

PATH GUIDING METHOD FOR WAVEFRONT PATH TRACING: A MEMORY
EFFICIENT APPROACH FOR GPU PATH TRACERS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

BORA YALÇINER

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF DOCTOR OF PHILOSOPHY
IN
COMPUTER ENGINEERING

MARCH 2024

Approval of the thesis:

**PATH GUIDING METHOD FOR WAVEFRONT PATH TRACING: A
MEMORY EFFICIENT APPROACH FOR GPU PATH TRACERS**

submitted by **BORA YALÇINER** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Halil Kalıpçılar
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Halit Oğuztüzün
Head of Department, **Computer Engineering**

Prof. Dr. Ahmet Oğuz Akyüz
Supervisor, **Computer Engineering, METU**

Examining Committee Members:

Prof. Dr. Uğur Gündükbay
Computer Engineering, Bilkent University

Prof. Dr. Ahmet Oğuz Akyüz
Computer Engineering, METU

Prof. Dr. Yusuf Sahillioğlu
Computer Engineering, METU

Prof. Dr. Tolga Kurtuluş Çapın
Computer Engineering, TED University

Assoc. Prof. Dr. Elif Sürer
Graduate School of Informatics, METU

Date:04.03.2024

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Bora Yalçın

Signature :

ABSTRACT

PATH GUIDING METHOD FOR WAVEFRONT PATH TRACING: A MEMORY EFFICIENT APPROACH FOR GPU PATH TRACERS

Yalçın, Bora

Ph.D., Department of Computer Engineering

Supervisor: Prof. Dr. Ahmet Oğuz Akyüz

March 2024, 114 pages

In this thesis, we propose a path-guiding algorithm to be incorporated into the wavefront style of path tracers. As the wavefront technique of path tracers is primarily implemented on Graphics Processing Units (GPUs), the proposed method aims to leverage the capabilities of GPUs while reducing the hierarchical data structure usage and memory requirements necessary for path-guiding methods. To achieve this, we propose a wavefront path guiding algorithm that only stores the outgoing irradiance (radiant exitance) on a single global Sparse Voxel Octree (SVO) data structure. Probability density functions required to guide the rays are generated *on-the-fly* using this data structure. The on-the-fly generation of the probability field is made practical by utilizing the ray-tracing hardware of the latest GPUs. Furthermore, the proposed approach significantly reduces the persistent memory requirements compared to other state-of-the-art path-guiding techniques. Comparisons suggest that the proposed method requires less memory while maintaining similar results compared to the state-of-the-art techniques.

Keywords: path tracing, GPGPU, ray tracing, light transport, hardware acceleration

ÖZ

CEHPHE BAZI YOL İZLEME TEKNİKLERİ İÇİN YOL REHBERLİĞİ YÖNTEMİ: GRAFİK İŞLEMCİ TABANLI YOL İZLEYİCİLERİ İÇİN BELLEK VERİMLİLİĞİ SAĞLAYAN BİR YAKLAŞIM

Yalçın, Bora

Doktora, Bilgisayar Mühendisliği Bölümü

Tez Yöneticisi: Prof. Dr. Ahmet Oğuz Akyüz

Mart 2024 , 114 sayfa

Bu tezde, cephe bazlı yol izleyicileri için bir yol yönlendirme algoritması öneriyoruz. Cephe bazlı yol izleyiciler öncelikle Grafik İşleme Birimlerinde (GPU'lar) uygulandığından; önerilen yol yönlendirme yöntemi de Grafik İşlemcilerinin yeteneklerini ön plana çıkaracak şekilde ve aynı zamanda gerekli olan hiyerarşik veri yapısı kullanımını ve bellek gereksinimlerini azaltmayı amaçlamaktadır. Bunu başarmak için, tek bir küresel seyrek voksel barındıran sekizli ağaç (SVO) tabanlı veri yapısında yalnızca giden ışınımı (radyant çıkış) depolayan bir dalga cephesi yol yönlendirme algoritması öneriyoruz. Işınları yönlendirmek için gereken olasılık yoğunluk fonksiyonları, bu veri yapısı kullanılarak simülasyon sırasında oluşturulur. Olasılık alanının anında oluşturulması, en yeni grafik işlemcilerin ışın izleme donanımı kullanılarak pratik hale getirilmiştir. Ayrıca, önerilen yaklaşım, diğer son teknoloji yol yönlendirme tekniklerine kıyasla kalıcı bellek gereksinimlerini önemli ölçüde azaltır. Karşılaştırmalar, son teknoloji tekniklere kıyasla benzer sonuçları korurken, daha az bellek gerektirdiğini göstermektedir.

Anahtar Kelimeler: yol izleme, grafik işlemcilerde genel hesaplama, ışın izleme, ışık ulaştırma, donanımsal ivmelendirme

To all people who like to see pretty pictures on a screen

ACKNOWLEDGMENTS

I would like to thank the thesis committee members Prof. Dr. Uğur Gdkbay and Prof. Dr. Yusuf Sahilliođlu for their valuable comments over multiple committee meetings, especially to Prof. Dr. Yusuf Sahilliođlu inducing idea that became the basis of this thesis, although unpurposefully.

Additionally, I would like to thank my advisor Prof. Dr. Ahmet Ođuz Akyz for his insightful comments and his experience that guided me throughout my PhD education. His lecture named “Special Topics: Advanced Ray Tracing” is the reason for me to change my MSc-related research of real-time computer graphics to offline-focused ray tracing.

Some of the reference images presented in this thesis’s Figures were generated at TUBITAK ULAKBIM, High Performance and Grid Computing Center (TRUBA resources). The CPU reference images and multiple of the CPU-based proposed methods’ images are generated using this multi-core system. Even then, generating some of these images took multiple days to complete. Thus, we appreciate the capability of the distributed system.

TABLE OF CONTENTS

ABSTRACT	v
ÖZ	vii
ACKNOWLEDGMENTS	x
TABLE OF CONTENTS	xi
LIST OF TABLES	xv
LIST OF FIGURES	xvi
LIST OF ABBREVIATIONS	xviii
CHAPTERS	
1 INTRODUCTION	1
1.1 Motivation and Problem Definition	1
1.2 Contributions and Novelties	2
1.3 The Outline of the Thesis	2
2 LITERATURE SURVEY	5
2.1 Rendering Equation	5
2.1.1 Terminology	7
2.2 Rendering Methodologies	8
2.2.1 Monte Carlo Integration and Path Tracing	9
2.2.1.1 Multiple Importance Sampling	9

2.2.1.2	Path Tracing	11
2.2.1.3	Next Event Estimation	11
2.2.1.4	Russian Roulette Path Termination	14
2.2.2	Bi-Directional Path Tracing	14
2.2.3	Metropolis Light Transport or Markov Chain Monte Carlo	16
2.2.4	Photon Mapping	18
2.2.5	Instant Radiosity & Virtual Point Lights	20
2.2.6	Path Guiding	22
2.2.7	Conclusion	24
2.3	Massively Parallel Architectures and GPGPU	27
2.3.1	Design Differences between CPUs and GPUs	28
2.4	Sparse Voxel Octrees & Cone Tracing	30
2.5	GPU Oriented Light Transport Proposal	32
3	PARTITION BASED WAVEFRONT PATH TRACING	33
3.1	Preliminaries	33
3.1.1	Taxonomy	35
3.2	GPU Oriented Parallel Design	36
3.2.1	Case Study: Reduction	36
3.2.2	Parallel Reduction	37
3.2.3	Massively Parallel Reduction	38
3.3	Path Tracing on the GPU	40
3.4	Wavefront Path Tracing	42
3.4.1	Queue-based Partitioning	45

3.4.1.1	Memory Management Issue	46
3.4.2	Consistency Issue	47
3.4.3	Sort-based Partitioning	48
3.4.3.1	Ray Payload & Key Parameter	48
3.4.3.2	The Algorithm	51
3.5	Final Words	53
4	WAVEFRONT PATH GUIDING	55
4.1	Brief Refresh of Path Guiding	55
4.2	Overview	57
4.3	Radiant Exittance Caching using Sparse Voxel Octree Structure	59
4.4	On-the-fly Generation of Radiance Field	63
4.4.1	Partitioning	63
4.4.2	Radiance Field Generation	65
4.5	Exposing BxDF Product the Radiance Field	69
5	IMPLEMENTATION AND RESULTS	73
5.1	Implementation	73
5.2	Parametrization	75
5.3	Path Guiding Visualization Tool	77
5.4	Profiling	80
5.5	Baseline Comparison	82
5.6	Comparison with Literature	84
5.7	Product Path Guiding	89
5.8	Limitations and Future Work	89

6 CONCLUSIONS	93
6.1 GPU Limitations	94
6.2 Final Words	94
REFERENCES	97
CURRICULUM VITAE	111

LIST OF TABLES

TABLES

Table 5.1	Timings of the wavefront path guiding stages.	81
Table 5.2	Comparisons between Ruppert et al.'s proposal and our method over two scenes.	87

LIST OF FIGURES

FIGURES

Figure 2.1	Illustration of two PDFs approximating different function domains.	10
Figure 2.2	A case demonstrating when NEE fails to improve the resulting scene.	12
Figure 2.3	Different kinds of path tracing techniques.	15
Figure 2.4	Photon mapping noise pattern.	19
Figure 2.5	Showing unique noise pattern of Virtual Point Light Techniques.	21
Figure 2.6	Specular-Diffuse-Specular interaction demonstration.	25
Figure 2.7	Multiple path integral approximation techniques.	26
Figure 2.8	Highly simplified block layout of a General purpose CPU.	28
Figure 2.9	Highly simplified block layout of a GPU.	30
Figure 3.1	C++ code snippet for reduction, using arithmetic add operation. .	36
Figure 3.2	C++ code snippet for traditional parallel reduction, using arithmetic add operation.	37
Figure 3.3	C++ code snippet for massively parallel reduction, using arithmetic add operation.	39
Figure 3.4	Wavefront path tracing algorithm overview.	43
Figure 3.5	The payload of a ray.	49

Figure 3.6	The material type key structure.	49
Figure 3.7	Array representation of the binary partitioning scheme and marking algorithms.	52
Figure 3.8	The final result of the sorting-based partitioning algorithm.	52
Figure 4.1	The top-down view of the entire path guiding algorithm.	58
Figure 4.2	Morton code encoding visualization.	61
Figure 4.3	K-means clustering algorithm sketch.	62
Figure 4.4	An example of generated marginal and conditional PDFs from the radiance field.	68
Figure 5.1	Aliasing illustration.	74
Figure 5.2	Learned or Generated Radiance field PDF of our method and practical path guiding method.	78
Figure 5.3	Convergence of Müller et al.’s method and our method.	79
Figure 5.4	Single sample variance of the proposed and traditional path-tracking methods.	83
Figure 5.5	Mean FLIP comparisons of the experimented sampling techniques.	85
Figure 5.6	Comparison between classical path tracing and Practical Path Guiding method of Müller et al.	88
Figure 5.7	Demonstration of what product path guiding prevents.	90

LIST OF ABBREVIATIONS

2D	2 Dimensional
3D	3 Dimensional
SVO	Sparse Voxel Octree
MIS	Multiple Importance Sampling
WFPG	Wavefront Path Tracing
WFPT	Wavefront Path Guiding
GPU	Graphics Processing Unit
CPU	Central Processing Unit
GPGPU	General Purpose Graphics Processing Unit

CHAPTER 1

INTRODUCTION

1.1 Motivation and Problem Definition

Path tracing techniques have become widely adopted in the field of computer graphics for creating highly realistic images [1]. These methods are preferred due to their ease of implementation and the high quality of images they produce. Moreover, with the advancements in graphics hardware, it is now possible to achieve interactive rendering using these techniques.

Although most of the state-of-the-art methods can be adapted to the graphics hardware, ground-up designed GPU algorithms for light transport simulation are not thoroughly researched. One reason is that hardware capability for light transport simulation, especially ray tracing-based capabilities, is very recent. Another reason is that GPUs are inherently real-time focused devices, which does not suit the complex nature of the light transport problems. Real-time systems either pre-generate complex light transport phenomena or rely on ad-hoc methods to fastly approximate indirect phenomena due to computation time limitations.

The recent shift towards ray tracing for real-time graphics is promising. Modern hardware can accelerate ray tracing in silicon. This hardware acceleration enables interactive light transport simulation that captures all light transport-related phenomena such as global illumination, indirect lighting, and caustics. Even real-time simulations that do not rely on ad-hoc methods or pre-computation are starting to emerge.

A traditional approach that simulates light in an unbiased way is path tracing. Path tracing can encapsulate all of the light transport phenomena in a simple and elegant

way. The path tracing approach is a de-facto standard for offline rendering cases in which CPUs are utilized for computation. Unlike designing a path tracing structure for CPUs, designing a GPU-based path tracing scheme is not straightforward due to differences between hardware.

Thus, ground-up designing a GPU-based light transport scheme that utilizes the recent GPUs' hardware accelerated ray tracing capabilities for the path tracing logic and accompanying methodologies that further accelerate light transport simulation is our main motivation.

1.2 Contributions and Novelties

To this end, we propose a wavefront path tracing scheme that utilizes sorting-based partitioning and an accompanying wavefront path guiding algorithm to utilize the GPU. Our main contributions are as follows;

- For GPU path tracing, a radix sort-based partitioning scheme enables consistent memory allocation.
- On-the-fly generation of radiance field by utilizing incident radiant exitance, which resides on Sparse Voxel Octree.
- Hardware-accelerated approximate cone tracing for an efficient query of the radiant exitance.
- GPU-oriented parallel product path guiding scheme that utilizes warp-level intrinsics.
- A heuristic that optimally combines the guided samples of the proposed method.

1.3 The Outline of the Thesis

This thesis will first explore the main family of light transport techniques in the literature. Furthermore, we will discuss general-purpose computing over graphics proces-

sor units. We conclude the first chapter by determining which family of light transport simulations is most suitable for a GPU-based method proposal.

The next chapter explains the parallelization schemes for the GPUs. We compare and contrast parallelization methodologies of GPUs and CPUs using a simple case. Moreover, we will discuss wavefront path tracing schemes and propose a novel sort-based path tracing algorithm for GPUs.

The third chapter proposes an accompanying path guiding technique for wavefront-style path tracers. This section will explain the proposed method thoroughly and express the design decisions required to implement the method using GPUs.

The fourth chapter will present the results of the proposal. We compare and contrast our proposal with the state-of-the-art path guiding techniques and express our observations.

Finally, we will conclude our findings by explaining the advantages and shortcomings of the. We will finalize the thesis with our concluding remarks.

CHAPTER 2

LITERATURE SURVEY

The light transport problem has been heavily researched in computer graphics literature for over 25 years. Although it has been researched heavily, proposals for robust techniques are still desired due to the complex nature of the light transport problem.

In this chapter, we will discuss the light transport problem. Then, we will explain the main types of algorithms to approximate this complex light transport problem and the accompanying techniques for these algorithms. Finally, we try to summarize what can be further improved for all of the explained algorithms.

Furthermore, the design principles of GPU-based light transport algorithms will be discussed. Such algorithms will be utilized to implement state-of-the-art rendering techniques and the novel technique proposed for this Ph.D. thesis.

2.1 Rendering Equation

The rendering equation can be considered a fundamental definition of the light contribution of a certain point on a surface [1]. The rendering equation is defined in angular form as below (Equation 2.1).

$$L_o(p, w_o) = L_e(p, w_o) + \int_{\Omega} f_x(p, w_i, w_o) L_i(p, w_i) \cos \theta_i dw_i \quad (2.1)$$

The rendering equation is an energy balance equation which is defined as all outgoing radiance $L_o(p, w_o)$ from a surface point should be equal to all incoming radiance towards that surface point and emitted radiance $L_e(p, w_o)$ if available.

Incoming Radiance depends on three main concepts. First is the incoming radiance $L_i(p, w_i)$, which is the radiance contribution from other objects throughout the scene. Second is the cosine term that exposes Lambert’s Cosine Law. Lambert’s Cosine Law states that incoming radiance contribution depends on the angle between the surface normal and radiance direction. Finally, the last term $f_x(p, w_i, w_o)$ is a bi-directional distribution function (BxDF). In the context of different surface compositions, BxDFs are further defined as bi-directional “reflectance” distribution function (BRDF) for opaque objects, bi-directional “scattering” distribution function (BSDF) for participating media, and bi-directional “subsurface scattering” distribution function (BSSRDF). Since all materials technically scatter light, “Scattering” can be used as a superset. All in all, the BxDF function defines how much of that incoming radiance is transferred towards that outgoing direction w_o .

Even if it is defined for a single surface, the rendering equation is computationally complex. Furthermore, each surface’s incoming radiance depends on the outgoing radiance of the other surfaces. If we assume the volume between objects is void, radiance along distances will not change. Then, we can recursively define a global equation that represents the entire radiance contribution of the scene over a single surface point.

In the Physically Based Rendering Book, such a definition is represented as in Equations 2.2 and 2.3 [2]. Although these equations are defined in the geometric form in the Physically Based Rendering Book, the hemispherical form is chosen for this thesis due to its clarity.

Equations 2.2 and 2.3 define the recursive integral in geometric form instead of a hemispherical form, which is the form of Equation 2.1. Both of these forms are equivalent to each other.

$$L(p_1, w_o) = \sum_{n=1}^{\infty} P(\bar{p}_n) \tag{2.2}$$

$$\begin{aligned}
P(\bar{p}_n) = & \underbrace{\int_{\Omega} \int \cdots \int_{\Omega}}_{n-1} L_e(p_n, w_{o_{n-1}}) \\
& \times \left(\prod_{j=1}^{n-1} f_x(p_{j+1}, w_{o_j}, w_{i_{j-1}}) \right) dw_{i_2} \cdots dw_{i_n}
\end{aligned} \tag{2.3}$$

Each $P(\bar{p}_n)$ represents n^{th} depth of the recursion. On each depth, chained integrals represent all incoming radiance from surfaces with exactly n bounces. Discretization of these chained integrals is called *paths*. Product term $\prod_{i=1}^{n-1} \cdots$ is often called the *path throughput*, which is the term that determines how much energy from the energy source is culled throughout the bounce process. Throughput will be important for Monte Carlo integration (see Section 2.2.1).

Technically, $L(p_1, w_o)$ can be defined on any scene surface point; however, it is most useful to define it from the camera sensor. Each camera sensor’s 2D discretization point (pixel) will be computed as a chain of integrals to determine how much radiance is accumulated over that pixel.

Equations 2.2 and 2.3 show how the light transport problem is mathematically complex. It is an infinite recursion with explicit base conditions (meaning that energy termination points change with respect to the scenes), and each recursion depth has chained integrals. Although this representation is complex, it is the mathematically profound equation for a given point that encapsulates all light-related phenomena. All upcoming techniques that will be discussed try to find an approximation to this equation.

2.1.1 Terminology

In order to explain the phenomena related to the light transport problem, we will define the terminology to ease the discussion. BRDF (BxDF with “reflectance” behavior, meaning transport occurs on the hemisphere) functions can be segmented into two main categories, *diffuse* and *specular*. Diffuse reflections uniformly scatter the incoming radiance throughout the entire hemisphere. On the other hand, specular reflections nearly fully obey the law of reflection, meaning that outgoing radiance is concentrated over the reflection of the incoming radiance. We will also inter-

change the term “material” with the BxDF function. Transmitting materials such as dielectrics (i.e., glass) can also behave specularly.

Additional material terms (like glossy materials) exist between diffuse and specular materials that exhibit reflectance behaviors similar to specular surfaces but disperse light even more around the reflected direction. However, we won’t delve into these definitions for this discussion. To illustrate, surfaces like chalk can be classified as diffuse, while mirrors fall into the specular category.

Caustics is a light transport phenomenon that occurs when light is reflected/transmitted through a single or series of specular objects over a diffuse surface. We mention this phenomenon in our discussion due to its inherent challenge when calculating using various methodologies.

Another term we want to explain is “scene dependency”. Scene-dependent phenomena or behavior means that it is hard to encapsulate such occurrence beforehand (i.e., scene initialization time) is non-trivial. For example, caustics are a scene-dependent phenomenon; given the scene’s parameters, determining the caustic regions would require computations as expensive as the light transport simulation. On the other hand, simpler forms of emitter sampling (see Section 2.2.1.3) are “scene-independent” meaning you can construct data structures beforehand for directly sampling the emitters and acquire acceptable improvements of image quality.

2.2 Rendering Methodologies

Almost all of the algorithms proposed throughout the literature try to approximate a solution to the rendering equation in one way or another. In this section, we will explore computational techniques employed to address the rendering equation, along with the diverse discretization approaches these methods utilize. Ultimately, the process of computation necessitates some form of discretization.

It should be noted that only the main light transport family of techniques is explained in this literature survey. There are various methods for light transport techniques that are not covered here. One main example is gradient-domain rendering for this family

of algorithms; we kindly refer the reader to Hua et al.’s survey of such methods [3].

2.2.1 Monte Carlo Integration and Path Tracing

Monte Carlo integration can be considered a fundamental numeric integral approximation algorithm. The Monte Carlo integration algorithm uniformly samples the integrand space and averages the results. An example of the Monte Carlo numeric approximation for a simple single integral can be on Equation 2.4. With enough samples N , the result will converge to the actual integral result.

$$\int_A f(x)dx = \frac{1}{N} \sum_{i=1}^N f(x_i) \quad (2.4)$$

Additionally, better to integrate with a probability density function (PDF), which roughly resembles the integrand. Equation 2.5 is equivalent to Equation 2.4 as long as the function $p(x)$ is non-zero where the actual function is non-zero over the domain of the integral. Additionally and by definition, PDF should integrate to one ($\int_A p(x) = 1$) over its domain.

$$\int_A f(x)dx = \frac{1}{N} \sum_{i=1}^N \frac{f(x_i)}{p(x_i)} \quad (2.5)$$

This process is called “importance sampling”. Such an approach could dramatically decrease the convergence time of the computation to an acceptable level. However, convergence time can be worse if a PDF that does *not* resemble the integrand results. Especially when the PDF is slightly off due to heuristical approximations. This observation will be important in Section 4,

2.2.1.1 Multiple Importance Sampling

It is not always practical to find a single function that can resemble the integrand and be able to sample from it. In this case, utilizing multiple PDFs would yield better results. The combination technique without any bias is called *Multiple Importance Sampling (MIS)* [4]. A basic example of this can be seen in Figure 2.1

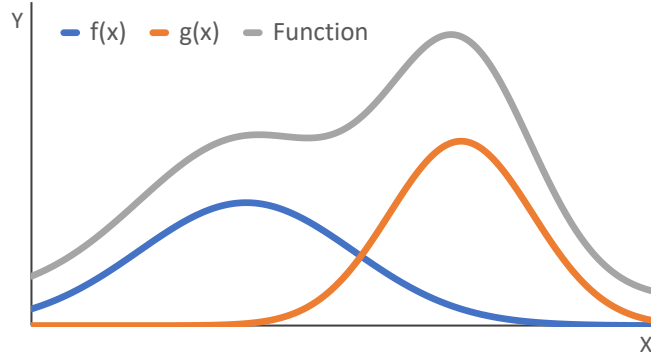


Figure 2.1: Illustration of two PDFs approximating different function domains. A one-dimensional function (the gray line) and two PDFs (in this case, two Gaussian functions) are represented as blue and orange lines. The integrand function closely resembles the combination of the two Gaussian. Using the Monte Carlo method to calculate the integral with only one of these PDF functions is ill-advised, but using them together will provide better results.

MIS uses weighting functions to combine multiple PDFs called *heuristics*. Veach et al. proposed balanced, power, cut-off, and maximum heuristics [4, 5]. Equation 2.6 shows the combination of two different estimators using weighting function $w(x)$ assuming $\int_{\Omega} f(x)dx$ is being approximated via two different sampling strategies with different PDFs.

$$\frac{1}{n_{p_1}} \sum_{i=1}^{n_{p_1}} \frac{f(X_i)w_{p_1}(X_i)}{p_1(X_i)} + \frac{1}{n_{p_2}} \sum_{i=1}^{n_{p_2}} \frac{f(Y_i)w_{p_2}(Y_i)}{p_2(Y_i)} \quad (2.6)$$

This method is not limited to only two PDF merges; multiple different PDFs can be combined using this way. The most popular weighting function is the balanced heuristic, and the formulation is given in Equation 2.7. n is the number of samples of each estimator.

$$\omega_s = \frac{n_s p_s(x)}{\sum_i^K n_i p_i(x)} \quad (2.7)$$

The balanced heuristic is usually optimal and reduces variance [4]. However; research about how other heuristics can be utilized while reducing variance in certain situations can be found in literature [6]. MIS is the fundamental variance reduction technique for path tracing algorithms due to its flexibility and extensibility.

2.2.1.2 Path Tracing

The Monte Carlo method is traditionally applied to light transport simulations as follows. While sampling the light integral, incremental construction of the path is randomly generated from the camera toward the scene. Each integral (Equation 2.1) is approximated via the Monte Carlo Method. When an emitting source is encountered, the total radiance contribution is calculated and stored on an image. Traditionally, the entire BxDF or a portion of the BxDF is utilized as a PDF for importance sampling. Such an approach is called *forward path tracing*.

Forward path tracing ensures that each random exploration of the path integral (Equation 2.3) is guaranteed to contribute to the final image by definition. However, the actual physical process is reversed; light travels from the emitting source ends on the camera sensor. Such an approach is applicable because of the bi-directional nature of the distribution function. The name “bi-directional distribution function” suggests internal scattering functions are reversible; additionally, outgoing radiance and incoming radiance are equal, assuming the traversing medium does not affect light (i.e., vacuum). Even if the traversing medium affects the radiance, its distance-related absorption/scattering function is also reversible.

2.2.1.3 Next Event Estimation

As one can observe, the main radiance contribution towards any surface should come from directly visible light sources (emitters). Due to this observation, directly sampling emitting sources throughout the scene and advancing paths is considered [7]. Such a method is called “next event estimation”. This method is also called “shadow ray casting” in real-time computer graphics literature. A direct emitter is sampled on every surface during the path construction, and its contribution is accumulated. One should take caution when a path construction encounters the sampled emitter; path construction should not accumulate its contribution due to double-counting.

For basic NEE, one can utilize a list of emitters as an array, which can be easily constructed during initialization time. On every surface, as in Monte Carlo Fashion, the NEE sampler can randomly select an emitter and sample from it while incorporating

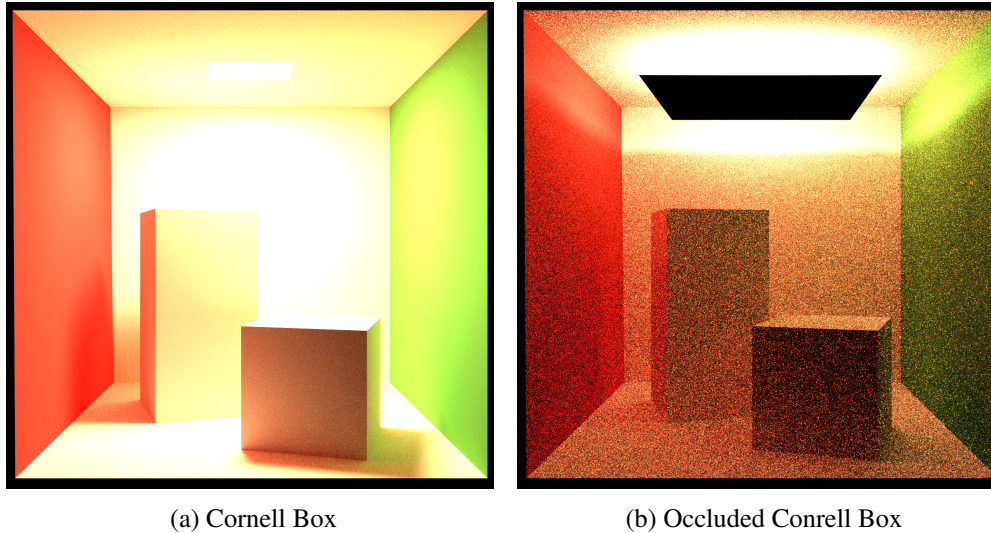


Figure 2.2: A case demonstrating when NEE fails to improve the resulting scene. Both images are generated using the same computation time using path tracing with NEE on. a shows that the scene converges well due to the mostly direct light contributions. On the other hand, in b, most of the surfaces are indirectly lit; thus, NEE does not improve image quality.

such discrete selection probability as importance sampling fashion.

Even in this simple form, NEE can dramatically improve image quality. However, it has a potential fallback when the scene is illuminated mostly by indirect illumination. Since the NEE is a direct sampling technique, regions that receive illumination indirectly will not benefit from this method.

Another problem with NEE is that in its most basic form, occlusion information is not incorporated into the sampling scheme due to its complexity. Occlusion information is scene-dependent; thus, it may require pre-processing or an on-the-fly approximation of the occlusion field. Guo et al. proposed a dense global visibility (occlusion) field for such purpose [8].

Specular surface reflected/refracted direct light sources are common in natural-looking scenes. For example, light bulbs have a specular refracting surface that encapsulates the actual light source. NEE will fail for such orientations since the actual light source is technically not directly visible. Such interaction minimally affects the energy of the source; thus, it may be the dominant contribution source and should be sampled. For such occurrences, Johannes et al. proposed Manifold exploration NEE in which a

light source is sampled and specular to specular interreflections (manifolds) are iteratively constructed [9]. Proper importance sampling weights were then incorporated. Different methods for tackling specular reflection exposed light source sampling also exist [10, 11].

Light Sampling (NEE) for volumetric mediums (participating media) is also a research area in computer graphics. Kulla and Farjardo proposed an equiangular sampling technique when sampling a light source inside of a participating media [12]. This method suggests an inverse square falloff for the pdf function for the importance sampling scheme. Another method for sampling light sources for participating media is proposed by Johannes et al. [13] in which an additional vertex is introduced to NEE calculations. This method can be considered a light transport analog of the “once-more collided flux estimation” for neutron transport simulations.

Another research area for NEE estimation is scalability. As the emitter count increases, the amount of calculation for direct light also increases. The emitters’ orientation and distance toward the sampled surface should be incorporated into the NEE sampling scheme. To this end, Estevez et al. proposed a BVH-like tree structure in which lights are partitioned with respect to their characteristics as cones [14]. A similar approach is incorporated for the GPU as well [15].

Bitterli et al. proposed a state-of-the-art GPU-oriented NEE scheme that utilizes reservoir sampling [16]. This sampling scheme is suited for the GPU since it has a static data structure that is a series of per-pixel reservoirs. The intuition of this method is to collaboratively utilize the neighboring reservoirs for sampling in addition to the calculated pixel’s reservoir. This approach reduces the variance of the direct light estimator using additional data without the extra work.

Overall, the NEE sampling scheme is a fundamental method that improves the quality of the image with minimal computational overhead. It is also required for analytic light sources, especially for point lights, since traditional Monte Carlo path tracing could not hit such lights.

2.2.1.4 Russian Roulette Path Termination

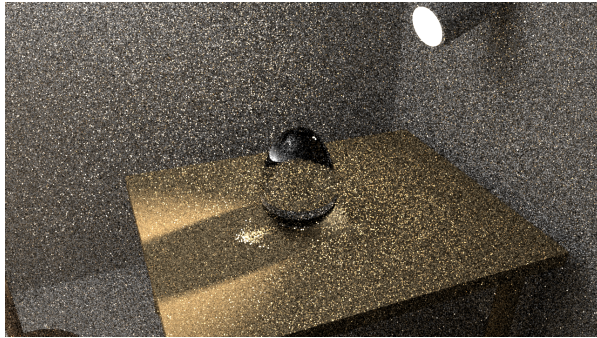
Equation 2.2 is an infinite-length integral that is not practical to calculate. One may statically determine a culling point up to a certain depth to eliminate this practical problem. However, every path is different because, after multiple bounces, some paths may not be in a state of negligible energy. In contrast, the energy contributions of other paths would have converged to zero. To dynamically cull paths with respect to a path-dependent parameter is called *Russian Roulette*, which was first introduced by Arvo et al. [17]. As the name suggests, paths are stochastically culled with respect to a certain parameter; in traditional cases, path throughput or surface albedo is used [17]. Importantly, unculled paths are weighted by this probabilistic function in order to preserve the unbiasedness of the Monte Carlo techniques. While it allows practical traversal of the entire path space in an unbiased manner, this method increases the variance. Another way to terminate paths is to check the total variance over the image and terminate accordingly [5].

Newly emerged research improves over the traditional termination methods by using paths' total expected contribution from a reference radiance field as a termination probability [18]. This is especially useful for path guiding techniques (See Section 2.2.6) since such methods' radiance field is readily available.

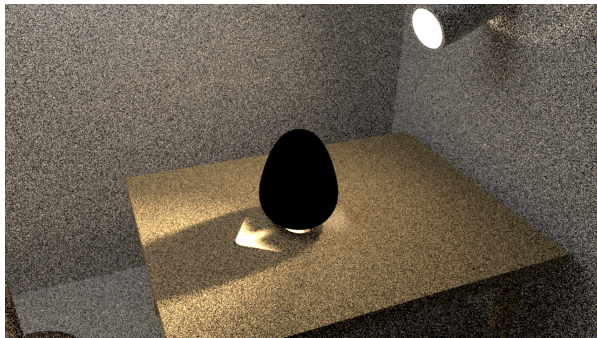
Russian roulette is another fundamental technique for path tracing. It tackles the practical limitations of the light transport simulation in an unbiased manner.

2.2.2 Bi-Directional Path Tracing

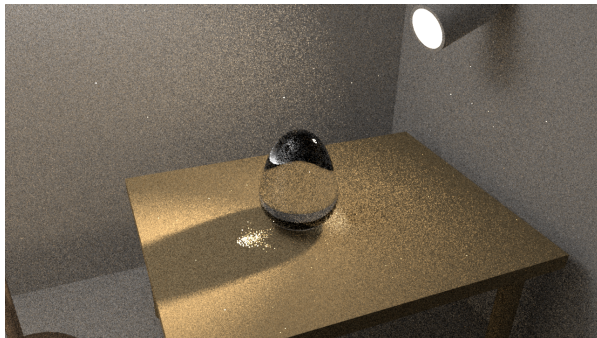
For caustics, forward path tracing has a hard time sampling such light paths (see Figure 2.3a) efficiently. The main reason for this is that local importance sampling techniques (material-related techniques) do not contribute to the occurrence of caustic phenomena. To be effective, the caustic receiving surface should sample the next path location using the incoming radiance field. Approaches that incorporate this incoming radiance field as an importance sampling method are called “path guiding” methods and will be discussed in Section 2.2.6.



(a) Forward Path Tracing



(b) Light Path Tracing



(c) Bi-directional Path Tracing

Figure 2.3: Different kinds of path tracing techniques. Each pixel is sampled with eight paths. This figure shows the discrepancies between techniques with respect to caustics. Best caustics are generated on light tracing. However, it cannot sample paths that are behind a specular object. On the other hand, the forward path tracer had bad caustic convergence. Bi-directional path tracers converge better overall.

One may argue that reversing the traversal scheme to light-to-camera would improve the caustics seen in Figure 2.3b). In this case, NEE works reversely and tries to sample the camera sensor to calculate the image contribution. However, as Figure 2.3b suggests, surfaces that are “occluded” by a specular transmissive surface could not be

sampled. The camera is a pinhole in these images, meaning the Monte Carlo sampler could not hit the camera stochastically, making these regions pitch black. This method is called “light tracing” or “light path tracing”.

A final argument can be the combination of forward and light path tracing. However, naively combining resulting images using simple averaging will not improve the results; noise from one method will creep into the combined image. Formally, simple averaging will not decrease the variance.

Bi-directional path tracing method tries to achieve this efficient combination of light tracing and path tracing [5, 19, 20]. Bi-directional path tracing *simultaneously* traces paths from both ends, and light contributions along all traced surfaces with respect to each other are combined (see Figure 2.3c). More importantly, bi-directional path tracing selectively combines such paths using MIS. Additionally, utilizing generated paths between different eye paths using MIS is also introduced [21]. Extensions for incorporating the participating media are also available [22].

2.2.3 Metropolis Light Transport or Markov Chain Monte Carlo

Metropolis Light Transport (MLT) technique was first introduced to the graphics literature by Veach et al. [23]. Generic explanation (not light path integral related) is proposed by Metropolis et al. [24] and Hastings [25].

In the most basic explanation, given an initial state and an integral, the algorithm generates another state solely depending on the previous state. Such a state is accepted or rejected, given a probability. Just like fundamental Monte Carlo, when enough states are traversed, the algorithm converges to the result of the integral. This hierarchy of states is defined as a *Markov Chain*, and the process of generating a Markov Chain is called “doing a *random walk*”.

Veach et al. proposed the path space Markov Chain, where we start with an initial state, a connected path from the camera to the light. Using this initial state, the entire path space is traversed (see Equations 2.2 and 2.3) by mutating the path slightly.

Path space is thoroughly explored via two functionalities of this method. The first

method is *mutations* in which the algorithm introduces or removes vertices over the path with a probability. This method makes the algorithm jump between $P(\bar{p}_n)$ functions on Equation 2.2. The second method is to *perturb* the given vertices slightly. Through this method, surface space is explored over the current path length. It should be noted that the proposed method uses a single chain to explore the entire image by perturbing the camera plane vertex as well.

Choosing a good initial state is important to eliminate start-up bias. For this purpose, Veach et al. proposed to use bi-directional path tracing (see Section 2.2.1) for initial state generation. In literature, methods that use the path space to generate states are entitled as *Path Space Metropolis Light Transport (PSMLT)*.

Most of the research in this area revolves around two main sections. The first is to introduce MLT to different spaces aside from path space. Another research area is introducing better mutation (or perturbation) strategies for different BxDFs and/or path compositions.

One such proposal is Primary Sample Space MLT (PSSMLT) [26]. In this method, the sample space of the Markov Chain, which is the space defined by the uniform samples generated while creating the initial path, is used. These uniform numbers are perturbed to introduce mutations. Improvement of this method is *Multiplexed Metropolis Light Transport (MMLT)* in which information of multiple importance sampling is incorporated as well [27].

Different mutation strategies are also proposed for MLT. *Manifold exploration method (MEMLT)* is proposed to tackle hard light paths that consist of specular or near-specular surfaces especially SDS paths [28].

A combination of Monte Carlo path tracing and metropolis light transport is also researched. Instead of using a single initial state, the algorithm finds a path using path tracing methods; then, such a path undergoes a series of short burst mutations and contributes to the image.

Instead of using primary sample space or path space, one can use half vector space instead [29] [30]. Gradient domain MLT approaches also exist [31] [32]. In such a case, the domain is the differential domain, meaning that the rate of change of

radiance is important instead of the actual amount.

For example, if we consider the shadow region of an area of light, gradient-domain metropolis light transport will have fewer mutations over the fully lit and occluded regions. Instead, most of the computational power will be concentrated on the penumbra region.

It should be noted that MLT algorithms are delicate. Introducing a bad mutation strategy would result in high variance or biased results. Moreover, not all mutation strategies would work consistently over all different scenes or path compositions.

The main issue of MLT is the temporal coherency of images. When an animation is on the scene, successive frames' pixels may have high variance, resulting in flickering. This results from the MLT process' traverse nature, making the image rendered by an MLT algorithm converge unpredictably. However, preventing this temporal incoherence is a research area as well [33].

We can see that the MLT algorithms have an advantage over MC approaches in terms of available information. MC algorithms are unaware of the next hit location since iteration continues incrementally. On the other hand, MLT algorithms start with a valid path. With this additional information, an algorithm can decide how to construct additional paths more effectively than Monte Carlo integration.

2.2.4 Photon Mapping

Photon Mapping algorithm [34, 35] tries to include additional light information about the scene by using photons. Photons are scattered throughout the scene over the diffuse surface points as a first pass. On the second pass, incoming radiance is approximated using these photons along the surface points.

There are two key advantages of this method. Firstly, the noise generated by the photon mapping algorithm is low-frequency. Second, photon mapping decouples the scene's complexity from radiance calculation since generated photons are accumulated regardless of occlusion, making the resulting image's convergence independent of the scene complexity.



(a) Path Traced Reference



(b) Stochastic Progressive Photon Mapping

Figure 2.4: Photon mapping noise pattern. Camera pixels calculate contributions from neighboring photons; thus, circular noise patterns may emerge if the photon count is low. b is specifically generated using a low amount of photons to demonstrate the noise pattern.

The algorithm assumes that the radiance of surface points that have not received any photons is correlated with the density of neighboring photons. However, this assumption is not always valid, thus making this approximation biased.

Additionally, a high amount of photons ($\sim 10^6$) are required to have an accurately converged image. However, this requires a high amount of memory. For this approach, improvements are made, consisting of progressively discarding and generating new photons instead of generating them in bulk. This approach is called *Progressive Photon Mapping* [36]. Camera samples can also be stochastically generated, which is

called *Stochastic Progressive Photon Mapping* [37]. These make the memory requirement constant while having a theoretically infinite amount of photons.

A large leap towards making photon mapping unbiased is made by Hachisuka et al. and Gergiev et al. [38, 39]. The Vertex connection and merging (VCM) algorithm combines photon mapping and bi-directional path tracing. The algorithm conditionally accepts neighboring pre-generated bi-directional light paths of similar surface points if such path is in a certain radius r similar to Photon Mapping. Then, this acceptance radius is reduced gradually, converging to the bi-directional path tracing algorithm.

Additionally, there are implementations of photon mapping that utilize GPU for photon generation [40] [41] or final gathering [42].

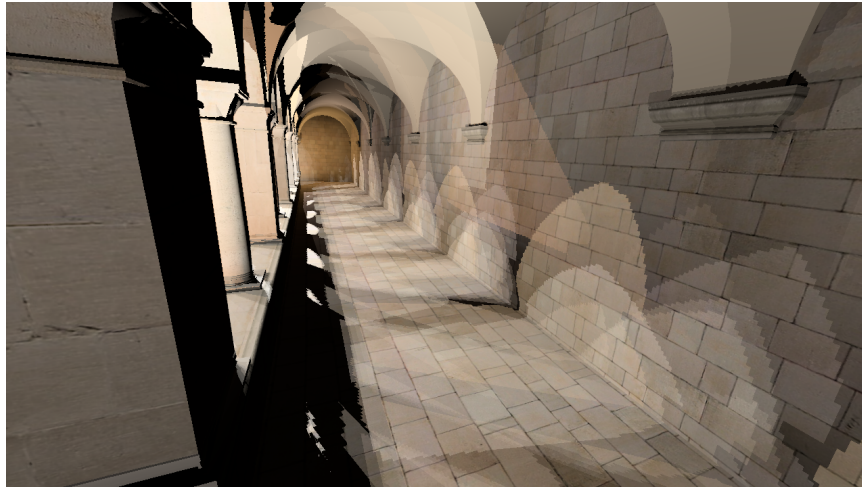
As shown in Figure 2.4, photon mapping techniques may have unique circular patterns due to the interpolation of light combinations of photons around the sampled camera ray. Such noise drops as the scene converges. However, the resulting images will be biased but consistent when generated by a stochastic progressive photon mapping method.

2.2.5 Instant Radiosity & Virtual Point Lights

In this section, we will explain Instant Radiosity (known as Virtual Point Lights (VPL) or many-light global illumination). First introduced by Keller, the Instant Radiosity method utilizes point lights to simulate global illumination. Many point light sources are introduced to the system by tracing light sub-paths of the primary light, and then a VPL is generated on that surface point. Secondly, each visible surface point is illuminated by all relevant point lights *directly*.

At first sight, one can assume that the instant radiosity algorithm is similar to the Photon Mapping algorithm, but there is a key difference. Photons in the photon mapping algorithm discretize *flux* of energy that is coming out of a light source. Instant radiosity algorithm discretizes the *radiance* along the light path using light sources.

Instant Radiosity has a unique noise pattern different from other approximation tech-



(a) 32 Virtual Point Lights



(b) 8192 Virtual Point Lights

Figure 2.5: Showing unique noise pattern of Virtual Point Light Techniques. Due to the hard shadow nature of the virtual point light, the under-sampled scene contains lines of different light gradients.

niques when less than optimal VPLs are present (see Figure 2.5). Shadows are not smooth because of the point light discretization nature of the algorithm since point lights have hard shadow edges. However, it smooths out with enough samples.

As the name suggests, the instant radiosity method does not handle specular objects naturally. Because of that, a hybrid approach is introduced by Udeshi et al. in order to approximate specular reflection and refraction [43]. However, it has limited performance in generating phenomena like caustics. Wald et al. generalize this algorithm

to support such phenomena [44]. Additionally, Virtual Ray Lights, an extension of this method, is proposed to overcome the singularities created by perfectly specular objects [45].

This algorithm is inherently acceleratable using GPU hardware [46, 47, 48]. After determining virtual point lights, GPU rasterization hardware can render the scene for each VPL and accumulate the result. Visibility of VPLs is determined by shadow maps [49], which can be accelerated by the GPU as well.

Since many lights are required to approximate the global rendering equation, acceleration structures, light segmentation, or caching are often essential for VPL algorithms. Traditionally, irradiance or radiance caching can be utilized [50], [51]. Data structure-based implementations to sample light sources are another research area [52, 53, 54, 55, 56]. GPU based many-light acceleration structures also exist [57, 58].

It is not possible to explain every research paper about virtual point lights. More comprehensive research is done by Dachsbacher et al., including VPL utilization for participating media and specular object handling [59]. Readers can refer to that paper for a more thorough explanation of many-light algorithms.

2.2.6 Path Guiding

As discussed in Section 2.2.1, traditional Monte Carlo Path tracing methods only rely on local portions of the rendering equation for importance sampling. New techniques have tried to overcome this problem of path tracing by exposing the global radiance field around the sampled location. It is not simple to sample the incoming radiance field since it is scene-dependent, and storing such information on a scene basis is required. Moreover, the global radiance field of the scene is a “super-set” of the camera-received radiance.

This family of algorithms is called *Path Guiding* techniques. Some implementations pre-generate such radiance field over the scene [60, 61, 62, 63] or progressively generate using already computed paths [64, 65] in order to estimate incoming radiance.

The key difference of these algorithms compared to other radiance, irradiance, or photon caching algorithms is that these algorithms store radiance *probability* (or use that radiance information as a probability) and use it to “guide” paths over the scene in an unbiased form. Scene radiance is a high dimensional data requiring three dimensions for spatial information and two for spherical (or hemispherical for opaque surfaces) direction information. Thus, proposing an efficient data structure to hold and sample such information is essential for this family of algorithms.

The initial proposal of such a method was made by Lafortune et al [66] and Jensen [67]. The former utilizes a dense 5D data structure to estimate the radiance field, and the latter utilizes a photon map (Section 2.2.4). Due to the dense nature of the 5D radiance field, memory constraints for such a method is high.

Voba et al. utilize Gaussian Mixtures that are projected on a 2D plane and stored spatially [60]. This structure is pre-generated using photon distributions and forward ray distributions. Product sampling, which is the sampling of not only the incoming radiance but the multiplication of BxDF and radiance, is proved to have better convergence in the case of Gaussian Mixture Model implementation [68]. Similarly, Dodik et al. utilize spatio-directional mixture models, enabling efficient incorporation with the BxDFs [69].

Product sampling is an important extension to path-guiding methods for specular since these surfaces would probably mask most of the incoming radiance field; it could dramatically reduce variance for such regions. However, if a surface is near perfect specular, that region may not require path guiding.

Moreover, Muller et al. proposed an on-the-fly approach named the practical path-guiding method. The algorithm utilizes spatio-dimensional trees (SD-trees) to capture radiance information [64]. Initially, this tree starts empty, and when a unidirectional path tracer finds a path, all of the paths’ vertices populate the tree with calculated radiance information, which progressively continues. Previous image samples are discarded to reduce variance.

Ruppert et al. use von Mises-Fisher distributions (vMF) for radiance field capture as another method. Such small distributions are held on a Kd-tree, and nearby sampling

locations combine nearby vMFs [70] and samples using the generated probability field. Due to the analytic nature of the directional field, product sampling can easily be incorporated into the technique by approximating the BxDF via multiple vMF distributions.

Machine learning methods are also proposed as path-guiding algorithms. Neural Importance Sampling method proposed by Muller et al. [61] utilizes neural networks to learn the incoming radiance, and it easily includes product importance sampling.

While Muller et al. train the network on a per-scene basis, Bako et al. pre-train a generic deep neural network with various scenes, and then this network is utilized over an independent scene [63]. Huo et al. also propose a similar approach as well [62]. Other reinforced learning methods are also utilized, such as the one proposed by Dahm et al. [71] in which the incoming radiance equation is similar to the Q-learning function. This method utilizes a point cloud backed by a nearest neighbor search data structure (for example, a KD tree), and each point holds a dense 2D radiance field. A Bayesian regression model is also applied in order to efficiently sample light sources for better NEE (Direct illumination) sampling [65].

Unlike other methods, Guo et al. propose a primary sample space path guiding scheme [72]. In this method, a structure is built by utilizing primary samples and the luminance of the paths. Primary sample space is a multi-dimensional space of random numbers that is used to sample the path.

2.2.7 Conclusion

We have discussed various families of rendering methods; in conclusion, none of the methods is perfect for every scene layout. In this section, we will try to express the capabilities and shortcomings of these methods.

The convergence rate of Monte Carlo techniques is independent of the dimensionality of the integral. For the light transport case, it is a strong advantage since Equations 2.2 and 2.3 consist of multiple infinite dimensional integrals. Moreover, the Monte Carlo approach is simple to implement and an unbiased technique.

Near-singular or singular objects have problems with Monte Carlo sampling when a path consists of specular-diffuse-specular connections. Traditional Monte Carlo techniques would not be able to find such paths since the paths are enforced by the perfectly specular surfaces (or multiple of such objects), and endpoints of the path are also singular (in this case, a point light or a pin-hole camera). To make such paths sampleable, the light source, camera, or specular object should not be singular. Even still, such paths would be hard to sample. These paths are defined as “Specular-Diffuse-Specular” (SDS) paths. An illustration of such paths can be seen in Figure 2.6. Making such paths easy to sample is one of the main research points in computer graphics. Path space regularization tries to overcome this problem by making such impossible paths sample-able by relaxing (by making them a narrow Gaussian) such singular BxDF and making them diffuse-like functions [73]. Such constraint gets stricter as the samples accumulate, making the final result consistent.

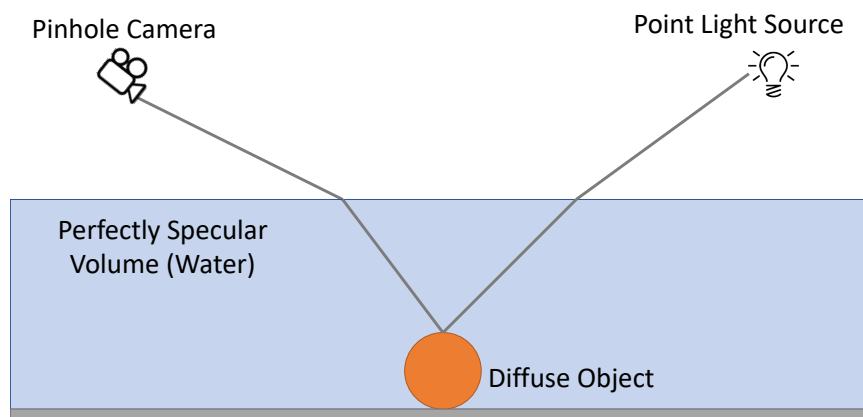


Figure 2.6: Specular-Diffuse-Specular interaction demonstration. Monte Carlo methods can not sample such paths due to singularity. Surfaces of the diffuse object (orange) can not “aim” toward the point light source directly by position due to the specular surface in between the surface and the light source.

The most common natural example of this phenomenon is pool caustics on a sunny day; although the sun is not a point light, it is high-frequency light compared to the entire incoming radiance field. A rendered example of this phenomenon can be seen in Figure 2.7.

This shows the fundamental drawback of general incremental path construction and path tracing techniques. Such methods are only locally aware of the scene. There is

no global information to act on to construct paths toward the required endpoints. As discussed before, path-guiding methods try to alleviate such lack of global information (see Section 2.2.6).

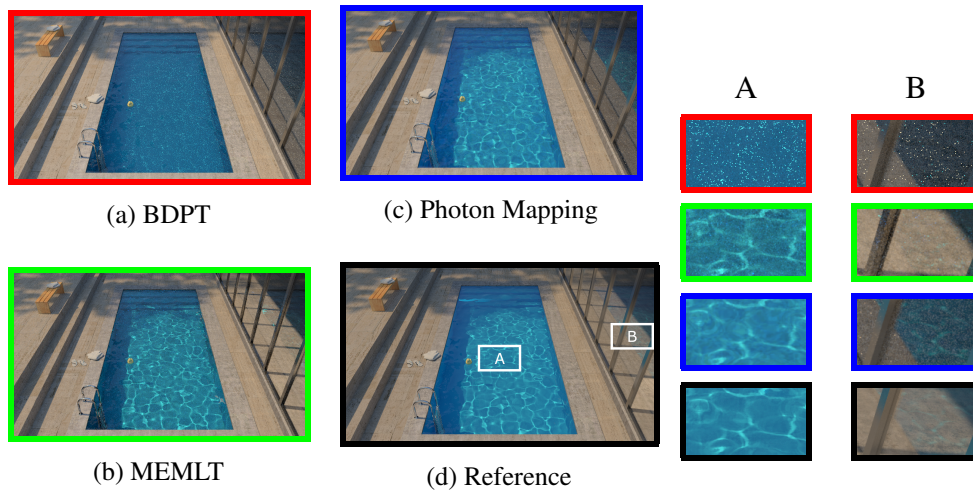


Figure 2.7: Multiple path integral approximation techniques. All of them took approximately the same amount of time (2.5 minutes). The reference image took 8 hours to render using the manifold exploration metropolis light transport technique. Segment A is zoomed in to show the hard Specular-Diffuse-Specular (SDS) paths. Section B is even harder, in which an SDS path is reflected. Bi-directional path tracing fails to sample those hard paths efficiently. Photon mapping can efficiently sample paths, but it is a biased method. Finally, the Manifold Exploration Metropolis Light Transport method can handle hard paths efficiently, but it is not temporally coherent.

In Figure 2.7, photon mapping and manifold-exploration metropolis light transport (MEMLT) could generate pool caustics quite well. Although a static MEMLT can not demonstrate it, it suffers from temporal incoherence, meaning that unless a specific path space is reached, that path cannot be thoroughly explored by the MLT system. This results in the spatial segmentation of highly converged regions and noisy regions.

Photon mapping, on the other hand, is a biased technique, which can be ill-suited when a high-precision comparison is needed with respect to an experimental result for a research.

Although an image is not available, path-guiding methods can generate comparable images with respect to both MEMLT and Photon Mapping, given enough samples to train. However, path-guiding methods are both unbiased, and convergence behavior is uniform. However, such methods can require a substantial amount of memory de-

pending on the scene; furthermore, path-guiding methods will require pre-processing or on-the-fly training.

With all this in mind, we proposed a path-guiding technique for this thesis. Path guiding techniques can be considered a “holy grail” of rendering techniques given enough time and space. However, such a claim is unfortunately not practical.

2.3 Massively Parallel Architectures and GPGPU

As we establish our method, we need to consider the implementing device. As hardware acceleration becomes available [74] on modern GPUs, proposing a GPU-oriented path guiding method seemed suitable. Path guiding techniques are similar to path tracing methods in terms of parallelization, and GPU path tracing implementations have become available [75, 76, 77].

Initially, GPUs are designed to do raster graphics-related tasks such as texture mapping and triangle rasterization. These GPUs have had a static pipeline with minimal programmability via changing switches exposed by the device. Due to the independent nature of the triangle rasterization with respect to other triangles, these devices are inherently designed to execute many triangles in parallel. Programmability is introduced to portions of the graphics rasterization pipeline in which vertices of the triangles and the fragments (potential pixels) that are generated by the rasterizes could be manipulated by writing programs using assembly language and after using high-level languages.

Initially, such programmability was not shared, meaning different hardware portions were responsible for vertex and fragment shading. As the hardware design progresses, this programmability is unified and exposed to the user. After the GPUs acquire programmability capability, proposals emerge to utilize such programmability for general-purpose floating point compute-heavy tasks. Buck et al. proposed to utilize the fragment shader for computing such tasks using an extended C-like language [78, 79]. Although unknown then, this computation model was highly similar to the post-processing model before the compute shaders were introduced to the graphics APIs. In Krüger and Westermann’s case, algorithms for linear algebra-

related problems are proposed. Even ray-tracing implementation using this programmability is proposed way before the actual ray-tracing capable hardware [80].

Hardware Graphics company NVIDIA directly exposed this functionality to C/C++ high-level language [81]. Instead of manipulating a raster graphics pipeline for general computing needs, a user can directly write code specifically. NVIDIA chose the name “Compute Unified Device Architecture” (CUDA). In CUDA, the user can write a single code that can be compiled into the CPU and the GPU. Consequently, GPUs become “General-Purpose Graphics Processing Units” (GPGPU).

2.3.1 Design Differences between CPUs and GPUs

In this section, we briefly explain the design differences between CPUs and GPUs. We deemed such an explanation is important due to establishing design differences of *algorithms* that solve the same problem over GPUs and CPUs.

GPUs are inherently designed to compute parallelization-heavy tasks. Such design principle is the byproduct of the raster graphics pipeline as discussed in Section 2.3. Because of that, GPUs allocate more area for computational units in the die than CPUs (see Figures 2.8 and 2.9).

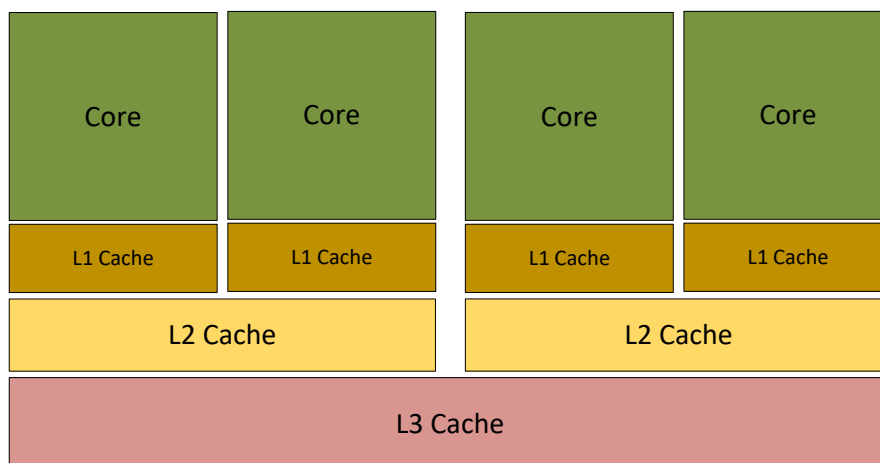


Figure 2.8: Highly simplified block layout of a General purpose CPU. The general design of the CPU reserves a larger die area for cores due to their complexity.

The reason CPUs could not reduce their core size and introduce additional cores is due to complexity. Modern CPUs are deeply pipelined, execute instructions in an out-of-order fashion, and are super-scalar. Super-scalar means that CPUs can issue multiple instructions simultaneously in a single core. Unlike GPUs, such multi-issue is determined by the instruction flow and calculated in real-time. This introduces high complexity to the design of the core. However, such a design enables streamlined performance with respect to any algorithm type.

GPUs on the other hand, could be delicate compared to CPUs. The design of the algorithms for a GPU should be highly parallelizable and have minimal branches. GPU algorithms should be divided into theoretical threads and threads registered to multiple “streaming multiprocessors” (SMs). A highly simplified version of a GPU block diagram is given in Figure 2.9. On each SM, instructions are issued in a “single instruction multiple threads” (SIMT) fashion, meaning multiple threads undergo the same instruction. These instructions are common for all the threads issued for a parallel algorithm. SM juggles threads over the cores to hide data dependency and latency. Due to SIMT design, intra-threads of the SIMT would require processing both branches if multiple threads take different paths of the branch. While threads process one part of the thread, the other threads are masked. Thus, branch-heavy code would reduce the performance.

CPUs alleviate this branch-heavy algorithm by utilizing sophisticated branch predictor hardware. However, such an application is not suitable for GPUs since all threads that take the SIMT should branch into the same path for branch prediction to be applicable.

Another design difference is the memory and cache hierarchy. CPUs have sophisticated multi-level caches that utilize temporal and spatial coherency of algorithms. When data is accessed, neighboring data is fetched onto the cache (spatial coherency). Likewise, data that has been recently touched (read or written) stays on the cache (temporal coherency). GPU memory hierarchy could not utilize such sophisticated hierarchies due to the sheer amount of raw data required to feed all GPU cores. Thus, GPU memory is throughput optimized, unlike CPU memory hierarchy, which is latency optimized. This observation means that computation-heavy algorithms would

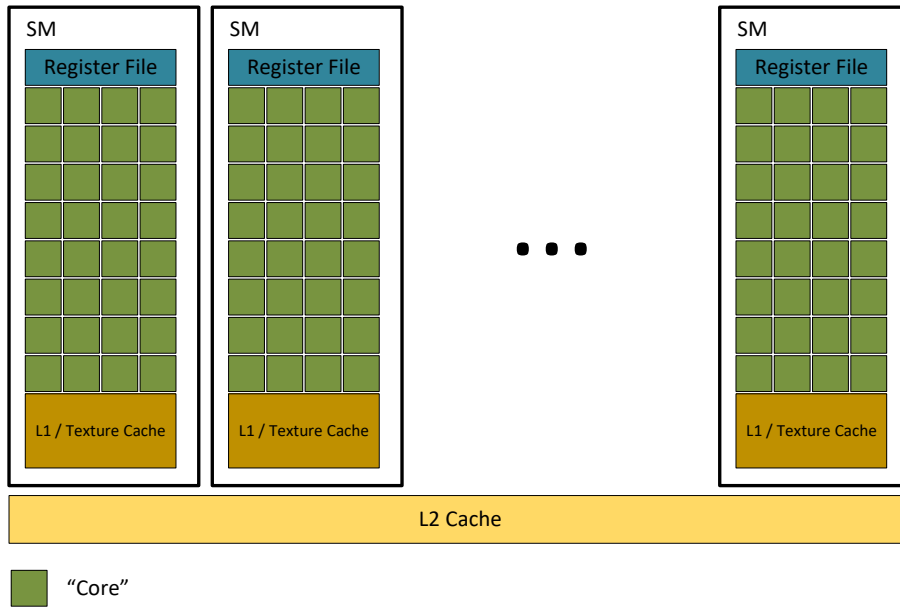


Figure 2.9: Highly simplified block layout of a GPU. GPU has a high amount of calculation units (Cores) with a smaller die area compared to CPU.

perform better on the GPUs.

Thus, GPUs expose a fast memory called “shared memory” shared across threads in an SM. The user can utilize this memory for the collaboration of threads inside shared memory or for specific caching needs. Such exposure is suitable for GPUs since hardware-demanded caching may not be suitable for different algorithms.

We will give a simple example, parallel reduction, to expose the differences between CPU and GPU design in Section 3.2. This explanation would create a baseline for the proposed method of this thesis.

2.4 Sparse Voxel Octrees & Cone Tracing

The proposal of this thesis will utilize Sparse Voxel Octrees (SVO). Thus; briefly mentioning the previous work is deemed necessary. Octree structures are predominantly used for volume rendering. Some methodologies utilize the GPU as well [82, 83]. Hybrid topology tree structures like those proposed by Museth are also used for volume representations [84]. Museth’s proposal is predominantly used in offline

graphics since it is open source. Recently, the GPU implementation of this method has become available [85]. These methods are similar to an octree; however, their topology is more similar to B+Trees.

Hybrid dense-sparse octrees are also researched for volume representation [86]. We would kindly draw the reader’s attention to a recent survey by Beyer et al. about volume rendering methodologies [87].

The generation of voxel data structure is important for non-volumetric entities. Hardware rasterizer can be utilized as a voxelization technique for triangle meshes [88, 89]. There are voxelization techniques for other solid representations as well [90].

Casting rays over the generated SVO structure may be required for certain tasks. Our proposal is one task requiring efficient ray casting over the generated SVO. The traditional approach utilizes a digital differential analysis (DDA) algorithm [91, 92]. In this methodology, empty spaces are skipped according to the leaf voxel sizes of the octree. Cone tracing algorithms are proposed for high solid-angle information retrieval [83], in which data is queried using cones; thus, only voxels that match the aperture of the cones are queried. This necessitates holding data not only on the leaves of the tree but on intermediate nodes as well. Advanced methods, such as the one proposed by Hadwiger et al. [93], combine the above-mentioned space skipping with generated ray segments using a rasterizer.

The proposed method discussed in Chapter 4 will have a similar basis to the real-time global illumination framework proposed by Crassin et al. [94]. In this case, regions’ incoming radiance and material properties are stored in an SVO. Cone tracing is conducted on camera-visible surfaces to query indirect illumination information. According to the results, a few large cones would be deemed sufficient for a realistic capture of the indirect illumination, which was a motivating finding for our work. Chapter 4 has a similar approach; however, instead of directly utilizing the stored data for estimation, which will be biased, we utilize the data as a guiding metric. Unlike Crassin et al., we hold radiant exitance over the voxels.

2.5 GPU Oriented Light Transport Proposal

Hardware acceleration of computer graphics has a long history. In this section, to the best of our capability, we try to do a general overview of the main family of rendering algorithms. Almost all of it has advantages and disadvantages, as discussed in Section 2.2). We concluded that proposing a path-guiding algorithm over massively parallel architecture like GPUs would be a motivating research area.

To this end, we propose a GPU-oriented path-guiding algorithm that utilizes the capabilities of GPUs. To the best of our knowledge, such an approach has not been researched thoroughly in rendering related computer graphics research. Our proposal will be ground-up designed for the GPUs that utilize the recent hardware acceleration capabilities and expose the parallelization capability of the GPUs. Before proposing a path-guiding scheme, we must tackle the light transport problem, which is the basis of path-guiding algorithms. The next chapter will discuss a parallelized path tracer that will be extended using a novel path-guiding approach.

CHAPTER 3

PARTITION BASED WAVEFRONT PATH TRACING

In this chapter, we explain the baseline of GPU-oriented design using the fundamental reduction algorithm. After that, we explain the parallelization scheme for path tracing utilizing a queue-based approach and the novel sorting-based approach. Finally, we conclude with the pros & cons of both methods.

3.1 Preliminaries

As we establish the underlying hardware in Section 2.3, utilized hardware is selected as GPU. However, the explanation of the proposal will be on a specific GPU vendor's hardware, namely NVIDIA GPUs. Such an explanation may not mean the proposal is for this specific hardware; GPU hardware can utilize this design. The reason for this specificity is to standardize terms to enable profound and concise explanations.

In recent years, hardware acceleration for computer graphics shifted towards ray tracing from rasterization. All modern graphics hardware do support hardware-accelerated ray tracing capability [95, 96, 97]. This capability is first exposed through graphics APIs such as DirectX [98]. Other graphics APIs, such as Vulkan, acquire this capability through extensions as well.

The main design scheme of all ray tracing is through a pipeline similar to a rasterization pipeline. Unlike the rasterization pipeline, the user feeds rays to the system instead of triangles. Rays undergo hardware-accelerated intersection tests, and like the raster pipeline, the user can program specific parts programmatically.

On a ray-tracing pipeline, rays are intersected with triangles via opaque acceleration

structures. Although the internals of these acceleration structures are not exposed to the user due to potential changes in the future, current-generation hardware utilizes Bounding Volume Hierarchies (BVHs) [99, 100, 101]. Thus, hardware can do axis-aligned bounding box (AABB) and triangle intersection tests in silicon. This region is also a potential research area, and initial research emerged to find more suitable acceleration structures suitable to design in hardware [102].

Due to the complete difference in the algorithm logic, programmability regions of the ray tracing pipeline are completely different from the raster pipeline's classic vertex/pixel shaders. Unlike the raster pipeline, rays can trigger shaders when any ray-triangle hits succeed or when the closest intersection is found. Finally, a shader trigger occurs when a ray completely misses the acceleration structure. These shaders are in the order called "any-hit shader", "closest-hit shader" and "miss shader"¹. Another programmable shader portion is the "ray generation shader" in which the user generates rays that can be fed to the pipeline. The reason for the any-hit shader is to accept/reject intersections programmatically. One prominent example is the alpha-masked objects, such as tree leaves.

Another difference is that the ray tracing pipeline is self-triggering. The ray generation or closest hit shader can trigger ray-tracing calls and re-cycle the pipeline for a newly created ray. This capability enables path-tracing algorithms to be implemented using the ray-tracing pipeline. Computation concludes when all of the rays in the pipeline are exhausted.

By observation, this design is proposed with path tracing in mind; an extensional approach to path tracing, such as path guiding, would not be able to utilize GPU hardware capabilities fully. Moreover, this pipeline does not expose the shared memory to the user on all programmable regions. This would limit collaborative approaches, where multiple rays in close regions on the scene would not amortize similar works.

To alleviate the issues discussed above, Laine et al. [103] proposed the wavefront style of path tracing. The proposal in this Chapter will have a similar basis with a different partitioning technique. We will thoroughly explain the methodology in later

¹ There is another custom shared phase named "intersection shader" which, can be utilized for non-triangle primitives (i.e., analytic primitives such as curves, spheres) which is skipped since raster pipeline only operates over triangles. It makes the compare-contrast convoluted.

sections. Before that, we deemed an introduction to GPU parallel design beneficial for a profound understanding.

3.1.1 Taxonomy

Before explaining the methodology, we will establish the terminology used in this chapter. We will utilize NVIDIA's taxonomy for GPU design constructs. Instead of using CPU and GPU, we use *host* for the CPU and *device* for the GPU. Hardware execution constructs are defined as *streaming multiprocessor (SM)*, *grid*, *block*, *wrap*, *thread*, and *kernel*.

A *kernel* is a piece of execution that occurs on the device. A kernel can be configured with parameters to tune parallelization. The *grid* is the main parametrization of the kernel. A grid consists of multiple *blocks*, and each block consists of multiple *threads*. This hierarchical parametrization enables efficiency between multiple device families. The user can parametrize the block count and thread count.

One aspect of the blocks is that threads inside of a block can communicate with each other via shared memory. A shared memory can be allocated on each block if required, and threads in each block can use that memory collaboratively.

A specific device family may concurrently handle multiple blocks on its internal processor block called *SM*. This is limited by the devices' resources (registers, static hardware capabilities). The ratio between the maximum blocks that can be run on an SM and the kernel's achieved block count per SM is called *occupancy*. This crucial metric directs the programmer to specific designs to minimize register usage, shared memory allocations, etc.

As we discussed before, GPUs run in SIMT fashion. Each block's threads are partitioned into *warps*, and warps execute the kernel instructions in lock-step fashion. For NVIDIA, the warp thread (warp size) did not change throughout the years and is 32. Branch-heavy code could not run efficiently on the GPU because of the warps' lock-step fashion execution. Intra-threads on a warp must run the same instruction by design, and if a branch mismatch occurs, both regions of the branch must be executed in a masked fashion. This limitation is a fundamental path-tracing constraint due to

random ray scattering. This limitation will be one of the main issues addressed in this chapter.

3.2 GPU Oriented Parallel Design

In this section, we will briefly explain a use case to explain the implementation design differences between the host and the device. To achieve this, we perform parallel reduction using a massively parallel architecture. Although, this explanation will be similar to Harris's explanation [104], it will be more computer science-focused and illustrative instead of performance-practical.

3.2.1 Case Study: Reduction

Parallel reduction is a key concept in functional programming, commonly employed in various fundamental algorithms as an intermediary process. In parallel reduction, a list of N variables is iteratively reduced to a single value using a binary operation (e.g., addition), with the condition that the operation is both associative and commutative. Associativity will be crucial for devising efficient parallelization strategies, which will be discussed later. The provided figure (3.1) illustrates a straightforward non-parallel implementation of this algorithm.

```
1  int ReductionAdd(const int a*, int N)
2  {
3      int result = 0;
4      for(int i = 0; i < N; i++)
5      {
6          result += a[i];
7      }
8      return result;
9  }
```

Figure 3.1: C++ code snippet for reduction, using arithmetic add operation. This reduction function acts over default integer types.

As can be observed, the non-parallel implementation of reduction is embarrassingly

straightforward. For a single-threaded CPU, this approach can be considered optimal. However, parallelizing this algorithm will introduce additional complexity. The destination value ("result") is a critical aspect to consider. When multiple threads attempt to write to it concurrently, it leads to data race issues. Hence, careful attention to managing the destination value is necessary for effective parallelization.

3.2.2 Parallel Reduction

Now, we will assume a traditional parallelization scheme for reduction operation. We will divide the data space (source space) into equally sized C portions. Each segment will independently compute its reduced value, and then these local results will be written into a shared variable using synchronization mechanisms like semaphores, mutexes, or atomic operations. An example is given in Figure 3.2. In this case, we assumed the hardware capability of atomic add operation is available and used such functionality. Most modern hardware has this capability.

```
1 // Region [begin, end)
2 // Called for each thread
3 void ReductionAddThreaded(int& result, const int* a,
4                           int begin, int end)
5 {
6     int localResult = 0;
7     for(int i = begin; i < end; i++)
8     {
9         localResult += a[i];
10    }
11    atomicAdd(result, localResult);
12 }
```

Figure 3.2: C++ code snippet for traditional parallel reduction, using arithmetic add operation. This function is called for each collaborating thread.

Implementation complexity compared to Figure 3.1 did not change as much. Although there is additional overhead to partitioning the operands into threads, which is not shown in Figure 3.2, it is comparably trivial, and it can be partitioned in constant time. This shown algorithm can be optimal for CPU-like systems, where C is rela-

tively low. The main bottleneck here is the atomic portion, where instructions must be sequential due to data dependency (in this case, destination dependency).

For distributed systems, where C is much higher, the algorithm in Figure 3.2 can still be suitable; however, additional improvements would be required. We will skip such improvements and concentrate where C is *asymptotically* comparable to N , meaning given $\mathcal{O}(N)$ work, computing system has $\mathcal{O}(N)$ threads. For CPU-like systems, even for distributed CPU systems, asymptotically tying C to N is not practical; however, this is the case for GPU systems.

3.2.3 Massively Parallel Reduction

Imagine C is asymptotically significant for the algorithm exposed in Figure 3.2. Since the reduction operation is binary, assume $C = N/2$. The algorithm above will collapse to sequential operations almost immediately; each thread will add two values and try to pound over the single resulting value. Thus, the above algorithm will not be efficient in massively parallel architectures such as GPUs.

Since C is asymptotically significant, it should be incorporated into the algorithm. One can propose a hierarchical approach in which C amount of operations occur and are written to the memory, on the second iteration $C/2$ amount of operations occur and so forth, which will fully utilize such massive parallelization schemes of the given system throughout the iterations. An example of this kind of approach is given in Figure 3.3

As shown in Figure 3.3, algorithm complexity increased significantly. We are required to allocate intermediate buffers, one of which can be skipped if the input list “ a ” is modifiable. Moreover, memory throughput is increased significantly; in the worst case, which is the first iteration, $\mathcal{O}(N)$ amount of memory operations would be required. However, the data dependency bottleneck is eliminated, and as one can notice, this algorithm would not require critical sections or atomic operations. In this case, the only needed parallel construct is a barrier that collects the launched threads issued by the “*ParallelIssue* < \dots >” command. We assumed such a barrier is incorporated into the function itself.


```

1  void ReduceIteration(int* outBuffer, const int* inBuffer,
2                          int location)
3  {
4      outBuffer[location] = inBuffer[2 * location] +
5                          inBuffer[2 * location + 1];
6  }
7
8  int ReduceGPU(const int* a, int N)
9  {
10     // Allocate intermediate buffers
11     std::vector<int> bufferIn(N), bufferOut(N/2);
12     std::copy(bufferIn.begin(), a.cbegin(), a.cend());
13     int iterationDataSize = N;
14     int* inData = bufferIn.data();
15     int* outData = bufferOut.data();
16     do
17     {
18         const int C = iterationDataSize / 2;
19         // Pseudo function that launches a C amount of
20         // threads that call the first argument.
21         // Each call will have a unique 'location'
22         // variable between [0, C)
23         ParallelIssue<C>(ReduceIteration,
24                         outData, inData);
25         // Swap the in and out buffers
26         Swap(inData, outData);
27         dataSize = dataSize / 2;
28     } while(dataSize > 1);
29     // Notice: due to the swap, we need to read the first
30     // element of inData instead of out
31     return inData[0];
32 }

```

Figure 3.3: C++ code snippet for massively parallel reduction, using arithmetic add operation. To simplify the illustration, assume N is even.

Although not indicated in the example algorithm, memory throughput can be utilized using shared memory on an actual GPU implementation. Each thread on a block can use the shared memory as a scratchpad to write and read internally locally. Such optimization in this illustrative example can be considered a low-level optimization technique and not included in the illustrative Figure.

This example perfectly illustrates the requirements for massively parallel design. We will list our observations below:

- GPUs are best utilized when parallelized regions are small, and the count of those regions is high.
- Small parallelized regions mean low register usage and actual hardware can issue more blocks over its SMs.
- Doing a “multi-pass” over these regions almost always requires an intermediate buffer.
- Such a multi-pass approach could eliminate data dependencies given the algorithm’s nature.
- Multi-pass approach would require high throughput memory.

One can notice why the design principles of the GPUs are chosen explicitly by this observation. The availability of a high throughput memory (shared memory) is one of these reasons. A path-tracing approach should consider all the necessities that came up with the massively parallel design.

3.3 Path Tracing on the GPU

To quickly reiterate, we re-issue the recursively expanded form of the rendering equation that is explained in Section 2.2.1 below in Equations 3.1 and 3.2.

$$L(p_1, w_o) = \sum_{n=1}^{\infty} P(\bar{p}_n) \tag{3.1}$$

$$\begin{aligned}
P(\bar{p}_n) = & \underbrace{\int_{\Omega} \int \cdots \int_{\Omega}}_{n-1} L_e(p_n, w_{o_{n-1}}) \\
& \times \left(\prod_{j=1}^{n-1} f_x(p_{j+1}, w_{o_j}, w_{i_{j-1}}) \right) dw_{i_2} \cdots dw_{i_n}
\end{aligned} \tag{3.2}$$

Traditional CPU-based path tracers would conduct a Monte Carlo simulation over this equation by iterating a single ray through path space and incrementally constructing a path and accumulating radiance. When parallelizing a path tracer, each path is processed independently in parallel. Each path calculation process ends up in a series of intersection and material evaluation routines until an emitting source is reached. As each thread operates on a distinct path, there are no data conflicts between threads, resulting in no inter-thread data dependencies ².

Although such an approach is free from data dependency, it is still ill-suited for GPU. On implementation discussed above, although the data path is independent, the execution path is *incoherent*. For example, *thread_a* may calculate a different material while a *thread_b* works on another because rays are scattered onto different surfaces that have different material characteristics. Since GPU threads work in SIMT fashion as discussed in Section 3.1, the different execution paths between warp’s threads will lead to serial executions; thus reducing efficiency.

On top of incoherent execution, memory accesses are incoherent as well. For example, neighboring threads (warp threads) may process the same intersection routine; however, they may end up in completely different regions of the scene due to scattering and act on different intersection data (such as bounding volume hierarchies). CPU can hide these memory accesses via sophisticated caches, but this is not true for GPU devices.

Finally, this material evaluation and intersection routine cycle; given many different material and surface properties, would pose a significant demand on GPU resources, potentially reducing GPU occupancy. This can be alleviated by separating intersection and material evaluation subroutines instead of calculating these on a single large kernel.

² There is a data dependency due to image filtering, but it is purposefully skipped to make the explanation concise.

Like the explanation of massively parallel reduction, we are required to segment the computation into multiple phases to reduce the register overhead. Moreover, incoherent execution should be handled via partitioning rays with respect to execution paths (such as material and surface characteristics). As in the reduction case, this means holding ray states in intermediate buffers. This approach is called “Wavefront Path Tracing” and will be discussed in the next section [103].

3.4 Wavefront Path Tracing

Our approach is similar in the design of both Laine et al. proposal [103] and ray tracing pipelines of modern GPUs [74]. Such approaches are called wavefront approaches. The analogy of the naming is unknown to us, but to speculate, waves represent the union state of the system, and those undergo operations in bulk. When a wave is exhausted, in metaphor, it breaks and perishes over the coastline, and another wave replaces its place. In path tracing, the bulk state is the ray list, and rays bounce around and create new waves, and those perish (i.e., hit an emitter).

This section will discuss the common parts of the wavefront path tracing between our proposal and Laine et al.’s. Implementation of ray tracing pipelines is hardware-dependent and proprietary; however, there should be a similar approach in these proprietary systems to alleviate naive path tracing issues.

The main overview of the algorithm can be seen in Figure 3.4. Wavefront path tracing generates rays in bulk; in the unidirectional path tracing case, rays are generated from the camera and, in a union, undergo operations together. In Figure 3.4, each red and teal box represents a kernel call. Rays are partitioned into groups, and a single kernel is called for each group. Vertically stacked boxes represent concurrent kernel calls.

More specifically, after rays are created in bulk by a custom ray generation kernel, rays undergo ray casting. The ray-casting algorithm is the same for all the rays. Thus, a single kernel is launched for all the rays. In the overview, it is segmented into two phases. This two-phase design enables efficient instancing of surfaces with different transforms. Assuming the base accelerator is a BVH, some leaves of the tree can refer to the same second-level acceleration structure with a different transform.

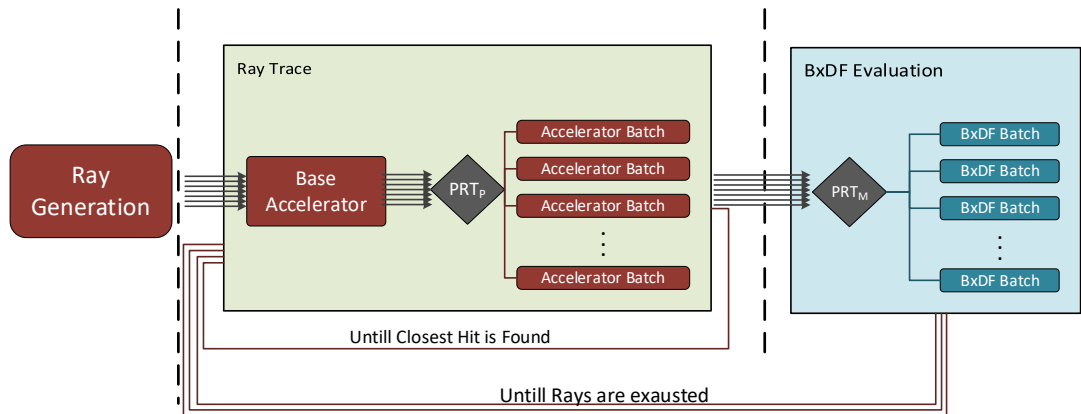


Figure 3.4: Wavefront path tracing algorithm overview. The entire path-tracing algorithm is separated into two distinct phases.

Additionally, this enables mixing and matching different acceleration structures for different primitive surfaces. An example of this can be for participating media such as fluids. Fluids may utilize octrees instead of BVHs since octrees may be more efficient in encapsulating volumetric mediums. A similar approach is also exposed on DirectX Ray Tracing (DXR) as well [98].

With the two-phase design in place, rays undergo partitioning to separate them into different acceleration structures. The base accelerator traversal stops upon reaching a leaf node, saving the traversal state before initiating traversal within inner accelerator structures. Once the inner accelerator traversal completes, rays return to the base accelerator and resume traversal. This cycle repeats until all rays either miss or intersect with a surface. Subsequently, all rays obtain information about the surface they hit and the material (BxDF) that needs evaluation.

The BxDF evaluation portion is responsible for evaluating material (BxDF) functions. Similar to the acceleration portion, rays are partitioned with respect to the material type, and a different kernel is launched for each specific type of material. Material evaluation is concluded by a ray generating zero or multiple new rays depending on the algorithm and the result of the material evaluation.

Finally, those newly generated rays return to the ray-casting step, and the system cycles again until no ray is left. Technically, as the rays die due to algorithm logic, another ray generation step can be issued between the cycles, and this step can fill

these empty spots. This enables full saturation for the underlying hardware.

The main advantage of this method is that it enables efficient and small parallel segments so that each ray can dynamically select the code paths that those rays are supposed to run. Branch-heavy portions of the code, such as material evaluation, are segregated, guaranteeing that no branch divergence can occur over those segregation logics.

In addition to the execution coherency, another advantage is the memory coherency. Materials may require multiple textures or data accesses. This design groups these memory accesses, enabling coalesced reading. Look-up tables can collectively be loaded into the shared memory and be read from that fast memory location, thus increasing efficiency.

Algorithm 1 Wavefront path tracing.

Input-Output

$R_1 = \{r_1, r_2 \dots\}$ ▷ Set of initial rays

Start

Initially Generate rays from the camera and populate R_1

for $i = 1$ to MaxDepth **do**

if $R_i = \emptyset$ **then**

Terminate.

end if

$NR_i = \{(n_1, R_{p_1}), (n_2, R_{p_2}) \dots\} \leftarrow \text{RAYTRACE}(R_i)$

$N_i = \{(n_1, R_{n_1}), (n_2, R_{n_2}) \dots\} \leftarrow \text{PARTITIONMATERIAL}(NR_i)$

for all $(n_j, R_{n_j}) \in N_i$ **do**

for all $r_k \in R_{n_j}$ **do**

$R_{i+1}^j \leftarrow \text{SAMPLEBXDF}(n_j, r_k)$

end for

end for

$R_{i+1} = \{R_{i+1}^1, R_{i+1}^2 \dots\}$ ▷ Next set of rays

end for

However, this approach also has its drawbacks. As each operation step is separated into different kernels, the state of rays must be stored in memory for persistence.

Given that GPUs are highly parallel devices, this scheme of saving state would demand a considerable amount of storage. This memory could have been used for higher-resolution scenes. CPU-based design amortizes this storage through their stack memory since each ray has a sophisticated CPU thread dedicated to itself.

We also provide an algorithmic representation of the method described in Algorithm 1. Ray tracing portion is encapsulated as a method named RAYTRACE routine to simplify the pseudocode. Operations happen inside the RAYTRACE is similar to the given code in structure.

In this case, there is an upper limit of the recursion dictated by the parameter “*MaxDepth*”. SAMPLEBXDF can also skip generating rays (i.e., when a ray reaches an emitter), thus terminating the system.

In the next section, we explain the partitioning schemes that can be utilized for the wavefront path tracing algorithms. We will first explain the method of Laine et al. and its advantages and disadvantages. Furthermore, we will explain the novel approach proposed in this thesis, which is a sort-based approach.

3.4.1 Queue-based Partitioning

Laine et al. proposed a queue-based approach for partitioning ray between operations [103]. Considering the material partitioning stage, each material batch will have its queue readily available for filling. When the ray tracing step determines the closest hit location and acquires the resulting material, the routine immediately writes its result into the appropriate material batch queue. Since the ray tracing routines are conducted in parallel, queue submission operations should be protected due to data races. Moreover, GPU hardware has fast atomic increment counters, and a single ATOMICINCREMENT operation can be used for lock-free enqueue operation.

Given a single global memory variable n that represents the number of items in the queue, applying ATOMICINCREMENT operation over n will increment n , then writes the incremented n to its corresponding memory location and finally return the *previous* value of n in a single atomic fashion.

Combining this with an empty array with N locations and n where $n = 0$, one can compose a queue. Enqueue operation will only call `ATOMICINCREMENT` over n , and in this case, operation will make n to 1 and return 0. The return value will give the enqueued location index. In other words, we created an atomic fine-grained allocator.

This approach enables efficient and easy-to-implement queues. Technically, using `ATOMICINCREMENT` and its sister operation `ATOMICDECREMENT`, we could only be able to create a stack. The name “queue” comes from the producer-consumer queues. In this scenario, the ray trace kernel act as a producer, inserting relevant material data into distinct queues. Subsequently, multiple kernels are launched for each queue, consuming the stored data in parallel.

The approach of this queue-based design is simple to implement and elegant. However, it has memory-related and consistency issues due to the usage of atomics. The following two sections will explain these shortcomings.

3.4.1.1 Memory Management Issue

Previous chapters and sections did not explain the fundamental memory management problem of GPUs, specifically GPU communicating APIs. On graphics-related APIs such as OpenGL or DirectX during the kernel execution (in OpenGL and DirectX terminology, in compute shaders), the user can not allocate memory on the heap. All the potentially required memory should be pre-allocated by the host device. CUDA alleviates this issue of in-kernel memory allocation; however, this functionality is not performant compared to static pre-allocation. Additionally, heaps of the *same* GPU memory allocated from the host and the device are different. This means you can not free the host-allocated GPU memory from the device (during kernel execution) and vice versa.

This limitation implies an implementation complexity for the suggested queues in Section 3.4.1. Due to API limitations, we could not implement an array-backed queue that dynamically grows by the threads of a kernel. Even if the hardware grants this capability, implementation of a dynamic allocation scheme will require mutexes and will deviate from the simple `ATOMICINCREMENT` based enqueue operation.

Limiting the queues to a static amount alleviates this dynamic allocation; however, it opens another issue. Determining the queue size beforehand is challenging since this amount ultimately depends on the scene. One can argue to run the enqueueing kernel twice; the first run will allocate but does not write the data to the queues, and the second run will write the data. The host will look at the incremented values between these kernel calls and allocate enough GPU memory. Unfortunately, this approach will increase the computation time because it calls the kernel twice.

Another approach could be to allocate enough memory for a worst-case situation. Assuming we have R amount of rays and M amount of queues, such a worst-case approach will require $R \times M$ amount of memory in consideration of the case when all of the rays happen to go into a single queue. Such an approach would be infeasible due to memory limitations.

There is no straightforward way to alleviate this problem for queue-based partitioning methods. One can choose a memory or a computation constraint to utilize this queue-based approach. Our method, however will require constant memory that does not depend on the partition count.

3.4.2 Consistency Issue

Another potential issue is with the computational consistency of the queue-based system. If we iterate over the same worst-case example that is discussed in Section 3.4.1.1, when all of the rays happen to go into the same queue, all of the enqueue operations will be sequentiality due to `ATOMICINCREMENT` operation. Such worst-case occurrences should not be as expected, especially in natural-looking and organic scenes. However, in scenes with more straightforward layouts with low amounts of different materials, this sequential behavior will reduce performance.

Authors suggest reducing this sequential behavior using shared memory and intra-warp level communication intrinsics; however, this availability may not exist on different GPU hardware vendors.

Our approach will have the opposite behavior; for a low amount of partitions, it performs *faster* compared to a high amount of partitions as one would expect, as opposed

to a queue-based approach that performs worse due to atomic pressure increase when the amount of partitions reaches to one.

3.4.3 Sort-based Partitioning

As we establish the shortcomings of the queue-based partitioning method, we will explain the proposed sorting-based method. As the name suggests, we sort the rays according to partition logic in the proposed method. In a path tracer, the logic will be the material evaluation. We partition rays via a material key, a single 32-bit integer. Each value of this queue represents both data and execution commonality. The layout of the Key Structure will be explained in Section 3.4.3.1. We employ an index-id pair sorting; thus, we do not touch the ray’s state during partitioning. The partitioning scheme will produce a shuffled list of IDs on one array and another that holds each partition’s start and end offsets. These offsets are relative to the generated ID array.

This eliminates the disadvantages of the queue-based approach; the sorting scheme for R amount of rays would require an amount that solely depends on R , and computation time is consistent.

As a choice of the sorting algorithm, radix sort is the preferred choice on GPU hardware [105]. The main reason for the selection is the performance; however, it exposes additional advantages when the partition count is low, which is a fortunate byproduct. To explain the performance gains using radix-sort, we are required to explain the underlying structure of the partitioning system, which will be explained in the next section.

3.4.3.1 Ray Payload & Key Parameter

Given a sequence of rays that happened to go through a ray tracing operation, rays will result in multiple data. In tracing terminology, this data is called “ray payload”. Figure 3.5 gives a stripped-down version of the proposed method’s ray payload.

For each ray, ray tracer kernel routines write multiple data, namely, “PartitionKey”, “PrimitiveId”, “TransformId”, and hit interpolation variables “Hit Float”. These vari-

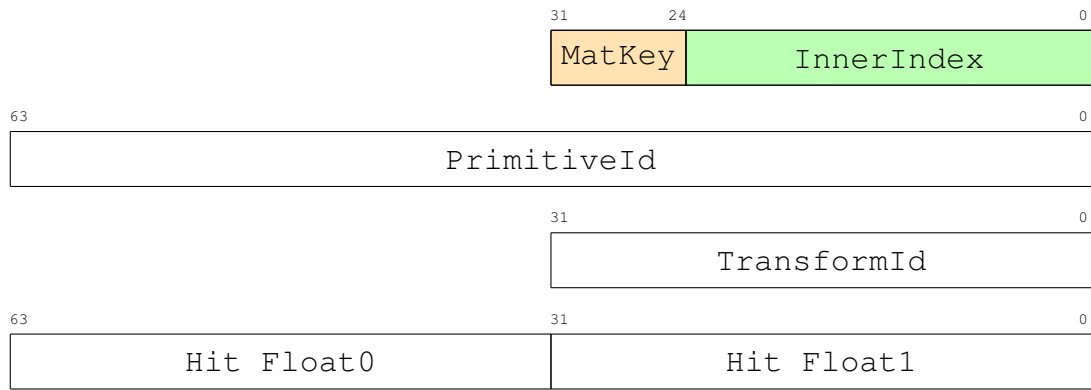


Figure 3.5: The payload of a ray. This representation excludes rendering method-related variables.

ables reside on the acceleration structure leaves and are written to the ray’s payload when the closest hit is found.

PrimitiveId represents a unique id of a primitive and is the index of that specific primitive over an array. To clarify further, every different type of primitive has its array, and this variable represents the index of that array. Distinguishing between various types of primitives is defined by the “PartitionKet” in addition to the material logic. “PrimitiveId” is a unique primitive identifier that will be utilized to access a certain primitive during material evaluation.

PartitionKey variable is split into two sections; the most significant 8-bit portion represents a unique primitive-material pair, and the lower bits represent the material ID for that specific material type. Again, the id here can be interchangeable with the index; every material type has a single array that holds all the different materials with the same type. Figure 3.6 gives a bit diagram of the Key Structure.

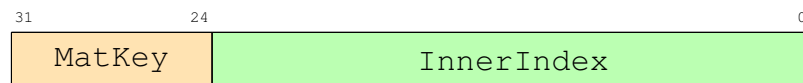


Figure 3.6: The material type key structure. The upper (most-significant) bits represent the code path, and the lower bits represent the data path.

We will clarify the data representation with a concrete example. Assume a scene consists mainly of triangles and some cubic splines. In this scene, multiple materials are present, which are purely diffuse, a mix of specular and diffuse, a sub-surface scattering material, and a custom hair material that simulates interreflections between

hair strands. All the materials have multiple data, such as texture maps and albedo colors.

Custom hair material is defined only over the spline primitives, and other material types only act upon the triangle primitives. In this case, there will be five different types of material-primitive pairs. An identifier of this is written on to high bits of the PartitionKey. All the other intra-type materials (i.e., blue diffuse, yellow diffuse) are uniquely identified by the “InnerIndex” portion of the key.

Hit values for triangle primitives are barycentric coordinates; for splines, a single value would suffice for inner-primitive interpolation (the t parameter). We conservatively allocate the hit buffer for the worst-case scenario. This conservative allocation has minimal impact, and the most interpolants are two. At most, the number of needed interpolants that occur when a volume representation is present on the scene can be three.

Since radix sort is stable and not every scene would require all the bits of the 32-bit word, we split sorting into two phases. First, the upper bits are sorted, and only the bits that conservatively encapsulate the different material-primitives pairs are sorted. If there is a m amount of different material types on the scene, it is enough to sort only $\lceil \log_2(m) \rceil$ bits on the higher level.

The least significant portion of the key (representing the data) does not need to be sorted. Execution logic is defined by the most significant portion of the key. We also chose to sort the least significant portion of the key due to the improvement of data coherency, which; in theory, should result in better performance due to cache coherency. Similarly, only the used bits are sorted. Since radix-sort is stable, we can sort the lower bits independently without scrambling the upper bits.

This concludes the sorting scheme of the partitioning routine. Since we only sort the required bits, performance should be better when the partition count is low. In the next Section, we will discuss the latter parts of the partitioning scheme.

3.4.3.2 The Algorithm

The rest of the algorithm is straightforward after the sorting is conducted. A kernel is launched to determine split locations. The kernel is launched with $n - 1$ threads. Each thread looks at its corresponding key-value pair and the next neighbor's pair. If a discrepancy between the upper 8-bit keys is detected, the *next* offset is written to an output array. Pseudocode of this process is given in Algorithm 2.

Algorithm 2 Mark Splits routine. Given n index-key pairs, the upper 8-bit is compared between the forward adjacent neighbor and the current pair. If a change is detected, $\text{offset}(j + 1)$ is written to an array; otherwise, zero is written.

Input-Output

$K = \{k_1, k_2 \dots k_n\}$ ▷ Set of keys
 $I = \{i_1, i_2 \dots i_n\}$ ▷ Set of corresponding ray indices
 $O = \{\}$ ▷ Empty offset output array

Start

for $j = 1$ to $N - 1$ **do**

$k'_j \leftarrow \text{ACQUIREUPPERBITS}(k_j)$

$k'_{j+1} \leftarrow \text{ACQUIREUPPERBITS}(k_{j+1})$

if $k'_j \neq k'_{j+1}$ **then**

$o_i \leftarrow j + 1$

else

$o_i \leftarrow 0$

end if

end for

Each operation in this process is independent and can be trivially parallelized by the GPU. The resulting buffer will have the starting positions of partitions with written zero values in between. In the end, the resulting data is sparsely filled on a buffer, and it should be converted into a dense structure. The following kernel call will handle these zeroes.

To clean away the zeroes, we employ a stable partitioning algorithm. This partitioning operates similarly to that used in the quick sort algorithm. It should not be confused with the general partitioning algorithm discussed in this Section. Our partitioning

algorithm does a “N-way partitioning,” and N is unknown. In this case, we do a binary partitioning that divides the structure into two segments. The algorithm checks the offset array and compares the values between zero and partitions. Figure 3.7 gives an array diagram of the operation.

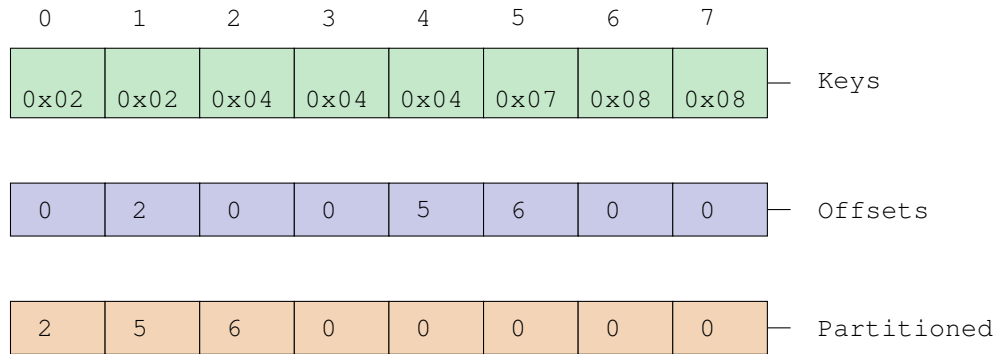


Figure 3.7: Array representation of the binary partitioning scheme and marking algorithms. The navy array is the result of the marking algorithm. The orange array is the result of the stable binary partitioning algorithm.

It should be noted that the binary partitioning algorithm should be stable, meaning the relative order of the value should not change. To iterate the example given in Figure 3.7, there are four partitions. The value of 4 is unknown until the last phase of the algorithm. After the mark and partition operations, we acquire three values representing starting offsets of the partitions of $p_2 \dots p_4$. The offset of the partition p_1 is implicit and is zero. As a final operation, we concatenate this implicit zero, and the ray counts to the resulting array’s start and end positions, respectively.

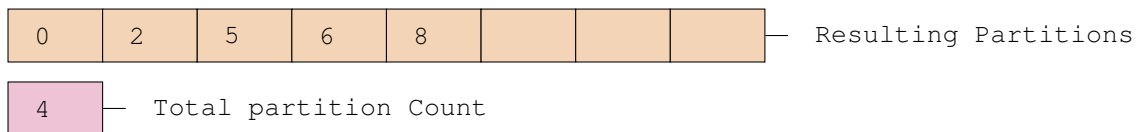


Figure 3.8: The final result of the sorting-based partitioning algorithm. All in all, a series of start and end offsets of each region is calculated.

The final result of the algorithm can be seen in Figure 3.8. Each adjacent pair in the resulting array represents a partition in which indices between $[p_i, p_{i+1})$ are members of the partition i .

The computation of the algorithm is consistent, meaning computation does not change

with respect to partition count. Memory usage is consistent as well. Given r amount of rays, this N-way partitioning system would require;

1. $2r$ for indices (input and output). The initial input is generated via *iota* routine.
2. r for keys (output). The input keys are readily available in the ray payload.
3. $O(sm^r)$ amount of extra memory for radix sorting. s is related to the SM count of the physical device.
4. r for writing offsets (Figure 3.7, blue array).
5. $O(sm^p)$ for binary partition. s is same as in the item 3.
6. $r + 1$ for the final output array and count.

Since all operations happen subsequently, the memory usage can be reduced by a smaller allocation. The allocation can be repurposed as needed. In the end, $2r + \max(O(sm^p), O(sm^r))$ amount of memory would suffice for the entire N-way partitioning operation.

All fundamental operations are done in GPU using CUDA [81]. Both the radix sort and partition algorithm used to construct this N-way partitioning algorithm are fundamental functional programming constructs and are readily implemented in CUDA's CUB Library [106]. The rest of the functions are trivial and implemented with custom kernels.

Finally, the resulting array and the partition count are transferred to the host device. Then, the host device launches a kernel for each specific partition. A specific BXDF-related kernel is launched for each partition in our material evaluation case.

3.5 Final Words

This section proposed an N-way partitioning algorithm for wavefront path tracing on GPU. Wavefront path tracing generates rays in bulk and segregates the ray tracing operation and the material evaluation portions of the scheme. Each ray undergoes

ray tracing routines using intersection accelerators to find the closest hit surface and the material of that surface. Then, each ray is partitioned with respect to surface and material pairs, and specific material-related routines are applied to each partition. Rays “bounce” over the surfaces and continue to different surfaces; thus, the entire path space is explored.

During material evaluation routines, rays that hit an emitter write their contribution to the corresponding pixel. Rays can cease to exist due to Russian Roulette as well.

In the next chapter, we explore the usage of this partitioning scheme for path tracing and other extension methods, such as path guiding.

CHAPTER 4

WAVEFRONT PATH GUIDING

This chapter will propose a novel approach that utilizes the partitioning scheme explained in Chapter 3. In short, we employ a path-guiding approach that generates an incoming radiance field *on the fly*. From this radiance field, a probability density function will be generated and used for importance sampling.

4.1 Brief Refresh of Path Guiding

This section briefly explains the path-guiding methods and their main motivation. The rendering equation is defined by the following integral [1].

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_{\Omega} f_x(p, \omega_i, \omega_o) L_i(p, \omega_i) \cos \theta_i \partial \omega_i \quad (4.1)$$

As discussed before, this integral is estimated over all of the camera-contributing surfaces over the scene. A Monte Carlo estimator for this function can be seen in the equation below.

$$\langle L_x(p, \omega_o) \rangle = \frac{1}{N} \sum_{i=1}^n \frac{f_x(p, \omega_i, \omega_o) L_i(p, \omega_i) \cos \theta_i}{p(\omega_i|x, \omega_o)} \quad (4.2)$$

Traditionally, $p(\omega_i|x, \omega_o)$ is related to the BxDF function of $f_x(p, \omega_i, \omega_o)$. Sampling over the BxDF is straightforward since all of the data is locally available during the sample evaluation time¹. On the other hand, sampling using $L_i(p, \omega_i)$ portion is not straightforward since the L_i term also depends on other inductions of Equation 4.1.

¹ $\cos \theta_i$ term as well

Path guiding approaches try to achieve precisely this, approximating a robust incoming radiance field *all* the camera contributing regions of the scene. As one may observe, the incoming radiance field across the entire scene constitutes a five-dimensional function: three dimensions for spatial information and two for directionality.

Another problem with the path-guiding approaches is that the required data (usually the incoming radiance field over a particular location) is initially unavailable. The radiance field requires an estimated path-tracing approach; however, we are trying to do path-tracing anyway. This situation gives rise to a classic “chicken and egg” dilemma. We need specific data to perform a calculation, yet this data is generated as an outcome of the calculation we aim to execute.

Because of that, path-guiding methods pre-generate this radiance field or use the acquired samples of the path tracer to estimate the radiance field during execution. For the latter case, efficient extrapolation data is required for optimal execution.

In theory, path-guiding methods can generate very high-quality samples at the expense of computation cost. However, more straightforward high-speed methods can outperform the path-guiding methods in equal time measure. Thus, a path-guiding method should optimally divide the computational resources between estimating the radiance field and doing the actual rendering.

The final issue arises in the concept of memory. Scene radiance field is, as discussed, a 5-dimensional function, and holding the radiance field naively, for example, over a dense 5D array, would require an excessive amount of memory due to the curse of dimensionality. This bottleneck is especially predominant on GPUs since GPUs have relatively low memory capacity compared to CPUs.

Most of the time, path-guiding approaches divide this 5D field into two portions. One portion is responsible for spatial subdivision, and each spatial subdivided region contains a . A classic example of this approach can be seen in Müller et al.’s work in which a binary tree is responsible for spatiality. And for directional portions, a quad-tree is used [64]. For the directional portion, a combination of integrable analytical functions can also be utilized [60].

Data structures used for path guiding have no information about the scene layout at the start. As the samples are fed into the system, these structures are adaptively refined to a method-defined threshold to prevent excessive memory usage. Such approaches are ill-suited for GPUs since adaptive data structures mean fine-grained dynamic memory management, which is not straightforward.

Another issue with dynamic data structures is that such structures almost always have branching behaviors (such as trees), and memory access to these data structures is incoherent. CPUs can hide this incoherent access via sophisticated caches; however, GPUs do not have this functionality.

Instead of separately sampling via BxDF and this radiance field, the product of these two values can also be used for sampling. Such an approach is called “product path guiding”. Not every path-guiding method trivially enables product sampling; discretely represented radiance fields require a non-trivial amount of computation to sample from the product. On the other hand, analytically represented radiance fields are more straightforward.

Considering this, we propose a GPU-oriented path-guiding method that seamlessly fits the previously explained wavefront path tracer. Unlike existing approaches, we estimate the radiant exitance² of the regions and utilize the radiant exitance to generate radiance field *on-the-fly*. This method will use GPU-specific functionalities, especially the hardware ray-tracing capabilities of the modern GPUs.

4.2 Overview

An overview of the design can be seen in Figure 4.1. This block diagram is similar to the 3.4 in Chapter 3 with additional extensions. Two extra phases are introduced for path guiding. The “guiding” phase occurs before the BxDF evaluation and generates guided directions via creating a radiance field, generating a PDF from that field, and sampling over this generated PDF.

Generating a radiance field for a single ray is unfeasible. Since the wavefront path

² Radiant exitance is the inverse of irradiance, which is the radiant flux *emitted* by a surface per unit area.

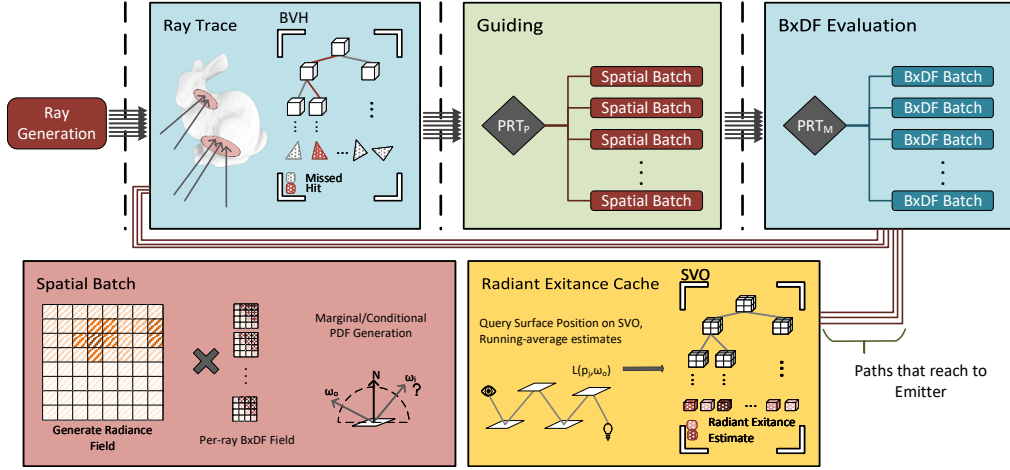


Figure 4.1: The top-down view of the entire path guiding algorithm. Blue rectangles of the image show the wavefront path tracing operations. Other colored parts are the additional steps required for guiding the rays. PRT_p and PRT_m sections represent partitioning the rays by position and material, respectively. Device code is executed for each spatial batch. Each batch generates an incoming radiance field incorporated into the sampling scheme. Paths that reach an emitter contribute to an approximation of the radiant exitance, which is cached on an SVO.

tracers already have many rays in circulation, we *amortize* the radiance field generation over the many rays. To accomplish this, rays are partitioned according to their positions. Then, the partitioned rays *collaboratively* generate an estimate of the incoming radiance field and the PDF.

The Pseudocode of the wavefront path guiding method is given in Algorithm 3. The overall layout of the algorithm is similar to the wavefront path tracing algorithm presented in Chapter 3 with additional extensions. PARTITION-S routine is precisely the same partitioning routine discussed in Chapter 3 but with a different key value. The generation of this key value will be discussed in Section 4.3. Another difference is that the BxDF evaluation routine is not responsible for generating new rays but only actually evaluates the reflectance function using the direction generated by the guiding routine.

Another extension to the path tracing method is the radiant exitance caching. In this case, the rays that reach an emitter deposit their radiance information onto a spatial radiant exitance cache. Later, the cached results will be utilized for approximating the

Algorithm 3 Wavefront path guiding.

Input-Output

$$R_1 = \{r_1, r_2 \dots\}$$

▷ Set of initial rays

Start**Initially** Generate rays from the camera and populate R_1 **for** $i = 1$ to MaxDepth **do**

$$B_i = \{(p_1, R_{p_1}), (p_2, R_{p_2}) \dots\} \leftarrow \text{PARTITION-S}(R_i)$$

$$N_i = \{(n_1, R_{n_1}), (n_2, R_{n_2}) \dots\} \leftarrow \text{PARTITION-M}(R_i)$$

for all $(p_j, R_{p_j}) \in B_i$ **do**

$$R_{i+1}^j \leftarrow \text{GUIDERAYS}((p_j, R_{p_j}))$$

end for**for all** $(n_j, R_{n_j}) \in N_i$ **do****for all** $r_k \in R_{n_j}$ **do**

$$\text{EVALUATEBXDF}(n_j, r_k)$$

end for**end for**

$$R_{i+1} = \{R_{i+1}^1, R_{i+1}^2 \dots\}$$

▷ Next set of rays

end for**for all** paths that reach an emitter **do**

$$\text{UPDATEEXITANCE}(SVO)$$

end for

radiance field over a specific region in the scene. In the next section, we will discuss this radiance exitance caching methodology.

4.3 Radiant Exitance Caching using Sparse Voxel Octree Structure

To explain the caching scheme, we need to expand the rendering equation. Equation 4.3 gives a recursively expanded rendering equation formulation [2].

$$\begin{aligned}
P(\bar{p}_n) &= \underbrace{\int_{\Omega} \int \cdots \int_{\Omega}}_{n-1} L_e(p_n, \omega_{o_{n-1}}) \\
&\times \left(\prod_{j=1}^{n-1} f_x(p_{j+1}, \omega_{o_j}, \omega_{i_{j-1}}) \cos \theta_i \right) \partial \omega_{i_2} \cdots \partial \omega_{i_n}
\end{aligned} \tag{4.3}$$

($\prod \dots$) term in this equation is defined as path throughput ($T(\bar{p}_n)$). While estimating the integral via Monte Carlo formulation throughput $T(\bar{p}_n)$ can be represented as follows.

$$T(\bar{p}_n) = \prod_{j=1}^{n-1} \frac{f_x(p_{j+1}, \omega_{o_j}, \omega_{i_{j-1}})}{p(\omega_{i_{j-1}} | x, \omega_{o_j})} \cos \theta_i \tag{4.4}$$

The outgoing radiance sample at point p_k can be extracted from the total throughput $T(\bar{p}_n)$ and the throughput of the path vertex $T(\bar{p}_k)$.

$$L_{o_k}(p_k, \omega_{o_{k-1}}) = \frac{T(\bar{p}_n)}{T(\bar{p}_k)} L_e(p_n, \omega_{o_{n-1}}) \tag{4.5}$$

This necessitates holding the entire path chain while path tracing. The payload of a ray holds its path chain internally; only the position and throughput of each path vertex are stored.

After an emitter is encountered on the path chain, the ray calculates the outgoing radiance of each vertex and deposits it onto a Sparse Voxel Octree (SVO) structure. SVO is used to approximate the scene layout; a voxel is created for each occupied scene region. Each voxel holds the radiant exitance and approximation of the surface normals.

SVO is constructed using Crassin et al.'s approach [88]. This operation utilizes a hardware rasterizer and is inherently GPU-oriented. Scene layout is pre-determined; thus, such generation can be done before the rendering. Initially, SVO will be empty; as the rays deposit the data, the scene's global radiant exitance estimate will be generated. The generation scheme also utilizes Karras et al.'s approach for constructing

linear bounding volume hierarchies (LBVH) [101]. Indices of the generated voxels are converted to Morton Codes (see Figure 4.2) and sorted. Discrepancies between these morton codes directly represent the subdivision structure of the SVO.

More specifically, each (x, y, z) triplet in the MortonCode represents the child index of that voxel. The most significant triplet shows the subdivision of the root. The next two triplets combined will give the second level node of that particular voxel. And this pattern continues. Since voxels are sorted, discrepancies between these bits will determine the change of parent hierarchy between nodes. By this observation, the hierarchy of the SVO can be generated.

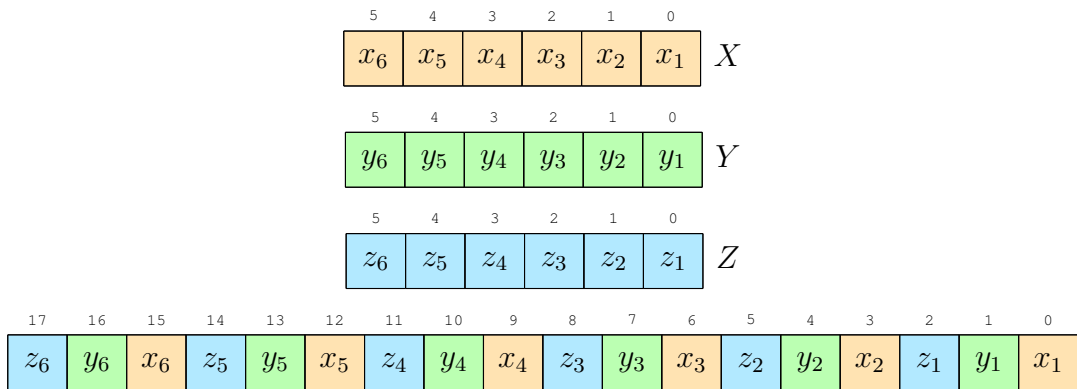


Figure 4.2: Morton code encoding visualization. Given a 6-bit voxel index of (x, y, z) , encoding interleaves the bits of each dimension $(z_6, y_6, x_6 \dots, z_1, y_1, x_1)$. In the actual case, the Morton Code width is much larger.

By definition, radiant exitance is directionless. Because of that, we are not required to hold a directional data structure on the leaves of the SVO. This drastically reduces memory requirements. As we discussed above, SVO is pre-generated. Thus, the rendering does not require subdivision, enabling simpler memory management.

Since a voxel is inherently a volumetric structure, representing surfaces creates challenges. Imagine a two-sided, infinitely thin object, such as a wall separating regions with drastic illumination discrepancies. Such voxel will transfer the radiance information incorrectly to the other side creating light leaks. The main reason is the resolution mismatch; we could not represent some surfaces perfectly because of memory limitations.

Thus, an approximate surface representation is required. To this end, we store the normals on the voxels. Again, because of the resolution mismatch, multiple triangles

may end up on the same voxel (Figure 4.3). In order to determine the approximate surface normal, we utilize simple k-means clustering where $k = 2$ to determine the normal.

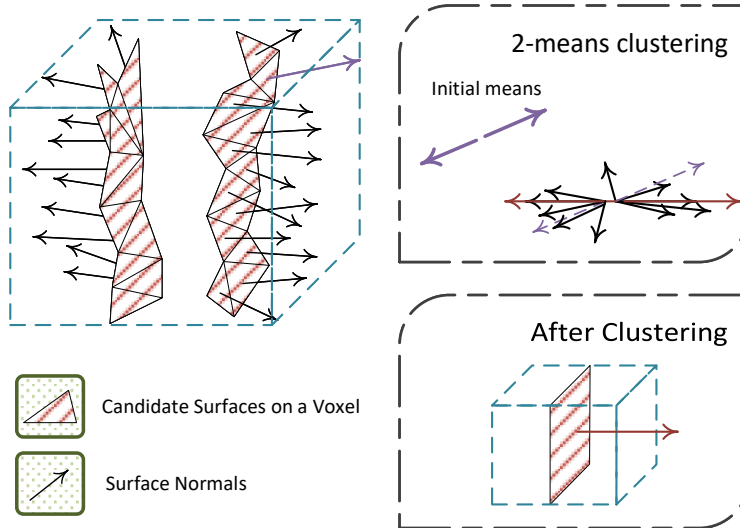


Figure 4.3: K-means clustering algorithm sketch. This figure assumes that the voxel resolution is lower than the triangle resolution in the context. Each triangle will be rasterized onto this single voxel. The approximate normal is calculated by conducting a 2-means clustering. This prevents normals of very thin, two-sided objects from being canceled out.

We select initial means \vec{N}_0 and \vec{N}_1 via selecting the first normal from the pool, namely \vec{N}' , \vec{N}_0 will be \vec{N}' and \vec{N}_1 will be $-\vec{N}'$. With that, normals of two surfaces oriented in opposite directions that share the same voxel will not cancel each other out. Figure 4.3 shows an example of this approach.

By introducing normals, we effectively created two sides of a voxel. Because of that, radiant exitance for both sides is stored in a voxel. While depositing, the ray will determine which side they should deposit into by comparing the direction with the normal of the voxel.

Radiant exitance is only deposited into the leaf voxels of the SVO. However, we require all nodes of the SVO to hold a radiant exitance value. The reason is to employ a cone tracing routine, which will be explained in Section 4.4. Thus, we filter the leaf values towards the root in a bottom-up fashion.

This concludes the radiant exitance caching scheme of the system. Rays that undergo guided path tracing will fill the structure during rendering. While rendering, rays will use this information to guide rays. The following section will specify the usage of the radiant exitance cache for path guiding.

4.4 On-the-fly Generation of Radiance Field

Our path tracing scheme is a wavefront scheme; thus, many rays are in circulation during rendering. During rendering, these rays may end up in similar locations with similar incoming radiance fields. Because of that, we partition these rays with respect to the positions. After partitioning is conducted, grouped rays collaboratively generate the radiance field and do sampling. Sections 4.4.1 and 4.4.2 will explain these processes.

4.4.1 Partitioning

To partition the rays with respect to position, we utilize the partitioning scheme discussed in Chapter 3, however, with a different key value. The generation of this key value will use the generated SVO itself. SVO already partitions the scene by volume, so using this already-generated data structure will be convenient. We will acquire the voxel index from the SVO (an integer index (x, y, z)) by descending towards the leaf. When we encounter the leaf, we increment an atomic counter to express the usage of that leaf.

Rays could be directly partitioned without the SVO by discretizing the position information. We specifically utilize SVO to merge the partitions indicated by the leaves further. Algorithm 4 explains the process in the pseudocode. This process will further amortize the radiance field generation cost by reducing the field count per ray.

After rays mark the leaves of the SVO, we bottom-up merge the nodes of the SVO. This collapse process is orchestrated by two user-defined parameters, namely “min-BinLevel” l_{min} and “binRayCount” c_{ray} . l_{min} puts an upper bound for this collapsing scheme where no collapsing would occur for levels above the SVO level l_{min} . On the

Algorithm 4 PARTITION-S Routine. Partition the paths that have hit p_i to series of bins b_j using an SVO with the depth d .

Input

$R = \{r_1, r_2 \dots\}$ ▷ Rays that are going to be partitioned

$SVO = \{(n_1)^1, (n_1, \dots)^2 \dots (n_1, \dots)^d\}$

Output

$B = \{(p_1, R_{p_1}), (p_1, R_{p_2} \dots)\}$ ▷ Pair of position and ray pools

Buffer

$I = \{b_1, b_2 \dots b_i\}$ ▷ Bin id for each ray

Start

for all $r_i \in R$ **do**

$p_i \leftarrow \text{RAYPOSITION}(r_i)$

$n_i^d \leftarrow \text{DESCENDLEAF}(p_i)$

$\text{ATOMICADD}(n_i^d, 1)$

$b_i \leftarrow \text{NODEID}(n_i^d)$

end for

for all $l \in SVO$ (in bottom-up fashion, up to l_{min}) **do**

for all $n^l \in (n \dots)^l$ in SVO level l **do**

$C = \{c_1, c_2, \dots, c_8\}$ ▷ node children's path count

$T \leftarrow c_1 + \dots + c_8$

if $T \geq c_{ray}$ **or** $l = l_{min}$ **then**

$\text{MARKNODE}(n_i^l)$

end if

end for

end for

for all $b_i \in I$ **do**

$n_i^d \leftarrow \text{TONODE}(b_i)$

$n_i^l \leftarrow \text{ASCENDANDFINDMARKED}(n_i^d)$

$b_i \leftarrow \text{NODEID}(n_i^l)$

end for

$B \leftarrow \text{PARTITION}(I, R)$

other hand, c_{ray} defines how many rays are deemed enough for amortization. When a node satisfies these constraints, it is marked.

Rays then again descend on the SVO and find this marked SVO node. When encountered, the node index of that voxel is written to a buffer. This node index value will be the key parameter for the partitioning scheme. Finally, we have partitioned the rays spatially.

4.4.2 Radiance Field Generation

Each partition will generate a single omnidirectional radiance field. The reason for the omnidirectional radiance field comes from the fact that we do not know which portions of the radiance field are required for the surfaces. We hold surface normal; however, such normal is only approximate; it can be an average normal for many small, differently oriented surfaces. Furthermore, these surfaces may be refractive, meaning a whole omnidirectional field is needed. For these reasons, we conservatively generate the entire omnidirectional radiance field.

We attach a single block to each partition. Threads on each block will generate a radiance field using the SVO, and then they read the ray information and sample directions using the generated radiance field. More importantly, the shared memory of each block will be used for temporary and fast memory, effectively eliminating the extra memory cost. Since shared memory does not persist between the kernel calls, it will automatically be discarded after a block is finished with a partition. The Overview of this process can be seen in algorithm 5.

To generate the incoming radiance field, we utilize the cone tracing approach. Given the location of the scene p_k , omnidirectional radiance field $L(p_k, w_i)$ is stratified to equal area patches of $\partial\omega$. For each patch, a cone is launched in that direction. The cone conducts its tracing and effectively finds a location p' , and then this location is queried on SVO by looking at the cone's aperture and position.

The reason we utilize the cone tracing approach has a practical basis. Due to computational concerns, we could only reasonably launch hundreds of rays for a given partition. Thus, we require an efficient estimation of the radiance field using a minimal

Algorithm 5 GUIDERAYS routine. Given a bin with partitioned rays, generate incident radiance field, generate PDF and CDF, and sample either using path guiding or BxDF via MIS.

Input

(p_j, R_{p_j}) ▷ Partitioned position and rays

Output

R_{i+1}^j ▷ Guided rays

Buffer

$L(p_i, \omega_i)$ ▷ Incoming Radiance Field on shared memory

$PDF(\omega_i), CDF(\omega_i)$ ▷ PDF and CDF on shared memory

Start

$p_o \leftarrow \text{SELECTORIGIN}(R_{p_j})$

for all $\omega_i \in \Omega$ **do**

$L(p_o, \omega_i) \leftarrow \text{CONETRACE}(SVO, p_o, \omega_i)$

end for

$CDF(\omega_i), PDF(\omega_i) \leftarrow \text{GENERATEPDF-CDF}(L)$

for all $r_k \in R_{p_j}$ **do**

$M \leftarrow \text{ACQUIREMATERIAL}(r_k)$

$r_{k_{next}} \leftarrow \text{MIS}(PDF(\omega_i), CDF(\omega_i), M)$

end for

$R_{i+1}^j = \{r_{1_{next}} \dots\}$

amount of ray-casting operations. By cone tracing, a single cone would accurately estimate large $\partial\omega$ patches.

The projected area with respect to the cone solid angle ω and the distance r can be formulated as in Equation 4.6.

$$A = r^2\omega \tag{4.6}$$

The solid angle area is then assumed to be a circle due to conal representation, and the relation between the voxel size of i^{th} level of the SVO v_i can be compared as follows.

$$\begin{aligned}
R^2 &= 4 * A/\pi \\
v_i^2 &\sim R^2
\end{aligned}
\tag{4.7}$$

This is the reason for filtering the leaf radiance exitance values over the upper levels of the tree. Such an approach estimates radiant exitance efficiently and minimizes the amount of tracing that would be needed.

A couple of approaches can be utilized to find the incident hit location. A volumetric estimation proposed by Crassin et al. [94] can be used. Despite being efficient, such an approach is prone to light leaks. Empty space skipping cone tracing, which is discussed by Laine et al. [82], can be utilized as well. In this approach, rays are marched traditionally in an empty-space skipping manner. As rays are marched further, the cone’s projected area over that area gets larger as well. When it matches the voxel area of an SVO tree level, tracing terminates, and radiant exitance can be directly queried from that level of the tree.

Although the above tracing schemes would reasonably estimate the incoming radiance field, such approaches are quite slow. Our path-guiding strategy relies on many ray-casting operations for the radiance field estimation over many locations on the scene. We propose a different approach that utilizes the device’s hardware-accelerated ray tracing capabilities.

Unlike other methods, we directly utilize the ray-tracing hardware on the device. Instead of tracing the SVO, we trace the scene’s acceleration structure and find a hit position. From that hit location, distance r is calculated, and Equations 4.6 and 4.7 are calculated as if we traced a cone. Such an approach is approximate since we could not incrementally check Equation 4.7.

Cone tracing occurs with a user-defined resolution $X \times Y$. This can be considered as a partition-local camera-generated environment map. By this observation, we utilize traditional environment sampling schemes. The radiance field is treated as a piecewise constant 2D function for which the traditional inverse sampling method is utilized. An example of the method can be found in the *Physically Rendering Book* [2] or Shirley’s chapter on Ray Tracing Gems [107]. An example of the process can be

seen in Figure 4.4.

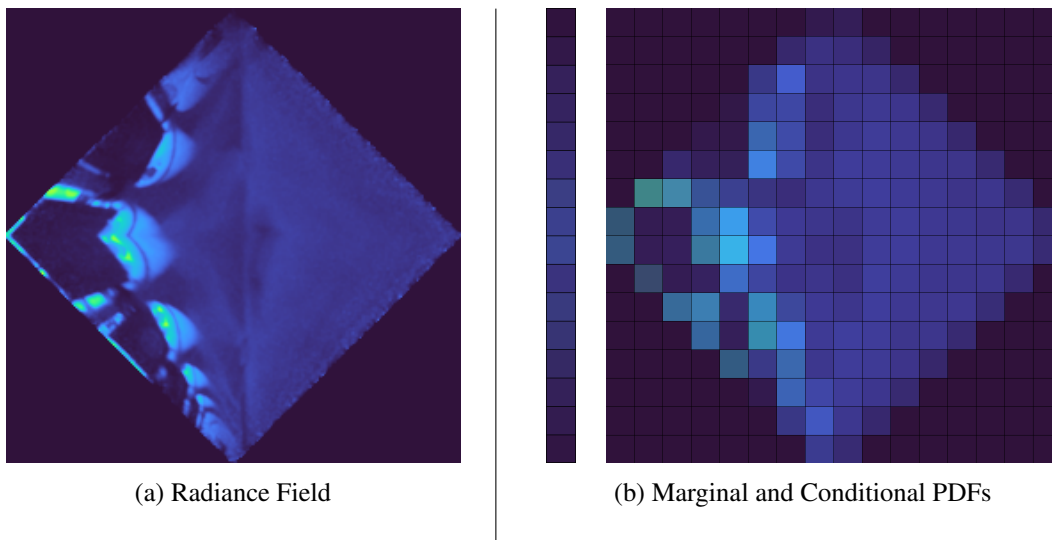


Figure 4.4: An example of generated marginal and conditional PDFs from the radiance field. For demonstration purposes, the PDFs are low-resolution. The marginal PDF function is only a one-dimensional array and is responsible for selecting a row to find a conditional PDF.

More specifically, the computation is done as follows. Each row of the generated 2D dense radiance field is considered a 1D piecewise constant (PWC) function. Each row is integrated to find the unnormalized CDF function. Since the function is a PWC integration, it corresponds to a “scan” (prefix-sum) operation. Scan operation is similar to reduction operation but has additional complexity. For a list $A = \{n_1, n_2, n_3, \dots, n_N\}$, it creates N element list where the element $n'_i = \sum_{j=1}^i n_j$.

Each block calculates the row integral in parallel. The last element of the resulting scan operation is the summation of all row values. We divide all the elements to create the actual CDF. Radiance field values are also divided to generate the row PDFs. This total value is also used to create the marginal PDF. For $X \times Y$ radiance field, we have a single Y sized function that undergoes the same operations described above.

Generating the CDF values and normalizing the radiance field to generate PDF is highly parallelizable; thus, block threads do these operations in parallel. The scan operation is a fundamental functional programming construct and a readily available CUDA routine. For our implementation, we utilized CUDA’s implementation.

Sampling from this 2D field requires a two-phase approach. Two random values ξ_1, ξ_2

are generated where $\xi \in [0, 1)$. First, a binary search is conducted over the marginal CDF function using ξ_1 , and a row is found. The row's CDF is also binary searched via ξ_2 , and the actual sampled cell is determined. The PDF value will correspond to the multiplication of the found marginal and conditional PDFs.

4.5 Exposing BxDF Product the Radiance Field

For discretized path-guiding methods, conducting a product sampling scheme is non-trivial. In the product path guiding method, we utilize the product of the BxDF and the radiance field as a sampler instead of directly sampling the radiance field. We utilize Estevez et al.'s approach [108], which is proposed for environment maps. Since the generated radiance field can be considered as an environment map this method is suitable for product path guiding.

In this approach, the radiance field is divided into two layers: one low-resolution layer and a high-resolution layer. The low-resolution layer is then multiplied with the discrete representation of BxDF. Similar approaches for sampling are conducted for the low-resolution level discussed in the section above. One difference is the binary search; we employ warp-level intrinsics for searching the CDF, which corresponds to a parallel brute-force search.

Unlike the non-product sampling method described above, an entire block could not collaboratively generate these low-resolution multiplication fields since each ray may have a different material. Because of that parallelization scheme is different. While sampling, a block issues a single warp for each ray. The radiance field query is still done at the block level. Generating the low-level version of the radiance field is also done at the block level. Then parallelization scheme is changed within the block, and a single warp becomes responsible for each ray. A warp calculates BxDF for each cell and multiplies it with the low-resolution radiance field. This multiplied field is sampled, and a region is determined. For this region, corresponding cells of the high-resolution field undergo the same sampling scheme. For example, given 32×32 radiance field and assuming 8×8 field is used for product portion, an inner 4×4 field will be sampled.

The pseudocode of the routine is given in Algorithm 6. Only the sampling scheme for one of the levels is given. For the pseudocode to be understandable, we must explain the intrinsics used in the routine.

Algorithm 6 PRODUCTSAMPLE Routine. Given two uniform numbers, sample a region from the outer radiance field that is multiplied by the reflectance. Returns the UV coordinates of the sampled location and the multiplied probability. w is thread per warp and $m = n^2/w$ (conditional pdf value per warp). t_{id} is the thread identifier (between $[0, w)$). The algorithm assumes $n < w$ and w is evenly divisible by n .

```

1: Input
2:  $\xi_0, \xi_1$  ▷ Random values between  $[0, 1)$  available only for  $t_{id} = 0$ 
3:  $\text{PDF}_X = \{px_1, px_2 \dots px_m\}$  ▷ Row PDF values
4:  $\text{CDF}_X = \{cx_1, cx_2 \dots cx_m\}$  ▷ Row CDF values
5:  $\text{PDF}_Y = p_y$  ▷ Marginal PDF values (First  $n$  threads has this)
6:  $\text{CDF}_Y = c_y$  ▷ Marginal CDF values (same as above)
7: Output
8:  $u, v$  ▷ Normalized 2D coordinates of the sampled location
9:  $pdf$  ▷ Final pdf value
10: Start
11: Sample Marginal PDF
12:  $\xi_0 \leftarrow \text{SHUFFLE}(\xi_0, 0)$ 
13:  $\text{mask} \leftarrow \text{BALLOT}(\xi_0 > c_y)$ 
14:  $\text{rowId} \leftarrow \text{FINDFIRSTSET}(\sim\text{mask}) - 1$ 
15:  $c_{y_{\text{sample}}} \leftarrow \text{SHUFFLE}(c_y, \text{rowId})$ 
16: if  $\text{rowId} == n$  then ▷ Eliminate edge case
17:    $c_{y_{\text{next}}} \leftarrow 1$ 
18: else
19:    $c_{y_{\text{next}}} \leftarrow \text{SHUFFLE}(c_y, \text{rowId} + 1)$ 
20: end if
21:  $v \leftarrow \left( \text{rowId} + \frac{\xi_0 - c_{y_{\text{sample}}}}{c_{y_{\text{next}}} - c_{y_{\text{sample}}}} \right) / n$ 

```

Shuffle intrinsic function broadcasts word-sized values (1st argument) to the threads from the thread that is distinguished by the 2nd argument. Finally, it returns the broadcasted value. *Ballot* routine bitwise packs and returns the provided predicate (1st ar-

gument) to all the threads on the warp, meaning each bit will be the result of the predicate calculated by a thread. Although not warp-related, *FindFirstSet* routine returns the index of the first set bit on a word. The order is from the least to the most significant bit.

Given these descriptions, an entire single-layer product sampling routine can be seen in Algorithm 6. Interestingly, rejection sampling can also be utilized for the inner region since the resolution of the inner region is relatively low compared to the number of threads in a warp.

Algorithm 7 PRODUCTSAMPLE Routine continued.

22: **Sample Conditional PDF**

23: $\text{myColId} \leftarrow t_{id} \% n$

24: $\text{myRowId} \leftarrow t_{id} / n$

25: $\text{localIndex} \leftarrow (\text{rowId} / m)$

26: $\xi_1 \leftarrow \text{SHUFFLE}(\xi_1, 0)$

27: $\text{mask} \leftarrow \text{BALLOT}(\xi_1 > \text{CDF}_X[\text{localIndex}])$

28: $\text{mask} \leftarrow \text{mask} \ll \text{rowId} \% m$

29: $\text{colId} \leftarrow \text{FINDFIRSTSET}(\sim \text{mask}) - 1$

30: $\text{colThrd} \leftarrow (\text{rowId} * n + \text{colId}) / w$

31: $c_{x_{\text{sample}}} \leftarrow \text{SHUFFLE}(\text{CDF}_X[\text{localIndex}], \text{colThrd})$

32: **if** $\text{colThrd} == n$ **then** ▷ Eliminate edge case

33: $c_{x_{\text{next}}} \leftarrow 1$

34: **else**

35: $c_{x_{\text{next}}} \leftarrow \text{SHUFFLE}(\text{CDF}_X[\text{localIndex}], \text{colThrd} + 1)$

36: **end if**

37:

38: $u \leftarrow \left(\text{colId} + \frac{\xi_1 - c_{x_{\text{sample}}}}{c_{x_{\text{next}}} - c_{x_{\text{sample}}}} \right) / n$

39:

40: **Find PDF**

41: $\text{pdf}_y \leftarrow \text{SHUFFLE}(p_y, \text{rowId})$

42: $\text{pdf}_x \leftarrow \text{SHUFFLE}(\text{PDF}_X[\text{localIndex}], \text{colThrd})$

43: $\text{pdf} \leftarrow \text{pdf}_x * \text{pdf}_y$

CHAPTER 5

IMPLEMENTATION AND RESULTS

In this section, we will compare state-of-the-art methods and our method and discuss the practical details of the proposed method. Throughout the chapter, our wavefront path guiding method will have an abbreviation of WFPG.

5.1 Implementation

We have implemented our algorithm using CUDA [81]. For hardware-accelerated ray tracing, we utilize the OptiX Framework [109]. The entire wavefront path tracing and path guiding implementation can be found publicly [110].

In this Section, we will discuss the practical concerns of the proposed wavefront path guiding method.

OptiX framework and shared memory. Unfortunately, OptiX does not expose ray-tracing capabilities on custom kernels. Graphics-related APIs like DirectX Ray Tracing and Vulkan Ray Tracing have inline ray tracing capabilities over the compute shaders. However, Optix only exposes a ray tracing pipeline style of the programming model. Because of that, we utilize a small persistent buffer to transfer the ray-traced radiance field between the OptiX kernel and the sampling kernel. The allocation amount depends on the SM count of the running device. In our experiments, 8 to 16 MiB of memory is enough to saturate a mid to high-end GPU.

Aliasing. As discussed in Chapter 4, after we generate the radiance field and use it for sampling, we discard it due to memory concerns. Similar to the real-time rendering paradigm, aliasing becomes an issue. Another reason is the relatively low resolution

of the generated radiance fields. High-frequency illumination or occlusion may not be captured with a single sample. An example of this is given in Figure 5.1.

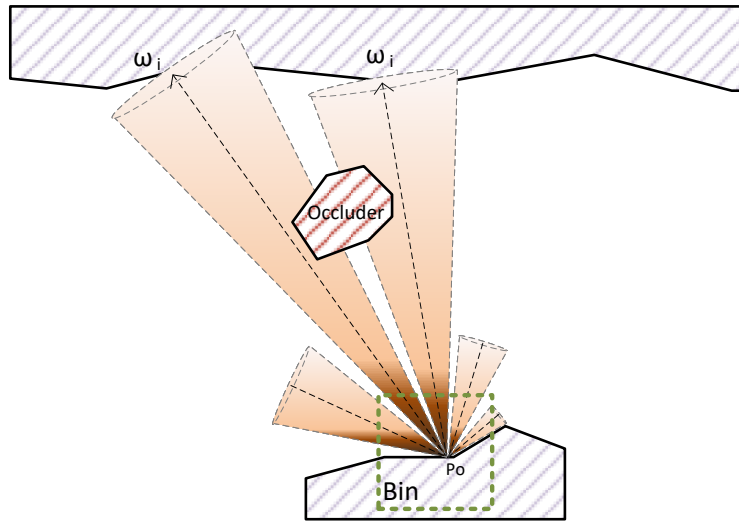


Figure 5.1: Aliasing illustration. Assuming the radiance field is generated over the volume represented by the green dashed square, The radiance field is generated from a point p_o . The contribution of a small occluder (shaded red) could not be captured due to the low-resolution radiance field. Cone rays miss the occluder, and the radiant exitance of the surface behind is queried.

Therefore, we employ a very basic and fast anti-aliasing scheme. We do a Gaussian Blur over the entire image and jitter the sampling directions while generating the radiance field.

Aliasing aside, this alleviates another primary issue; it matures the estimated radiance field and reduces harsh variance estimates over high-frequency changing regions. With that, all rays in the regions have a usable field. Since the generated radiance field corresponds precisely to a single point over the region, rays that trail slightly different parts of the scene (but in the same bin) would require slightly different fields.

Radiance field capture origin. Selecting the “rendering” origin for the radiance field is not simple. One may directly select the center point of the partitioned region for a captured origin. However, such utilization may create self-occlusions or create “variance seams”. Instead, we randomly select a candidate ray’s hit position as

origin every time a radiance field for that region is needed to be generated. Such an approach minimizes or eliminates the variance seams, making the rendering image more suitable for a potential denoising post-process.

Multiple Importance Sampling (MIS). Path guiding methods would require impractical memory or computation time to be “zero variance” sampling schemes. Our method is no different; unlike other path-guiding proposals, our method will have fewer memory concerns but more significant computational concerns. Due to that, we utilize the path-guiding method as a sister sampling method with the traditional BxDF sampling method using MIS. This reveals a practical computational hiccup for the proposed method. Since we only partition with respect to position, we require both the BxDF’s PDF and the radiance fields’ PDF to sample using MIS. However, we did not partition for material; thus, branching discrepancies would occur. Such discrepancies are minimal in practice since nearby regions mostly have the same material. This approach is nonexistent for the product sampling scheme since the entire warp is responsible for a single ray; thus, no branch mismatches happen in a warp.

Radiance field projection onto 2D Cartesian Space. We do not explain the projection scheme in Chapter 4. Since we create a relatively low-resolution radiance field, every pixel is important. To this end, we utilize an equi-area sampling method proposed by Clarberg [111]. Concentric octahedral mapping would give better results on lower resolutions than other classical projection techniques, such as Spherical Projection.

5.2 Parametrization

There are multiple parametrization variables available for the implementation, mainly c_{ray} (“BinRayCount”) and l_{min} (“MinSVOLevel”) as discussed in Chapter 4. Additional parameters are “BxDF-WFPGMISRatio”, “OctreeLevel”, “FieldResolution” and “FieldFilterAlpha”.

“BinRayCount” and “MinSVOLevel” handle how many rays should reside in a bin. Higher “BinRayCount” and lower “MinSVOLevel” create spatially larger bins and reduce bin count. Ultimately, this results in faster computation. However, having

bins that cover larger portions of the scene may result in a suboptimal radiance field for all of the rays inside the bin. Having fine-grained bins will result in inefficient utilization of the GPU since the generated radiance fields will be utilized by fewer rays, thus reducing performance. Therefore, tuning of these parameters is essential. In our experiments, “BinRayCount” of 512 and “MinSVOLevel” of “OctreeLevel” - 2 result in optimal performance in the resulting scenes.

“OctreeLevel” defines an upper bound for the leaf level of the SVO. Increasing this parameter will result in a higher resolution radiant exitance field for the entire scene, but it will also increase memory cost. For scenes that have high fidelity, increasing this parameter will be required to capture the high-frequency changes of the radiant exitance field. Additionally, large scenes will require high-resolution SVO, although local portions of the scene may not necessitate such a high resolution.

Like the other path-guiding approaches, we do not solely utilize WFPG as an importance-sampling method. “BXDF-WFPGMISRatio” defines the sampling ratio between BxDF and path guiding. This parameter is defined between 0 and 1 (inclusive). Similar to the other approaches, the value of 0.5 is used throughout the test scene. This means samples are equally generated between BxDF and WFPG.

As we discussed in Section 5.1, the radiance fields are filtered via a simple low-pass filter (a Gaussian Filter). Parameter “FieldFilterAlpha” is the value α in Equation 5.1.

$$y = e^{-\frac{x^2}{\alpha^2}} \tag{5.1}$$

Higher alpha will result in high blurring, which may result in high-frequency feature loss. On the other hand, lower alpha values would make the generated radiance field inefficient for rays that are far from the radiance field capture origin. In our experiments, we utilized the value of 0.8, which is strongly tied to the resolution of the generated radiance fields.

The “FieldResolution” parameter defines the width and height of the radiance field. Since we utilize the equi-area projection method, both width and height are required to be equal; thus, a single parameter would suffice to express the resolution. Higher radiance field resolution will enable the capture of high-frequency features; however,

it will increase computation time due to additional ray-casting operations. Radiance fields that are low-resolution may not capture important illumination sources, resulting in an increased variance. Additionally, this parameter is tied to the capabilities of the GPU because shared memory is utilized for storing radiance fields. For the GPU utilized (NVIDIA RTX 3070 Mobile) for tests in this Chapter, we would be able to get the maximum resolution of 128×128 .

We utilize a hierarchical selection of the radiance field due to high-resolution radiance fields taking a substantial portion of the method’s computation time. The first bounce will utilize maximum resolution because these rays contribute the highest to the resulting image. Subsequent bounces will reduce the field resolution by two. There is a lower limit of 16×16 that will be utilized by the fourth bounce and onwards.

5.3 Path Guiding Visualization Tool

In order to determine the correctness of the generated radiance fields and ease the development process, we created a path-guiding visualization tool. The implemented renderer can generate “reference” radiance fields, which can be used to compare the sampling fields of different methods. Reference radiance field generation occurs in a per-pixel fashion; the projected area of a camera pixel undergoes traditional path tracing. However, samples are not accumulated over the pixel but accumulated over an omnidirectional 2D radiance field image. Figure 5.2 compares our method and the method proposed by Müller et al. [64].

Our approach enables a uniform high-resolution radiance field due to lower memory constraints than Müller et al.’s approach. Müller et al.’s approach subdivide the radiance so that each leaf of the quadtree has a similar radiance value; thus, the normalized view in Figure 5.2 shows similarly colored pixels. However, the subdivision trend matches the trend of the reference image in an unnormalized form. Even still, the resolution of the calculated radiance field is comparably low.

We compare the “learning” schemes of our method with the method of Müller et al. as well. Such demonstration can be seen in Figure 5.3, which exposes the advantage of our process: our method does not require subdivision schemes for adapting the

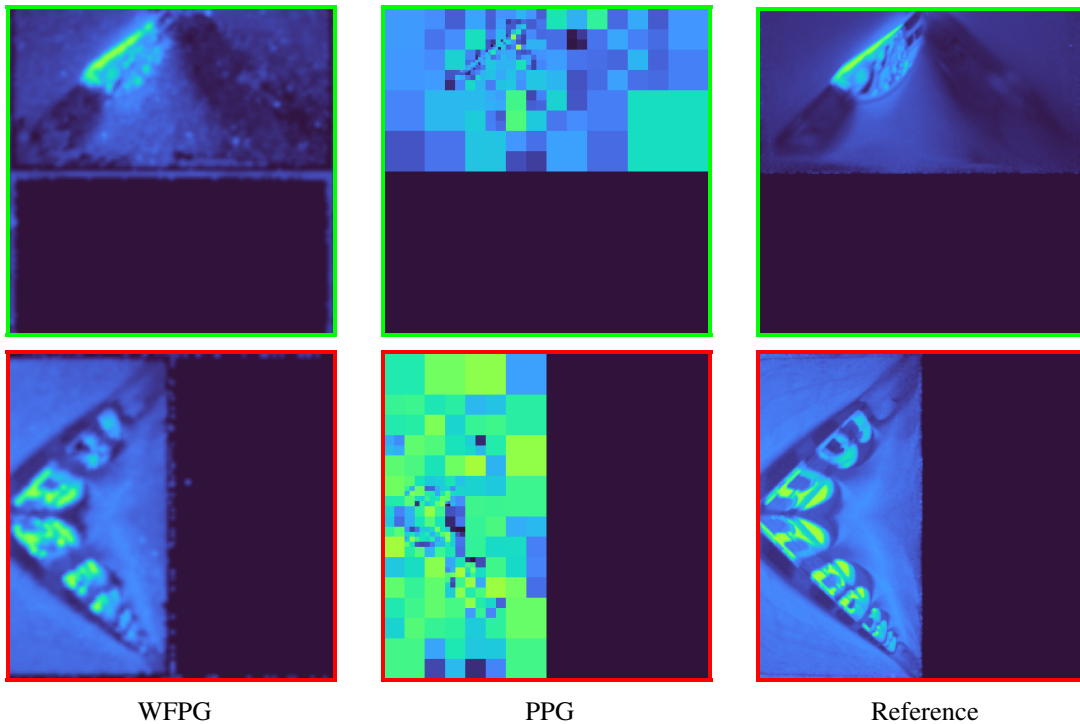


Figure 5.2: Learned or Generated Radiance field PDF of our method (abbreviated as WFPG) and practical path guiding method (abbreviated as PPG). Our method generates the radiance field 128^2 resolution. Both methods are “trained” with an equal number of samples (2048 samples per pixel). The reference radiance field is generated via path tracing and has a resolution of 256^2 (2^{16} samples per pixel).

data structure to the incoming radiance field. The directional data structure of our method is generated with a static discretization scheme and has a 64×64 resolution.

The adaptation scheme for our method represents more of a noise pattern of path tracing techniques. The first results immediately generate the radiance field’s structure,

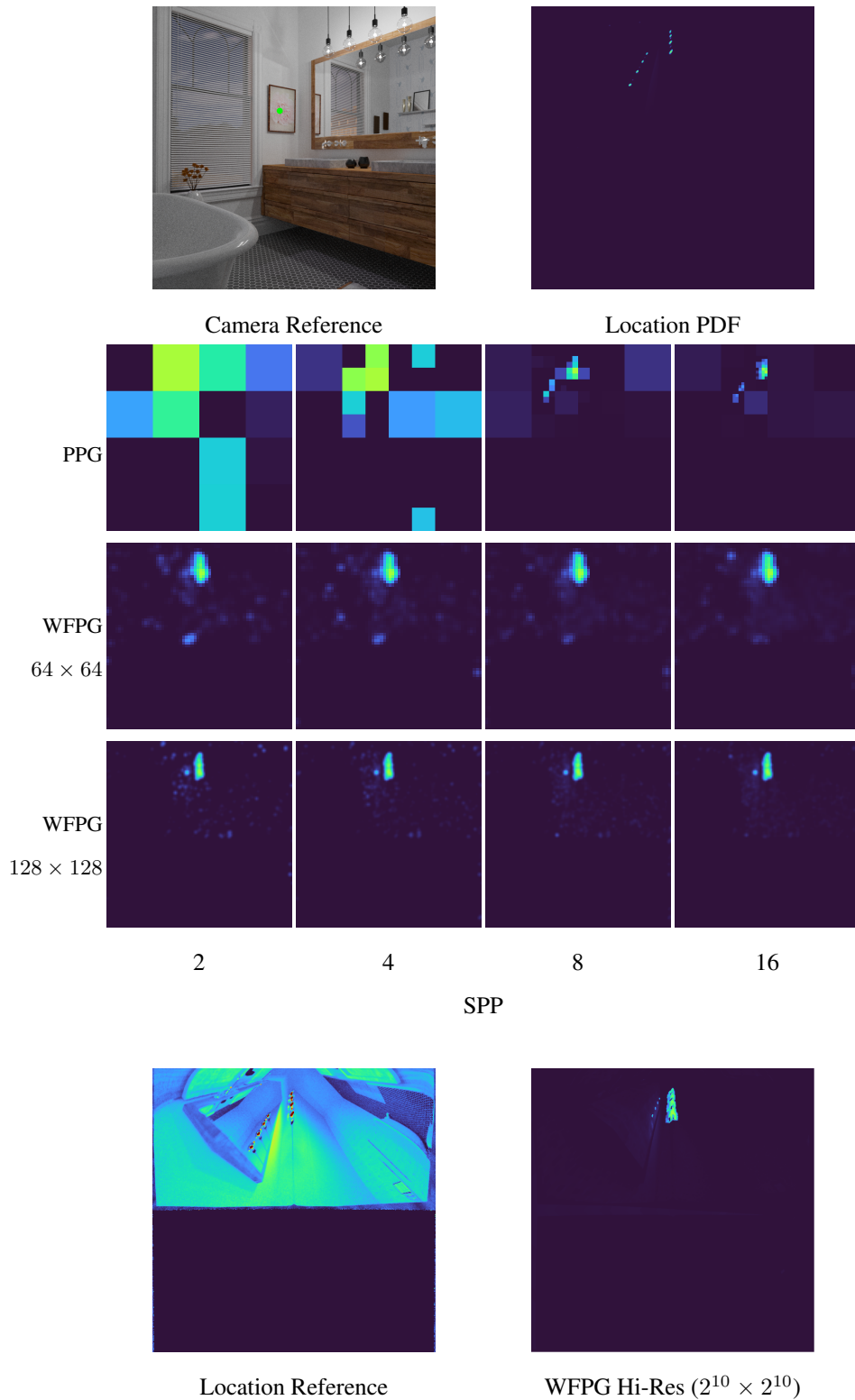


Figure 5.3: Convergence of Müller et al.’s method and our method. Reference is generated using path tracing over that region. For our method, 64×64 and 128×128 radiance fields are generated. Müller et al.’s method uses default parameters.

and additional samples tend to reduce the noise of the generated fields.

Our dense generation scheme can be considered a disadvantage for scenes with high-frequency radiance fields. Figure 5.3 also demonstrates the issue. Incoming radiance on the green dot has point light-like illumination. Reference PDF shows the reference radiance field. If our method generates the radiance field on low resolution, it generates a “smeared” radiance field, which is not exact. Still, even in this form, this radiance field is better than having no radiance field. This is especially true for this scene since dielectric surfaces encapsulate all emitters. Methods such as next event estimation (casting shadow rays) do not work since this kind of illumination is not a direct illumination.

Another comment on the resulting reference images is the resolution of the SVO. As can be seen on the bottom right of Figure 5.3, even with an excessive resolution for the radiance field (in this case 1024×1024 nearly an HD image), voxel representation of the scene emerges on to the generated field. Thus, the resolution of the SVO and the generated radiance field should be adjusted for high resolutions.

5.4 Profiling

To expose the method’s overhead, multiple measurements are conducted over different scenes. these can be seen in Table 5.1. Since our method generates radiance fields on the fly, one can argue that it can have impractical computation times. Although computation time increases, it is not practically low, for scenes with hard light interactions and high memory consumption would require this extra computation time instead of high memory usage.

Table 5.1 shows the extended timing table of the entire method compared to wavefront path tracing, denoted as “PT”. All of the measurements are done using an NVIDIA 3070ti Mobile GPU. Overall, our method doubles the computation cost of each sample. It should be noted that in the range of 32 to 90 million, additional rays are traced as cones to generate the radiance fields. Even with that extra cost, only $\sim 100\%$ increase in computation cost is understandable. However, our memory cost compared to other methods is nearly an order of magnitude lower, which can be desired for

Table 5.1: Timings of the wavefront path guiding stages. Each $Depth_n$ box has three values from top-down, which corresponds to “total bin count”, “average ray per bin”, and the total computation time of Algorithm 5, no product path guiding is conducted in these measurements. SVO Column values are as follows from top-down; “SVO Resolution”, “Node Count”, “Total Memory”, and “Construction Time”. The Miscellaneous portion includes partitioning routines with respect to both position and material and the material evaluation routines.

Scene (1920 × 1080)	PT	SVO	WFPG ($l_{min} = 5, c_{ray} = 512$)						
			$Depth_1$ (128 × 128)	$Depth_2$ (64 × 64)	$Depth_3$ (32 × 32)	$Depth_4$ (16 × 16)	Update Exitance	Misc.	Total
CRYSPONZA 	91.43ms	256 ³	2005.00	2712.83	2603.67	2444			
		638,763	1034.28	570.19	444.14	257.21			
		17.33 MiB	48.29ms	25.36ms	16.68ms	10.96ms	4.21ms	79.42ms	180.71ms
VEACHDOOR 	80.8ms	256 ³	2143.9	2651.28	2881.76	2684.25			
		199,362	967.39	703.44	612.12	598.50			
		5.36 MiB	35.76ms	21.23ms	16.84ms	17.25ms	4.07ms	74.87ms	165.95ms
CBOXOCCLUDE 	54.42ms	256 ³	5534.9	7172.8	7173.01	7172			
		638,395	207	152.49	140.87	125.63			
		17.07 MiB	67.13ms	26.36ms	14.91ms	10.09ms	6.68ms	50.38ms	177.87ms
BATHROOM 	61.19ms	256 ³	2448.92	3394.67	3163.44	3050.25			
		313,791	824.69	516.69	472.53	422.94			
		8.51 MiB	49.2ms	26.09ms	19.11ms	13.5ms	4.59ms	55.33ms	166.82ms
35.54ms									

scenes with high memory. This is the classical computation-cost vs. memory-usage tradeoff. Arguably, the memory is more precious on the GPU than CPU counterparts, especially for production-level scenes.

5.5 Baseline Comparison

For a baseline comparison, we profiled our method between classical path tracing. This comparison is used to make equal-time and equal-sample comparisons to check the method outperforms the classical path tracing. Equal-time comparisons are essential since the path tracer can outperform our method due to the sheer amount of extra rays generated due to computational efficiency.

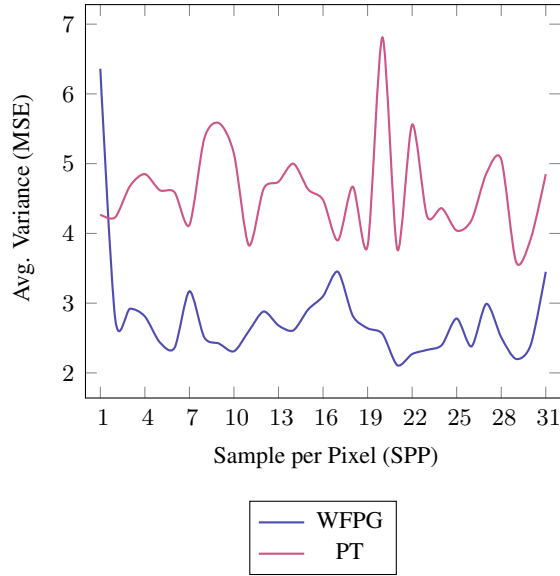
Figure 5.4 shows the single sample of the generated method over time. Since path-guiding methods train over time, the latter samples have higher sample quality than the previous ones. The first sample has an even higher variance than the path-tracing sample. This is expected since, initially, SVO is devoid of radiance information. Subsequent samples immediately outperform the path tracer.

Interestingly, newly generated samples after the first samples do not improve as dramatically as the very first samples. This can be tied to the generated radiance fields' convergence nature (Figure 5.3 WFPG row). By this observation, we experimented with multiple heuristics to combine the generated samples.

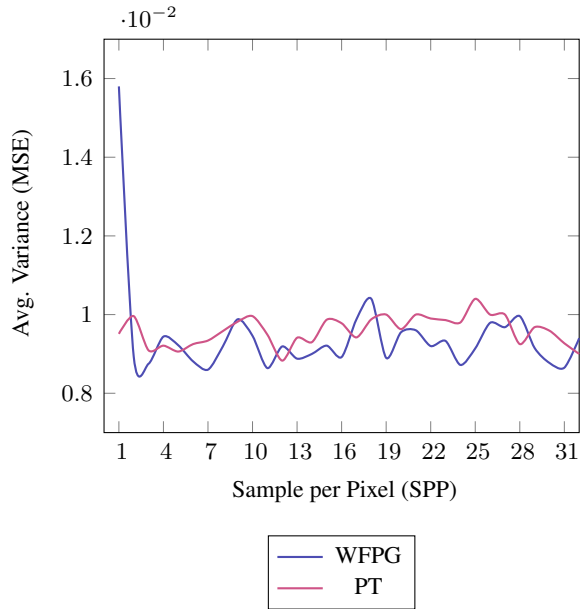
Given a set of n pre-filtered full-image samples and weights $S_n = \{(s_1, w_1), (s_2, w_2), \dots, (s_n, w_n)\}$, the resulting radiance-field of the generated image I_n can be computed with the given heuristics function $h(i)$ as follows.

$$I_n = \frac{\sum_{i=1}^n w_i s_i h(i)}{\sum_{i=1}^n w_i h(i)} \quad (5.2)$$

Several heuristic functions are shown below. If heuristic function $h(i)$ is constant, it corresponds to having no sample combination strategy being applied. The final heuristic, which did not have its heuristic function, is slightly different. It also uses a constant heuristic function; however, the first sample directly comes from the first path-tracing sample without applying the path-guiding method. In Figure 5.5, it is named ‘‘PT First’’.



(a) VeachDoor



(b) Sponza

Figure 5.4: Single sample variance of the proposed and traditional path-tracking methods. Each sample on the graph did not accumulate with the previous samples. This graph exposes the learning scheme of our method. The general trend of the radiance field is immediately learned in a couple of samples, especially after the very first sample.

$$h(i) = \begin{cases} i & i < 5 \\ 5 & \text{otherwise} \end{cases} \quad (\text{Linear})$$

$$h(i) = \begin{cases} i^2 & i < 5 \\ \frac{83}{25} & \text{otherwise} \end{cases} \quad (\text{Quadratic})$$

$$h(i) = \begin{cases} 1 & i = 1 \\ 2 & \text{otherwise} \end{cases} \quad (\text{One-Two})$$

Figure 5.5a graph exposes the mean FLIP of the sample combination strategies [112]. It can be observed that among the proposed strategies, the best combination strategy is “PT First”. However, except the “Quadratic” heuristic, all other heuristics outperform the non-weighted sampling strategy.

5.4, has two scenes namely “Sponza” and “VeachDoor”. The camera angle for the VeachDoor scene is the exact angle shown in Figure 5.6. For the Sponza scene, the camera angle is similar in Table 5.1, but it is shifted slightly towards the right.

Figure 5.5b shows the equal time comparison of the selected “PT First” heuristic and the path tracing. Although the proposed method is computationally more expensive, it outperforms path tracing in equal time.

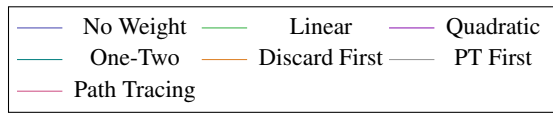
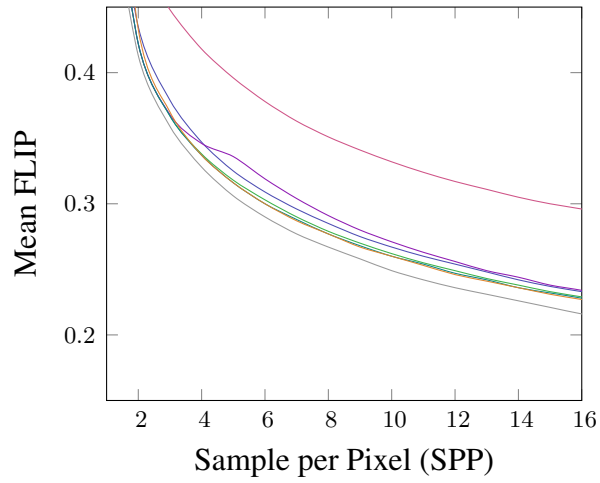
For Figure 5.5, we only demonstrated the VeachDoor scene. This scene’s camera angle is dominated by indirect illumination. Thus, we only provided the results of the Veach Door scene for the equal time comparisons. For the Sponza scene, camera-captured regions are a mix of directly and indirectly illumination-dominated regions. For that scene, results are similar but have a tighter gap between curves.

Similar to the approaches discussed in Müller et al. [64, 113], we did not choose to discard previous samples due to the interactive nature of the proposed method due to underlying hardware. This proposed heuristic does not discard samples; thus, it aligns pretty well with the interactive paradigm of the GPUs, which enables fast authoring of scenes with challenging lighting.

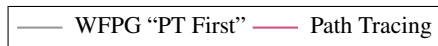
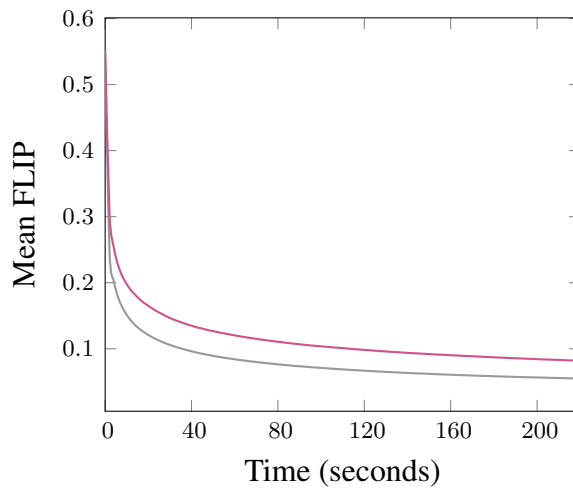
5.6 Comparison with Literature

We have conducted a per-sample comparison between Ruppert et al.’s method [70] as well as Müller et al.’s method [64]. Müller and Ruppert et al.’s methods have a public CPU implementation over the Mitsuba Renderer [114]. We refrain from conducting an equal-time comparison due to hardware differences. GPU implementations would outperform CPU implementations and would not make a fair comparison.

Results can be seen in Figure 5.6 and Table 5.2. In both cases, we utilize the HDR-Flip comparison tool [112, 115]. Comparisons between Ruppert et al. provide the



(a) VeachDoor - Methods



(b) VeachDoor - Time

Figure 5.5: Mean FLIP comparisons of the experimented sampling techniques. Additionally, equal-time comparisons are presented between the comparably best technique (“PT First”) and the classical path tracing technique. At the two-minute mark, the total number of combined samples is 928 for path tracing and 422 for WFGP.

FLIP heat map, which also shows the per-pixel difference values of the compared images. The user-adjustable parameters are on default values except for the training and sample count variables of both methods.

The entirety of the bathroom scene is dominated by indirect illumination. One reason is that the scene has specularly encapsulated light sources (i.e., lightbulbs). This essentially disables next-event estimation; thus, classical path tracing without path guiding would have difficulty generating images efficiently.

The Sponza scene has a good mix of directly and indirectly illuminated regions. Indirectly illuminated regions are cavities of the scene where illumination is obscured via pillars and arches. The main hall of the scene is also mostly illuminated via indirect illumination. However, this region has an easier time reaching the directly illuminated region.

In all scenes, the WFPG and Ruppert et al.’s methods achieve similar outcomes with equivalent sample counts. However, Ruppert et al.’s method involves an additional preprocessing phase that involves training samples to adaptively subdivide the Kd-tree and von Mises-Fisher Mixtures (vMM). After construction, these data structures are used for path guiding. This vMM mixture and Kd-tree training do not directly impact the final image quality; however, they consume a similar computational time.





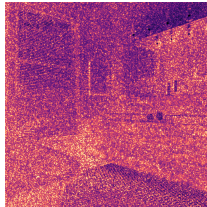
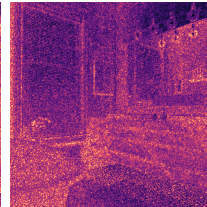
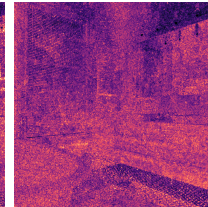







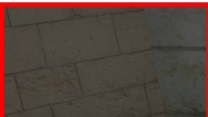







Considering this, we included extra samples for the WFPG and PT methods as compensation for that extra computational cost. This operation is also proper for the comparisons in Figure 5.6. This is false for the Sponza case since the total sample count is low. Giving additional training samples for that comparison would be unfair.

In summary, Muller and Ruppert et al.’s methods produce similar results with the exact sample count. The training requirement in Ruppert et al. and the discarding sample scheme of Muller et al. necessitates additional computation time without directly impacting the image quality.

Memory usage efficiency is another demonstration of Figure 5.6. Our method $\times 5$ improvement in the memory end due to the on-the-fly generation of radiance fields.

It should be noted that while computing Müller et al. and Ruppert et al.’s FLIP val-

Table 5.2: Comparisons between Ruppert et al.’s proposal and our method over two scenes. The FLIP method is conducted for comparisons. Sample per pixel for each method and training sample count is given on the second row. For the method of Ruppert et al., default parameters are used. For our method, $l_{min} = 5$, $c_{ray} = 512$ and SVO resolution of 256^3 is used. The maximum ray depth of the bathroom scene is 10. For the Sponza scene, the maximum ray depth is 4. Reported mean FLIP values are for the entire image. For the Sponza scene, two zoom-in sub-images are provided over regions dominated by indirect illumination.

	Reference	PT	WFPG	Ruppert et al.
		1536spp		512t + 1024spp
BATHROOM				
	FLIP Map			
	FLIP Mean	0.504965	0.408846	0.423945
		32spp		16t + 32spp
CRYSPONZA				
	FLIP Map			
	Flip Mean	0.227257	0.196408	0.212571
				
				

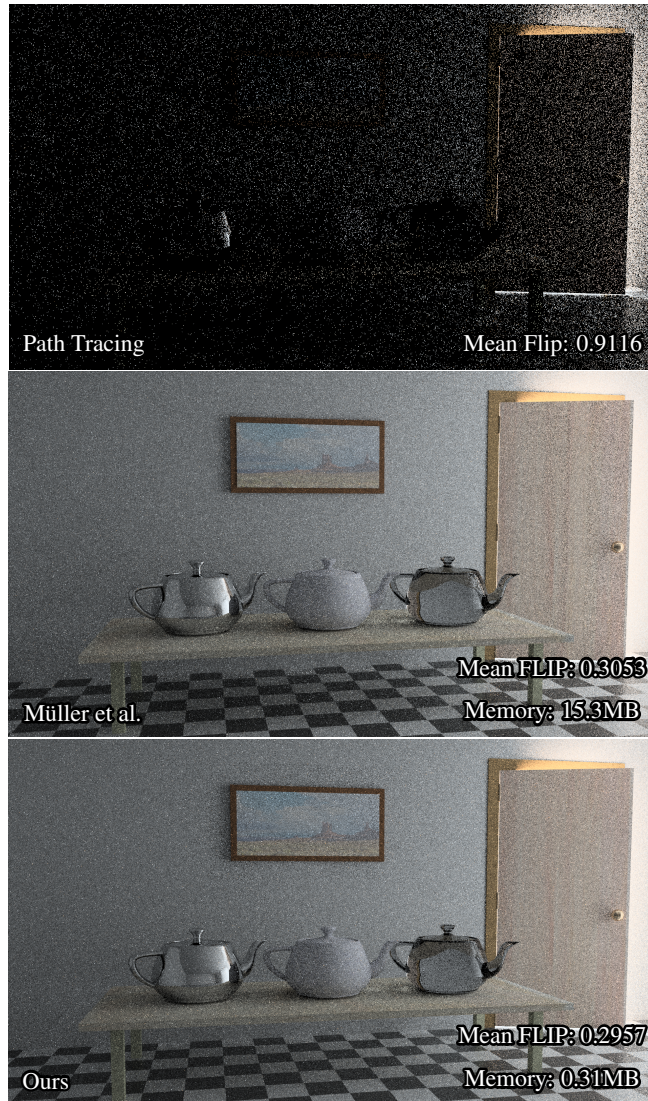


Figure 5.6: Comparison between classical path tracing and Practical Path Guiding method of Müller et al. [64]. and the proposed method. All methods are rendered without the next event estimation, and the sample per pixel count is 96. Practical path guiding method parameters are default, and it is without the extensions proposed on the course [113]. All other parameters are the default parameters. Our proposed method’s parameters are as follows. $l_{min} = 6$, $c_{ray} = 512$, SVO resolution of 128^3 , and using the proposed sample combination method. The proposed method has $5x$ less memory utilization with similar errors. Errors are measured using HDR-FLIP method [115, 112]. Image resolutions are 1920×1080 .

ues, reference images are generated with the Mitsuba Renderer’s Path Tracer. In contrast, our method’s reference image is from implementing a GPU Path Tracer. As the images suggest, there are minimal discrepancies between the renderers. These discrepancies can be attributed to implementation differences and variations in material evaluation, which have the potential to skew the results. Consequently, we employed

different reference images with distinct underlying architectures to ensure accurate comparisons and evaluations.

5.7 Product Path Guiding

In Table 5.1, timings are measured without the product path guiding method discussed in Chapter 4. With the same parameters and outer product field of 8×8 , it increases the computational times up to an additional 50% due to the per-ray calculation of the BxDF function. One potential future work could be combining the BxDFs of bins to do a union product sampling.

In our experiments, such low-resolution (8×8) product field mainly eliminates the sidedness problems that occur due to infinitely thin surfaces. When these thin surfaces have dramatic illumination differences between them, light leaks will occur. This light leak differs from the one discussed in Chapter 4; in that case, light leaks occur due to a resolution mismatch between the scene and the SVO. In this case, this mismatch is due to a volumetric partitioning scheme and the omnidirectional radiance field generation.

Figure 5.7 demonstrates the phenomena the product path guiding prevents. The wall with the painting has darkening when the product path guiding is off.

A higher product resolution can be chosen to capture the BxDF trend better for specular surfaces, but it will be computationally expensive. This can be a future work where regions will be marked with a specular metric, and higher-resolution product fields will be generated for these regions.

5.8 Limitations and Future Work

The proposed method has multiple limitations, some of which can be addressed by future works.

Densely generated radiance fields. One main limitation is the densely generated radiance field. Not every region in the scene may require high-resolution radiance

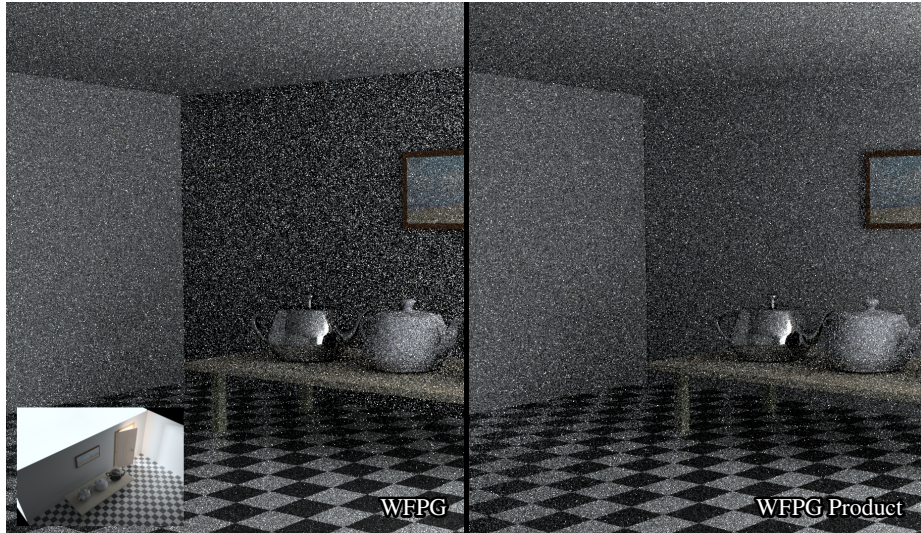


Figure 5.7: Demonstration of what product path guiding prevents. The image is purposely generated with a low sample count (16spp) to make the phenomena apparent. The picture on the left corner visually demonstrates the scene layout. Path guiding omnidirectionally captures the radiance field and wrongly launches rays towards the wall for the rays in the same room as the camera. The product with the reflectance prevents this issue.

fields. The opposite is also true; a higher resolution of a radiance field may be required to capture essential high-frequency features such as a reflection of point light.

Generation of radiance fields requires $\mathcal{O}(n^2)$ casting operations. Dynamizing this casting process can be a future work. One implementation can use the data structure proposed by Ditterbrant et al. named the “Compressed Directional Quadtree” (CQD) to hold a very low-resolution casting discretization scheme on each voxel [116].

Binned rays can query the SVO to find the nearby CQD and dynamically launch different-sized cones to estimate the radiance field efficiently. Such an approach can direct the required computational needs instead of equally dividing them uniformly over the region.

Radiant exitance and highly specular objects. Another limitation comes from the radiant exitance usage. Radiant exitance is inherently a directionless definition; thus, it saves memory on the SVO. However, highly specular objects reflect in a high-frequency fashion, meaning outgoing radiance changes drastically when the viewing angle changes.

However, determining regions that have specular objects is not a scene-dependent process. Thus, it can be obtained by looking at the scene definition. While generating the SVO, voxelized triangles can deposit this information onto the SVO during voxelization. A potential future work can utilize this process and allocate a small outgoing radiance field (again, maybe utilizing CDQ) for the required regions.

However, such a method should be used with caution if most regions require such a radiance field; approaches that cache the incoming radiance field, such as Muller et al.'s approach, would give better results. This is because one can hold incoming radiance instead of outgoing radiance and eliminate cone tracing. Thus, the incoming radiance field can be sampled directly instead.

Large scenes and Sparse Voxel Octree. This is a classical problem for scene accompanying data structures. For large scenes, the required data structure can be massive, meaning some form of the out-of-core process is required to be employed due to memory limitations. Adaptive subdivision schemes employed by other path-guiding methods can be utilized, but as demonstrated, these schemes would mean longer “training” times. A pre-generated, on-demand loadable SVO can be proposed as a future work.

Volumetric subdivision of the scene. We utilize volumetric subdivision methods for region binning and radiant exitance caching schemes. However, most natural-looking scenes have mostly reflective materials, and a surface base subdivision scheme could be better for most scenes.

Volumetric subdivision creates problems when the sampled surface is reflective and infinitely thin. Light does not get through these objects, but light leaks would occur due to volumetric subdivision schemes. Product path guiding can somewhat eliminate this issue, but a better approach would be to use a low-resolution surface representation of the scene directly.

This would mean a triangle tessellation for large triangles and triangle decimation for small triangles. However, such a regularized surface representation may not be trivial for most scenes.

Determine the regions that would require path guiding. Since we generate radi-

ance fields on the fly, determining regions that require path guiding would dramatically improve the computation time. Regions dominated by direct illumination would not require path guiding if a state-of-the-art next-event estimator is utilized. Perfectly specular regions (i.e., perfect mirror) would not require path guiding as well because incoming radiance does not contribute to the sampling scheme as much as BxDF. As a future work, one can define a heuristic to determine these regions and store it on the SVO. Binned regions then disregard the path-guiding process by looking to this value.

This approach may not always be helpful, such as the scene *VeachDoor* that is discussed in Section 5.6, but for other scenes, it would be highly beneficial. Also, finding a profound potential heuristic that captures all phenomena relating to path-guiding skipping is not trivial.

CHAPTER 6

CONCLUSIONS

Light transport simulations are hard. Many methods have many drawbacks and advantages, but not all methods are efficient for all scenes. Path tracing is a traditional method that captures every phenomenon but can be computationally expensive for hard light interactions. Path-guiding methods try to reduce the convergence time of the generated images of the path tracer. Those methods require estimating the 5D global radiance field as well as generating the actual image that is being rendered.

Such an ambitious extent means additional complex data management, either learning this global field while rendering or pre-generation. Especially for GPU-oriented renderers such dynamic memory management may not be easy to attach to a path tracer. Moreover, memory is arguably more precious on the GPU due to comparably low memory availability than CPUs.

In our case, we proposed a ground-up GPU-oriented path-guiding method that alleviates the memory requirement by converting such requirement into computational cost. The proposed path-guiding method is designed for wavefront-style path tracers, which are predominantly used in GPUs. Wavefront methods already require additional memory due to the design principles of the GPUs, and high memory using accompanying methods such as state-of-the-art path guiding methods may not be suitable for GPUs.

To this end, we proposed a wavefront path-guiding method that generates a local radiance field on the fly and alleviates the high-memory requirement. This radiance field generation utilizes the recent accelerated hardware ray tracing capabilities of modern GPUs. With this utilization, on-the-fly generation is practical in terms of

computation time.

6.1 GPU Limitations

Aside from the limitations discussed in Chapter 4, we will present the potential limitations of the GPU-oriented light transport simulations in this section.

The main limitation of the GPU renderers is due to the GPU itself. GPUs are highly parallel devices and development for these devices is tedious, unlike CPUs. CPUs are general-purpose machines; even a suboptimal code runs decently on a CPU. However, on GPU, the computational gap between suboptimal and optimized code can be tremendous. This practical concern would make development on the GPU comparably harder than a CPU-oriented design.

In addition, memory management on GPUs is not as flexible as CPU counterparts, and complex memory systems, such as trees and hash tables are not as performant as regular data structures such as arrays. GPUs require regular data structures, but some algorithms' asymptotic complexity would be tied to their usage of these complex data structures. Thus, it creates a dilemma. Although rarely, an asymptotically less efficient algorithm for a problem may perform better due to memory access regularity and simplicity.

However, when a GPU is fully utilized with an algorithm that is parallelizable, it is hard to beat the computational efficiency of the GPUs.

6.2 Final Words

This thesis demonstrated how the path tracing method can be applied to GPUs. The proposed method has an efficient scheme and static memory requirements, unlike queue-based methods. Additionally, we propose a GPU-oriented accompanying path-guiding method that leverages the hardware ray-tracing capabilities of the recent GPUs.

Although the proposed path-guiding method requires minimal memory compared to

other CPU-based path-guiding methods, it has higher computational time requirements. Such computational discrepancy, however, is parallelizable and can be amortized with a high amount of rays. Such an approach is suitable for the GPUs.

REFERENCES

- [1] J. T. Kajiya, “The rendering equation,” in *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH ’86, (New York, NY, USA), p. 143–150, Association for Computing Machinery, 1986.
- [2] M. Pharr, W. Jakob, and G. Humphreys, *Physically Based Rendering: From Theory to Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 3rd ed., 2016.
- [3] B.-S. Hua, A. Gruson, V. Petitjean, M. Zwicker, D. Nowrouzezahrai, E. Eisemann, and T. Hachisuka, “A survey on gradient-domain rendering,” *Computer Graphics Forum*, vol. 38, no. 2, pp. 455–472, 2019.
- [4] E. Veach and L. J. Guibas, “Optimally combining sampling techniques for monte carlo rendering,” in *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH ’95, (New York, NY, USA), p. 419–428, Association for Computing Machinery, 1995.
- [5] E. Veach, *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, Stanford University, Stanford, CA, USA, 1998. AAI9837162.
- [6] K. Ivo, P. Vévoda, P. Grittmann, T. Skřivan, P. Slusallek, and J. Křivánek, “Optimal multiple importance sampling,” *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2019)*, vol. 38, pp. 37:1–37:14, July 2019.
- [7] P. Shirley and C. Wang, “Direct lighting calculation by monte carlo integration,” in *Photorealistic Rendering in Computer Graphics* (P. Brunet and F. W. Jansen, eds.), (Berlin, Heidelberg), pp. 52–59, Springer Berlin Heidelberg, 1994.
- [8] J. J. Guo, M. Eisemann, and E. Eisemann, “Next event estimation++: Visibility

- mapping for efficient light transport simulation,” *Computer Graphics Forum*, vol. 39, no. 7, pp. 205–217, 2020.
- [9] J. Hanika, M. Droske, and L. Fascione, “Manifold next event estimation,” *Computer Graphics Forum*, vol. 34, p. 87–97, jul 2015.
- [10] G. Loubet, T. Zeltner, N. Holzschuch, and W. Jakob, “Slope-space integrals for specular next event estimation,” *ACM Transactions on Graphics*, vol. 39, nov 2020.
- [11] B. Walter, S. Zhao, N. Holzschuch, and K. Bala, “Single scattering in refractive media with triangle mesh boundaries,” *ACM Transactions on Graphics*, vol. 28, jul 2009.
- [12] C. Kulla and M. Fajardo, “Importance sampling techniques for path tracing in participating media,” *Computer Graphics Forum*, vol. 31, no. 4, pp. 1519–1528, 2012.
- [13] J. Hanika, A. Weidlich, and M. Droske, “Once-more scattered next event estimation for volume rendering,” *Computer Graphics Forum*, vol. 41, no. 4, pp. 17–28, 2022.
- [14] A. Conty Estevez and C. Kulla, “Importance sampling of many lights with adaptive tree splitting,” *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, vol. 1, aug 2018.
- [15] P. Moreau, M. Pharr, and P. Clarberg, “Dynamic many-light sampling for real-time ray tracing,” in *Proceedings of the Conference on High-Performance Graphics, HPG ’19*, (Goslar, DEU), p. 21–26, Eurographics Association, 2022.
- [16] B. Bitterli, C. Wyman, M. Pharr, P. Shirley, A. Lefohn, and W. Jarosz, “Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting,” *ACM Transactions on Graphics*, vol. 39, aug 2020.
- [17] J. Arvo and D. Kirk, “Particle transport and image synthesis,” *SIGGRAPH Computer Graphics*, vol. 24, p. 63–66, Sept. 1990.
- [18] J. Vorba and J. Křivánek, “Adjoint-driven russian roulette and splitting in light transport simulation,” *ACM Transactions on Graphics*, vol. 35, July 2016.

- [19] E. Veach and L. Guibas, “Bidirectional estimators for light transport,” in *Photorealistic Rendering Techniques* (G. Sakas, S. Müller, and P. Shirley, eds.), pp. 145–167, 1995.
- [20] E. Lafortune and Y. Willems, “Bi-directional path tracing,” *Proceedings of Third International Conference on Computational Graphics and Visualization Techniques (Compugraphics)*, vol. 93, 01 1998.
- [21] S. Popov, R. Ramamoorthi, F. Durand, and G. Drettakis, “Probabilistic connections for bidirectional path tracing,” *Computer Graphics Forum*, vol. 34, p. 75–86, July 2015.
- [22] E. P. Lafortune and Y. D. Willems, “Rendering participating media with bidirectional path tracing,” in *Rendering Techniques ’96* (X. Pueyo and P. Schröder, eds.), (Vienna), pp. 91–100, Springer Vienna, 1996.
- [23] E. Veach and L. J. Guibas, “Metropolis light transport,” in *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH ’97*, (USA), p. 65–76, ACM Press/Addison-Wesley Publishing Co., 1997.
- [24] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, “Equation of state calculations by fast computing machines,” *The Journal of Chemical Physics*, vol. 21, no. 6, pp. 1087–1092, 1953.
- [25] W. K. Hastings, “Monte carlo sampling methods using markov chains and their applications,” *Biometrika*, vol. 57, no. 1, pp. 97–109, 1970.
- [26] C. Kelemen, L. Szirmay-Kalos, G. Antal, and F. Csonka, “A simple and robust mutation strategy for the metropolis light transport algorithm,” *Computer Graphics Forum*, vol. 21, no. 3, pp. 531–540, 2002.
- [27] T. Hachisuka, A. S. Kaplanyan, and C. Dachsbacher, “Multiplexed metropolis light transport,” *ACM Transactions on Graphics*, vol. 33, July 2014.
- [28] W. Jakob and S. Marschner, “Manifold exploration: A markov chain monte carlo technique for rendering scenes with difficult specular transport,” *ACM Transactions on Graphics*, vol. 31, no. 4, pp. 58:1–58:13, 2012.

- [29] A. S. Kaplanyan, J. Hanika, and C. Dachsbacher, “The natural-constraint representation of the path space for efficient light transport simulation,” *ACM Transactions on Graphics*, vol. 33, July 2014.
- [30] J. Hanika, A. Kaplanyan, and C. Dachsbacher, “Improved half vector space light transport,” *Computer Graphics Forum*, vol. 34, p. 65–74, July 2015.
- [31] J. Lehtinen, T. Karras, S. Laine, M. Aittala, F. Durand, and T. Aila, “Gradient-domain metropolis light transport,” *ACM Transactions on Graphics*, vol. 32, July 2013.
- [32] M. Manzi, F. Rousselle, M. Kettunen, J. Lehtinen, and M. Zwicker, “Improved sampling for gradient-domain metropolis light transport,” *ACM Transactions on Graphics*, vol. 33, Nov. 2014.
- [33] J. Van de Woestijne, R. Frederickx, N. Billen, and P. Dutré, “Temporal coherence for metropolis light transport,” in *Proceedings of the Eurographics Symposium on Rendering: Experimental Ideas & Implementations*, EGSR ’17, (Goslar, DEU), p. 55–63, Eurographics Association, 2017.
- [34] H. W. Jensen, “Importance driven path tracing using the photon map,” in *Rendering Techniques ’95* (P. M. Hanrahan and W. Purgathofer, eds.), (Vienna), pp. 326–335, Springer Vienna, 1995.
- [35] Jensen, Henrik Wann, “Global illumination using photon maps,” in *Proceedings of the Eurographics Workshop on Rendering Techniques ’96*, (Berlin, Heidelberg), p. 21–30, Springer-Verlag, 1996.
- [36] T. Hachisuka, S. Ogaki, and H. W. Jensen, “Progressive photon mapping,” in *ACM SIGGRAPH Asia 2008 Papers*, SIGGRAPH Asia ’08, (New York, NY, USA), Association for Computing Machinery, 2008.
- [37] T. Hachisuka and H. W. Jensen, “Stochastic progressive photon mapping,” *ACM Transactions on Graphics*, vol. 28, p. 1–8, Dec. 2009.
- [38] I. Georgiev, J. Křivánek, T. Davidovič, and P. Slusallek, “Light transport simulation with vertex connection and merging,” *ACM Transactions on Graphics*, vol. 31, Nov. 2012.

- [39] T. Hachisuka, J. Pantaleoni, and H. Jensen, “A path space extension for robust light transport simulation,” *ACM Transactions on Graphics*, vol. 31, p. 1, 11 2012.
- [40] M. McGuire and D. Luebke, “Hardware-accelerated global illumination by image space photon mapping,” in *Proceedings of the Conference on High Performance Graphics 2009*, HPG ’09, (New York, NY, USA), pp. 77–89, ACM, 2009.
- [41] 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 *Proceedings of the 21st Eurographics Conference on Rendering*, EGSR’10, (Aire-la-Ville, Switzerland, Switzerland), pp. 1315–1324, Eurographics Association, 2010.
- [42] T. Ritschel, T. Engelhardt, T. Grosch, H.-P. Seidel, J. Kautz, and C. Dachsbacher, “Micro-rendering for scalable, parallel final gathering,” *ACM Transactions on Graphics*, vol. 28, pp. 132:1–132:8, Dec. 2009.
- [43] T. Udeshi and C. D. Hansen, “Towards interactive photorealistic rendering of indoor scenes: A hybrid approach,” in *Proceedings of the 10th Eurographics Conference on Rendering*, EGWR’99, (Goslar, DEU), p. 63–76, Eurographics Association, 1999.
- [44] I. Wald, T. Kollig, C. Benthin, A. Keller, and P. Slusallek, “Interactive global illumination using fast ray tracing,” in *Proceedings of the 13th Eurographics Workshop on Rendering*, EGRW ’02, (Goslar, DEU), p. 15–24, Eurographics Association, 2002.
- [45] J. Novák, D. Nowrouzezahrai, C. Dachsbacher, and W. Jarosz, “Virtual ray lights for rendering scenes with participating media,” *ACM Transactions on Graphics*, vol. 31, July 2012.
- [46] C. Dachsbacher and M. Stamminger, “Reflective shadow maps,” in *Proceedings of the 2005 Symposium on Interactive3D Graphics and Games*, I3D ’05, (New York, NY, USA), pp. 203–231, ACM, 2005.
- [47] T. Ritschel, T. Grosch, H.-P. Kim, M. H. and Seidel, C. Dachsbacher, and J. Kautz, “Imperfect shadow maps for efficient computation of indirect illu-

- mination,” *ACM Transactions on Graphics*, vol. 27, pp. 129:1–129:8, Dec. 2008.
- [48] T. Ritschel, E. Eisemann, I. Ha, J. D. K. Kim, and H.-P. Seidel, “Making imperfect shadow maps view-adaptive: High-quality global illumination in large dynamic scenes,” *Computer Graphics Forum*, vol. 30, no. 8, pp. 2258–2269, 2011.
- [49] L. Williams, “Casting curved shadows on curved surfaces,” *SIGGRAPH Computer Graphics*, vol. 12, pp. 270–274, Aug. 1978.
- [50] G. J. Ward, F. M. Rubinstein, and R. D. Clear, “A ray tracing solution for diffuse interreflection,” in *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH ’88, (New York, NY, USA), p. 85–92, Association for Computing Machinery, 1988.
- [51] J. Krivanek, P. Gautron, S. Pattanaik, and K. Bouatouch, “Radiance caching for efficient global illumination computation,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 11, p. 550–561, Sept. 2005.
- [52] A. Conty Estevez and C. Kulla, “Importance sampling of many lights with adaptive tree splitting,” *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, vol. 1, Aug. 2018.
- [53] M. Hašan, F. Pellacini, and K. Bala, “Matrix row-column sampling for the many-light problem,” *ACM Transactions on Graphics*, vol. 26, July 2007.
- [54] C. Sun and E. Agu, “Many-lights real time global illumination using sparse voxel octree,” in *Advances in Visual Computing: 11th International Symposium, ISVC 2015* (G. Bebis, R. Boyle, B. Parvin, D. Koracin, I. Pavlidis, R. Feris, T. McGraw, M. Elendt, E. Kopper, Regis and Ragan, Z. Ye, and G. Weber, eds.), (Cham), pp. 150–159, Springer International Publishing, 2015.
- [55] B. Walter, S. Fernandez, A. Arbree, K. Bala, M. Donikian, and D. P. Greenberg, “Lightcuts: A scalable approach to illumination,” *ACM Transactions on Graphics*, vol. 24, p. 1098–1107, July 2005.
- [56] C. Yuksel, “Stochastic lightcuts,” in *High-Performance Graphics (HPG 2019)*, The Eurographics Association, 2019.

- [57] P. Moreau and P. Clarberg, “Importance sampling of many lights on the gpu,” in *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs* (E. Haines and T. Akenine-Möller, eds.), pp. 255–283, Berkeley, CA: Apress, 2019.
- [58] T. Davidovič, I. Georgiev, and P. Slusallek, “Progressive lightcuts for GPU,” in *ACM SIGGRAPH 2012 Talks*, SIGGRAPH ’12, (New York, NY, USA), Association for Computing Machinery, 2012.
- [59] C. Dachsbacher, J. Křivánek, M. Hašan, A. Arbree, B. Walter, and J. Novák, “Scalable realistic rendering with many-light methods,” *Computer Graphics Forum*, vol. 33, no. 1, pp. 88–104, 2014.
- [60] J. Vorba, O. Karlík, M. Šik, T. Ritschel, and J. Křivánek, “On-line learning of parametric mixture models for light transport simulation,” *ACM Transactions on Graphics*, vol. 33, July 2014.
- [61] T. Müller, B. McWilliams, F. Rousselle, M. Gross, and J. Novák, “Neural importance sampling,” *ACM Transactions on Graphics*, vol. 38, Oct. 2019.
- [62] Y. Huo, R. Wang, R. Zheng, H. Xu, H. Bao, and S.-E. Yoon, “Adaptive incident radiance field sampling and reconstruction using deep reinforcement learning,” *ACM Transactions on Graphics*, vol. 39, Jan. 2020.
- [63] S. Bako, M. Meyer, T. DeRose, and P. Sen, “Offline deep importance sampling for monte carlo path tracing,” *Computer Graphics Forum*, 2019.
- [64] T. Müller, M. Gross, and J. Novák, “Practical path guiding for efficient light-transport simulation,” *Computer Graphics Forum*, vol. 36, p. 91–100, July 2017.
- [65] P. Vévoda, I. Kondapaneni, and J. Křivánek, “Bayesian online regression for adaptive direct illumination sampling,” *ACM Transactions on Graphics*, vol. 37, July 2018.
- [66] E. P. Lafortune and Y. D. Willems, “A 5D tree to reduce the variance of monte carlo ray tracing,” in *Rendering Techniques ’95 (Proceedings of the 6th Eurographics Workshop on Rendering)*, pp. 11–20, 1995.

- [67] H. W. Jensen, “Importance driven path tracing using the photon map,” in *Rendering Techniques*, 1995.
- [68] S. Herholz, O. Elek, J. Vorba, H. Lensch, and J. Křivánek, “Product importance sampling for light transport path guiding,” *Computer Graphics Forum*, vol. 35, p. 67–77, July 2016.
- [69] A. Dodik, M. Papas, C. Öztireli, and T. Müller, “Path guiding using spatio-directional mixture models,” *Computer Graphics Forum*, vol. 41, no. 1, pp. 172–189, 2022.
- [70] L. Ruppert, S. Herholz, and H. P. A. Lensch, “Robust fitting of parallax-aware mixtures for path guiding,” *ACM Transactions on Graphics*, vol. 39, aug 2020.
- [71] K. Dahm and A. Keller, “Learning light transport the reinforced way,” in *ACM SIGGRAPH 2017 Talks*, SIGGRAPH ’17, (New York, NY, USA), Association for Computing Machinery, 2017.
- [72] J. Guo, P. Bauszat, J. Bikker, and E. Eisemann, “Primary sample space path guiding,” in *Eurographics Symposium on Rendering - EI & I* (W. Jakob and T. Hachisuka, eds.), pp. 73–82, Eurographics, The Eurographics Association, July 2018. doi: 10.2312/sre.20181174.
- [73] A. Kaplanyan and C. Dachsbacher, “Path space regularization for holistic and robust light transport,” *Computer Graphics Forum*, vol. 32, 05 2013.
- [74] 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,” *ACM Transactions on Graphics*, vol. 29, jul 2010.
- [75] M. Pharr, W. Jakob, and G. Humphreys, *Physically Based Rendering: From Theory to Implementation*. Cambridge, MA, USA: The MIT Press, 4th ed., 2023.
- [76] M. Nimier-David, D. Vicini, T. Zeltner, and W. Jakob, “Mitsuba 2: A retargetable forward and inverse renderer,” *ACM Transactions on Graphics*, vol. 38, nov 2019.

- [77] S. Zheng, Z. Zhou, X. Chen, D. Yan, C. Zhang, Y. Geng, Y. Gu, and K. Xu, “Luisarender: A high-performance rendering framework with layered and unified interfaces on stream architectures,” *ACM Transactions on Graphics*, vol. 41, nov 2022.
- [78] I. Buck, T. Foley, D. Horn, J. SUGERMAN, K. Fatahalian, M. Houston, and P. Hanrahan, “Brook for GPUs: Stream computing on graphics hardware,” *ACM Transactions on Graphics*, vol. 23, p. 777–786, aug 2004.
- [79] J. Krüger and R. Westermann, “Linear algebra operators for gpu implementation of numerical algorithms,” *ACM Transactions on Graphics*, vol. 22, p. 908–916, jul 2003.
- [80] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan, “Ray tracing on programmable graphics hardware,” *ACM Transactions on Graphics*, vol. 21, p. 703–712, jul 2002.
- [81] J. Nickolls, I. Buck, M. Garland, and K. Skadron, “Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for?,” *Queue*, vol. 6, p. 40–53, mar 2008.
- [82] S. Laine and T. Karras, “Efficient sparse voxel octrees – analysis, extensions, and implementation,” NVIDIA Technical Report NVR-2010-001, NVIDIA Corporation, Feb. 2010.
- [83] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann, “Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering,” in *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games, I3D ’09*, (New York, NY, USA), p. 15–22, Association for Computing Machinery, 2009.
- [84] K. Museth, “VDB: High-resolution sparse volumes with dynamic topology,” *ACM Transactions on Graphics*, vol. 32, jul 2013.
- [85] K. Museth, “NanoVDB: A gpu-friendly and portable vdb data structure for real-time rendering and simulation,” in *ACM SIGGRAPH 2021 Talks*, SIGGRAPH ’21, (New York, NY, USA), Association for Computing Machinery, 2021.

- [86] M. Labschütz, S. Bruckner, M. E. Gröller, M. Hadwiger, and P. Rautek, “Jit-tree: A just-in-time compiled sparse gpu volume data structure,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 22, no. 1, pp. 1025–1034, 2016.
- [87] J. Beyer, M. Hadwiger, and H. Pfister, “State-of-the-art in gpu-based large-scale volume visualization,” *Computer Graphics Forum*, vol. 34, p. 13–37, dec 2015.
- [88] C. Crassin and S. Green, *Octree-Based Sparse Voxelization Using The GPU Hardware Rasterizer*, ch. 22. CRC Press, Patrick Cozzi and Christophe Riccio, 2012.
- [89] J. Pantaleoni, “Voxelpipe: A programmable pipeline for 3d voxelization,” in *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG ’11, (New York, NY, USA), p. 99–106, Association for Computing Machinery, 2011.
- [90] G. Young and A. Krishnamurthy, “Gpu-accelerated generation and rendering of multi-level voxel representations of solid models,” *Computers & Graphics*, vol. 75, pp. 11–24, 2018.
- [91] K. Sung, “A DDA Octree Traversal Algorithm for Ray Tracing,” in *EG 1991-Technical Papers*, Eurographics Association, 1991.
- [92] K. Museth, “Hierarchical digital differential analyzer for efficient ray-marching in opendb,” in *ACM SIGGRAPH 2014 Talks*, SIGGRAPH ’14, (New York, NY, USA), Association for Computing Machinery, 2014.
- [93] M. Hadwiger, A. K. Al-Awami, J. Beyer, M. Agus, and H. Pfister, “Sparseleap: Efficient empty space skipping for large-scale volume rendering,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 24, no. 1, pp. 974–983, 2018.
- [94] C. Crassin, F. Neyret, M. Sainz, S. Green, and E. Eisemann, “Interactive indirect illumination using voxel cone tracing: A preview,” in *Symposium on Interactive 3D Graphics and Games*, I3D ’11, (New York, NY, USA), p. 207, Association for Computing Machinery, 2011.

- [95] Intel Corporation, “Introduction to the Xe-HPG Architecture,” tech. rep., Intel Corporation, 2022.
- [96] Nvidia Corporation, “Nvidia Turing GPU Architecture,” tech. rep., Nvidia Corporation, 2018.
- [97] AMD Corporation, *RDNA3 Instruction Set Architecture, Reference Guide*. AMD Corporation, 2023.
- [98] “DirectX Raytracing (DXR) Functional Spec.” <https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html>. Accessed: 2023-08-05.
- [99] M. Stich, H. Friedrich, and A. Dietrich, “Spatial splits in bounding volume hierarchies,” in *Proceedings of the Conference on High Performance Graphics 2009*, HPG ’09, (New York, NY, USA), p. 7–13, Association for Computing Machinery, 2009.
- [100] I. Wald, “On fast construction of SAH-based bounding volume hierarchies,” in *2007 IEEE Symposium on Interactive Ray Tracing*, pp. 33–40, 2007.
- [101] T. Karras and T. Aila, “Fast parallel construction of high-quality bounding volume hierarchies,” in *Proceedings of the 5th High-Performance Graphics Conference*, HPG ’13, (New York, NY, USA), p. 89–99, Association for Computing Machinery, 2013.
- [102] E. Vasiou, K. Shkurko, E. Brunvand, and C. Yuksel, “Mach-RT: A many chip architecture for high-performance ray tracing,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 28, no. 3, pp. 1585–1596, 2020.
- [103] S. Laine, T. Karras, and T. Aila, “Megakernels considered harmful: Wavefront path tracing on GPUs,” in *Proceedings of the 5th High-Performance Graphics Conference*, HPG ’13, (New York, NY, USA), p. 137–143, Association for Computing Machinery, 2013.
- [104] M. Harris *et al.*, “Optimizing parallel reduction in CUDA,” *Nvidia developer technology*, vol. 2, no. 4, p. 70, 2007.

- [105] D. P. Singh, I. Joshi, and J. Choudhary, “Survey of gpu based sorting algorithms,” *Int. J. Parallel Program.*, vol. 46, p. 1017–1034, dec 2018.
- [106] “NVIDIA CUB Documentation.” <https://nvlabs.github.io/cub/index.html>. Accessed: 2023-08-06.
- [107] P. Shirley, S. Laine, D. Hart, M. Pharr, P. Clarberg, E. Haines, M. Raab, and D. Cline, *Sampling Transformations Zoo*, ch. Sampling, pp. 223–246. Berkeley, CA: Apress, 2019.
- [108] A. Conty Estevez and P. Lécocq, “Fast product importance sampling of environment maps,” in *ACM SIGGRAPH 2018 Talks*, SIGGRAPH ’18, (New York, NY, USA), Association for Computing Machinery, 2018.
- [109] S. G. Parker, H. Friedrich, D. Luebke, K. Morley, J. Bigler, J. Hoberock, D. McAllister, A. Robison, A. Dietrich, G. Humphreys, M. McGuire, and M. Stich, “Gpu ray tracing,” *Commun. ACM*, vol. 56, p. 93–101, May 2013.
- [110] B. Yalçiner, “Mray: Gpu based research renderer,” 2023. <https://github.com/yalcinerbora/meturay>.
- [111] P. Clarberg, “Fast Equal-Area Mapping of the (Hemi)Sphere using SIMD,” *Journal of Graphics Tools*, vol. 13, no. 3, pp. 53–68, 2008.
- [112] P. Andersson, J. Nilsson, P. Shirley, and T. Akenine-Möller, “Visualizing Errors in Rendered High Dynamic Range Images,” in *Eurographics 2021 - Short Papers* (H. Theisel and M. Wimmer, eds.), The Eurographics Association, 2021.
- [113] J. Vorba, J. Hanika, S. Herholz, T. Müller, J. Křivánek, and A. Keller, “Path guiding in production,” in *ACM SIGGRAPH 2019 Courses*, SIGGRAPH ’19, (New York, NY, USA), Association for Computing Machinery, 2019.
- [114] W. Jakob, “Mitsuba renderer,” 2010. Accessed 22 April 2023.
- [115] P. Andersson, J. Nilsson, T. Akenine-Möller, M. Oskarsson, K. Åström, and M. D. Fairchild, “FLIP: A difference evaluator for alternating images,” *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, vol. 3, aug 2020.

- [116] A. Dittebrandt, J. Hanika, and C. Dachsbacher, “Temporal Sample Reuse for Next Event Estimation and Path Guiding for Real-Time Path Tracing,” in *Eurographics Symposium on Rendering - DL-only Track* (C. Dachsbacher and M. Pharr, eds.), The Eurographics Association, 2020.

CURRICULUM VITAE

PERSONAL

NAME : Bora Yalçiner
PLACE : Ankara, Turkey
E-MAIL : yalcinerbora@outlook.com
WEBSITE : yalcinerbora.github.io

WORK EXPERIENCE

- | | |
|------------------------|--|
| DEC. 2020 | Research Assistant
METU Department of Computer Engineering, Turkey |
| FEB. 2020
FEB. 2018 | Software Developer
R&D Department of VERI-SIS Co., Turkey
<i>Development of NLSW Equations Solver for Marine-related Events</i>
Development of non-linear shallow water (NLSW) equations solver for marine events such as tsunamis, storm surges, and tropical cyclones. Solver has already been applied to marine hazard analysis of critical coastal plants such as nuclear power plants. |
| FEB. 2018
MAY 2015 | Researcher
METU Ocean Engineering Research Center, Turkey
<i>Development of NLSW Equations Solver and Real-Time Renderer</i>
Developed non-linear shallow water (NLSW) equations solver for academic and applied research. Solver enables analysis of tsunamis. Additionally, it provides real-time data visualization of generated outputs. |
| MAR. 2014
JUNE 2014 | Project Assistant
METU Ocean Engineering Research Center, Turkey
<i>Development of Wind Data Acquiring GUI Application</i>
Implemented wind data fetch GUI Application for EU FP7 Funded COCONET (Towards COast to COast <u>NET</u> works of marine protected areas: from the shore to the deep sea). The application provides the acquisition and visualization of wind data on Mediterranean waters. |

SUMMER 2012 | **Summer Intern**
Zinek Coding House, Turkey
Worked on linear algebra math library and accelerated it using x86 SSE instruction set. Also worked on their rendering engine in which an abstraction layer is designed to support OpenGL implementation and the currently available DirectX 9 renderer.

SUMMER 2011 | **Summer Intern**
LST Software, Turkey
Tested the website implementation of the Social Security Institution of Turkey. Additionally, I was involved in portlet development for the site.

EDUCATION

MAR. 2024 | Ph.D. in COMPUTER ENGINEERING
Graduate School of Natural and Applied Sciences
Middle East Technical University (METU), Turkey
Thesis: “Path Guiding Method for Wavefront Path Tracing: A Memory Efficient Approach for GPU Path Tracers”
Advisor: Ahmet Oğuz AKYÜZ

JULY 2016 | MSc. Degree in MULTIMEDIA INFORMATICS
Graduate School of Informatics
Middle East Technical University (METU), Turkey
Thesis: “Dynamic Voxelization to Aid Illumination of Real-Time Scenes”
Advisor: Yusuf SAHILLIOGLU

JULY 2013 | Bs Degree in COMPUTER SCIENCE
Faculty of Engineering
Bilkent University, Turkey

PUBLICATIONS

Journal publications are annotated with the “**J**” prefix.

J Yalciner B., Akyuz A. O. “Path Guiding for Wavefront Path Tracing: A Memory Efficient Approach for GPU Path Tracers”. Computers & Graphics. **Under Review.**

J Dogan G. G., Yalciner A. C., Annunziato A., **Yalciner B.**, Necmioglu O. “Global propagation of air pressure waves and consequent ocean waves due to the Jan-

uary 2022 Hunga Tonga-Hunga Ha'apai eruption". *Ocean Engineering* 2023; 267:113174.

J Dogan, G. G., Pelinovsky, E., Zaytsev, A., Metin, A. D., Ozyurt Tarakcioglu, G., Yalciner, A. C., **Yalciner B.**, Didenkulova, I. (2021). "Long Wave Generation and Coastal Amplification due to Propagating Atmospheric Pressure Disturbances". *Natural Hazards*, 106(2), 1195–1221. doi:10.1007/s11069-021-04625-9

Dogan, G. G., Probst, P., **Yalciner, B.**, Annunziato, A., Zahibo, N., Yalciner, A. C. (2020). "Numerical Modeling of Tropical Cyclone Generated Waves; Case studies of Irma, Maria and Dorian". In *EGU General Assembly Conference Abstracts* (pp. 11200).

J **Yalciner, B.**, Sahillioğlu, Y. "Voxel Transformation: Scalable Scene Geometry Discretization for Global Illumination". *J Real-Time Image Proc* 17; 1585–1596 (2020).

Yalciner, A. C., Suzen L. M., Tufekci Enginar D., Dogan G. G., Kolat C., Celikbas B., **Yalciner B.**, Cabuk O., Bas M., Kilic O., Yahya Mentese E., Tarih A., Zaytsev, A., Pelinovski, E. (2019). "Complete Tsunami Hazard Assessment, Vulnerability and Risk Analysis for the Marmara Coast of Istanbul Metropolitan Area". In *EGU General Assembly Conference Abstracts* (pp. 16743).

Zahibo, N., Krien Y., Arnau G., **Yalciner B.**, Zaytsev A., Yalciner A. C., ... Cabuk, O.(2019). "Storm Surge Analysis for French West Indies". 14th MEDCOAST Congress on Coastal and Marine Sciences, Engineering, Management and Conservation, MEDCOAST 2019 (pp.703-710). Marmaris, Turkey

Tufekci Enginar D., Suzen L. M., Yalciner A. C., Kolat C., Dogan G. G., **Yalciner, B.**, Zaytsev A. (2018). "Tsunami Human Vulnerability Assessment of Silivri District, Istanbul". In *EGU General Assembly Conference Abstracts* (pp. 1068).

J Zaytsev A., Beresnev P., Flatov V., Makarov V., Tyugin D., Zeziulin D., Pelinovsky E., Yalciner A. C., **Yalciner B.**, Oshmarina O., Kurkin A. (2017). "Coastal

Monitoring of the Okhotsk Seas Using an Autonomous Mobile Robot”. Science of Tsunami Hazards, 1–12.

Yalciner, B., Zaytsev A. Assessment of Efficiency and Performance in Tsunami Numerical Modeling with GPU. European Geosciences Union(EGU) General Assembly 2017; Tsunami (co-organized) NH5.1/OS4.13.

Yalciner, B., Zaytsev A., Yalciner A. C. Accelerated Solutions in Tsunami Simulation and Visualization with Case Studies. International Tsunami Symposium (ITS) 2017, Analytical; experimental and numerical methods and applications (003).

PROGRAMMING

Expert : C++, CUDA
Advanced : C++20, C, OpenGL 2/3/4
Intermediate : Autodesk® Maya, Unity®, Arnold®, Python, Git, L^AT_EX, HTML
Beginner : Javascript, Php, POSIX API, Java, C#, GNU Gimp, DirectX 10/11
Basic : Matlab, ASP.NET, SQL, Bash Script, Batch Script, Assembly (MIPS, x86, Intel8051)

INTERESTS AND ACTIVITIES

- CMAS Two Star Scuba Diver.
- Licensed amateur captain since February 2009.
- Long-term experience in yachting, navigation, and fishing
- An avid cook and culinary enthusiast