] R.L. Cook. Stochastic sampling in computer graphics. *ACM Transactions on Graphics*, 5(1):51–72, 1986.

[CPC84] R.L. Cook, T. Porter, and L. Carpenter. Distributed ray tracing. In *ACM SIGGRAPH 1984*, pages 137–145, 1984.

[Cro77] F.C. Crow. Shadow algorithms for computer graphics. In *ACM SIGGRAPH 1977*, pages 242–248, 1977.

[CS07] R. Cochran and J. Steele. Second-order illumination in real-time (student paper). In *ACM Southeast 2007*, pages 13–18, 2007.

[CSE06] D. Cline, K. Steele, and P. Egbert. Lightweight bounding volumes for ray tracing. *Journal of Graphics Tools*, 11(4):61–71, 2006.

[CSKSN05] S. Czuczor, L. Szirmay-Kalos, L. Szécsi, and L. Neumann. Photon map gathering on the GPU. In *Eurographics 2005 Short Papers*, pages 117–120, 2005.

[CT08] D. Cederman and P. Tsigas. On dynamic load balancing on graphics processors. In *Graphics Hardware 2008*, pages 57–64, 2008.

[DBB06] P. Dutré, K. Bala, and P. Bekaert. *Advanced Global Illumination*. AK Peters, second edition, 2006.

[DBMS02] K. Dmitriev, S. Brabec, K. Myszkowski, and H.-P. Seidel. Interactive global illumination using selective photon tracing. In *EGWR 2002*, pages 21–34, 2002.

[DGR+09] Z. Dong, T. Grosch, T. Ritschel, J. Kautz, and H.-P. Seidel. Real-time indirect illumination with clustered visibility. In *Vision, Modeling, and Visualization 2009*, pages 187–196, 2009.

[DHK08] H. Dammertz, J. Hanika, and A. Keller. Shallow bounding volume hierarchies for fast SIMD ray tracing of incoherent rays. In *EGSR 2008*, pages 1225–1233, 2008.

[DK08] H. Dammertz and A. Keller. The edge volume heuristic - robust triangle subdivision for improved BVH performance. In *RT 2008*, pages 155–158, 2008.

[DKH09] P. Djeu, S. Keely, and W. Hunt. Accelerating shadow rays using volumetric occluders and modified kd-tree traversal. In *HPG 2009*, pages 69–76, 2009.

[DS84] M. Dippé and J. Swensen. An adaptive subdivision algorithm and parallel architecture for realistic image synthesis. In *ACM SIGGRAPH 1984*, pages 149–158, 1984.

[DS05] C. Dachsbacher and M. Stamminger. Reflective shadow maps. In *i3D 2005*, pages 203–231, 2005.

[DS06] C. Dachsbacher and M. Stamminger. Splatting indirect illumination. In *i3D 2006*, pages 93–100, 2006.

[DSDD07] C. Dachsbacher, M. Stamminger, G. Drettakis, and F. Durand. Implicit visibility and antiradiance for interactive global illumination. In *ACM SIGGRAPH 2007*, pages 61:1–61:10, 2007.

[DYN00] Y. Dobashi, T. Yamamoto, and T. Nishita. Interactive rendering method for displaying shafts of light. In *Pacific Graphics 2000*, pages 31–37, 2000.

[DYN02] Y. Dobashi, T. Yamamoto, and T. Nishita. Interactive rendering of atmospheric scattering effects using graphics hardware. In *Graphics Hardware 2002*, pages 99–108, 2002.

[ED10] T. Engelhardt and C. Dachsbacher. Epipolar sampling for shadows and crepuscular rays in participating media with single scattering. In *i3D 2010*, pages 119–125, 2010.

[EG07] M. Ernst and G. Greiner. Early split clipping for bounding volume hierarchies. In *RT 2007*, pages 73–78, 2007.

[EG08] M. Ernst and G. Greiner. Multi bounding volume hierarchies. In *RT 2008*, pages 35–40, 2008.

[Epa69] V.A. Epanechnikov. Non-parametric estimation of a multivariate probability density. *Theory of Probability and its Applications*, 14(1):153–158, 1969.

[Eve01] C. Everitt. Interactive order-independent transparency, 2001.

[EWM08] M. Eisemann, C. Woizischke, and M. Magnor. Ray tracing with the single slab hierarchy. In *Vision, Modeling, and Visualization 2008*, pages 373–381, 2008.

[Fab06] B. Fabianowski. Efficient GPU-based multi-modal medical volume registration. Master's thesis, Universität Dortmund, Dortmund, Germany, 2006.

[Fat09] R. Fattal. Participating media illumination using light propagation maps. *ACM Transactions on Graphics*, 28(1):7:1–7:11, 2009.

[FC07] C. Fowler and S. Collins. Implementing the RT^2 real-time ray-tracing system. In *Eurographics Ireland 2007*, pages 1–8, 2007.

[FD09a] B. Fabianowski and J. Dingliana. Compact BVH storage for ray tracing and photon mapping. In *Eurographics Ireland 2009*, pages 1–8, 2009.

[FD09b] B. Fabianowski and J. Dingliana. Interactive global photon mapping. *Computer Graphics Forum*, 28(4):1151–1159, 2009.

[FFD09] B. Fabianowski, C. Flower, and J. Dingliana. A cost metric for scene-interior ray origins. In *Eurographics 2009 Short Papers*, pages 49–52, 2009.

[FKN80] H. Fuchs, Z.M. Kedem, and B.F. Naylor. On visible surface generation by a priori tree structures. In *ACM SIGGRAPH 1980*, pages 124–133, 1980.

[Fly72] M.J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9):948–960, 1972.

[FM86] F. Fabbrini and C. Montani. Autumnal quadtrees. *The Computer Journal*, 29(5):472–474, 1986.

[FS05] T. Foley and J. Sugerman. Kd-tree acceleration structures for a GPU raytracer. In *Graphics Hardware 2005*, pages 15–22, 2005.

[FTI86] A. Fujimoto, T. Tanaka, and K. Iwata. ARTS: Accelerated ray-tracing system. *IEEE Computer Graphics and Applications*, 6(4):16–26, 1986.

[FTYK83] K. Fujimura, H. Toriya, K. Yamaguchi, and T.L. Kunii. An enhanced oct-tree data structure, operations for solid modeling. In *NASA Symposium on Computer-Aided Geometry Modeling 1983*, pages 279–288, 1983.

[Gar09] K. Garanzha. The use of precomputed triangle clusters for accelerated ray tracing in dynamic scenes. In *EGSR 2009*, pages 1199–1206, 2009.

[GBP07] P. Gautron, K. Bouatouch, and S. Pattanaik. Temporal radiance caching. *IEEE Transactions on Visualization and Computer Graphics*, 13(5):891–901, 2007.

[GDW00] X. Granier, G. Drettakis, and B. Walter. Fast global illumination including specular effects. In *EGWR 2000*, pages 47–58, 2000.

[GGH02] X. Gu, S.J. Gortler, and H. Hoppe. Geometry images. In *ACM SIGGRAPH 2002*, pages 355–361, 2002.

[GK09] V. Gassenbauer and J. Křivánek. Spatial directional radiance caching. In *EGSR 2009*, pages 1189–1198, 2009.

[GKBP05] P. Gautron, J. Křivánek, K. Bouatouch, and S. Pattanaik. Radiance cache splatting: A GPU-friendly global illumination algorithm. In *EGSR 2005*, pages 55–64, 2005.

[GKPB04] P. Gautron, J. Křivánek, S. Pattanaik, and K. Bouatouch. A novel hemispherical basis for accurate and efficient rendering. In *EGSR 2004*, pages 321–330, 2004.

[GL10] K. Garanzha and C. Loop. Fast ray sorting and breadth-first packet traversal for GPU ray tracing. In *Eurographics 2010*, pages 289–298, 2010.

[GMAS05] D. Gutierrez, A. Muñoz, O. Anson, and F.J. Seron. Non-linear volume photon mapping. In *EGSR 2005*, pages 291–300, 2005.

[GMF09] P. Gautron, J.-E. Marvie, and G. François. Volumetric shadow mapping. In *ACM SIGGRAPH 2009 Sketches*, page 49, 2009.

[GMSJ03] D. Gutierrez, A. Muñoz, F.J. Seron, and E. Jimenez. Global illumination in inhomogeneous media based on curved photon mapping. In *Visualization, Imaging, and Image Processing 2003*, 2003.

[Gou71] H. Gouraud. Continuous shading of curved surfaces. *IEEE Transactions on Computers*, C-20(6):623–628, 1971.

[GPSS07] J. Günther, S. Popov, H.-P. Seidel, and P. Slusallek. Realtime ray tracing on GPU with BVH-based packet traversal. In *RT 2007*, pages 113–118, 2007.

[GR08] C.P. Gribble and K. Ramani. Coherent ray tracing via stream filtering. In *RT 2008*, pages 59–66, 2008.

[GSHG98] G. Greger, P. Shirley, P.M. Hubbard, and D.P. Greenberg. The irradiance volume. *IEEE Computer Graphics and Applications*, 18(2):32–43, 1998.

[GSMA08] D. Gutierrez, F.J. Seron, A. Muñoz, and O. Anson. Visualizing underwater ocean optics. In *Eurographics 2008*, pages 547–556, 2008.

[GTGB84] C.M. Goral, K.E. Torrance, D.P. Greenberg, and B. Battaile. Modeling the interaction of light between diffuse surfaces. In *ACM SIGGRAPH 1984*, pages 213–222, 1984.

[GWS04] J. Günther, I. Wald, and P. Slusallek. Realtime caustics using distributed photon mapping. In *EGSR 2004*, pages 111–121, 2004.

[GWS05] J. Günther, I. Wald, and H.-P. Seidel. Precomputed light sets for fast high quality global illumination. In *ACM SIGGRAPH 2005 Sketches*, page 108, 2005.

[Hal60] J.H. Halton. On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals. *Numerische Mathematik*, 2:84–90, 1960.

[Han86] P. Hanrahan. Using caching and breadth-first search to speed up ray-tracing. In *Graphics Interface 1986*, pages 56–61, 1986.

[Hav00] V. Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, České vysoké učení technické v Praze, Prague, Czech Republic, 2000.

[HB02] V. Havran and J. Bittner. On improving kd-trees for ray shooting. In *WSCG 2002*, pages 209–217, 2002.

[HBHS05] V. Havran, J. Bittner, R. Herzog, and H.-P. Seidel. Ray maps for global illumination. In *EGSR 2005*, pages 43–54, 2005.

[HDI+10] W. Hu, Z. Dong, I. Ihrke, T. Grosch, G. Yuan, and H.-P. Seidel. Interactive volume caustics in single-scattering media. In *i3D 2010*, pages 109–117, 2010.

[Hec90] P.S. Heckbert. Adaptive radiosity textures for bidirectional ray tracing. In *ACM SIGGRAPH 1990*, pages 145–154, 1990.

[HHC+06] R. Hong, T.-C. Ho, J.-H. Chuang, R-M. Shiu, and R. Kuo. A real-time analytic lighting model for anisotropic scattering. In *Computer Graphics Workshop 2006*, 2006.

[HHK+07] R. Herzog, V. Havran, S. Kinuwaki, K. Myszkowski, and H.-P. Seidel. Global illumination using photon ray splatting. In *Eurographics 2007*, pages 503–513, 2007.

[HHS05] V. Havran, R. Herzog, and H.-P. Seidel. Fast final gathering via reverse photon mapping. In *Eurographics 2005*, pages 323–333, 2005.

[HHS06] V. Havran, R. Herzog, and H.-P. Seidel. On the fast construction of spatial hierarchies for ray tracing. In *RT 2006*, pages 71–80, 2006.

[HJ09] T. Hachisuka and H.W. Jensen. Stochastic progressive photon mapping. In *SIGGRAPH Asia 2009*, pages 141:1–141:8, 2009.

[HJW+08] T. Hachisuka, W. Jarosz, R.P. Weistroffer, K. Dale, G. Humphreys, M. Zwicker, and H.W. Jensen. Multidimensional adaptive sampling and reconstruction for ray tracing. In *ACM SIGGRAPH 2008*, pages 33:1–33:10, 2008.

[HKRS02] J. Hurley, A. Kapustin, A. Reshetov, and A. Soupikov. Fast ray tracing for modern general purpose CPU. In *Graphicon 2002*, 2002.

[HL09] D.M. Hughes and I.S. Lim. Kd-jump: A path-preserving stackless traversal for faster isosurface raytracing on GPUs. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1555–1562, 2009.

[HLJH09] J. Hoberock, V. Lu, Y. Jia, and J.C. Hart. Stream compaction for deferred shading. In *HPG 2009*, pages 173–180, 2009.

[HMHB06] E. Hubo, T. Mertens, T. Haber, and P. Bekaert. The quantized kd-tree: Efficient ray tracing of compressed point clouds. In *RT 2006*, pages 105–113, 2006.

[HMS06] W.A. Hunt, W.R. Mark, and G. Stoll. Fast kd-tree construction with an adaptive error-bounded heuristic. In *RT 2006*, pages 81–88, 2006.

[HMS09] R. Herzog, K. Myszkowski, and H.-P. Seidel. Anisotropic radiance-cache splatting for efficiently computing high-quality global illumination with lightcuts. In *Eurographics 2009*, pages 259–268, 2009.

[HOJ08] T. Hachisuka, S. Ogaki, and H.W. Jensen. Progressive photon mapping. In *SIGGRAPH Asia 2008*, pages 130:1–130:8, 2008.

[HOS+10] M. Harris, J.D. Owens, S. Sengupta, S. Tseng, Y. Zhang, and A. Davidson N. Satish. CUDPP. http://gpgpu.org/developer/cudpp/, 2010.

[HP01] H. Hey and W. Purgathofer. Global illumination with photon map compensation. Technical Report TR-186-2-01-04, Technische Universität Wien, Wien, Austria, 2001.

[HP02a] H. Hey and W. Purgathofer. Advanced radiance estimation for photon map global illumination. In *Eurographics 2002*, pages 541–545, 2002.

[HP02b] H. Hey and W. Purgathofer. Importance sampling with hemispherical particle footprints. In *SCCG 2002*, pages 107–114, 2002.

[HSHH07] D.R. Horn, J. Sugerman, M. Houston, and P. Hanrahan. Interactive k-d tree GPU raytracing. In *i3D 2007*, pages 167–174, 2007.

[HSZ+10] Q. Hou, X. Sun, K. Zhou, C. Lauterbach, D. Manocha, and B. Guo. Memory-scalable GPU spatial hierarchy construction. *IEEE Transactions on Visualization and Computer Graphics*, 2010. to appear.

[Hun08] W.A. Hunt. Corrections to the surface area metric with respect to mail-boxing. In *RT 2008*, pages 77–80, 2008.

[HVAPB08] M. Hašan, E. Velázquez-Armendáriz, F. Pellacini, and K. Bala. Tensor clustering for rendering many-light animations. In *EGSR 2008*, pages 1105–1114, 2008.

[IDN02] K. Iwasaki, Y. Dobashi, and T. Nishita. A fast rendering method for refractive and reflective caustics due to water surfaces. In *Eurographics 2002*, pages 601–609, 2002.

[IEE08] IEEE Computer Society. *IEEE Standard for Floating-Point Arithmetic*, 2008.

[Ige99] H. Igehy. Tracing ray differentials. In *ACM SIGGRAPH 1999*, pages 179–186, 1999.

[Int09] Intel Corporation. *Introduction to Intel's 32nm Process Technology*, 2009.

[Int10] Intel Corporation. Intel unveils new product plans for high-performance computing. Press Release, 2010.

[IST07] International Business Machines Corporation, Sony Computer Entertainment Incorporated, and Toshiba Corporation. *Cell Broadband Engine Architecture*, 1.02 edition, 2007.

[IZT+07] I. Ihrke, G. Ziegler, A. Tevs, C. Theobalt, M. Magnor, and H.-P. Seidel. Eikonal rendering: Efficient light transport in refractive objects. In *ACM SIGGRAPH 2007*, pages 59:1–59:9, 2007.

[JC95] H.W. Jensen and N.J. Christensen. Photon maps in bidirectional Monte Carlo ray tracing of complex objects. *Computers & Graphics*, 19(2):215–224, 1995.

[JC98] H.W. Jensen and P.H. Christensen. Efficient simulation of light transport in scences with participating media using photon maps. In *ACM SIGGRAPH 1998*, pages 311–320, 1998.

[JDZJ08] W. Jarosz, C. Donner, M. Zwicker, and H.W. Jensen. Radiance caching for participating media. *ACM Transactions on Graphics*, 27(1):7:1–7:11, 2008.

[Jen95] H.W. Jensen. Importance driven path tracing using the photon map. In *EGWR 1995*, pages 326–335, 1995.

[Jen96] H.W. Jensen. Global illumination using photon maps. In *EGWR 1996*, pages 21–30, 1996.

[Jen01] H.W. Jensen. *Realistic Image Synthesis Using Photon Mapping*. AK Peters, 2001.

[JW89] D. Jevans and B. Wyvill. Adaptive voxel subdivision for ray tracing. In *Graphics Interface 1989*, pages 164–172, 1989.

[JZJ08a] W. Jarosz, M. Zwicker, and H.W. Jensen. The beam radiance estimate for volumetric photon mapping. In *Eurographics 2008*, pages 557–566, 2008.

[JZJ08b] W. Jarosz, M. Zwicker, and H.W. Jensen. Irradiance gradients in the presence of participating media and occlusions. In *EGSR 2008*, pages 1087 – 1096, 2008.

[Kaj86] J.T. Kajiya. The rendering equation. In *ACM SIGGRAPH 1986*, pages 143–150, 1986.

[KBPŽ06] J. Křivánek, K. Bouatouch, S. Pattanaik, and J. Žára. Making radiance and irradiance caching practical: Adaptive caching and neighbor clamping. In *EGSR 2006*, pages 127–138, 2006.

[KBW06] J. Krüger, K. Bürger, and R. Westermann. Interactive screen-space accurate photon tracing on GPUs. In *EGSR 2006*, pages 319–329, 2006.

[KD10] A. Kaplanyan and C. Dachsbacher. Cascaded light propagation volumes for real-time indirect illumination. In *i3D 2010*, pages 99–108, 2010.

[Kel97a] A. Keller. Instant radiosity. In *ACM SIGGRAPH 1997*, pages 49–56, 1997.

[Kel97b] A. Keller. *Quasi-Monte Carlo Methods for Photorealistic Image Synthesis*. PhD thesis, Technische Universität Kaiserslautern, Kaiserslautern, Germany, 1997.

[Ken07] A. Kensler. Comments on uniform sampling and cones. *Ray Tracing News*, 20(1), 2007.

[KGBP05] J. Křivánek, P. Gautron, K. Bouatouch, and S. Pattanaik. Improved radiance gradient computation. In *SCCG 2005*, pages 155–159, 2005.

[KGPB05] J. Křivánek, P. Gautron, S. Pattanaik, and K. Bouatouch. Radiance caching for efficient global illumination computation. *IEEE Transactions on Visualization and Computer Graphics*, 11(5):550–561, 2005.

[KH01] A. Keller and W. Heidrich. Interleaved sampling. In *EGWR 2001*, pages 269–276, 2001.

[KIB05] B. Kang, I. Ihm, and C. Bajaj. Extending the photon mapping method for realistic rendering of hot gaseous fluids. *Journal of Visualization and Computer Animation*, 16(3–4):353–363, 2005.

[KK86] T.L. Kay and J.T. Kajiya. Ray tracing complex scenes. In *ACM SIGGRAPH 1986*, pages 269–278, 1986.

[KK02] T. Kollig and A. Keller. Efficient multidimensional sampling. In *Eurographics 2002*, pages 557–564, 2002.

[KM63] M.G. Kendall and P.A.P. Moran. *Geometrical Probability*. Charles Griffin & Company, 1963.

[KMKY09] T.-J. Kim, B. Moon, D. Kim, and S.-E. Yoon. RACBVHs: Random-accessible compressed bounding volume hierarchies. *IEEE Transactions on Visualization and Computer Graphics*, 15(6), 2009.

[KS09] J. Kalojanov and P. Slusallek. A parallel algorithm for construction of uniform grids. In *HPG 2009*, pages 23–28, 2009.

[KvH84] J.T. Kajiya and B.P. von Herzen. Ray tracing volume densities. In *ACM SIGGRAPH 1984*, pages 165–174, 1984.

[KW00] A. Keller and I. Wald. Efficient importance sampling techniques for the photon map. In *Vision, Modeling, and Visualization 2000*, pages 271–279, 2000.

[Lai10] S. Laine. Restart trail for stackless BVH traversal. In *HPG 2010*, pages 107–111, 2010.

[LAM05] T. Larsson and T. Akenine-Möller. A dynamic bounding volume hierarchy for generalized collision detection. In *VRIPHYS 2005*, pages 91–100, 2005.

[Lan02] H. Landis. Production-ready global illumination. In *ACM SIGGRAPH 2002 Courses*, pages 16:87–16:102, 2002.

[LC03] B.D. Larsen and N.J. Christensen. Optimizing photon mapping using multiple photon maps for irradiance estimates. In *WSCG 2003 Posters*, pages 77–80, 2003.

[LC04] B.D. Larsen and N.J. Christensen. Simulating photon mapping for real-time applications. In *EGSR 2004*, pages 123–131, 2004.

[LD08] A. Lagae and P. Dutré. Compact, fast and robust grids for ray tracing. In *EGSR 2008*, 2008.

[LGS+09] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast BVH construction on GPUs. In *Eurographics 2009*, pages 375–384, 2009.

[LKM01] E. Lindholm, M.J. Kilgard, and H. Moreton. A user-programmable vertex engine. In *ACM SIGGRAPH 2001*, pages 149–158, 2001.

[LP02] F. Lavignotte and M. Paulin. A new approach of density estimation for global illumination. In *WSCG 2002*, pages 263–270, 2002.

[LP03] F. Lavignotte and M. Paulin. Scalable photon splatting for global illumination. In *GRAPHITE 2003*, pages 203–210, 2003.

[LQ65] D.O. Loftsgaarden and C.P. Quesenberry. A nonparametric estimate of a multivariate density function. *Annals of Mathematical Statistics*, 36(3):1049–1051, 1965.

[LURM02] M. Lastra, C. Ureña, J. Revelles, and R. Montes. A particle-path based method for Monte Carlo density estimation. In *EGWR 2002 Posters*, pages 33–40, 2002.

[LW94] E.P. Lafortune and Y.D. Willems. Using the modified Phong reflectance model for physically based rendering. Technical Report CW 197, Katholieke Universiteit Leuven, Leuven, Belgium, 1994.

[LWL06] B. Liu, E. Wu, and X. Liu. Interactively rendering dynamic caustics on GPU. In *CGI 2006*, pages 136–147, 2006.

[LYM07] C. Lauterbach, S.-E. Yoon, and D. Manocha. Ray-strips: A compact mesh representation for interactive ray tracing. In *RT 2007*, pages 19–26, 2007.

[LYT06] C. Lauterbach, S.-E. Yoon, and D. Tuft. RT-DEFORM: Interactive ray tracing of dynamic scenes using BVHs. In *RT 2006*, pages 39–46, 2006.

[LYTM08] C. Lauterbach, S.-E. Yoon, M. Tang, and D. Manocha. ReduceM: Interactive and memory efficient ray tracing of large models. In *EGSR 2008*, pages 1313–1321, 2008.

[LZT+08] J. Lehtinen, M. Zwicker, E. Turquin, J. Kontkanen, F. Durand, F.X. Sillion, and T. Aila. A meshless hierarchical representation for light transport. In *ACM SIGGRAPH 2008*, pages 37:1–37:9, 2008.

[Mah05] J.A. Mahovsky. *Ray Tracing with Reduced-Precision Bounding Volume Hierarchies*. PhD thesis, University of Calgary, Calgary, AB, Canada, 2005.

[Max94] N. Max. Efficient light propagation for multiple anisotropic volume scattering. In *EGWR 1994*, pages 87–104, 1994.

[MB90] J.D. MacDonald and K.S. Booth. Heuristics for ray tracing using space subdivision. *The Visual Computer*, 6(3):153–166, 1990.

[MBC79] M.D. McKay, R.J. Beckman, and W.J. Conover. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 21(2):239–245, 1979.

[McC94] W.R. McCluney. *Introduction to Radiometry and Photometry*. Artech House, 1994.

[Min06] K. Min. An efficient photon mapping algorithm for rendering light-emitting fluids. In *International Symposium on Visual Computing 2006*, pages 850–859, 2006.

[MKS07] J. Mortensen, P. Khanna, and M. Slater. Light field propagation and rendering on the GPU. In *AFRIGRAPH 2007*, pages 15–23, 2007.

[ML09] M. McGuire and D. Luebke. Hardware-accelerated global illumination by image space photon mapping. In *HPG 2009*, pages 77–90, 2009.

[MM02] V.C.H. Ma and M.D. McCool. Low latency photon mapping using block hashing. In *Graphics Hardware 2002*, pages 1–11, 2002.

[MM06] J.T. Moon and S.R. Marschner. Simulating multiple scattering in hair using a photon mapping approach. In *ACM SIGGRAPH 2006*, pages 1067–1074, 2006.

[MMAM07] E. Månsson, J. Munkberg, and T. Akenine-Möller. Deep coherent ray tracing. In *RT 2007*, pages 79–85, 2007.

[Moo65] G.E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965.

[MT97] T. Möller and B. Trumbore. Fast, minimum storage ray/triangle intersection. *Journal of Graphics Tools*, 2(2):25–30, 1997.

[Mun10] A. Munshi. *The OpenCL Specification*, 1.1 edition, 2010.

[NFJ02] D.Q. Nguyen, R. Fedkiw, and H.W. Jensen. Physically based modeling and animation of fire. In *ACM SIGGRAPH 2002*, pages 721–728, 2002.

[NFLM07] P.A. Navrátil, D.S. Fussell, C. Lin, and W.R. Mark. Dynamic ray scheduling to improve ray coherence and bandwidth utilization. In *RT 2007*, pages 95–104, 2007.

[Nic65] F.E. Nicodemus. Directional reflectance and emissivity of an opaque surface. *Applied Optics*, 4(7):767–775, 1965.

[Nil08] A. Nilsson. Two hybrid methods of volumetric lighting. Master's thesis, Lunds Universitet, Lund, Sweden, 2008.

[NPG03] M. Nijasure, S. Pattanaik, and V. Goel. Interactive global illumination in dynamic environments using commodity graphics hardware. In *Pacific Graphics 2003*, pages 450–454, 2003.

[NRH77] F.E. Nicodemus, J.C. Richmond, and J.J. Hsia. *Geometrical Considerations and Nomenclature for Reflectance*, 1977.

[NSW09] G. Nichols, J. Shopf, and C. Wyman. Hierarchical image-space radiosity for interactive global illumination. In *EGSR 2009*, pages 1141–1149, 2009.

[NVI10a] NVIDIA Corporation. *NVIDIA CUDA™Best Practices Guide*, 3.1 edition, 2010.

[NVI10b] NVIDIA Corporation. *NVIDIA CUDA™Programming Guide*, 3.1 edition, 2010.

[NW09] G. Nichols and C. Wyman. Multiresolution splatting for indirect illumination. In *i3D 2009*, pages 83–90, 2009.

[NW10] G. Nichols and C. Wyman. Interactive indirect illumination using adaptive multiresolution splatting. *IEEE Transactions on Visualization and Computer Graphics*, 16(5):729–741, 2010.

[ORM08] R. Overbeck, R. Ramamoorthi, and W.R. Mark. Large ray packets for real-time whitted ray tracing. In *RT 2008*, pages 41–48, 2008.

[Par62] E. Parzen. On estimation of a probability density function and mode. *Annals of Mathematical Statistics*, 33(3):1065–1076, 1962.

[PARN04] S. Premože, M. Ashikhmin, R. Ramamoorthi, and S. Nayar. Practical rendering of multiple scattering effects in participating media. In *EGSR 2004*, pages 363–374, 2004.

[PBD+10] S.G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich. OptiX: A general purpose ray tracing engine. In *ACM SIGGRAPH 2010*, pages 66:1–66:13, 2010.

[PBMH02] T.J. Purcell, I. Buck, W.R. Mark, and Hanrahan. Ray tracing on programmable graphics hardware. In *ACM SIGGRAPH 2002*, pages 703–712, 2002.

[PBSP08] V. Pegoraro, C. Brownlee, P.S. Shirley, and S.G. Parker. Towards interactive global illumination effects via sequential Monte Carlo adaptation. In *RT 2008*, pages 107–114, 2008.

[PDC+03] T.J. Purcell, C. Donner, M. Cammarano, H.W. Jensen, and P. Hanrahan. Photon mapping on programmable graphics hardware. In *Graphics Hardware 2003*, pages 41–50, 2003.

[PGDS09] S. Popov, I. Georgiev, R. Dimov, and P. Slusallek. Object partitioning considered harmful: Space subdivision for BVHs. In *HPG 2009*, pages 15–22, 2009.

[PGSS06] S. Popov, J. Günther, H.-P. Seidel, and P. Slusallek. Experiences with streaming construction of SAH kd-trees. In *RT 2006*, pages 89–94, 2006.

[PGSS07] S. Popov, J. Günther, H.-P. Seidel, and P. Slusallek. Stackless kd-tree traversal for high performance GPU ray tracing. In *Eurographics 2007*, pages 415–424, 2007.

[Pho75] B.T. Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, 1975.

[PL10] J. Pantaleoni and D. Luebke. HLBVH: Hierarchical LBVH construction for real-time ray tracing of dynamic geometry. In *HPG 2010*, pages 87–95, 2010.

[Pla00] M. Planck. Entropy and temperature of radiant heat. *Annalen der Physik*, 1(4):719–737, 1900.

[PP98] I. Peter and G. Pietrek. Importance driven construction of photon maps. In *EGWR 1998*, pages 269–280, 1998.

[PP09a] C. Papadopoulos and G. Papaioannou. Realistic real-time underwater caustics and godrays. In *GraphiCon 2009*, pages 89–95, 2009.

[PP09b] V. Pegoraro and S.G. Parker. An analytical solution to single scattering in homogeneous participating media. In *Eurographics 2009*, pages 329–335, 2009.

[PSP10] V. Pegoraro, M. Schott, and S.G. Parker. A closed-form solution to single scattering for general phase functions and light distributions. In *EGSR 2010*, pages 1365–1374, 2010.

[PWP08] V. Pegoraro, I. Wald, and S.G. Parker. Sequential Monte Carlo adaptation in low-anisotropy participating media. In *EGSR 2008*, pages 1097–1104, 2008.

[RAA+03] J. Revelles, N. Aguilera, J. Aguado, M. Lastra, R. García, and R. Montes. A spatial representation for ray-scene intersection test improvement in complex scenes. In *Eurographics 2003 Posters*, pages 97–100, 2003.

[RCC+06] R.M. Ramanathan, R. Curry, S. Chennupaty, R.L. Cross, S. Kuo, and M.J. Buxton. *Extending the World's Most Popular Processor Architecture*, 2006.

[REG+09] T. Ritschel, T. Engelhardt, T. Grosch, H.-P. Seidel, J. Kautz, and C. Dachsbacher. Micro-rendering for scalable, parallel final gathering. In *SIGGRAPH Asia 2009*, pages 132:1–132:8, 2009.

[Res06] A. Reshetov. Omnidirectional ray tracing traversal algorithm for kd-trees. In *RT 2006*, pages 57–60, 2006.

[RGS09] T. Ritschel, T. Grosch, and H.-P. Seidel. Approximating dynamic global illumination in image space. In *i3D 2009*, pages 75–82, 2009.

[RH01] R. Ramamoorthi and P. Hanrahan. An efficient representation for irradiance environment maps. In *ACM SIGGRAPH 2001*, pages 497–500, 2001.

[RK07] R.Y. Rubinstein and D.P. Kroese. *Simulation and the Monte Carlo Method*. Wiley Series in Probability and Statistics. Wiley-Interscience, second edition, 2007.

[Rob81] J.T. Robinson. The k-d-B-tree: A search structure for large multidimensional dynamic indexes. In *SIGMOD*, pages 10–18, 1981.

[RW80] S.M. Rubin and T. Whitted. A 3-dimensional representation for fast rendering of complex scenes. In *ACM SIGGRAPH 1980*, pages 110–116, 1980.

[RZLG08] Z. Ren, K. Zhou, S. Lin, and B. Guo. Gradient-based interpolation and sampling for real-time rendering of inhomogeneous, single-scattering media. In *Pacific Graphics 2008*, pages 1945–1953, 2008.

[SA07] P. Shanmugam and O. Arikan. Hardware accelerated ambient occlusion techniques on GPUs. In *i3D 2007*, pages 73–80, 2007.

[SA10] M. Segal and K. Akeley. *The OpenGL®Graphics System: A Specification*, 4.1 (core profile) edition, 2010.

[SB97] W. Stürzlinger and R. Bastos. Interactive rendering of globally illuminated glossy scenes. In *EGWR 1997*, pages 93–102, 1997.

[Sch03] R. Schregle. Bias compensation for photon maps. *Computer Graphics Forum*, 22(4):729–742, 2003.

[Sch08] C. Schlier. On scrambled halton sequences. *Applied Numerical Mathematics*, 58(10):1467–1478, 2008.

[Sch09] L. Schjøth. *Anisotropic Density Estimation in Global Illumination*. PhD thesis, Københavns Universitet, København, Denmark, 2009.

[SCL05] J. Steinhurst, G. Coombe, and A. Lastra. Reordering for cache conscious photon mapping. In *Graphics Interface 2005*, pages 97–104, 2005.

[SCL08] J. Steinhurst, G. Coombe, and A. Lastra. Reducing photon-mapping bandwidth by query reordering. *IEEE Transactions on Visualization and Computer Graphics*, 14(1):13–24, 2008.

[SCS+08] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: A many-core x86 architecture for visual computing. In *ACM SIGGRAPH 2008*, pages 18:1–18:15, 2008.

[Sea96] D. Seaman. Re: N-dim spherical random number drawing. news:wmEgVDGO0VUtMEhUwg@andrew.cmu.edu, 1996.

[Seg08] B. Segovia. Radius-CUDA. http://www710.univ-lyon1.fr/~bsegovia/demos/radius-cuda.zip, 2008.

[SF07a] S. Singh and P. Faloutsos. The photon pipeline revisited. *The Visual Computer*, 23(7):479–492, 2007.

[SF07b] S. Singh and P. Faloutsos. SIMD packet techniques for photon mapping. In *RT 2007*, pages 87–94, 2007.

[SFD09] M. Stich, H. Friedrich, and A. Dietrich. Spatial splits in bounding volume hierarchies. In *HPG 2009*, pages 7–13, 2009.

[SFES07] L. Schjøth, J.R. Frisvad, K. Erleben, and J. Sporring. Photon differentials. In *GRAPHITE 2007*, pages 179–186, 2007.

[SHG08] S. Sengupta, M. Harris, and M. Garland. Efficient parallel scan algorithms for GPUs. Technical Report NVR-2008-003, NVIDIA Corporation, 2008.

[SHG09] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *IEEE International Parallel & Distributed Processing Symposium 2009*, pages 1–10, 2009.

[SHZO07] S. Sengupta, M. Harris, Y. Zhang, and J.D. Owens. Scan primitives for GPU computing. In *Graphics Hardware 2007*, pages 97–106, 2007.

[SIMP06] B. Segovia, J.C. Iehl, R. Mitanchey, and B. Péroche. Non-interleaved deferred shading of interleaved sample patterns. In *Graphics Hardware 2006*, pages 53–60, 2006.

[SIP07] B. Segovia, J.C. Iehl, and B. Péroche. Metropolis instant radiosity. In *Eurographics 2007*, pages 425–434, 2007.

[SJ09] B. Spencer and M.W. Jones. Into the blue: Better caustics through photon relaxation. In *Eurographics 2009*, pages 319–328, 2009.

[SKALP05] L. Szirmay-Kalos, B. Aszódi, I. Lazányi, and M. Premecz. Approximate ray-tracing on the GPU with distance impostors. In *Eurographics 2005*, pages 695–704, 2005.

[SKLUT09] L. Szirmay-Kalos, G. Liktor, T. Umenhoffer, and B. Tóth. Fast approximation of multiple scattering in inhomogeneous participating media. In *Eurographics 2009 Short Papers*, pages 53–56, 2009.

[SKP99] L. Szirmay-Kalos and W. Purgathofer. Analysis of the quasi-Monte Carlo integration of the rendering equation. In *WSCG 1999*, pages 281–288, 1999.

[SKP07] M. Shah, J. Konttinen, and S. Pattanaik. Caustics mapping: An image-space technique for real-time caustics. *IEEE Transactions on Visualization and Computer Graphics*, 13(2):272–280, 2007.

[SKSU05] L. Szirmay-Kalos, M. Sbert, and T. Umenhoffer. Real-time multiple scattering in participating media with illumination networks. In *EGSR 2005*, pages 277–282, 2005.

[SKUP+09] L. Szirmay-Kalos, T. Umenhoffer, G. Patow, L. Szécsi, and M. Sbert. Specular effects on the GPU: State of the art. *Computer Graphics Forum*, 28(6):1586–1617, 2009.

[Smi98] B. Smits. Efficiency issues for ray tracing. *Journal of Graphics Tools*, 3(2):1–14, 1998.

[Sob67] I.M. Sobol'. On the distribution of points in a cube and the approximate evaluation of integrals. *USSR Computational Mathematics and Mathematical Physics*, 7(4):86–112, 1967.

[SRNN05] B. Sun, R. Ramamoorthi, S.G. Narasimhan, and S.K. Nayar. A practical analytic single scattering model for real time rendering. In *ACM SIGGRAPH 2005*, pages 1040–1049, 2005.

[SSK07] M. Shevtsov, A. Soupikov, and A. Kapustin. Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes. In *Eurographics 2007*, pages 395–404, 2007.

[SSS02] A. Scheel, M. Stamminger, and H.-P. Seidel. Grid based final gather for radiosity on complex clustered scenes. In *Eurographics 2002*, pages 547–556, 2002.

[SSSK04] M. Sbert, L. Szécsi, and L. Szirmay-Kalos. Real-time light animation. In *Eurographics 2004*, pages 291–300, 2004.

[ST94] W. Stürzlinger and R.F. Tobler. Two optimization methods for raytracing. In *SCCG 1994*, pages 104–107, 1994.

[STK08] A. Schmitz, M. Tavenrath, and L. Kobbelt. Interactive global illumination for deformable geometry in CUDA. In *Pacific Graphics 2008*, pages 1979–1986, 2008.

[SW00] F. Suykens and Y.D. Willems. Density control for photon maps. In *EGWR 2000*, pages 23–34, 2000.

[SW01] F. Suykens and Y.D. Willems. Path differentials and applications. In *EGWR 2001*, pages 257–268, 2001.

[SWH+95] P. Shirley, B. Wade, P.M. Hubbard, D. Zareski, B. Walter, and D.P. Greenberg. Global illumination via density-estimation. In *EGWR 1995*, pages 187–199, 1995.

[SWS02] J. Schmittler, I. Wald, and P. Slusallek. SaarCOR — a hardware architecture for ray tracing. In *Graphics Hardware 2002*, pages 27–36, 2002.

[TCE05] J. Talbot, D. Cline, and P. Egbert. Importance resampling for global illumination. In *EGSR 2005*, pages 139–146, 2005.

[TDJ+02] J.M. Tendler, J.S. Dodson, J.S. Fields Junior, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 26(1):5–25, 2002.

[Ter01] P. Terdiman. Memory-optimized bounding-volume hierarchies, 2001.

[TL04] E. Tabellion and A. Lamorlette. An approximate global illumination system for computer generated films. In *ACM SIGGRAPH 2004*, pages 469–476, 2004.

[TMG09] R. Torres, P.J. Martín, and A. Gavilanes. Ray casting using a roped BVH with CUDA. In *SCCG 2009*, pages 107–114, 2009.

[TMS04a] T. Tawara, K. Myszkowski, and H.-P. Seidel. Efficient rendering of strong secondary lighting in photon mapping algorithm. In *Theory and Practice of Computer Graphics 2004*, pages 174–178, 2004.

[TMS04b] T. Tawara, K. Myszkowski, and H.-P. Seidel. Exploiting temporal coherence in final gathering for dynamic scenes. In *CGI 2004*, pages 110–119, 2004.

[TPO10] S. Tzeng, A. Patney, and J.D. Owens. Task management for irregular-parallel workloads on the GPU. In *HPG 2010*, pages 29–37, 2010.

[TPWG02] P. Tole, F. Pellacini, B. Walter, and D.P. Greenberg. Interactive global illumination in dynamic scenes. In *ACM SIGGRAPH 2002*, pages 537–546, 2002.

[Tsa09] J.A. Tsakok. Faster incoherent rays: Multi-BVH ray stream tracing. In *HPG 2009*, pages 151–158, 2009.

[TU09] B. Tóth and T. Umenhoffer. Real-time volumetric lighting in participating media. In *Eurographics 2009 Short Papers*, pages 57–60, 2009.

[UPSK08] T. Umenhoffer, G. Patow, and L. Szirmay-Kalos. Caustic triangles on the GPU. In *CGI 2008*, pages 222–228, 2008.

[VBS99] M. Vanco, G. Brunnett, and T. Schreiber. A hashing strategy for efficient k-nearest neighbors computation. In *CGI 1999*, pages 120–128, 1999.

[VD08] V. Volkov and J.W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Supercomputing 2008*, pages 31:1–31:11, 2008.

[Vea96] E. Veach. Non-symmetric scattering in light transport algorithms. In *EGWR 1996*, pages 82–91, 1996.

[VG94] E. Veach and L. Guibas. Bidirectional estimators for light transport. In *EGWR 1994*, pages 147–162, 1994.

[VG97] E. Veach and L.J. Guibas. Metropolis light transport. In *ACM SIGGRAPH 1997*, pages 65–76, 1997.

[vH67] H. von Helmholtz. *Handbuch der Physiologischen Optik*, volume 9 of *Allgemeine Encyklopädie der Physik*. Leopold Voss, 1867.

[vOS83] A. van Oosterom and J. Strackee. The solid angle of a plane triangle. *IEEE Transactions on Biomedical Engineering*, BME-30(2):125–126, 1983.

[WABG06] B. Walter, A. Arbree, K. Bala, and D.P. Greenberg. Multidimensional lightcuts. In *ACM SIGGRAPH 2006*, pages 1081–1088, 2006.

[Wal04] I. Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, 2004.

[Wal05] I. Wald. DIRmaps: Discretized incident radiance maps for high-quality global illumination walkthroughs in complex environments. Technical Report UUSCI-2005-010, University of Utah, Salt Lake City, UT, USA, 2005.

[Wal07] I. Wald. On fast construction of SAH-based bounding volume hierarchies. In *RT 2007*, pages 33–40, 2007.

[War96] G.J. Ward. *The Materials and Geometry Format*, 1.1 edition, 1996.

[WBB08] I. Wald, C. Benthin, and S. Boulos. Getting rid of packets - efficient SIMD single-ray traversal using multi-branching BVHs -. In *RT 2008*, pages 49–57, 2008.

[WBS03] I. Wald, C. Benthin, and P. Slusallek. Interactive global illumination in complex and highly occluded environments. In *EGSR 2003*, pages 74–81, 2003.

[WBS07] I. Wald, S. Boulos, and P. Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics*, 26(1):6:1–6:18, 2007.

[WD06] C. Wyman and C. Dachsbacher. Improving image-space caustics via variable-sized splatting. Technical Report UICS-06-02, University of Utah, Salt Lake City, UT, USA, 2006.

[WFA+05] B. Walter, S. Fernandez, A. Arbree, K. Bala, M. Donikian, and D.P. Greenberg. Lightcuts: A scalable approach to illumination. In *ACM SIGGRAPH 2005*, pages 1098–1107, 2005.

[WFM+05] I. Wald, H. Friedrich, G. Marmitt, P. Slusallek, and H.-P. Seidel. Faster isosurface ray tracing using implicit kd-trees. *IEEE Transactions on Visualization and Computer Graphics*, 11(5):562–572, 2005.

[WGBK05] I. Wald, C. Gribble, S. Boulos, and A. Kensler. SIMD ray stream tracing - SIMD ray traversal with generalized ray packets and on-the-fly re-ordering -. Technical Report UUSCI-2005-010, University of Utah, Salt Lake City, UT, USA, 2005.

[WGS04] I. Wald, J. Günther, and P. Slusallek. Balancing considered harmful – faster photon mapping using the voxel volume heuristic –. In *Eurographics 2004*, pages 595–603, 2004.

[WH92] G.J. Ward and P.S. Heckbert. Irradiance gradients. In *EGWR 1992*, pages 85–98, 1992.

[WH06] I. Wald and V. Havran. On building fast kd-trees for ray tracing, and on doing that in O(n log n). In *RT 2006*, pages 61–69, 2006.

[WHG84] H. Weghorst, G. Hooper, and D.P. Greenberg. Improved computational methods for ray tracing. *ACM Transactions on Graphics*, 3(1):52–69, 1984.

[Whi80] T. Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, 1980.

[Wil78] Lance Williams. Casting curved shadows on curved surfaces. In *ACM SIGGRAPH 1978*, pages 270–274, 1978.

[WK06] C. Wächter and A. Keller. Instant ray tracing: The bounding interval hierarchy. In *EGSR 2006*, pages 139–149, 2006.

[WKB+02] I. Wald, T. Kollig, C. Benthin, A. Keller, and P. Slusallek. Interactive global illumination using fast ray tracing. In *EGWR 2002*, pages 15–24, 2002.

[WMH+07] I. Wald, W.R. Mark, W. Hunt, J. Günther, S.G. Parker, S. Boulos, P. Shirley, and T. Ize. State of the art in ray tracing animated scenes. In *Eurographics 2007 State of the Art Reports*, pages 89–116, 2007.

[WMHL65] E. Woodcock, T. Murphy, P. Hemmings, and S. Longworth. Techniques used in the GEM code for Monte Carlo neutronics calculations in reactors and other systems of complex geometry. In *Conference on the Application of Computing Methods to Reactor Problems 1965*, pages 557–579, 1965.

[WMS06] S. Woop, G. Marmitt, and P. Slusallek. B-kd trees for hardware accelerated ray tracing of dynamic scenes. In *Graphics Hardware 2006*, pages 67–77, 2006.

[WN09] C. Wyman and G. Nichols. Adaptive caustic maps using deferred shading. In *Eurographics 2009*, pages 309–318, 2009.

[Woo90] A. Woo. *Graphics Gems*, chapter Fast Ray-Box Intersection, pages 395–396. Academic Press, 1990.

[WPSAM10] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *IEEE International Symposium on Performance Analysis of Systems and Software 2010*, pages 235–246, 2010.

[WR08] C. Wyman and S. Ramsey. Interactive volumetric shadows in participating media with single-scattering. In *RT 2008*, pages 87–92, 2008.

[WRC88] G.J. Ward, F.M. Rubinstein, and R.D. Clear. A ray tracing solution for diffuse interreflection. In *ACM SIGGRAPH 1988*, pages 85–92, 1988.

[WS03] M. Wand and W. Straßer. Real-time caustics. In *Eurographics 2003*, pages 611–620, 2003.

[WSBW01] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive rendering with coherent ray tracing. In *Eurographics 2001*, pages 153–164, 2001.

[WSE04] D. Weiskopf, T. Schafhitzel, and T. Ertl. GPU-based nonlinear ray tracing. In *Eurographics 2004*, pages 625–633, 2004.

[WvG92] J. Wilhelms and A. van Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11(3):201–227, 1992.

[WWZ+09] R. Wang, R. Wang, K. Zhou, M. Pan, and H. Bao. An efficient GPU-based approach for interactive global illumination. In *ACM SIGGRAPH 2009*, pages 91:1–91:8, 2009.

[Wym05] C. Wyman. An approximate image-space approach for interactive refraction. In *ACM SIGGRAPH 2005*, pages 1050–1053, 2005.

[Wym08] C. Wyman. Hierarchical caustic maps. In *i3D 2008*, pages 163–171, 2008.

[WZHB09] B. Walter, S. Zhao, N. Holzschuch, and K. Bala. Single scattering in refractive media with triangle mesh boundaries. In *ACM SIGGRAPH 2009*, pages 92:1–92:8, 2009.

[XSXZ07] Q. Xu, M. Sbert, L. Xing, and J. Zhang. A novel adaptive sampling by Tsallis entropy. In *CGIV 2007*, pages 5–10, 2007.

[YFSZ06] M. Yang, G. Fei, M. Shi, and Y. Zhan. A simple, efficient method for real-time simulation of smoke shadow. In *International Conference on Artificial Reality and Telexistence 2006*, pages 633–642, 2006.

[YWC+10] C. Yao, B. Wang, B. Chan, J. Yong, and J.-C. Paul. Multi-image based photon tracing for interactive global illumination of dynamic scenes. In *EGSR 2010*, pages 1315–1324, 2010.

[ZC03] C. Zhang and R. Crawfis. Shadows and soft shadows with participating media using splatting. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):139–149, 2003.

[ZGHG08] K. Zhou, M. Gong, X. Huang, and B. Guo. Highly parallel surface reconstruction. Technical Report MSR-TR-2008-53, Microsoft Research, 2008.

[ZHG+07] K. Zhou, Q. Hou, M. Gong, J. Snyder, B. Guo, and H.-Y. Shum. Fogshop: Real-time design and rendering of inhomogeneous, single-scattering media. In *Pacific Graphics 2007*, pages 116–128, 2007.

[ZHWG08] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time kd-tree construction on graphics hardware. In *SIGGRAPH Asia 2008*, pages 126:1–126:11, 2008.