Path Guiding in Production and Recent Advancements

SEBASTIAN HERHOLZ, Intel Corporation, Germany
LEA REICHARDT, Walt Disney Animation Studios, Canada
MARCO MANZI, DisneyResearch|Studios, Switzerland
MARTIN ŠIK, Chaos, Czech Republic
BRIAN GREEN, Walt Disney Animation Studios, USA
YINING KARL LI, Walt Disney Animation Studios, USA

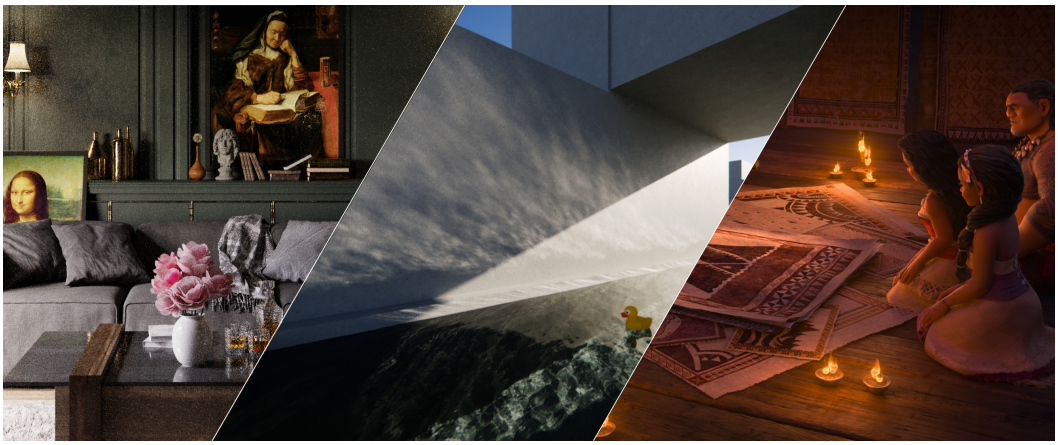


Fig. 1. Examples of different integrations of *path guiding* algorithms in production renderers, such as Blender's Cycles, Chaos' Corona, and Disney Animation's Hyperion. The images showcase different *guiding* specific use cases, such as complex multi-bounce indirect illumination and caustics. First image: scene by Jesús Sandoval. Third image: *Moana 2* ©2024 Disney.

Over the last decade, path guiding algorithms found their way from the scientific realm into production renderers. These algorithms enable the rendering of challenging lighting effects (e.g., complex indirect illumination, caustics, volumetric multi-scattering, and occluded direct illumination from multiple lights), which are crucial for generating high-fidelity images. The fact that these algorithms primarily focus on optimizing local importance sampling decisions makes it possible to integrate them into a path tracer, the de facto standard rendering algorithm used in production today. The theory behind these algorithms has been presented and discussed on various occasions (e.g., in presentations or research papers), and their practical applications in production have been explored in the previous course on: *Path Guiding in Production*.
Nevertheless, the implementation details or challenges associated with integrating them into a production render are usually unknown or not publicly discussed.
This course aims to provide a deeper understanding of how specific guiding algorithms are integrated into and utilized in various production renderers, including Blender's Cycles, Chaos's VRay and Corona, SideFX's Karma, and Disney Animation's Hyperion. The first chapter (Chap. 1) provides deeper insights into the nitty-gritty details that must be considered when integrating a path-guiding framework into a production renderer, details that are usually unknown. In the second chapter (Chap. 2), Disney shares their experiences and challenges when integrating path guiding into their wavefront production renderer Hyperion. The last chapter (Chap. 3) presents Chaos's recent advancements to their specialized guiding estimator for caustics.

Authors' Contact Information: Sebastian Herholz, sebastian.herholz@intel.com, Intel Corporation, Karlsruhe, Germany; Lea Reichardt, lea.reichardt@disneyanimation.com, Walt Disney Animation Studios, Vancouver, Canada; Marco Manzi, marco.manzi@disneyresearch.com, DisneyResearch|Studios, Zürich, Switzerland; Martin Šik, sik@corona-renderer.com, Chaos, Prague, Czech Republic; Brian Green, brian.green@disneyanimation.com, Walt Disney Animation Studios, Burbank, USA; Yining Karl Li, karl.li@disneyanimation.com, Walt Disney Animation Studios, Burbank, USA.

Contents

Frontmatter 1
Abstract 1
Contents 2

Ch. 1: Integrating Path Guiding into a Production Render: The Nitty Gritty Details 4
1 Introduction 5
2 Local Path Guiding 6
2.1 OpenPGL: An Open-Source Production-Ready local Path Guiding Framework 7
3 Guided Directional Sampling 7
3.1 General Advice 8
3.2 MIS-based Guided Sampling 9
3.3 RIS-based Guided Sampling 10
4 Path Guiding and Russian Roulette 13
4.1 Traditional Throughput-based Russian Roulette 14
4.2 Guided Russian Roulette 16
5 Combining Path Guiding with Direct Light Next-Event Estimation 19
6 Scattering Event Types and Light Path Expressions 20
7 Improving Path Guiding Robustness via Path Space Regularization 22
8 Other Integration Challenges 23
8.1 Training Data Generation 24
8.2 Special Production Renderer Features 25
8.3 General Integration Advice 25
9 Conclusion and Future Work 26
Acknowledgments 27
References 27

Ch. 2: Path Guiding Surfaces and Volumes in Disney's Hyperion Renderer: A Case Study 30
1 Introduction 30
2 The Challenges of Path Guiding in a Wavefront Production Renderer 31
2.1 Incompatibilities between Wavefront Architectures and Path Guiding 32
2.2 Hyperion Architecture Review 33
2.3 Previous Version of Path Guiding in Hyperion 33
3 Recording Radiance in a Wavefront Renderer 35
3.1 A Simple Example 35
3.2 A More Complex Example 36
3.3 Radiance Recording Compared to a Depth-First Renderer 39
3.4 Radiance Recorder Data Structure 40
4 Debugging Path Guiding in Hyperion 42
4.1 Radiance Recorder Debug Output 42
4.2 Path Guiding Visualizer 43
4.3 Visualizing the Directional Models 44
4.4 Visualizing Additional Statistics 47
5 Lessons from Path Guiding in Production 48
5.1 Vertex Splatting Depth 48
5.2 Render Iteration Schedule 51
5.3 Practical Realities of Production Rendering 55
5.4 Discussion of Production Shots Rendered with Path Guiding 60

6 Future Work 63
6.1 Improving Sampling Techniques 63
6.2 Sample Combination Scheme 64
6.3 Estimating Efficiency 64
6.4 Going Neural 65
6.5 Guiding Other Sampling Decisions 65
7 Conclusion 65
Acknowledgments 65
References 66

Ch. 3: Efficient Rendering of Caustics: Photon Guiding at Corona 68
1 Introduction 68
1.1 Path guiding vs Corona's caustics solver 69
2 Corona's caustics solver - initial implementation 70
2.1 Simplified Vertex Connection And Merging 70
2.2 Photon guiding 71
2.3 Number of photon paths 72
2.4 Additional details 72
3 Caustics solver improvements 73
3.1 Light sources stratification 73
3.2 Adaptive emission 74
3.3 Photon count estimation 75
3.4 Misc 77
4 Volumetric caustics 77
4.1 Beam radiance estimator 78
4.2 Volume photons. 78
5 Summary and Future work 79
References 80

Integrating Path Guiding into a Production Render: The Nitty Gritty Details

SEBASTIAN HERHOLZ, Intel Corporation, Germany



Fig. 1. Comparing equal sample count (128*spp*) renderings of the DARKINTERIOR using standard path tracing (**left**) with two different integrations of path guiding into Blender's production renderer Cycles: **center-left** a naïve integration and **center-right** our advanced integration that considers all nitty gritty details, presented in this work. It is worth noting that both renderings utilize the same guiding framework and cache.

In recent years, local path guiding algorithms have transitioned from research to the production rendering community. This transition enabled the simulation of challenging light transport effects, such as complex indirect illumination, volumetric multi-scattering, and even caustics, using the widely adopted path tracing rendering algorithm. With the introduction of Intel's Open Path Guiding Library (OpenPGL), it has become even easier than ever to integrate state-of-the-art, production-ready path guiding algorithms not only in research but also in production rendering systems.

While the details of most path guiding algorithms (e.g., guiding structures and training algorithms) are explained in their corresponding research literature, the details about how to integrate them efficiently into a production renderer are unknown or not discussed publicly. Especially the interaction of path guiding with other rendering techniques, such as Russian Roulette, next-event estimation, and roughening, can significantly influence the efficiency of path guiding. In the worst case, these interactions can so severely harm the effectiveness of path guiding that it becomes impractical in production.

In this section of the course, we provide some nitty-gritty details on how to integrate path guiding into production renderers efficiently. The details were gathered during the integration of the OpenPGL path guiding framework into multiple production and research renderers such as Blender's Cycles, Chaos's VRay, SideFX's Karma, Disney Animation's Hyperion (Chap. 2), Mitsuba, and PBRT. Our most important insights include a modified RIS-based guided directional sampling (Sec. 3.3) strategy, followed by an in-depth analysis of the interactions of path guiding with multiple other rendering techniques such as Russian Roulette (Sec. 4), next-event estimation (Sec. 5), scattering event types (Sec. 6), and roughening (Sec. 7), and how to combine them with path guiding without harming its efficiency, or making it unpredictable by changing the final look of the rendering.

1 INTRODUCTION

Accurate simulation of physically-based light transport is crucial when generating high-fidelity images. Today, path tracing-based algorithms [Kajiya 1986; Veach 1998] have become the de facto industry standard in production [Fascione et al. 2019, 2018]. Although multiple research works have improved the efficiency of path tracing to render scenes with moderate light transport complexity, complex effects like long indirect bounces, volumetric multiple scattering, or caustics are still challenging. Rendering these effects either takes a long time to converge/finish, limiting their accurate simulation to only a small set of shots in a production, or they have to be faked by artists (e.g., using virtual light sources or gobos). For a decade now, path guiding has found its way into production rendering [Vorba et al. 2019] and has been shown to significantly improve performance when rendering these challenging effects. By including learned information about a scene's radiance distribution, these algorithms improve the sampling quality, enabling the rendering of challenging effects (e.g., long complex indirect illumination, caustics, or multiple scattering in volumes) more efficiently. The information about the scene's light transport distribution is unknown upfront and, therefore, has to be learned either online during rendering or in a preprocessing step. Over recent years, much research has been done on the topic of path guiding by the rendering research community. These works mainly focused on developing new guiding structures and distributions [Dodik et al. 2022; Herholz et al. 2019; Müller et al. 2017; Ruppert et al. 2020; Vorba et al. 2014], their training processes, or new target functions [Rath et al. 2020, 2022] to not only reduce the variance but also improve the overall efficiency of the renderer. However, the path guiding algorithims presented in these works are usually evaluated in isolation by integrating them into scientific rendering frameworks, such as PBRT [Pharr et al. 2016] or Mitsuba [Jakob 2010], and on scenes primarly containing complex light transport constellations, where standard path tracing fails. In addition, standard rendering features such as next-event estimation or Russian roulette are often disabled to focus more on the expressiveness of the used representations and the improvements in sampling quality.

Unfortunately, production renderers are usually more complex than these scientific rendering frameworks and make heavy use of a wide variety of additional rendering features/techniques (e.g., next-event estimation, Russian roulette, roughening, light linking, etc.) to improve rendering efficiency. All these methods can interfere with path guiding and influence its efficiency, making the results shown in these research works not directly transferable to production renderers. When aiming to integrate path guiding as an always-on solution in a production renderer, it is essential that the sampling quality and efficiency only increase and do not decrease when compared to disabling path guiding.

This is especially important for scenes that do not contain complex light transport setups and can be rendered without path guiding. Unfortunately, details or experiences on how to integrate path guiding into a production rendering system efficiently and robustly are usually not discussed in path guiding literature. In this section of the course, we close this knowledge gap by presenting a set of critical insights and some nitty-gritty details gathered from multiple integrations of path guiding into production renderers. These insights include explorations of the interaction between path guiding and other techniques, such as Russian roulette, next-event estimation, or labeling scattering types, and how to combine them robustly, ensuring that the sampling quality and efficiency are never worse than standard unguided path tracing. These integration details are typically not discussed in research work. They are likely unknown to the majority of engineers and researchers who integrate path guiding into their renderers for the first time. Nevertheless, these nitty-gritty details can make a significant difference in the effectiveness and usability of path guiding in a production rendering environment. Fig. 1 shows the difference between a straightforward integration of path guiding,

as presented in multiple research papers, and an integration following our advice given in this chapter.

2 LOCAL PATH GUIDING

The standard importance sampling techniques used in path tracing have access only to local scene information, such as material/volume properties or light source setup, to build their sampling distribution. Since global information about the indirect light distribution is unknown, the resulting sampling distributions can only consider certain aspects of the integrand of the rendering equations for surface-based [Kajiya 1986] and volumetric light transport [Chandrasekhar 1960; Kajiya and Von Herzen 1984]. This limits their ability to sample some challenging light transport effects efficiently, especially when the distribution of the full integrand (i.e., full product) significantly diverges from the distribution of its sampled components (e.g., BSDF or phase function). Path guiding approaches overcome this limitation by incorporating additional information about a scene's light transport into the sampling process. This additional information is usually learned during rendering or in a preprocessing step. In principle, these algorithms can be classified into two categories, *global* and *local path guiding* algorithms. Global path guiding approaches [Guo et al. 2018a; Simon et al. 2018; Zheng and Zwicker 2019] aim to learn the optimal sampling distribution of a path all at once. These methods face the problem of finding an adequate representation and training procedure to learn a high-dimensional distribution, which is challenging. To our knowledge, these methods have not been adopted in production rendering yet. Local path guiding algorithms take another approach: They interpret the sampling/construction process of a full path as a series of nested estimators $\langle L_\mathrm{o} \rangle$[1]:

$$\langle L_\mathrm{o}(\mathbf{x}, \omega_\mathrm{o}) \rangle = L_\mathrm{e}(\mathbf{x}, \omega_\mathrm{o}) + \frac{f_\mathrm{s}(\mathbf{x}, \omega_\mathrm{o}, \omega_\mathrm{i}) \cdot |\cos(\theta_\mathrm{i})| \cdot \langle L_\mathrm{i}(\mathbf{x}, \omega_\mathrm{i}) \rangle}{p(\omega_\mathrm{i} \mid \mathbf{x}, \omega_\mathrm{o})} \tag{1}$$

where the estimator for the incoming radiance, in the case of surface-based light transport, is equal to the estimator of the outgoing radiance at the next path vertex $\mathbf{x}'$: $\langle L_\mathrm{i}(\mathbf{x}, \omega_\mathrm{i}) \rangle = \langle L_\mathrm{o}(\mathbf{x}', \omega'_\mathrm{o}) \rangle$ with $\omega'_\mathrm{o} = -\omega_\mathrm{i}$. These guiding approaches focus on optimizing the local sampling distribution $p(\omega_\mathrm{i} \mid \mathbf{x}, \omega_\mathrm{o})$ at every path vertex to achieve a globally optimal path distribution. To improve the local sampling distribution, these approaches learn an approximation/representation of the scene's light distribution. This distribution can be queried at each position in the scene for an approximation of the local incoming radiance distribution (Sec. 2.1), which is then used to build advanced *guided* importance sampling strategies (Sec. 3). By only relying on an approximation of the incoming radiance distribution, the dimensionality of the guiding distribution used by local path guiding approaches is significantly reduced compared to global path guiding approaches. The resulting guiding representations are usually only 5-dimensional (i.e., 3D-spatial and 2D-directional), making them easier to represent and learn. In the next section, we briefly introduce an open-source path guiding framework (OpenPGL) that offers functionality to represent and learn guiding distributions and to easily implement multiple local path guiding techniques in a rendering system (professional or scientific). Later, in Section 3, we then show how the guiding distributions provided by such a guiding framework are used to build guided directional importance sampling strategies that improve the sampling quality of the local estimators (Eq. 1) leading to a significant variance reduction.

[1]In this work, we mainly focus our explanations on surface-based light transport and refer the reader to the works of Herholz et al. [Herholz et al. 2019] and Xu et al. [Xu et al. 2024] for the volumetric case.

2.1 OpenPGL: An Open-Source Production-Ready local Path Guiding Framework

Intel's Open Path Guiding Library (OpenPGL) [Herholz and Dittebrandt 2022] is an open-source path guiding framework that is used by multiple production renderers (VRay, Blender, Karma, Hyperion (Chap. 2)) as well as a few research rendering frameworks (PBRT [Pharr et al. 2016][2], Mitsuba v0.6 [Jakob 2010][3]). The library adopted multiple techniques developed by the research community [Herholz et al. 2019; Müller et al. 2017; Rath et al. 2022; Ruppert et al. 2020; Vorba and Křivánek 2016; Xu et al. 2024], and extended and optimized them to make them production-ready. These extensions and optimizations focused on robustness, performance, quality improvements, and ensuring determinism, a critical feature in production rendering. OpenPGL is not only useful for production but also for the research community to develop new algorithms, which can easily be adopted later in production. A recent example is the work by Xu et al. [Xu et al. 2024] on guiding volume scattering probabilities, which was developed using OpenPGL and got immediately adopted after publication (i.e., v0.8.0). The library provides a local path guiding framework for learning and updating a 5D guiding structure incrementally (i.e., after every rendering iteration). At the moment, the guiding structure uses a KD-tree to subdivide the 3D spatial domain of the scene and supports 2D directional quad trees [Müller et al. 2017] and parallax-aware von Mises-Fisher mixture models [Ruppert et al. 2020] to represent the directional guiding distributions. The structure stores information about a scene's surface-based and volumetric light transport. At each position $\mathbf{x}$ in the scene, the structure can be queried to return a directional guiding distribution g that is proportional to an approximation $\tilde{L}_\mathrm{i}$ of the local incoming radiance at $\mathbf{x}$:

$$g(\omega_\mathrm{i} \mid \mathbf{x}) \propto \tilde{L}_\mathrm{i}(\mathbf{x}, \omega_\mathrm{i}). \tag{2}$$

This guiding distribution is used by the renderer to implement a guided directional sampling strategy (see Sec. 3). In addition to the guiding distribution, OpenPGL offers the ability to learn and query radiance caches for the incoming and outgoing or in-scattered radiance, quantities which either can be used directly (i.e., for biased or viewport rendering) or for other guiding techniques such as guided Russian Roulette (Sec. 4). In summary OpenPGL can be used to integrate the following path guiding techniques: directional guiding on surfaces or inside volume, guided Russian Roulette or volume scattering probability guiding (for primary and secondary rays), and to query the following light transport quantities: incoming and outgoing radiance, irradiance (depending on the surface normal), in-scattered radiance (for the Henyey-Greenstein phase functions) and an estimate for the pixel value (Sec. 4.2). In the following Secs. 3–7, we will explain how a guiding framework that supports similar features as OpenPGL can be robustly integrated into a production renderer. The main goal of the integration is to ensure that path guiding does not only improve rendering in challenging scenarios (e.g., caustics or complex indirect lighting) as usually presented in research papers, but that it also works well in standard scenarios without decreasing the sampling quality when compared to standard *unguided* path tracing.

3 GUIDED DIRECTIONAL SAMPLING

The optimal guided sampling distribution p_opt to reduce the variance of the outgoing radiance estimator (Eq. 1) needs to be proportional to the integrand of the reflected radiance of the rendering

[2]Fork of PBRTv4 integrating OpenPGL: https://github.com/OpenPathGuidingLibrary/pbrt-v4
[3]Fork of Mitsuba 0.6 integrating OpenPGL: https://github.com/sherholz/mitsuba-path-guiding

equation:

$$p_{\text{opt}}(\omega_\text{i} \mid \mathbf{x}, \omega_\text{o}) \propto f_\text{s}(\mathbf{x}, \omega_\text{o}, \omega_\text{i}) \cdot |\cos\theta_\text{i}| \cdot L_\text{i}(\mathbf{x}, \omega_\text{i}). \tag{3}$$

This optimal distribution would be proportional to the product of the BSDF, the cosine term, and the incoming radiance. Theoretically speaking, if such a distribution were possible to achieve and if it were consequently applied at every path vertex, the resulting estimator would have zero variance, meaning a single sample would be enough to render a converged image. Since this would require a perfect estimate of the incoming radiance, and we only have access to an approximation of the incoming radiance, this is not achievable in practice.
Nevertheless, any guided directional sampling that results in a distribution approaching the optimal one will significantly improve the sampling quality and, consequently, also reduce variance.

In the research community, multiple works proposed ways to approximate the optimal guided sampling distribution as defined in Eq. 3. Some achieve this by reconstructing or re-weighting the representation of directional guiding distribution, approximating the incoming radiance to represent the product with the BSDF and the cosine term on-the-fly at every path vertex ([Bashford-Rogers et al. 2012; Hey and Purgathofer 2002; Jensen 1995; Lafortune and Willems 1995]). Unfortunately, these approaches require multiple evaluations of the BSDF, e.g., one for each quad tree or histogram cell, photon, or photon footprint, or cosine lobe, which significantly increases the computational costs, especially when using complex BSDFs, which is usually the case in production renderers.

Other works eliminate this cost overhead by utilizing closed-form features, such as analytical solutions to calculate product distributions or convolutions, as provided by multiple parametric mixture model-based representations ([Dodik et al. 2022; Herholz et al. 2016, 2019]). For each BSDF model used in the scene, an intermediate representation is fitted that can be used to calculate the product distribution efficiently on-the-fly at every path vertex. While these methods show promising results in scientific rendering frameworks in scenes of medium complexity (e.g., number of different BSDF models) and relatively simple BSDF models (i.e., one or two parameters), they are impractical in production use cases. The reason for that is the required preprocessing phase in combination with the sheer number of different BSDF models and their complexity (i.e., number of parameters), leading to an enormous preprocessing time before rendering the first pixel. Nevertheless, in special cases such as volume rendering, where the phase functions are simple, such closed-form product approaches ([Herholz and Dittebrandt 2022; Herholz et al. 2019]) are usable, even in production.
Due to their limitations and impracticalities, these product-based guided sampling methods are not widely adopted in production rendering. In practice, other guided sampling approaches that are based on MIS (Sec. 3.2) or RIS (Sec. 3.3) are used to combine standard BSDF sampling with guided sampling, based on the approximation of the incoming radiance distribution (Eq. 2).

3.1 General Advice

Before diving into the details of how to implement guided directional sampling, we would like to provide some additional guidance on when to use it and when not to use it upfront. In some research works, guided directional sampling is applied always regardless of the type or parameters (e.g., roughness) of the BSDF at the current path vertex. This leads to insufficient sampling results when the BSDFs are specular (i.e., Dirac-delta) or nearly specular (i.e., small roughness value), since directional samples proposed by the guiding distribution usually lead to zero contributions when evaluating these BSDFs. In practice, it is, therefore, advisable to only apply guiding when the maximum roughness of a BSDF or the mean cosine of a phase function passes a given threshold. In our experiments, we used a roughness threshold of $\alpha > 0.05$ and a mean cosine threshold of $g < 0.95$ to activate guided sampling. It is worth noting that these thresholds can vary based on the renderer and its definition of roughness.

3.2 MIS-based Guided Sampling

Using single-sample multiple importance sampling (MIS) [Veach and Guibas 1995] is probably the most commonly used approach to implement guided directional sampling ([Bashford-Rogers et al. 2012; Dodik et al. 2022; Herholz et al. 2019; Hey and Purgathofer 2002; Jensen 1995; Lafortune and Willems 1995; Müller et al. 2017; Vorba et al. 2014]). Single-sample MIS combines BSDF-based and guiding distribution-based sampling by either drawing a sample from the BSDF or from the guiding distribution. The probability of either picking a sample from the BSDF (i.e., p_{f_s}) or the guiding distribution (i.e., g) is defined by the BSDF selection weight α, which results in the following sampling PDF for a direction ω_i:

$$p_{\text{guide}}(\omega_i \mid \mathbf{x}, \omega_o) = \alpha \cdot p_{f_s}(\omega_i \mid \mathbf{x}, \omega_o) + (1 - \alpha) \cdot g(\omega_i \mid \mathbf{x}, \omega_o), \tag{4}$$

where p_{f_s} is proportional to the BSDF and the cosine product:

$$p_{f_s}(\omega_i \mid \mathbf{x}, \omega_o) \propto f_s(\mathbf{x}, \omega_o, \omega_i) \cdot |\cos \theta_i|. \tag{5}$$

Since the guiding distribution g does not necessarily represent the optimal sampling distribution (Eq. 3), alternating between sampling from the BSDF and the guiding distribution ensures that the resulting estimator is unbiased, as long as α is greater than zero. A fixed selection weight of $\alpha = 0.5$ is used in practice, resulting in a defensive sampling strategy [Hesterberg 1995] leading to an upper bound for the variance of the resulting estimator of four times the lowest variance of the two sampling strategies. Such an MIS-based combination of the two sampling strategies compensates for many drawbacks of only sampling based on the BSDF and not the full product. This is especially true when indirect illumination dominates the product distribution. It is worth noting that the guiding distribution $g(\omega_i \mid \mathbf{x}, \omega_o)$ does not necessarily need to be proportional to an approximation of the full incoming radiance (Sec. 5). If supported by the guiding framework and the used representation, a guiding distribution proportional to the product of the approximation of the incoming radiance and parts of the integrand (e.g., cosine term or phase function) [4] can be used to improve the guided sampling quality further. Using this MIS-based guided sampling strategy leads to the following MIS estimator for the outgoing radiance:

$$\langle L_o(\mathbf{x}, \omega_o) \rangle_{\text{MIS}} = L_e(\mathbf{x}, \omega_o) + \frac{f_s(\mathbf{x}, \omega_o, \omega_i) \cdot |\cos(\theta_i)|}{p_{\text{guide}}(\omega_i \mid \mathbf{x}, \omega_o)} \cdot \langle L_i(\mathbf{x}, \omega_i) \rangle. \tag{6}$$

Importantly, this formulation is already a simplified version when using the balance heuristic for calculating the MIS weights, which is only applicable if the PDFs of both sampling procedures can be evaluated separately for a given direction. For some BSDFs (e.g., [Guo et al. 2018b]), this is not the case; for example, when their sampling procedure relies on a random walk process and the BSDF evolution and PDF are unknown, but the sampling weight, the division of the BSDF and the PDF, is. In these cases, the original non-simplified version of the MIS estimator needs to be used using an approximation p'_{f_s} of the actual BSDF sampling PDF. This leads to the following MIS estimator:

$$\langle L_o(\mathbf{x}, \omega_o) \rangle_{\text{MIS}} = L_e(\mathbf{x}, \omega_o) + \underbrace{\frac{p'_{f_s}(\omega_i \mid \mathbf{x}, \omega_o)}{\alpha \cdot p'_{f_s}(\omega_i \mid \mathbf{x}, \omega_o) + (1 - \alpha) \cdot g(\omega_i \mid \mathbf{x}, \omega_o)}}_{\text{balance heuristic MIS weight}} \cdot \underbrace{\frac{f_s(\mathbf{x}, \omega_o, \omega_i) \cdot |\cos(\theta_i)|}{p_{f_s}(\omega_i \mid \mathbf{x}, \omega_o)}}_{\text{BSDF sample weight}} \cdot \langle L_i(\mathbf{x}, \omega_i) \rangle \tag{7}$$

[4] These are features provided by OpenPGL if directional representations based on the von-Mises-Fisher mixtures are used.

Note: This formulation is only needed for the samples generated via BSDF sampling. Samples that are generated using the guiding distribution can still use the simplified version from Eq. 4 but need to replace p_{f_s} with p'_{f_s}. If p'_{f_s} is equal to p_{f_s}, the BSDF PDFs in the numerator of the MIS weight and in the denominator of the BSDF sample weight cancel out, resulting in a sample weight that uses the MIS guiding PDF p_{guide} from Eq. 4.

3.2.1 Selection Weight Optimization Although the MIS-based approach to implement guided directional sampling is widely used, it does not guarantee that it will always improve the sampling quality compared to standard BSDF-based importance sampling. In the worst case, it can significantly decrease the sampling quality, leading to a significant variance increase, making path guiding less efficient than standard path tracing.
This behavior is often not recognizable when rendering scenes with challenging light transport constellations like the ones often used in path guiding research papers, where the variance of standard path tracing is extremely high due to its inability to sample important light contributions robustly. In scenes with medium light transport complexity the inefficiency of MIS-based guided sampling can become more prominent. This is especially the case on surfaces with mainly glossy, specular, or nearly specular BSDFs. In these cases, the directional samples proposed by the guiding distribution are often outside the BSDF lobe, and the evaluation of the BSDF returns zero or a value close to zero. These samples lead to early path terminations, often resulting in zero contributions to the final estimate, which increases the variance of the estimator.

There have been multiple approaches to counter this problem. A simple solution is to use a heuristic based on the roughness of the BSDF at the current path vertex as presented by Schüßler et al. [Schüßler et al. 2022]. This heuristic intends to use more samples from the guiding distribution when the material is rough (i.e., diffuse) and fewer when it is glossy or even none when it is almost specular (i.e., Dirac-delta).

Another approach was presented by Müller [Müller 2019], who proposed fitting an optimal selection weight that reduces the variance of the MIS estimator (Eq. 6). These optimized weights are fitted for each cell of the spatial subdivision structure, while training the guiding structure. Since each cell only stores one weight, this optimized weight represents a marginalization over all BSDFs of the region covered by the cell. The resulting weight is, therefore, only the average optimal weight, which might still be insufficient in cases where a cell contains BSDFs with highly varying roughness values.

While both of these methods can improve the performance of MIS-based guided sampling compared to a fixed selection weight (e.g., $\alpha = 0.5$), they do not guarantee to always be optimal or never lead to a worse sampling distribution than standard path tracing using only BSDF sampling. One of the main limitations of MIS-based guided sampling is that, as long as, the guiding distribution only represents a component of the product and not the full product (Eq. 3), such as the incoming radiance, the potential improvement of MIS is limited: $\alpha \cdot A + (1 - \alpha) \cdot B \neq A \cdot B$.

3.3 RIS-based Guided Sampling

Resampled Importance Sampling (RIS) [Talbot et al. 2005] is another way to build a guided sampling strategy with a distribution close to the optimal product distribution Eq. 3. Unlike the previously mentioned product approaches, RIS does not rebuild an entire product distribution on-the-fly or require any preprocessing of representations of all BSDFs used in the scene. In the context of path guiding, RIS was first introduced by Deng et al. [Deng et al. 2020] for guiding volumetric light transport using a guiding distribution that does not support closed-form product sampling [Herholz et al. 2019]. Our RIS approach differs from the one presented by Deng et al. [Deng et al. 2020]

mainly in the target function Eq. 10 used to drive the resampling process and in a discussion about how to combine RIS sampling with next-event estimation Sec. 3.3.1.

The first step of RIS generates a set of sample candidates $\{c_1, ..., c_N\}$, which will be resampled later based on some target function. To generate the set, we draw N_{f_s} samples from the BSDF and N_g candidate samples from our guiding distribution. This process results in a set of $N = N_{f_s} + N_g$ resampling candidates, where each candidate c_i corresponds to direction $\omega_{i,i}$ in which the path can continue (i.e., $c_i = \omega_{i,i}$). The PDF for generating a candidate is given by:

$$p(c_i) = \frac{1}{N} \cdot \left(N_{f_s} \cdot p_{f_s}(\omega_{i,i} \mid \mathbf{x}, \omega_o) + N_g \cdot g(\omega_{i,i} \mid \mathbf{x})\right). \tag{8}$$

In our experience, we already achieve sufficient and efficient sampling results by only using two candidates, one from the BSDF (i.e., $N_{f_s} = 1$) and one from the guiding distribution (i.e., $N_g = 1$). This simplifies the candidate PDF to the following:

$$p(c_i) = 0.5 \cdot p_{f_s}(\omega_{i,i} \mid \mathbf{x}, \omega_o) + 0.5 \cdot g(\omega_{i,i} \mid \mathbf{x}). \tag{9}$$

It is worth noting that the guiding distribution used to sample candidates does not need to be proportional to the incoming radiance (Eq. 2). For example, when using OpenPGL and the parallax-aware von-Mises Fisher mixtures models (PAVMM) as directional distribution, we use the product of the incoming radiance and the normal oriented cosine as sampling distribution, to generate candidates. Due to the PAVMMs, such a product distribution can be sampled and evaluated efficiently in closed form. A general rule of thumb is that the closer the sampling distribution of the candidates is to the target distribution, the more efficient RIS will become.

The traditional RIS method presented by Talbot [Talbot et al. 2005] uses the product of the BSDF, the cosine term, and the incoming radiance as the target function for the resampling process. This is possible since Talbot uses an exact estimate of the incoming radiance by directly querying light sources (e.g., an environment map). In path guiding, we only have access to an approximation $\tilde{L}_i$ of the incoming radiance and therefore, need to pursue a more conservative approach to avoid insufficient or even biased sampling due to inaccuracies in the incoming radiance approximation. To account for such inaccuracies, our target function combines our incoming radiance estimate $g_{\tilde{L}_i}$ with a constant (i.e., $\frac{1}{4\pi}$):

$$t(c_i) = f_s(\mathbf{x}, \omega_o, \omega_{i,i}) \cdot |\cos(\theta_{i,i})| \cdot \underbrace{\left(\alpha \cdot \frac{1}{4\pi} + (1-\alpha) \cdot g_{\tilde{L}_i}(\mathbf{x}, \omega_{i,i})\right)}_{\text{defensive } L_i \text{ estimate}}. \tag{10}$$

Since our incoming radiance estimate is based on a normalized distribution, the weighted combination with the constant of $\frac{1}{4\pi}$ leads to a defensive RIS sampling strategy. It has a similar effect as the one described by Hesterberg [Hesterberg 1995] for MIS, where α influences the probability of selecting a BSDF sample during the resampling step. In our experiments, we found that using $\alpha = 0.5$ serves as a robust defensive sampling weight. It is worth noting that, based on the integration into the renderer, the used approximation of the incoming radiance $g_{\tilde{L}_i}$ does not necessarily need to be proportional to the complete incoming radiance but could also contain a fraction of it (Sec. 5).

Using our defensive target function (Eq. 10) and the sampling PDF of a candidate (Eq. 8), we calculate a candidate's resampling weight as follows:

$$w(c_i) = \frac{t(c_i)}{p(c_i)} \tag{11}$$

In the next step, we resample our final candidate sample c_i from the candidate set proportional to the resampling weights, which results in the following RIS estimator:

$$\langle L_\mathrm{o}(\mathbf{x}, \omega_\mathrm{o})\rangle_\mathrm{RIS} = L_\mathrm{e}(\mathbf{x}, \omega_\mathrm{o}) + \frac{f_\mathrm{s}(\mathbf{x}, \omega_\mathrm{o}, \omega_{\mathrm{i},i}) \cdot |\cos(\theta_{\mathrm{i},i})|}{t(c_i)} \cdot \left[\frac{1}{N} \cdot \sum_{i=1}^{N} w(c_i)\right] \cdot \langle L_\mathrm{i}(\mathbf{x}, \omega_\mathrm{i})\rangle. \tag{12}$$

This equation for the RIS estimator can be simplified by first converting the resampling weights to resampling probabilities through normalization:

$$P_{rs}(c_i) = \frac{w(c_i)}{\frac{1}{N} \cdot \sum_{i=1}^{N} w(c_i)}, \tag{13}$$

which, after some reformulation, leads to the following RIS estimator:

$$\langle L_\mathrm{o}(\mathbf{x}, \omega_\mathrm{o})\rangle_\mathrm{RIS} = L_\mathrm{e}(\mathbf{x}, \omega_\mathrm{o}) + \frac{f_\mathrm{s}(\mathbf{x}, \omega_\mathrm{o}, \omega_\mathrm{i}) \cdot |\cos(\theta_\mathrm{i})|}{p(c_i) \cdot P_{rs}(c_i)} \cdot \langle L_\mathrm{i}(\mathbf{x}, \omega_\mathrm{i})\rangle. \tag{14}$$

This notation of the RIS estimator is similar to the traditional Monte-Carlo estimator (Eq. 1 or Eq. 6), where the integrand of the rendering equation is in the numerator and the PDF for the sample is in the denominator. In RIS, the PDF for the final sample is the PDF of generating the candidate sample c_i multiplied by the probability of resampling the candidate from the candidate set. It is worth noting that, due to the stochastic candidate generation process, this PDF is only valid for this specific instance of a sample for the direction $\omega_{\mathrm{i},i}$, namely the instance generated from exactly this candidate set. Consequently, this PDF cannot be used directly for calculating MIS weights in the context of next-event estimation (see Sec. 3.3.1). Fig. 2 demonstrates the improved sampling quality of RIS-based against two versions of MIS-based guided sampling: one with fixed selection weights ($\alpha = 0.5$) and one with roughness-driven ones. To focus only on the differences in sampling quality, we used the same pre-trained guiding cache (one for each scene) for each sampling method. As we can see, MIS-based sampling with a fixed selection weight ($\alpha = 0.5$) results in the lowest sampling quality between the three strategies. Especially on the glossy metallic surfaces in the DarkInterior scene (**top**), MIS can lead to insufficient sampling behavior since the samples proposed by the incoming radiance-based guiding distribution generate invalid samples (i.e., samples with low contributions or outside the BSDF lobe). The roughness-based MIS strategy (**center**) can improve sampling quality by reducing the number of samples drawn from the guiding distribution, but it does not achieve the same quality as the RIS-based strategy (**right**).

3.3.1 RIS and MIS for Next-Event Estimation As previously mentioned, the PDF for an RIS sample is part of a stochastic process and is tied to a specific instance of a directional sample. This means that this PDF can not be evaluated for a given direction afterwards: for example, to calculate MIS weights for next-event estimation. It is possible to use an estimate of the PDF by repeating the stochastic candidate generation process to estimate the selection probability $P(c_i)$ for a given direction $c_i = \{\omega_{\mathrm{i},i}\}$. Unfortunately, such a process quickly becomes inefficient when the size of the candidate set or the number of next-event estimation samples grows.

Our solution to this problem is relatively simple. It is based on the fact that, when calculating the MIS weights, it is not absolutely necessary to use the actual PDF of a direction, as long as its representative is the same when calculating the MIS weights for each technique. In our experiments, we found that using the candidate generation PDF from Eq. 8 or Eq. 9 as a representative PDF for the MIS weight calculations leads to sufficient weighting between the contributions of the different estimators. It is worth noting that, while the resulting MIS weights perform well in practice, they are not optimal, presenting an excellent opportunity for future research to improve them further.

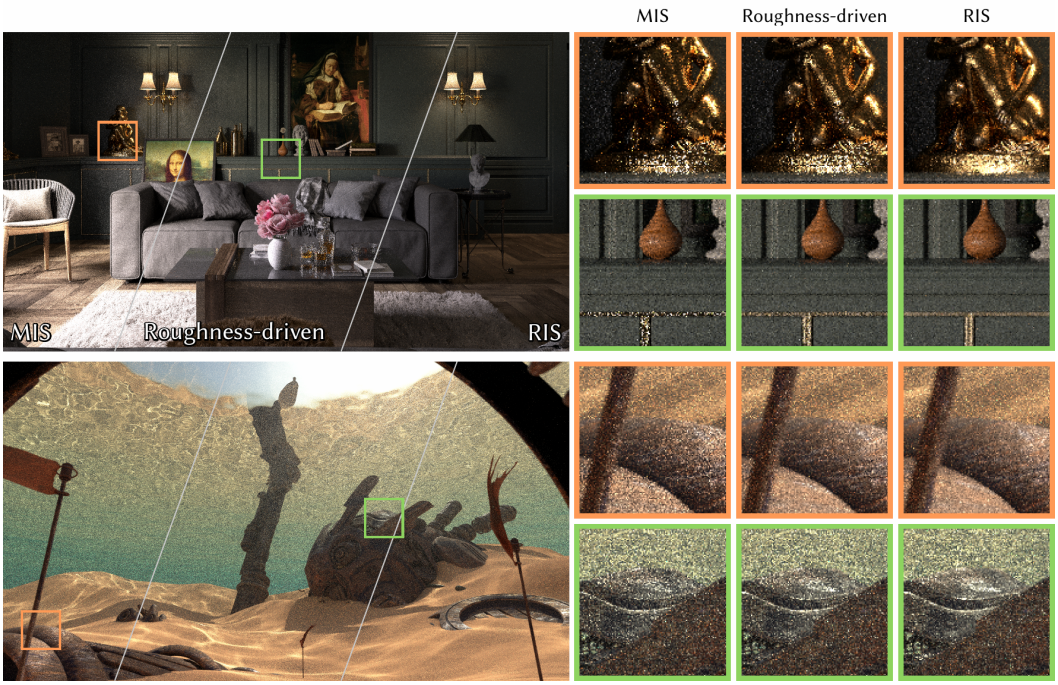


Fig. 2. Comparing the quality of different guided sampling strategies using equal sample count (64spp) rendering of the DarkInterior (**top**) and ShallowUnderWater scenes. Both scenes are rendered using Blender's Cycles rendering engine. The **left** shows MIS-based guided sampling (Sec. 3.2) using a fixed selection weight of $\alpha = 0.5$, while the **center** adjusts the MIS selection weight based on the surface roughness. The **righ** shows our RIS-based strategy as presented in Sec. 3.3.

4 PATH GUIDING AND RUSSIAN ROULETTE

Russian Roulette (RR) is an important and commonly used technique in rendering to improve the computational efficiency of path tracing by terminating paths stochastically. Through this termination, RR inevitably leads to an increase in variance (a.k.a. noise) when comparing equal sample count renderings that do not use RR. An efficiency increase is achieved when the ratio of saved compute time is higher than the ratio of the variance increase. Ideally, RR only terminates paths with a low contribution to the final pixel estimate. Unfortunately, the contribution of the suffix path is unknown upfront, and it is therefore common practice to rely on heuristics to estimate it. The most widely used heuristic is the one introduced by Arvo and Kirk [Arvo and Kirk 1990], which is based on the Monte-Carlo throughput weight (Sec. 4.1). Although this heuristic often works well within a standard path tracer, it can lead to a significant increase in variance within a guided path tracer when applied directly. This shortcoming of throughput weight-based RR was first pointed out by Vorba et al. [Vorba et al. 2014] but is not well-known in the rendering community. Therefore, it is standard practice in path guiding papers to either disable RR [Dodik et al. 2022; Müller et al. 2017; Ruppert et al. 2020], use a more defensive RR heuristic [Vorba et al. 2014], or use a more advanced RR strategies [Herholz et al. 2019; Rath et al. 2022; Vorba and Křivánek 2016] (Sec. 4.2). Unfortunately, all three approaches have their shortcomings. The first two can increase the average path length, which, in the worst case, can make guided path tracing less efficient than standard path tracing using throughput weight-based RR. The third approach, on the other hand, requires a

more complex guiding framework since these methods rely on estimates of additional quantities (e.g., pixel estimates, adjoints, or variance estimates).

In the following two sections, we first revisit the traditional throughput weight-based RR heuristic as presented by Arvo and Kirk [Arvo and Kirk 1990], identify why guided path tracing leads to insufficient termination behavior, and propose a simple workaround to fix this and restore the initial throughput weight-based termination behavior (Sec. 4.1.1). In the second section (Sec. 4.2), we discuss more advanced guided RR strategies and how these can be efficiently integrated into a production renderer using the functionality provided by a path guiding framework such as OpenPGL [Herholz and Dittebrandt 2022].

4.1 Traditional Throughput-based Russian Roulette

As mentioned before, Arvo and Kirk [Arvo and Kirk 1990] proposed an RR heuristic based on the Monte-Carlo throughput weight t of a path $\bar{\mathbf{x}}$ to calculate the paths' survival probability q at its current end vertex $\mathbf{x}_k$[5]:

$$q(\mathbf{x}_k) = t(\bar{\mathbf{x}}) = \underbrace{\left[\prod_{i=1}^{k-1} \frac{f_\mathrm{s}(\mathbf{x}_i, \omega_{\mathrm{o},i}, \omega_{\mathrm{i},i}) \cdot |\cos\theta_{\mathrm{i},i}|}{q(\mathbf{x}_i) \cdot p(\omega_{\mathrm{i},i} \mid \mathbf{x}_i, \omega_{\mathrm{o},i})}\right]}_{\text{prefix path } \bar{\mathbf{x}}_{k-1}} \cdot \underbrace{\frac{f_\mathrm{s}(\mathbf{x}_k, \omega_{\mathrm{o},k}, \omega_{\mathrm{i},k}) \cdot |\cos\theta_{\mathrm{i},k}|}{p(\omega_{\mathrm{i},k} \mid \mathbf{x}_k, \omega_{\mathrm{o},k})}}_{\text{path segment weight}}. \tag{15}$$

The throughput weight of that path is the product of all local Monte-Carlo weights at each path vertex $\{\mathbf{x}_1, ..., \mathbf{x}_k\}$ which, in case of surface-based rendering, are defined by the evaluation of the rendering equation at each path vertex $\mathbf{x}_i$ divided by the PDF for sampling the direction towards the next path vertex. When RR is applied, the weights of the prefix path vertices $\{\mathbf{x}_1, ..., \mathbf{x}_{k-1}\}$ need to include the division by the survival probability at each vertex $\mathbf{x}_i$ where RR was applied.

In traditional path tracing, a new direction is sampled using a PDF that tries to be as proportional as possible to the BSDF times cosine at the current path vertex (see Sec. 3 Eq. 5). As a result, we can reinterpret the local Monte-Carlo weight as an estimate for the directional albedo:

$$\bar{f}_\mathrm{s}(\mathbf{x}, \omega_\mathrm{o}) = \int f_\mathrm{s}(\mathbf{x}, \omega_\mathrm{o}, \omega_\mathrm{i}) \cdot |\cos\theta_\mathrm{i}| \, \mathrm{d}\omega_\mathrm{i} \approx \frac{f_\mathrm{s}(\mathbf{x}, \omega_\mathrm{o}, \omega_\mathrm{i}) \cdot |\cos\theta_\mathrm{i}|}{p_{f_\mathrm{s}}(\omega_\mathrm{i} \mid \mathbf{x}, \omega_\mathrm{o})}, \tag{16}$$

which, due to the energy conservation of the BSDF, is defined to be less than or equal to one.

It is therefore valid to assume that the RR-corrected throughput weight of the prefix path converges to around one, making the local RR decision at the current path vertex $\mathbf{x}_k$ mainly depends on the estimate of the local directional albedo. The overall path probability to generate a path with the length k is therefore the product of the local directional albedos at each path vertex. As a result ,throughput weight-based RR leads to a heuristic that estimates the future path contribution solely on the statistics of the prefix path, estimating a high future contribution if the prefix path mainly interacted with bright materials and a low contribution if the prefix path mainly interacted with darker materials.

When using path guiding, the directional sampling PDF is not proportional to the BSDF anymore, making the local Monte-Carlo weight at the vertex not an estimate for the local directional albedo. In fact, path guiding can lead to high PDF values towards important directions, which do not necessarily result in high BSDF values. Since the guiding PDF is used in the denominator in Eq. 16 the value of the directional albedo is significantly underestimate leading to a higher chance to terminate the path towards important directions. Through this *early path termination*, naively applying traditional throughput weight-based RR, therefore, acts contrary to the initial goal of

[5]About the t parameter which in practice is usually set to 1 which leads to this simplified version.

path guiding. Examples of this effect, where throughput-weight-based RR leads to a significant variance increase when naively integrated into a guided path tracer is shown in the **center** in Fig. 3. In the following, we introduce a simple fix that reestablishes the expected termination behavior of throughput weight-based RR when using path guiding.

4.1.1 Russian Roulette Correction Factor Our solution to reestablish the intended behavior of throughput weight-based RR when using path guiding introduces a correction factor c_i that corrects the difference between the guiding PDF p_{guide} and the BSDF PDF p_{f_s} at the i-th path vertex:

$$c_i = \frac{p_{\text{guide}}(\omega_{\text{i},i} \mid \mathbf{x}_i, \omega_{\text{o},i})}{p_{f_s}(\omega_{\text{i},i} \mid \mathbf{x}_i, \omega_{\text{o},i})}. \tag{17}$$

Multiplying c_i with the current path segment weight (Eq. 15) results in the estimator for the local directional albedo (Eq. 16). The correction factor for a path, with the current end point at $\mathbf{x}_k$, is defined by the product of the local correction factors from each previous path vertex $\mathbf{x}_i$:

$$c(\mathbf{x}_k) = \prod_{i=1}^{k} c_i. \tag{18}$$

With the help of our RR correction factor for the current path, we can calculate the fixed path survival probability q' at the current endpoint point $\mathbf{x}_k$ of the path:

$$q'(\mathbf{x}_k) = c(\mathbf{x}_k) \cdot t(\mathbf{x}_k). \tag{19}$$

By using q' as the path's survival probability, RR behaves as intended initially by Arvo and Kirk [Arvo and Kirk 1990] when using a guided path tracer. It is worth noting that when simulating long light paths (e.g., > 60 when simulating clouds), the paths' correction factor can cause numerical problems and instabilities when defined as float. A way to avoid this is to define $c(\mathbf{x}_k)$ as a double or by using an estimate of the local directional albedo directly instead of the throughput weight. Examples of how our RR correction factor fixes the termination behavior of throughput weight-based RR when using a guided path tracer is demonstrated on the right in Fig. 3.

In the CountryKitchen scene (**top**), path guiding without our correction only leads to minor, almost not visible, improvements when compared to standard path tracing with RR. This is mainly due to the early termination of important paths. For example, standard path tracing with RR yields an average path length of 3.2, while guiding and RR without our correction reduces the average path length to 3.0. With our correction path guiding, RR does not terminate important paths early, which leads to an average path length of 3.1 and a significant quality improvement (i.e., noise reduction). The render time increase compared to standard path tracing is +13% for path guiding without and +14% with our correction[6].

In the ShallowUnderWater scene (**bottom**), standard path tracing is not able to resolve the caustics, both the directly visible ones on the floor and the ones reflected on the water surface. To resolve these caustics, path guiding is required. Since we only apply RR at the second path vertex (after the primary hit point), path guiding with RR with our correction can resolve the directly visible caustic but fails to resolve the reflected ones at later path vertices due to early terminations of important paths. With our RR correction, path guiding can robustly resolve both caustics, the directly visible and the reflected ones. In this scene, the changes in average path length and render times compared to standard path tracing with RR are minimal. The average path length changes from 2.97 to 2.83 and 2.86, while the render times for both guiding methods only increase by +2.5% [6].

[6]Since we are using the same pre-trained guiding cache for all tests, the render times exclude the training times of the guiding caches.

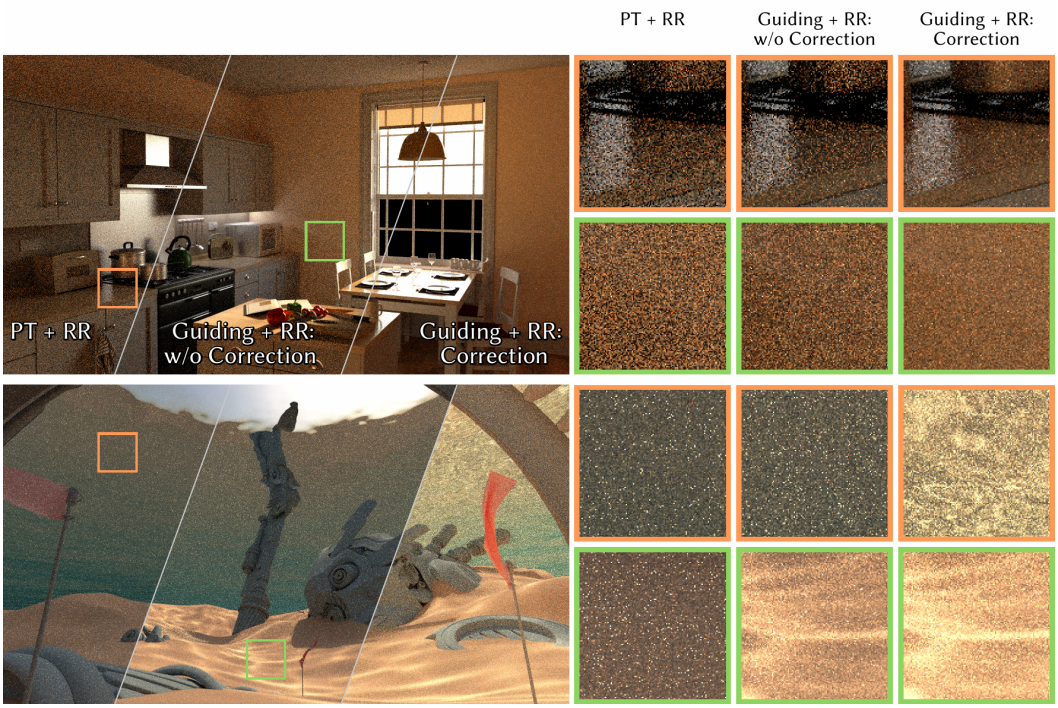


Fig. 3. Equal sample count (64*spp*) renderings of the CountryKitchen and ShallowUnderWater scenes, comparing standard path tracing without guiding and RR (**left**) against path tracing with path guiding and RR: not using (**center**) and using (**right**) our correction factor (Eq. 19). All renderings are using the exact same guiding cache, and RR is applied at the second path vertex (i.e., after the primary hit point). Both scenes are rendered using our extended version of PBRT [2].

4.2 Guided Russian Roulette

More advanced RR techniques, such as *adjoint-driven* or *efficiency-aware* Russian Roulette and splitting (ADRRS [Herholz et al. 2019; Vorba and Křivánek 2016] and EARS [Rath et al. 2022]) do not suffer from early path termination introduced by a change in the path tracers' sampling behavior. These methods are not based on heuristics but on mathematical derivations and concepts, such as the zero-variance theory [Hoogenboom 2008], that consider the full Monte-Carlo process, including the used path sampling procedures. In the following, we only discuss adjoint-driven RR (ADRR), since ADRRS and EARS mainly differ in their splitting behavior and only lead to minimal differences in their termination behavior. Additionally, the explanation of ADRR and its integration into a path guiding framework is much simpler and intuitive.

4.2.1 Adjoint-driven Russian Roulette ADRR was first introduced to the rendering community by Vorba et al. [Vorba and Křivánek 2016] to counter the early path termination problem of traditional throughput weight-based RR when using path guiding. While initially derived for surface-based light transport, the technique was later extended to work for volumetric light transport by Herholz et al. [Herholz et al. 2019]. According to ADRR, the optimal path survival probability q_{ad} at $\mathbf{x}_k$ is based on the ratio of the *expected path contribution* to the final pixel value I_p:

$$q_{\text{ad}}(\bar{\mathbf{x}}_k) = \frac{t(\bar{\mathbf{x}}_{k-1}) \cdot \mathrm{E}[\langle L_{\text{o}}(\mathbf{x}_k, \omega_{\text{o},k})\rangle]}{q_{\text{ad}}(\mathbf{x}_{k-1}) \cdot I_p}. \tag{20}$$

The expected path contribution is estimated using the current path throughput weight $t(\bar{\mathbf{x}}_{k-1})$ until the previous path vertex $\mathbf{x}_{k-1}$ and an estimate of adjoint quantity which, in case of surface-based light transport, is the outgoing radiance L_o at $\mathbf{x}_k$ towards the outgoing direction $\omega_{\mathrm{o},k}$. Since our definition of the throughput weight in Eq. 15 does not include the division by the survival probability at $\mathbf{x}_{k-1}$, we need to add the product with $q_\mathrm{ad}(\mathbf{x}_{k-1})$ in the denominator. When ADRR is applied in the context of volume rendering the throughput weight needs to include the transmittance weight from $\mathbf{x}_{k-1}$ towards $\mathbf{x}_k$ and the adjoint for the outgoing radiance needs to be replaced by the product of the local scattering coefficient at $\mathbf{x}_k$ and an estimate of the in-scattered radiance ([Herholz et al. 2019] Eq. 27). Both quantities, the estimate for the outgoing and in-scattered radiance, can be learned during the training phase of the guiding structure and can be queried on-the-fly during rendering. Based on the way RR and NEE are integrated into the renderer (i.e., if RR is performed before or after NEE), the L_o estimate should either contain the full direct illumination scattered at x or the MIS weighted one (see Sec. 5).

The final pixel value I_p can be estimated during rendering using the pixel samples from previous rendering iterations/progressions. Since these estimates will be noisy, a denoiser such as Intel's OpenImageDenoise (OIDN) [Áfra 2019] can be used to improve the quality of the pixel estimates. In our experiments, we found out that updating the pixel estimates after $\{2^0, ..., 2^N\}$ rendering iteration/progression and denoising the results using OIDN leads to adequate pixel estimates. Since OIDN can run on both CPU and GPU, supporting Intel, Nvidia, AMD, and Apple GPUs, the overhead of updating the current pixel estimate buffer is negligible for CPU-based rendering when a GPU is equipped in the system. An example series of denoised pixel estimates after multiple update iterations is presented in Fig. 4. A comparison of traditional RR and adjoint-driven RR is



Fig. 4. Examples of the pixel estimate buffers at different rendering iterations: 2spp (**top left**), 4spp (**top right**), 8spp (**bottom left**), 16spp (**bottom right**). Even at lower sample counts, like 2 or 4spp, the pixel estimate buffers are already providing acceptable information to drive guided RR decisions (Eq. 20).

presented in Fig. 5. It is worth noting that the comparison uses the same guiding cache and random number sequences that are bound to the pixel and the path indices. This results in a path generation

process which only differs in path length due to RR, meaning that the difference (i.e., variance increase) in the images (center and right) compared to using no RR (left) is solely due to the used RR strategy. In both scenes, CountryKitchen and DeepUnderWater, guided RR (**center**), when compared to using no RR (**left**), only leads to a minimal variance increase, resulting in a similar noise behavior/pattern. On the other hand, throughput weight-based RR with our correction factor (Sec. 4.1.1) significantly increases the variance, leading to visibly more noise (**right**).

Since the CountryKitchen scene is a closed environment, disabling RR leads to a high average path length of 11.3. Applying RR significantly reduces the average path length to 4.3 for guided RR and 3.1 for throughput weight-based RR. This shortening of the average path lengths has a significant influence on render times, which is the reason why RR was introduced into rendering in the first place. Guided RR reduces the render time by −61.6%, while leading to similar noise behavior, while throughput weight-based RR reduces render times by −74.0%, while significantly increasing noise.

Since the DeepUnderWater scene is a more open world setup, the differences in average path lengths are not that drastic. Here, disabling RR leads to an average path length of 4.7 while using guided RR leads to an average path length of 3.7 and 2.9 for throughput weight-based RR. The changes in render times are therefore minimal, leading to a −22.2% decrease of guided RR and −43.1% for throughput weight-based RR. It is worth noting that the effects on the rendering times are directly correlated with the changes in average path length across all RR techniques.

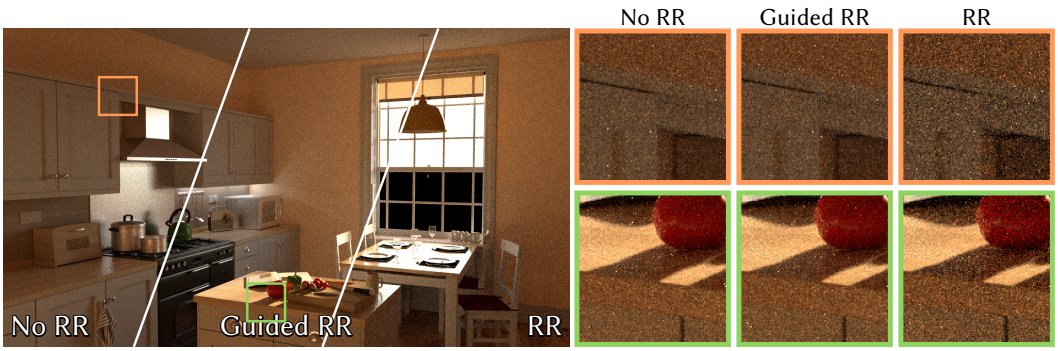


Fig. 5. Comparing equal sample count (64spp) renderings of the CountryKitchen and DeepUnderWaterscenes using path guiding, without using RR (**left**), guided RR (**center**), and throughput weight-based RR (**right**). The guided RR technique uses adjoint-driven RR (Sec. 4.2), while the throughput weight-based RR uses our RR correction factor (Sec. 4.1.1). Both scenes are rendered using our extended version of PBRT [2].

General Advice: Most guided RR techniques require approximations of incoming or outgoing directional quantities (e.g., radiance, variance, or second moment). At path vertices with glossy, almost specular (i.e., Dirac-delta) materials, these approximations have to be precise to avoid making an insufficient decision, which would negatively affect the efficiency of the renderer. Such precise approximations are hard to achieve due to the expressiveness of the used model to represent (e.g., parametric mixtures, quad-trees, neural networks) or the spatial marginalization of the spatial sub-division structure. We, therefore, found out that in cases where the BSDF closures are almost specular, below a given roughness threshold (e.g., $\alpha < 0.05$), it is advisable not to use guided RR and either fall back to a local directional albedo-based approach (Eq. 16) or to keep the survival probability high ($q(\bar{\mathbf{x}}_k) = 0.95$), since surviving previous guided RR decisions already means that the contribution of this glossy or specular suffix path is important.

5 COMBINING PATH GUIDING WITH DIRECT LIGHT NEXT-EVENT ESTIMATION

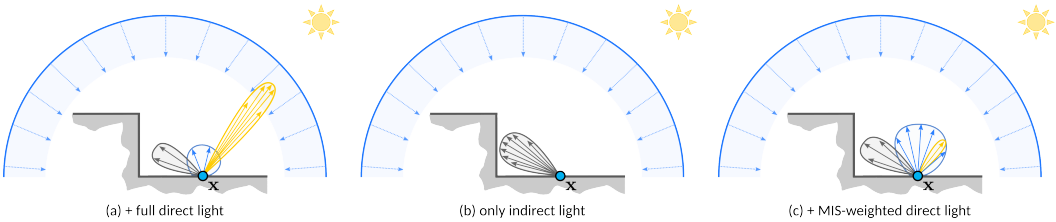


Fig. 6. Visualizing the different approaches to include direct light information into the guiding distribution and their resulting sampling behavior towards direct and indirect lighting. The scene setup utilizes an environment illuminated by a uniform sky and a sun, a configuration where the sun's contribution is typically well-sampled using next-event estimation (NEE). The left **(a)** shows a guiding distribution that includes the complete direct and indirect radiance information. Here, many samples are directed towards the sun, which are later down-weighted by MIS, resulting in low or even insignificant contributions. The center **(b)** shows a guiding distribution only containing indirect radiance information. Here, all guided samples are focused on exploring indirect illumination and rely entirely on next-event estimation to sufficiently collect direct light contributions. In cases where next-event estimation is not possible, for example, if NEE focuses all samples towards the sun but an object blocks it, this leads to a non-optimal sampling distribution. The right **(c)** shows a guiding distribution that includes the indirect and MIS-weighted direct radiance information. Here, the guided samples explore the indirect radiance and the direct radiance that NEE does not optimally handle.

Combining multiple estimators via multiple importance sampling (MIS) [Veach and Guibas 1995] is a commonly used technique to improve the efficiency of a path tracing-based renderer. The most commonly used combination is one that combines directly sampling light sources, also known as *next-event estimation* (NEE), with forward path tracing. In this setup the direct light contribution from a light source at a path vertex $\mathbf{x}_i$ is either evaluated by direct sampling a the contribution from a light source or by continuing the path in a randomly sampled direction where the next intersection x_{i+1} is on a light source. To ensure unbiasedness (i.e., avoid double contributions) and to reduce the variance of the resulting estimator, both contributions are then weighted via MIS weights. These weights increase the contribution from the technique with lower variance (i.e., higher PDF) and decrease the contribution from the method with the higher variance (i.e., lower PDF). Since a standard path tracer has no additional knowledge about the incoming radiance, it samples new directions mainly based on the local BSDF, making this approach extremely efficient. When using path guiding, we have access to additional knowledge about the incoming radiance (both direct and indirect), and we face the challenge of utilizing this knowledge wisely to integrate NEE efficiently. Unfortunately, there has been limited research on how to combine path guiding with NEE to ensure the resulting estimator is most efficient and does not result in inferior sampling quality compared to standard path tracing. In practice, there are three different approaches that are used throughout different path guiding research papers. The first approach fully includes the direct light contribution when building the guiding distribution:

$$g_{\tilde{\mathrm{L}}_\mathrm{i}}(\omega_\mathrm{i} \mid \mathbf{x}) \propto \tilde{L}_\mathrm{i}^{\mathrm{indirect}}(\mathbf{x}, \omega_\mathrm{i}) + \tilde{L}_\mathrm{i}^{\mathrm{direct}}(\mathbf{x}, \omega_\mathrm{i}). \tag{21}$$

Using a guiding distribution that fully includes direct illumination neglects the existence of NEE and the usage of MIS weights to potentially downweight the direct light contribution collected via path guiding. In regions with strong direct illumination (i.e., when the direct is much stronger than the indirect illumination), a majority of the paths are guided towards the light sources. If, at

the same time, the contribution of these light sources can be sampled efficiently using next-event estimation, the contribution of these guided paths is significantly downweighted via MIS. As a result, the contributions of these paths to the final estimate are negligible, and the computations used to evaluate them are more or less wasted. This can significantly decrease the efficiency of a guided path tracer, which can make it even less efficient than standard path tracing. An example of such a constellation is visualized in Fig. 6 on the left. It is worth noting at this point that many research works on path guiding use this setting, but at the same time, they disable next-event estimation during their evaluation to focus on the expressiveness and robustness of their models ([Dodik et al. 2022; Dong et al. 2023; Müller et al. 2017]). By doing so, they implicitly avoid this problem.

The second approach completely neglects the direct light contribution, building a guiding distribution that solely considers indirect light:

$$g_{\tilde{L}_i^{\text{indirect}}}(\omega_i \mid \mathbf{x}) \propto \tilde{L}_i^{\text{indirect}}(\mathbf{x}, \omega_i). \tag{22}$$

While this approach avoids the previous problem of over-exploring direct light contributions, which are then downweighted via MIS, it does so by completely neglecting all direct light contributions independent of their importance/contribution to the final estimate. As a result, the guided path tracer relies solely on the defensive BSDF-based sampling strategy (see Sec. 3) to collect important direct light contributions that are not sampled well by next-event estimation (e.g., if no advanced light sampling strategy is used or at shadow boundaries). This leads to inefficient sampling, especially if the direct light contribution outweighs the indirect light contribution, since most of the paths focus on exploring the indirect light contribution instead of the more important direct light contribution. An example of such constellation is visualized in Fig. 6 in the center.

A third approach is the heuristic proposed by Ruppert et al. [Ruppert et al. 2020] that includes the MIS-weighted direct light as well as the indirect direct light contribution:

$$g_{\tilde{L}_i^{\text{MIS}}}(\omega_i \mid \mathbf{x}) \propto \tilde{L}_i^{\text{indirect}}(\mathbf{x}, \omega_i) + w_{MIS}(\mathbf{x}, \omega_i) \cdot \tilde{L}_i^{\text{direct}}(\mathbf{x}, \omega_i), \tag{23}$$

where w is the MIS weight for sampling the direct light with forward path tracing. The intuition behind this heuristic is that path guiding should only explore the residual direct light that is contributed to the final estimate after MIS weighting. In cases where next-event estimation has a high chance of sampling the direct light contribution, the MIS weight will downweight the direct light component in the guiding distribution, while it will not downweight and, therefore, explore more direct light contributions that are not sampled well via next-event estimation. While this approach is based solely on intuition without any mathematical derivation or proof, we found that it works well in practice and at least does not lead to worse sampling or variance behaviors than standard unguided path tracing. A visualization of this approach is shown in Fig. 6 on the right. Deriving a proper solution that combines next-event estimation with guided directional sampling for direct illumination, based on a solid theoretical foundation, is an interesting and valuable task for future research. Such an approach has the potential to further enhance the usability and efficiency of path guidance in production. First attempts might explore applying some of the recently introduced MIS-compensation techniques ([Ivo et al. 2019; Karlík et al. 2019]) to this problem.

6 SCATTERING EVENT TYPES AND LIGHT PATH EXPRESSIONS

In production, scattering event types (SETs) and *light path expressions* (LPEs) are essential features of any rendering workflow. SETs describe the types of scattering events happening at every vertex of a path and their order. For example, a scattering event at a vertex can happen at a surface (S) or inside a volume (V), and the type of scattering event can be diffuse (D), glossy (G), or specular (S).

This path information is used for different purposes. Artists use the scattering type of the previous vertex to change the behavior of a material at the current path vertex (e.g., explicitly changing its roughness, making it diffuse, or replacing the complete material to achieve specific effects). Another important use case uses regular expressions on the list of scattering types of a path to separate different light transport types or classes contributing to the image. These regular expressions are called *light path expressions* (LPEs) and their contributions are usually stored in separate *arbitrary output variable* (AOV) buffers that can be post-processed differently before merging back together in the compositing stage.

The type of a scattering event at each vertex is usually defined by its BSDF or phase function. Complex BSDF models are built through a set of different mixed or layered BSDF closures C_i of various types. In this case, the scattering type of the model is defined by the scattering type of the closure used to sample the next path direction. Since the path guiding sampling process is not directly connected to the BSDF and its individual closures (Sec. 3), the scattering type for a sampled direction is undefined. As a result, we need to estimate the scattering type separately after sampling a new direction using path guiding. When estimating the scattering type, the statistics and characteristics of the SETs must match those generated with standard BSDF sampling. For example, when a path vertex has a BSDF composed of a glossy and diffuse closure, the ratio of glossy and diffuse scattering events needs to be the same for a given direction, independent of whether BSDF-based or guided sampling is used. Any discrepancy between these ratios will lead to inconsistency and bias, resulting in visually different rendering results when path guiding is turned on or off. Such an inconsistency would not be tolerable and would make path guiding impractical in production rendering.

In our previously mentioned example, the scattering event type at a path vertex and for a given direction depends on the location of $\mathbf{x}$ and the selected BSDF or phase function closure used to sample the new direction ω_i. The selection of these closures is a stochastic process, and therefore, so is the resulting scattering event type for a sampled direction. To estimate the type of a scattering event after guided sampling, we replicate the original process by first calculating the following probability for each BSDF closure C_i:

$$P_{C_i} = \frac{\alpha_i \cdot p_{f_\mathrm{s},\mathrm{C_i}}(\omega_\mathrm{i} \mid \mathbf{x}, \omega_\mathrm{o})}{\sum \alpha_j \cdot p_{f_\mathrm{s},\mathrm{C_j}}(\omega_\mathrm{i} \mid \mathbf{x}, \omega_\mathrm{o})}, \tag{24}$$

which depends on the sampled direction ω_i and the current scattering event position $\mathbf{x}$. This probability defines the likelihood that the direction ω_i was generated by sampling the i-th closure. Using these probabilities, we can estimate the scattering type for a sampled direction ω_i by first randomly selecting a BSDF closure proportional to its probability and returning the scattering type of the selected closure. If a renderer employs a different approach to identify the scattering event type at a path vertex interaction, our presented approach must be adapted to match the initial behavior of the renderer. Fig. 7 shows the effect of incorrectly labeled scattering event types in a scene where the artist made extensive use of manipulating materials based on the previous scattering event type. In both scenes, BarberShop and DarkInterior, the scattering event types of previous path vertices are used to define the direct visibility of virtual light sources. As we can see, incorrectly labeling scattering events (**left**) leads to a significant change in the scene's look, when compared to guiding with correct scattering event type labels (**center**) and the reference (**right**). This change in look is caused by changes in the light visibilities and, therefore, contributions of the scene's virtual light sources to a shading point (i.e., path vertex).

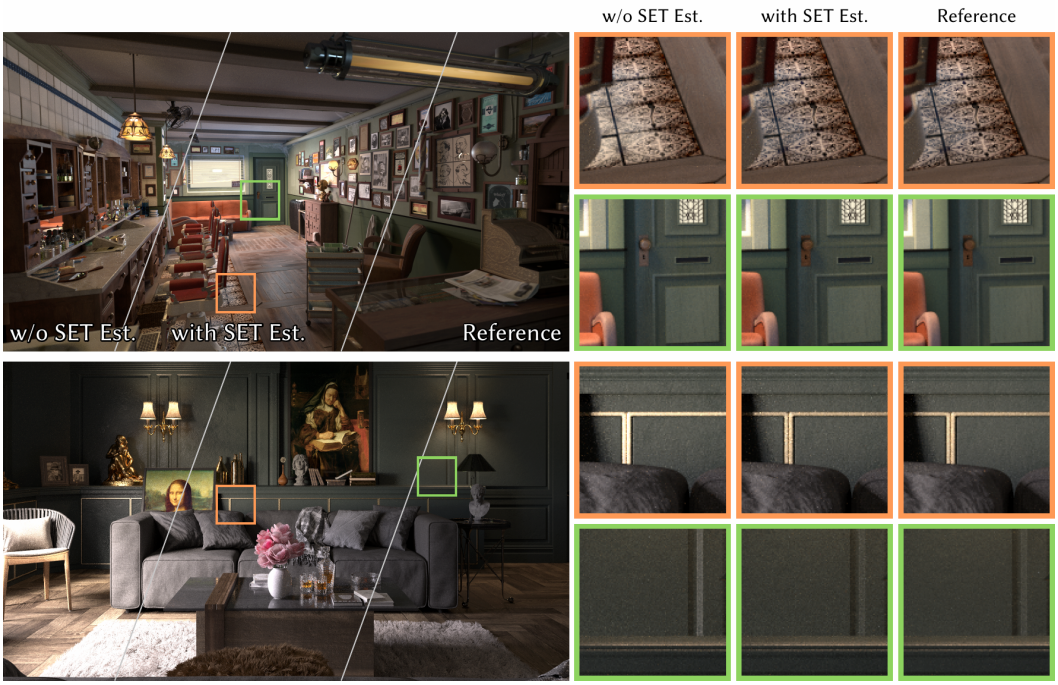


Fig. 7. Showing the effect of incorrect labeling of scattering event types (SET) for scattering events sampled using path guiding. The **left** demonstrates how incorrect labeling of the SETs can change the look of a scene. Our proposed SET estimation (Sec. 6) in the **center** leads to correct labeling of the SETs leading to a same look as the reference not using path guiding (**right**). Both scenes, BarberShop and DarkInterior, are rendered with equal sample counts (512spp) using Blender's Cycles rendering engine.

7 IMPROVING PATH GUIDING ROBUSTNESS VIA PATH SPACE REGULARIZATION

While local path guiding techniques can lead to significant efficiency improvements of a renderer, especially when rendering complex light transport scenarios, this efficiency gain mainly depends on the quality of the learned guiding structure (e.g., incident radiance field) (Sec. 2.1). The quality of the guiding structure depends not only on the expressiveness of the model used to represent the incident radiance field but also on the data used to train it. Online learning-based path guiding frameworks, such as OpenPGL, learn their radiance field progressively from training data generated during rendering. They, therefore, rely on standard *unguided* path tracing to initially discover challenging light transport effects, which are then learned and further explored in upcoming iterations. The ability to robustly learn complex light transport effects is, therefore, directly related to the ability of standard path tracing to initially discover or sample these effects. Some highly complex effects can be hard to discover by standard path tracing, making the occurrence of training samples representing these effects extremely rare. As a result, it can even take several rendering iterations for the effect to be discovered and learned. Examples of such effects are water caustics generated by a perfect dielectric water surface (i.e., $\alpha = 0$) and a physically correct simulated sun (i.e., radius of 0.5°) or multi-bounce effects through multiple perfect dielectric surfaces. In the worst case, an effect might not be discovered at all during the training phase of a frame, leading to temporally inconsistent rendering results, where the effect is discovered in one frame but not

in another. A way to counter this problem and assist the training process of the path guiding framework is the use of *path space regularization* or *roughening* techniques [Georgiev et al. 2018; Jendersie and Grosch 2019; Kaplanyan and Dachsbacher 2013; Weier et al. 2021]. The goal of these methods is to simplify the light transport complexity by increasing the roughness of the materials to increase the probability of sampling important light transport paths using standard path tracing.

In our experiments, we found that even a naïve/simplified version of the *optimized path space regularization* (OPSR) method by Weier et al. [Weier et al. 2021] can significantly improve the robustness of the forward learning approaches of path guiding frameworks when facing challenging light transport effects/constellations. Our simplified version is based on the *accumulated roughness* as described in Eq. 11 in their paper, considering only the roughness from the previous path vertex (i.e., using $k = 1$). Using this formula the updated roughness α' at the current path vertex $\mathbf{x}_i$ is defined as follows:

$$\alpha`(\mathbf{x}_k) = 1 - \Big(\big(1 - \alpha(\mathbf{x}_k)\big) \cdot \big(1 - \gamma \cdot \alpha`(\mathbf{x}_{k-1})\big)\Big), \tag{25}$$

where α defines the actual roughness at $\mathbf{x}_k$ and $\alpha`(\mathbf{x}_{k-1})$ the regularized roughness at the previous path vertex $\mathbf{x}_{k-1}$. Here, the parameter $\gamma = [0, 1]$ defines the strength of the regularization, where one means complete and zero means no regularization. In our experiments, we have found that a value of $\gamma = 0.01$ often already leads to a significant improvement in the robustness of the guiding structure's training, while having only a minimal visual effect in the rendering. One crucial part of OPSR is that the roughness from the previous path vertex is defined by the roughness of the closure used to sample the direction towards the current path vertex and not the average, minimum, or maximum roughness of the full BSDF at x_{k-1}. In cases where a BSDF closure is selected for sampling proportional to its contribution, the roughening effect of OPSR on objects will only be minimal when they are visible through specular closures (i.e., coatings), even if the BSDF at the previous path vertex has a diffuse component. Similar to the scattering event types, when using path guiding, the sampled roughness has to be estimated stochastically (Sec. 6). Fig. 8 shows examples of applying our naïve version of OPSR on a scene with challenging and non-challenging light transport and how it affects the quality/robustness of the trained guiding structure.

In the ShallowUnderWater scene (**top**), roughening significantly improves the convergence of the directly visible and reflected caustics. These caustics only lose a little bit of their sharpness (**right**) compared to when using a perfect dielectric water surface [7] (**left**).

The CountryKitchen scene (**bottom**)does not contain challenging caustic-like light transport. Here, roughening has almost no visible effect on the final look of the scene, and the robustness of the guiding caches remains similar, as they can already be trained robustly without using roughening (**left**).

8 OTHER INTEGRATION CHALLENGES

The previous sections explained the most essential nitty-gritty details we encountered when integrating path guiding using the OpenPGL framework in various production renderers. The following section will briefly introduce some additional challenges that need to be considered when integrating path guiding into a rendering system. The section finishes with a short list of advice that we consider to be useful for a developer or researcher when integrating path guiding into their rendering system.

[7] It is worth noting, at this point, that in reality, perfect specular (dielectric or conductor) surfaces do not exist.

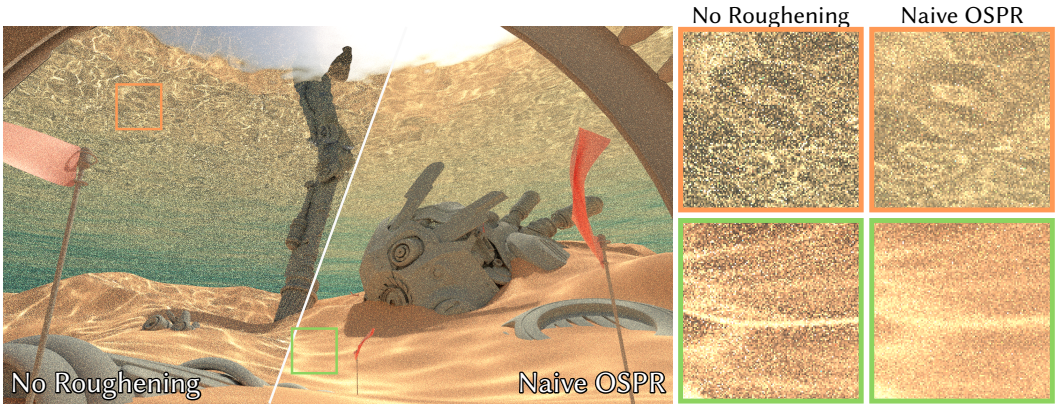


Fig. 8. Comparing equal sample count (64spp) rendering of the ShallowUnderWater and CountryKitchen scene using path guiding with and without our naïve implementation of OPSR-based roughening ($\gamma = 0.01$). In the ShallowUnderWater scene (**top**), even such a small roughening strength leads to a significant variance reduction while causing only a minimal visual change by slightly blurring the directly visible and reflected caustics. In the CountryKitchen scene, which does not contain complex specular light transport, this small roughening strength does not affect either the visual appearance or the variance of the rendering. Both scenes are rendered using our extended version of PBRT [2].

8.1 Training Data Generation

The effectiveness of a path guiding solution mainly depends on the quality and correctness of the guiding structure, which, as mentioned in Sec. 2.1 and Sec. 5, usually represents an approximation of the scene's light distribution (e.g., for the incoming and/or outgoing radiance). In practice, these guiding structures are learned from training data (i.e., radiance samples) generated by the renderer using information from the random walks generated during the path tracing process. The correctness of these radiance samples directly influences the correctness and quality of the learned radiance field approximation. It is, therefore, essential to pay special attention when generating and validating the training samples as well as the trained guiding structures. For example, when radiance samples are generated via Monte-Carlo path tracing, the training of the radiance distribution needs to consider that the training data are Monte-Carlo samples. Explicitly, this means that not only samples with non-zero radiance values but also zero-valued samples need to be considered during training to avoid structurally biased estimates [8]. In our experience, debugging and ensuring the correctness of the generated training data can take up to $70 - 80\,\%$ of the time when integrating an already existing path guiding framework, such as OpenPGL, into a renderer. It is worth noting at this point that even inaccurate training data can often lead to guiding distributions that improve rendering efficiency in complex light transport scenes (e.g., caustics or long indirect light transport), such as the ones used in many research papers. The adverse effects on rendering efficiency often become apparent only when testing the path guiding integration in more moderate day-to-day scenarios. A detailed description of how to inspect and validate the training data and the learned guiding structures can be found in Chap. 2 Sec. 4.2.

[8]The approximation will potentially be biased due to the limitation of the expressiveness of the used representation and spatial and directional marginalization.

8.2 Special Production Renderer Features

Other challenges when integrating path guiding into a production renderer include interactions with path guiding and special rendering features used in production to make a scene renderable by the given renderer within a given time and compute budget. These features often bend or break the physical correctness of the rendering, making it challenging to ensure that enabling path guiding leads to the same result as disabling it. When this is not the case, it is difficult to determine whether the problem is caused by a bug in the integration of path guiding, the path guiding framework itself, or the production rendering feature that breaks the physical correctness of the light transport simulation. Examples of such features are *radiance clamping*, *shadow path visibilities*, or *aggressive roughening*.

Radiance Clamping: Radiance clamping is a commonly used technique to reduce noise in the rendered image caused by high-intensity outlier samples. If not clamped, these outlier samples often require one or multiple orders of magnitude more samples than available in the given sample budget to reach an acceptable noise level. Clamping introduces bias since it removes energy from the image. The amount of introduced bias thereby depends on the renderer's ability to sample the challenging light transport, which causes the outlier sample. Since path guiding usually improves the sampling quality of challenging light transport paths, the bias introduced by radiance clamping also changes when path guiding is enabled. An example and discussion for the interaction of path guiding and radiance clamping is presented in Chap. 2 Sec. 5.3.2 (Example: Throughput Clamping).

Shadow Path Visibility: Shadow path or light visibility is a commonly used feature in production to enable or disable general shadow casting from an object or when evaluating shadow rays for next-event estimation. It is often used to allow connections during light sampling through glass windows (i.e., neglecting the refractions at the glass surface). When implementing these features, it is challenging to calculate the MIS weights correctly. If not done correctly, and the MIS-weights do not sum up to one, for example, due to inconsistencies in the directional sampling PDF, the look of the rendering changes as soon as the directional sampling PDF changes (e.g., when path guiding is used). It is, therefore, essential to validate the consistency and correctness of the MIS-weight calculation in scenarios where the shadow path visibility feature is used before integrating path guiding.

Aggressive Roughening: The positive effects of minimal or conservative roughening on path guiding were already discussed in Sec. 7. When using smaller regularization factors that do not significantly influence the overall look of the scene, the robustness and efficiency of path guiding often improve. In production, it is not uncommon to use more aggressive regularization parameters to simplify the scene's light transport, allowing for the rendering of the scene within a given time budget using standard path tracing. If such aggressive roughening approaches are used, path guiding might not lead to significant efficiency improvements. A more detailed discussion about aggressive roughening and path guiding, and why path guiding might still be beneficial in these cases, is presented in Chap. 2 Sec. 5.3.2 (Example: Path Simplification).

8.3 General Integration Advice

Integrating path guiding into a complex rendering system can be challenging. When the integration is not working as expected or does not directly show an efficiency improvement, it is often hard to judge if the problem is caused by bugs in the implementation of the path guiding framework/algorithm, its integration into the renderer, or if path guiding, in general, is not helpful in solving the given rendering challenges. To help developers during the integration process of a path guiding framework into their rendering system, we came up with a collection of advice.

A helpful attitude during integration is to focus on correctness and robustness first and consider efficiency and completeness later.

Russian Roulette: As pointed out in Sec. 4, using Russian Roulette can harm the efficiency of path guiding when the two techniques are not properly combined. It is, therefore, advisable to start the integration of path guiding with Russian Roulette disabled and only enable it when the path guiding integration is finished and works as expected.

Production Rendering Features: The challenges of combining path guiding with additional production rendering features such as radiance clamping, light visibility, or aggressive roughening were already mentioned in Sec. 8.2. Similar to Russian Roulette, it is advisable to start integrating path guiding without the use of these advanced production rendering features and to enable and combine them with path guiding one by one.

Equal Time vs. Equal Samples: When first testing the integration of a new rendering feature, one has two options: Comparing equal-time or equal-sample-count renderings. Using equal-time renderings directly reflects the efficiency gains or penalties when using path guiding. It is, therefore, the ideal evaluation method for testing the efficiency of the final path guiding integration into the rendering system. During the early stages of the integration process, on the other hand, focusing too much on equal-time evaluations can be misleading. As pointed out in the previous Secs. 3–7, path guiding can only unfold its full potential if all little nitty-gritty details and interactions with other techniques (e.g., Russian Roulette or next-event estimation) are considered correctly. In cases where not all of these aspects are already considered, equal-time comparisons may not show the expected efficiency improvements. It is, therefore, advisable to start with equal sample count evaluations first and include equal-time evaluations later if individual steps of the path guiding integration are completed and their correct functionality has been evaluated.

Tracking Additional Statistics: When evaluating a path guiding integration, it is possible that using path guiding can lead to a significant variance reduction in equal sample count renderings. Still, it may result in lower efficiency when using equal-time comparisons. Sometimes, the reason for the efficiency decrease is not obvious, for example, when the measured overhead of training the guiding structure or guided sampling is minimal. In this case, it makes sense to examine other per-path or path segment statistics, such as changes in the *average path length* or the ratio of *valid vs. invalid* directional samples and how they impact a renderer's efficiency.

Be Patient: Our last advice is to be patient and not give up too early. Implementing a path guiding framework or integrating one into a renderer can be a tedious task. Initial integrations often show promising results, e.g., when evaluating the integration on scenes with extremely challenging light transport (e.g., caustics or complex indirect illumination). However, when tested on more moderate scenes, the results may no longer seem as promising. As often, the devil lies in the details when it comes to path guiding; even minor bugs or inaccuracies in the implementation or integration can have significant effects on efficiency. Finding these bugs and inaccuracies is tedious and can take a lot of time and patience, but trust us: It is worth it.

9 CONCLUSION AND FUTURE WORK

In this part of the course, we presented a collection of nitty-gritty details that are important when integrating a path guiding framework into a production or research renderer. Most of our findings were not previously documented or openly discussed, but they have a significant impact on the effectiveness of path guiding. We hope that the provided integration details and insights

will further improve the adoption of path guiding in production rendering environments and help researchers enhance the robustness and quality of their path guiding integration. While our presented techniques demonstrate significant additional improvements when integrating path guiding into a production environment, we believe that there is still room for further improvements. We hope that further research and development will lead to a point where path guiding can be seen as an *always-on* solution, enabling artists to build more scenes with complex light transport effects without the need for hacks or ad hoc solutions to fake these effects. Ideally, this would enable artists to focus on optimizing the appearance of the hero shots and final artistic nuances rather than establishing or faking an overall physically plausible indirect lighting effect using virtual light rigs. We view this work as a first step in this direction and look forward to what future developments in path guiding for production rendering will bring.

An overview of open challenges of path guiding in production is presented in Chap. 2 Sec. 6. Caustics, for instance, remain challenging effects to simulate. While classical local path guiding techniques make it possible to resolve caustic effects using forward path tracing, they still may need a large number of samples to be resolved sufficiently. For these cases, specialized bidirectional guiding techniques that focus on solving caustics may be preferable in production. Such an approach is presented in the third chapter of this course (Chap. 3).

ACKNOWLEDGMENTS

We thank Christoph Peters, Lucas Alber, and Stefan Werner for reviewing this chapter of the course. Their valuable feedback and advice helped to improve this work. In addition, we thank the artists of the scenes used in this work: Jesús Sandoval for DarkInterior, Pablo Vazquez for the original Nishita Sky Demo scene used to generate ShallowUnderWater and DeepUnderWater, Jay-Artist for CountryKitchen, and Blender Studio for BarberShop.

REFERENCES

Attila T. Áfra. 2019. Intel® Open Image Denoise. https://www.openimagedenoise.org/.

James Arvo and David Kirk. 1990. Particle Transport and Image Synthesis. *ACM SIGGRAPH Computer Graphics (Proc. of SIGGRAPH).* 24, 4 (Sept. 1990), 63–66. doi:10.1145/97880.97886

Thomas Bashford-Rogers, Kurt Debattista, and Alan Chalmers. 2012. A significance cache for accelerating global illumination. *Computer Graphics. Forum* 31, 6 (Sept. 2012), 1837–1851. doi:10.1111/j.1467-8659.2012.02099.x

S. Chandrasekhar. 1960. *Radiative Transfer.* Dover Publications.

Hong Deng, Beibei Wang, Rui Wang, and Nicolas Holzschuch. 2020. A Practical Path Guiding Method for Participating Media. *Computational Visual Media* 6 (March 2020), 37–51. doi:10.1007/s41095-020-0160-1

Ana Dodik, Marios Papas, Cengiz Öztireli, and Thomas Müller. 2022. Path Guiding Using Spatio-Directional Mixture Models. *Computer Graphics Forum (Proc. of Eurographics)* 41, 1 (Feb. 2022), 172–189. doi:10.1111/cgf.14428

Honghao Dong, Guoping Wang, and Sheng Li. 2023. Neural Parametric Mixtures for Path Guiding. In *SIGGRAPH '23 Conference Proceedings* (Los Angeles, CA, USA). 10 pages. doi:10.1145/3588432.3591533

Luca Fascione, Johannes Hanika, Daniel Heckenberg, Christopher Kulla, Marc Droske, Jorge Schwarzhaupt, Wenzel Jakob, Andrea Weidlich, Rob Pieké, and Hanzhi Tang. 2019. Path Tracing in Production. In *ACM SIGGRAPH 2019 Courses.* Article 19. doi:10.1145/3305366.3328079

Luca Fascione, Johannes Hanika, Rob Pieké, Ryusuke Villemin, Christophe Hery, Manuel Gamito, Luke Emrose, and André Mazzone. 2018. Path Tracing in Production. In *ACM SIGGRAPH 2018 Courses.* Article 15. doi:10.1145/3214834.3214864

Iliyan Georgiev, Thiago Ize, Mike Farnsworth, Ramón Montoya-Vozmediano, Alan King, Brecht Van Lommel, Angel Jimenez, Oscar Anson, Shinji Ogaki, Eric Johnston, Adrien Herubel, Declan Russell, Frédéric Servant, and Marcos Fajardo. 2018. Arnold: A Brute-Force Production Path Tracer. *ACM Transactions on Graphics* 37, 3, Article 32 (Aug. 2018). doi:10.1145/3182160

Jerry Guo, Pablo Bauszat, Jacco Bikker, and Elmar Eisemann. 2018a. Primary Sample Space Path Guiding. In *Eurographics Symposium on Rendering - EI & I (EGSR '18).* 73–82. doi:10.2312/sre.20181174

Yu Guo, Miloš Hašan, and Shuang Zhao. 2018b. Position-free monte carlo simulation for arbitrary layered BSDFs. *ACM Transactions on Graphics (Proc. of SIGGRAPH Asia)* 37, 6, Article 279 (Dec. 2018). doi:10.1145/3272127.3275053

Sebastian Herholz and Addis Dittebrandt. 2022. Intel® Open Path Guiding Library. http://www.openpgl.org.

Sebastian Herholz, Oskar Elek, Jiří Vorba, Hendrik Lensch, and Jaroslav Křivánek. 2016. Product importance sampling for light transport path guiding. *Computer Graphics Forum (Proc. of Eurographics Symposium on Graphics)* 35, 4 (June 2016), 67–77. doi:10.1111/cgf.12950

Sebastian Herholz, Yangyang Zhao, Oskar Elek, Derek Nowrouzezahrai, Hendrik P. A. Lensch, and Jaroslav Křivánek. 2019. Volume Path Guiding Based on Zero-Variance Random Walk Theory. *ACM Transactions on Graphics (Proc. of SIGGRAPH)* 38, 3, Article 25 (June 2019). doi:10.1145/3230635

Tim Hesterberg. 1995. Weighted Average Importance Sampling and Defensive Mixture Distributions. *Technometrics* 37, 2 (May 1995), 185–194. doi:10.1080/00401706.1995.10484303

Heinrich Hey and Werner Purgathofer. 2002. Importance sampling with hemispherical particle footprints. In *Proc. of SCCG*. 107–114. doi:10.1145/584458.584476

J. Eduard Hoogenboom. 2008. Zero-variance Monte Carlo schemes revisited. *Nuclear Science and Engineering* 160, 1 (Sept. 2008), 1–22. doi:10.13182/NSE160-01

Kondapaneni Ivo, Petr Vévoda, Pascal Grittmann, Tomáš Skřivan, Philipp Slusallek, and Jaroslav Křivánek. 2019. Optimal Multiple Importance Sampling. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2019)* 38, 4 (July 2019), 37:1–37:14. doi:10.1145/3306346.3323009

Wenzel Jakob. 2010. Mitsuba renderer. http://www.mitsuba-renderer.org.

Johannes Jendersie and Thorsten Grosch. 2019. Microfacet Model Regularization for Robust Light Transport. *Computer Graphics Forum (Proc. of Eurographics Symposium of Rendering)* 38, 4 (July 2019), 39–47. doi:10.1111/cgf.13768

Henrik Wann Jensen. 1995. Importance driven path tracing using the photon map. In *Rendering Techniques (Proc. of Eurographics Workshop on Rendering)*. doi:10.1007/978-3-7091-9430-0_31

James T. Kajiya. 1986. The Rendering Equation. *ACM SIGGRAPH Computer Graphics (Proc. of SIGGRAPH)*. 20, 4 (Aug. 1986), 143–150. doi:10.1145/15886.15902

James T. Kajiya and Brian P Von Herzen. 1984. Ray tracing volume densities. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '84)*. Association for Computing Machinery, New York, NY, USA, 165–174. doi:10.1145/800031.808594

Anton S. Kaplanyan and Carsten Dachsbacher. 2013. Path Space Regularization for Holistic and Robust Light Transport. *Computer Graphics Forum (Proc. of Eurographics)* 32, 2 (May 2013), 63–72. doi:10.1111/cgf.12026

Ondřej Karlík, Martin Šik, Petr Vévoda, Tomáš Skřivan, and Jaroslav Křivánek. 2019. MIS compensation: optimizing sampling techniques in multiple importance sampling. *ACM Trans. Graph.* 38, 6, Article 151 (Nov. 2019), 12 pages. doi:10.1145/3355089.3356565

Eric P. Lafortune and Yves D. Willems. 1995. A 5D tree to reduce the variance of Monte Carlo ray tracing. In *Rendering Techniques (Proc. of Eurographics Workshop on Rendering)*. 11–20. doi:10.1007/978-3-7091-9430-0_2

Thomas Müller. 2019. "Practical Path Guiding" in Production. In *ACM SIGGRAPH 2019 Courses: Path Guiding in Production*. Article 18, 18:35–18:48 pages. doi:10.1145/3305366.3328091

Thomas Müller, Markus Gross, and Jan Novák. 2017. Practical path guiding for efficient light-transport simulation. *Computer Graphics Forum (Proc. of Eurographics Symposium on Rendering)* 36, 4 (June 2017), 91–100. doi:10.1111/cgf.13227

Matt Pharr, Wenzel Jakob, and Greg Humphreys. 2016. *Physically based rendering: From theory to implementation* (3rd ed.). Morgan Kaufmann.

Alexander Rath, Pascal Grittmann, Sebastian Herholz, Petr Vévoda, Philipp Slusallek, and Jaroslav Křivánek. 2020. Variance-Aware Path Guiding. *ACM Transactions on Graphics (Proc. of SIGGRAPH)* 39, 4, Article 151 (July 2020). doi:10.1145/3386569.3392441

Alexander Rath, Pascal Grittmann, Sebastian Herholz, Philippe Weier, and Philipp Slusallek. 2022. EARS: Efficiency-Aware Russian Roulette and Splitting. *ACM Transactions on Graphics (Proc. of SIGGRAPH)* 41, 4, Article 81 (July 2022). doi:10.1145/3528223.3530168

Lukas Ruppert, Sebastian Herholz, and Hendrik P. A. Lensch. 2020. Robust Fitting of Parallax-Aware Mixtures for Path Guiding. *ACM Transactions on Graphics (Proc. of SIGGRAPH)* 39, 4, Article 147 (Aug. 2020). doi:10.1145/3386569.3392421

Vincent Schüßler, Johannes Hanika, Alisa Jung, and Carsten Dachsbacher. 2022. Path Guiding with Vertex Triplet Distributions. *Computer Graphics Forum (Proc. of Eurographics Symposium on Rendering)* 41, 4 (July 2022), 1–15. doi:10.1111/cgf.14582

Florian Simon, Alisa Jung, Johannes Hanika, and Carsten Dachsbacher. 2018. Selective guided sampling with complete light transport paths. *ACM Transactions on Graphics (Proc. of SIGGRAPH Asia)* 37, 6, Article 223 (Dec. 2018). doi:10.1145/3272127.3275030

Justin F. Talbot, David Cline, and Parris Egbert. 2005. Importance resampling for global illumination. In *Proc. of Eurographics Symposium on Rendering (EGSR '05)*. 139–46. doi:10.2312/EGWR/EGSR05/139-146

Eric Veach. 1998. *Robust Monte Carlo methods for light transport simulation.* Ph. D. Dissertation. Stanford University. Advisor(s) Guibas, Leonidas J.

Eric Veach and Leonidas J. Guibas. 1995. Optimally Combining Sampling Techniques for Monte Carlo Rendering. In *Proc. of SIGGRAPH (SIGGRAPH '95)*. 419–428. doi:10/d7b6n4

Jiří Vorba, Johannes Hanika, Sebastian Herholz, Thomas Müller, Jaroslav Křivánek, and Alexander Keller. 2019. Path Guiding in Production. In *ACM SIGGRAPH 2019 Courses*. Article 18. doi:10.1145/3305366.3328091

Jiří Vorba, Ondřej Karlík, Martin Šik, Tobias Ritschel, and Jaroslav Křivánek. 2014. On-line learning of parametric mixture models for light transport simulation. *ACM Transactions on Graphics (Proc. of SIGGRAPH)* 33, 4, Article 101 (July 2014). doi:10.1145/2601097.2601203

Jiří Vorba and Jaroslav Křivánek. 2016. Adjoint-driven Russian roulette and splitting in light transport simulation. *ACM Transactions on Graphics (Proc. of SIGGRAPH)* 35, 4, Article 42 (July 2016). doi:10.1145/2897824.2925912

Philippe Weier, Marc Droske, Johannes Hanika, Andrea Weidlich, and Jiří Vorba. 2021. Optimised Path Space Regularisation. *Computer Graphics Forum (Proc. of Eurographics Symposium on Rendering)* 40, 4 (July 2021), 139–151. doi:10.1111/cgf.14347

Kehan Xu, Sebastian Herholz, Marco Manzi, Marios Papas, and Markus Gross. 2024. Volume Scattering Probability Guiding. *ACM Transactions on Graphics (Proc. of SIGGRAPH Asia)* 43, 6, Article 184 (Nov. 2024). doi:10.1145/3687982

Quan Zheng and Matthias Zwicker. 2019. Learning to Importance Sample in Primary Sample Space. *Computer Graphics Forum (Proc. of Eurographics)* 38, 2 (May 2019). doi:10.1111/cgf.13628

Path Guiding Surfaces and Volumes in Disney's Hyperion Renderer: A Case Study

LEA REICHARDT, Walt Disney Animation Studios, Canada
BRIAN GREEN, Walt Disney Animation Studios, USA
YINING KARL LI, Walt Disney Animation Studios, USA
MARCO MANZI, DisneyResearch|Studios, Switzerland



Fig. 1. A production scene from *Moana 2*, rendered using path guiding in Disney's Hyperion Renderer. From left to right: reference baseline, 64 SPP without path guiding, 64 SPP with path guiding, and visualization of the path guiding spatio-directional field at 256 SPP. © 2025 Disney

We present our approach to implementing a second-generation path guiding system in Disney's Hyperion Renderer, which draws upon many lessons learned from our earlier first-generation path guiding system. We start by focusing on the technical challenges associated with implementing path guiding in a wavefront style path tracer and present our novel solutions to these challenges. We will then present some powerful visualization and debugging tools that we developed along the way to both help us validate our implementation's correctness and help us gain deeper insight into how path guiding performs in a complex production setting. Deploying path guiding in a complex production setting raises various interesting challenges that are not present in purely academic settings; we will explore what we learned from solving many of these challenges. Finally, we will look at some concrete production test results and discuss how these results inform our large scale deployment of path guiding in production. By providing a comprehensive review of what it took for us to achieve this deployment on a large scale in our production environment, we hope that we can provide useful lessons and inspiration for anyone else looking to similarly deploy path guiding in production, and also provide motivation for interesting future research directions.

1 INTRODUCTION

In these course notes, we present a case study of what it took to build Hyperion's modern path guiding system and what we have learned along the way. We first implemented a form of path guiding in Disney's Hyperion Renderer during the production of Disney's *Frozen 2*; this initial implementation was based on Practical Path Guiding (PPG) [Müller et al. 2017], with various

improvements and extensions [Müller 2019] discovered along the way. This first version of path guiding in Hyperion only saw limited production usage, but nonetheless provided many of the lessons presented here; these lessons proved to be invaluable in shaping our modern path guiding system.

Our modern path guiding system is built upon Intel's Open Path Guiding Library (OpenPGL) [Herholz and Dittebrandt 2022], which in addition to PPG implements other techniques such as those proposed by Herholz et al. [Herholz et al. 2019] and Ruppert et al. [Ruppert et al. 2020]. However, nothing we present in these course notes is specific to OpenPGL; we believe everything here is generally applicable and could be of interest to anyone looking to implement any modern path guiding technique in a complex production renderer. Collectively, the work presented here has allowed us to reach the point where we are now deploying our modern path guiding system on a large scale on Disney Animation projects currently in production.

We will begin in Section 2 with a discussion of the fundamental challenges that Hyperion's wavefront architecture presents towards implementing an effective path guiding system; these challenges directly motivated the Radiance Recorder data structure we describe in Section 3, which underpins our entire modern path guiding system and our OpenPGL integration. In addition to architectural challenges, debugging path tracing in a complex production setting is another major challenge, for which we have developed new tools that we present in Section 4. We then present how everything comes together in deploying path guiding in production in Section 5. Section 5 has four major subsections. Section 5.1 discusses how the Radiance Recorder is actually used in practice, and Section 5.2 discusses how we solve the scheduling issues that arise from Section 2. Section 5.3 discusses the various practical challenges that we encountered in deploying path guiding in a complex production environment; we found that these challenges do not diminish the intrinsic value of path guiding, but do require considerable care to navigate. In Section 5.4, we provide deeper dives into several specific interesting production results. Finally, in Section 6, we give a preview of where we are taking our modern path guiding system next.

2 THE CHALLENGES OF PATH GUIDING IN A WAVEFRONT PRODUCTION RENDERER

Disney's Hyperion Renderer [Burley et al. 2018] uses a sorted deferred shading architecture [Eisenacher et al. 2013], which fits under the broader definition of a wavefront style path tracing architecture. Typically, path tracing is implemented in a *depth-first* manner, where each processor core traces one complete path in a depth-first manner from the camera through the scene to either a light source or the path's termination before starting on another path. Conversely, wavefront path tracing architectures execute path tracing in a *breadth-first* manner, where a large number of rays are traced in parallel against the scene for a single bounce, results written to memory, then all of the shading points processed in parallel for a single bounce, results written to memory, and repeat until all paths are completed, with optional sorting and potentially other processing happening between each step.

Wavefront, or breadth-first, path tracing architectures have gained traction in both CPU and GPU-based production renderers for reasons primarily centered around extracting both better memory access coherency and better execution coherency than depth-first architectures can achieve. However, wavefront path tracing presents several major challenges to implementing more advanced path tracing techniques, such as path guiding; in this section, we'll briefly recap these challenges.

2.1 Incompatibilities between Wavefront Architectures and Path Guiding

Wavefront architectures generally only fit well with purely *feed-forward* path tracing implementations. In this context, feed-forward means that at each path vertex, information can only flow to future path vertices further down the path and can never flow backwards to earlier path vertices because earlier path vertices are discarded as each path progresses. Feed-forward is typically a requirement because wavefront path tracing requires keeping very large numbers of paths in flight simultaneously in order to offer enough work to extract coherency from, and all of these paths must have their path state stored in memory between sets of traversal and shading operations. All of these path states can consume a large quantity of storage space, making a feed-forward approach necessary in order to allow for freeing earlier path vertices from memory.

Modern path guiding techniques [Herholz and Dittebrandt 2022; Herholz et al. 2019; Müller 2019; Müller et al. 2017; Rath et al. 2020; Ruppert et al. 2020] work by learning an approximation of the scene's radiance field, from which an approximation of the incident radiance distribution is queried to improve importance sampling at each surface shading point or volumetric scattering location. Commonly, this scene radiance field is learned on the fly from paths that the renderer constructs and is continuously updated and refined throughout the course of the render. When a path is completed, the renderer can calculate all radiance flowing back along the path from connected light sources back to the camera, and can then feed the path guiding system an estimate of total incident radiance at each vertex on the path. However, in a feed-forward wavefront path tracer, the renderer does not have a complete path history at any given point in the rendering process, since earlier path vertices are discarded as the path progresses. Without complete path history, there actually is no way to correctly report a single incident radiance value per path vertex to a path guiding system; we explain in detail why this is the case in Section 3.1 and provide a breakdown of a common production example in Section 3.2.

Yalçıner and Akyüz [Yalçıner and Akyüz 2024] present an interesting implementation of path guiding in a wavefront system using radiant exitance instead of incident radiance. Since radiant exitance is a directionless quantity, using radiant exitance allows for guiding using a simpler sparse voxel octree (SVO) [Crassin et al. 2009; Laine and Karras 2010] based spatial data structure for storage which is populated by partitioning rays spatially and having rays collaboratively generate an estimated radiance field per partition. However, this approach requires extensive modifications to both the renderer's architecture and to the guiding techniques being used, performs worse than the methods from both Müller et al. [Müller et al. 2017] and Ruppert et al. [Ruppert et al. 2020] in many cases, and has difficulty adapting to highly complex production scenes due to the SVO's fixed resolution. Furthermore, this approach still requires recording full path histories.

Unlike Yalçıner and Akyüz [Yalçıner and Akyüz 2024], we wanted to keep the same underlying spatio-directional guiding data structure that most guiding approaches rely on and we did not want to significantly rework the renderer's ray scheduling architecture. But similar to their work, we also found that the most straightforward path to solve the feed-forward problem is to simply loosen the feed-forward requirement and store enough path history along with the current ray in each path to allow for reconstructing a single incident radiance value per path vertex. However, actually implementing this approach in practice is difficult because the underlying memory usage requirements that drive the need for a feed-forward system still exist; determining what to store ends up being a careful balancing act between maximizing the amount of information that is useful to path guiding while still minimizing the total quantity of memory required. Solving this problem is one of the key contributions of the work presented in these course notes.

2.2 Hyperion Architecture Review

To provide the relevant context for the rest of these course notes, we first provide a high-level summary of Hyperion's architecture, which goes as follows:

- Rays are generated from the camera and placed into batches organized by primary cardinal direction.
- In the past, non-active batches were potentially written to disk and offloaded from main memory using mmap, but today this is no longer the case as all batches are kept in memory. The most recently filled batch is always the next waiting batch.
- The next waiting full batch is activated. Rays within the batch are sorted by origin, time, and direction, and then traversed through the scene's BVH. Traversal is parallelized over rays, with each thread tracing a single ray at a time.
- Hit points are sorted by scene object and mesh face index, and then shading is carried out parallelized by scene object. Shaders spawn new secondary rays and light sample rays, both of which are queued into batches.
- Each path segment or bounce in Hyperion is called a *wave*; a single batch can contain millions of rays from multiple waves, with the specific number being an empirically determined "sweet spot".

All of the steps above are carried out in a series of iterations, with an iteration representing some number of samples per pixel (SPP). Various data structure updates happen between iterations; we discuss this with more detail in Section 5.2.1. Hyperion's light transport is a purely feed-forward, unidirectional path tracer with next-event estimation (NEE). This feed-forward approach means that operations such as next event estimation must be implemented in such a way that does not require spawned rays to immediately return an answer, since all rays are queued before being traced.

Modern Hyperion has several subsystems where locality is intrinsically expected and the computational cost of deferring operations would outweigh the benefit; these subsystems are implemented to bypass the wavefront architecture. Examples include subsurface scattering, volume rendering, ambient occlusion in shaders, and a few others. Hyperion relies on path traced subsurface scattering [Chiang et al. 2016], and for volumes, Hyperion uses a null-scattering theory based volumetric path tracing approach [Huang et al. 2021; Kutz et al. 2017] that is optimized for efficiently calculating high-order volumetric scattering. In both of these systems, potentially hundreds or thousands of bounces can occur, but the actual shading calculations that have to happen per bounce to determine the next scattering decision are essentially non-programmable by artists and are vastly simpler and therefore faster to execute than standard surface shading, which in turn changes the execution behavior enough to give a depth-first approach the advantage. However, when paths exit subsurface scattering or volumes and encounter a surface again, paths are then placed back into the sorted deferred batching system.

2.3 Previous Version of Path Guiding in Hyperion

Our first implementation of path guiding in Hyperion using PPG worked around the feed-forward problem by simply storing a fixed number of path vertices with the current ray state. By default, we set the number of stored path vertices to four, which was chosen to keep total memory usage for storing ray states bounded. Hyperion's ray states require upfront fixed memory allocations and the system cannot dynamically allocate more memory per ray once paths are started; therefore, in order to prevent non path guiding renders from needing to allocate memory for storing additional path vertices per ray state, we created separate path guiding and non path guiding ray types

and templatized the entire renderer on these different ray types, which introduced considerable additional implementation complexity into the renderer.

Over time, we found that our first implementation of path guiding did not gain significant traction in production due to several major reasons:

- Path guiding improved convergence speed in various difficult cases, sometimes by significant amounts. But in other cases, such as scenes dominated by direct illumination, path guiding also often increased total render time without providing significant improvements in noise when compared with standard unidirectional path tracing at the same SPP count. As a result, artists had a difficult time predicting whether or not path guiding would be worthwhile on a given scene. A complicating factor is that path guiding's performance is best characterized as improvements in *convergence rate*, but our artists are used to thinking about renderer performance in terms of raw speed to complete a render to a target SPP count but not necessarily in terms of convergence rate.
- For particularly complex scenarios, limiting path guiding training to four path vertices degraded results for deep paths with many bounces, making path guiding less attractive for even the cases where it helps the most.
- While in academic renderers path guiding typically shows significant advantages in convergence rate over unguided unidirectional path tracing, production renderers typically contain various optimizations that narrow the gap between a guided and unguided result; we discuss many of these in detail in Section 5.3. We found that understanding how these production optimizations and factors would contribute to the overall "value" of enabling path guiding was typically too much to ask of lighting artists already under time pressure to deliver shots.
- Our initial path guiding implementation only supported guiding for surfaces and not for volumes. As our films have become more and more reliant on extensive volumes in almost every shot, ranging from complex effects to simple atmospherics, the lack of support for guiding in volumes further decreased the usefulness of the system. Because volumetric scattering typically requires many more bounces than surface-to-surface scattering, even if we had implemented support for guiding in volumes in our initial implementation, the problems caused by low limits on available path vertices for training would have been further exacerbated.

As the complexity of our studio's films continues to increase, we set forth to build a second-generation path guiding system designed to apply the lessons we learned from the first version, with the goal of arriving at a system that could see widespread deployment and provide significant speedups for production. From the outset we understood that the ability to guide paths inside of volumes would be crucial to a viable system, which led us to build on top of OpenPGL's extensive existing volume guiding capabilities. We knew that we needed to be able to train guiding data structures using much deeper paths, but also wanted to simplify how path vertices are stored and remove the need for multiple ray types in the renderer, which led directly to the creation of the Radiance Recorder data structure described in Section 3. Even with the Radiance Recorder, integrating OpenPGL into Hyperion's existing architecture still required significant care; we describe these steps in Sections 5.1 and 5.2. In order for our development team to provide better recommendations to artists on when to use path guiding, we had to better understand ourselves how guiding performs on a variety of production scenes — both simple and complex — and why guiding performs the way it does; to this end, we developed better visualization and debugging tools for path guiding, which we describe in Section 4. These debugging tools allowed us to then gain many of the insights that we share in Sections 5.3 and 5.4.

3 RECORDING RADIANCE IN A WAVEFRONT RENDERER

In this section, we use two specific examples to describe how incident radiance is computed from Hyperion's per-wave shader results. These examples demonstrate the central requirement that incident radiance at a path vertex is only knowable once the entire path containing that path vertex completes. We go on to show that this requirement is nearly trivial to satisfy in a depth-first path tracer. However, under Hyperion's wavefront architecture, a secondary structure is needed. We call this structure the Radiance Recorder. The details of this structure are presented in the final part of this section.

3.1 A Simple Example

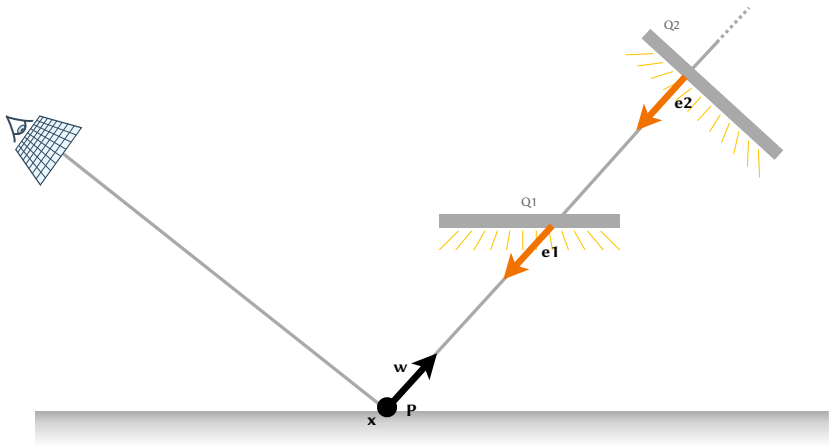


Fig. 2. Simple direct lighting setup. The goal is to compute the incident radiance at path vertex **P** at location **x** in direction **w** due to two virtual lights Q1 and Q2. **e1** and **e2** are the emission values at the light intersection points in direction -**w**.

Figure 2 is a simple case. We have two virtual light sources Q1 and Q2. Common in production, a virtual light source is a surface that emits light, but does not occlude light. They are also typically sampleable via light sampling techniques when performing next event estimation (NEE). Let **P** be the path vertex at **x** in direction **w**. A path vertex is fundamentally defined by its position and direction, but it can store pretty much any arbitrary data that we want. We'll adopt the notation $\mathbf{P}_{\mathbf{quantity}}$ to denote any quantity stored at path vertex **P**. See Table 1 for a symbol overview. For purposes of this discussion, $\mathbf{P}_{\mathbf{tp}}$ will denote the path throughput at **P**, $\mathbf{P}_{\mathbf{x}}$ the position, $\mathbf{P}_{\mathbf{w}}$ the direction, $\mathbf{P}_{\mathbf{ray}}$ denotes the ray at position $\mathbf{P}_{\mathbf{x}}$ in direction $\mathbf{P}_{\mathbf{w}}$, $\mathbf{P}_{\mathbf{em}}$ the accumulated shader emission values from light sources, and $\mathbf{P}_{\mathbf{rad}}$ the incident radiance at **P**. The task is to compute $\mathbf{P}_{\mathbf{rad}}$ and record this single result as part of our training data.

In Hyperion, a light shader produces emission values. The emission values accumulated by a path vertex relate to the incident radiance according to the formula

$$\mathbf{P}_{\mathbf{em}} = \mathbf{P}_{\mathbf{rad}} \times \mathbf{P}_{\mathbf{tp}}$$

Thus, to compute the incident radiance at **P** we take the incoming emission at **P** and divide by the path throughput at **P**. The emission value is convenient in a wavefront architecture because it can

Symbol	Description
$\mathbf{x}$	a position
$\mathbf{w}$	a direction
$\mathbf{P}$	a path vertex (has a position $\mathbf{x}$ and direction $\mathbf{w}$)
$\mathbf{P_x}$	position of $\mathbf{P}$
$\mathbf{P_w}$	direction of $\mathbf{P}$
$\mathbf{P_{ray}}$	ray at position $\mathbf{P_x}$ in direction $\mathbf{P_w}$
$\mathbf{P_{tp}}$	throughput at $\mathbf{P}$
$\mathbf{P_{em}}$	accumulated light emission at $\mathbf{P}$
$\mathbf{P_{rad}}$	incident radiance at $\mathbf{P}$

Table 1. Symbol overview. We generally use the notation $\mathbf{P_{quantity}}$ to denote any quantity stored at path vertex $\mathbf{P}$.

immediately be splatted to the framebuffer results. It is unfortunately required that we divide the emission result by the path throughput to capture the incident radiance value needed for training path guiding.
Breaking down the path in Figure 2 into waves:

- Wave 0: camera ray hits at $\mathbf{P_x}$
- Wave 1: $\mathbf{P_{ray}}$ hits Q1 producing emission $\mathbf{e1}$
- Wave 2: $\mathbf{P_{ray}}$ hits Q2 producing emission $\mathbf{e2}$
- Wave 3: $\mathbf{P_{ray}}$ hits the environment (0 emission in this example)

For training, it is tempting to compute and record training values as

- Wave 0: camera ray hits at $\mathbf{P_x}$: record nothing
- Wave 1: $\mathbf{P_{ray}}$ hits Q1 producing emission $\mathbf{e1}$: record $\mathbf{P_{rad}} = \frac{\mathbf{e1}}{\mathbf{P_{tp}}}$
- Wave 2: $\mathbf{P_{ray}}$ hits Q2 producing emission $\mathbf{e2}$: record $\mathbf{P_{rad}} = \frac{\mathbf{e2}}{\mathbf{P_{tp}}}$
- Wave 3: $\mathbf{P_{ray}}$ hits environment (0 emission in this example): record nothing

But this is wrong! For a single path vertex, we should record a single incident radiance value. With the above algorithm we are producing two training samples at $\mathbf{P}$, both of which are likely too small. The correct, single training value to record for this path is

$$\mathbf{P_{rad}} = \frac{\mathbf{e1} + \mathbf{e2}}{\mathbf{P_{tp}}}$$

This means we really can't record any value for $\mathbf{P}$ until the path completes. Our algorithm needs to look like:

- Wave 0: camera ray hits at $\mathbf{P_x}$: record nothing
- Wave 1: $\mathbf{P_{ray}}$ hits Q1 producing emission $\mathbf{e1}$: record nothing, but "remember" $\mathbf{e1}$
- Wave 2: $\mathbf{P_{ray}}$ hits Q2 producing emission $\mathbf{e2}$: record nothing, but "remember" $\mathbf{e2}$
- Wave 3: $\mathbf{P_{ray}}$ hits environment (0 emission in this example): record $\mathbf{P_{rad}} = \frac{\mathbf{e1+e2}}{\mathbf{P_{tp}}}$

3.2 A More Complex Example

Figure 3 is a more complex example. It includes three virtual light sources (Q1, Q2, Q3), environment light emission (e2, e4, e5, e8) and six path vertices (P0, P1, P2, P3, P4, P5)

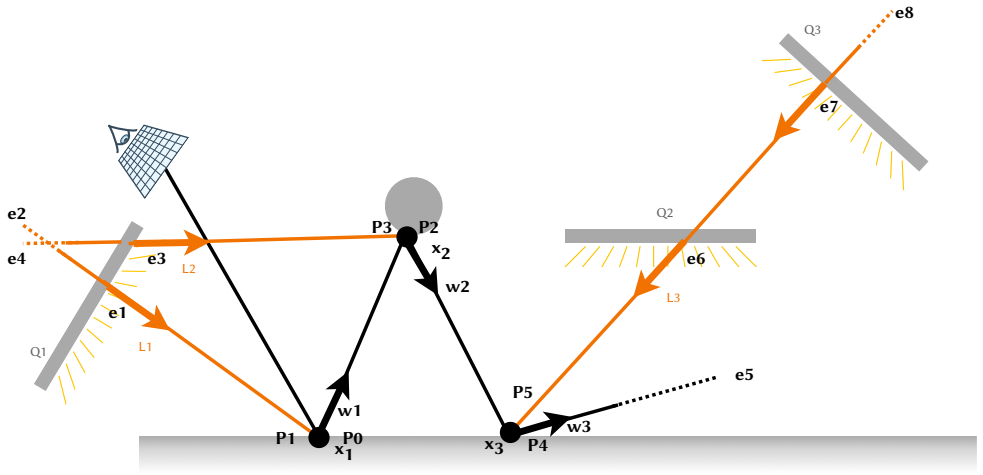


Fig. 3. A more complex lighting setup with direct and indirect illumination. We show a single path that includes both NEE rays (orange) and path continuation rays (black). The goal is to compute the full incident radiance at each path vertex **P0** - **P5**.

Because we have six path vertices, we expect to record six values. It is also worth noting that half of our path vertices correspond to NEE rays (shown in orange). This is a distinction we record, as it can be useful for training purposes.
Applying our algorithm, which is visualized in Figure 4:

- Wave 0: camera ray hits at **x1**: shade : create **P0** and **P1**
- Wave 1: $\mathbf{P0}_{\text{ray}}$ hits at **x2**: shade : create **P2** and **P3**
  $\mathbf{P1}_{\text{ray}}$ hits Q1 : remember **e1** : continue **P1**
- Wave 2: $\mathbf{P1}_{\text{ray}}$ hits environment: evaluates **e2** : record $\mathbf{P1}_{\text{rad}} = \frac{\mathbf{e1}+\mathbf{e2}}{\mathbf{P1}_{\text{tp}}}$
  $\mathbf{P2}_{\text{ray}}$ hits at **x3**: shade : create **P4** and **P5**
  $\mathbf{P3}_{\text{ray}}$ hits Q1 : remember **e3** : continue **P3**
- Wave 3: $\mathbf{P3}_{\text{ray}}$ hits environment: evaluate **e4** : record $\mathbf{P3}_{\text{rad}} = \frac{\mathbf{e3}+\mathbf{e4}}{\mathbf{P3}_{\text{tp}}}$
  $\mathbf{P4}_{\text{ray}}$ hits environment: evaluate **e5** : record $\mathbf{P4}_{\text{rad}} = \frac{\mathbf{e5}}{\mathbf{P4}_{\text{tp}}}$
  $\mathbf{P5}_{\text{ray}}$ hits Q2 : remember **e6** : continue **P5**
- Wave 4: $\mathbf{P5}_{\text{ray}}$ hits Q3 : remember **e7** : continue **P5**
- Wave 5: $\mathbf{P5}_{\text{ray}}$ hits environment : evaluate **e8** : record $\mathbf{P5}_{\text{rad}} = \frac{\mathbf{e6}+\mathbf{e7}+\mathbf{e8}}{\mathbf{P5}_{\text{tp}}}$
  record $\mathbf{P2}_{\text{rad}} = \frac{\mathbf{e5}+\mathbf{e6}+\mathbf{e7}+\mathbf{e8}}{\mathbf{P2}_{\text{tp}}}$
  record $\mathbf{P0}_{\text{rad}} = \frac{\mathbf{e3}+\mathbf{e4}+\mathbf{e5}+\mathbf{e6}+\mathbf{e7}+\mathbf{e8}}{\mathbf{P0}_{\text{tp}}}$

A more intuitive way to think about it is that we must collect all the emission values that might contribute to $\mathbf{P\#}_{\text{rad}}$ and then divide by $\mathbf{P\#}_{\text{tp}}$. For some shorter path segments, such as that at **P1**, only a small amount of storage is needed (for **e1**, and **e2**) that can be freed once that path segment is complete. This is an interesting special case. For others, such as that at **P0**, we can only know that we have collected all relevant emissions when the entire path completes. Neither these individual emission components nor the path throughput for prior waves is typically stored in a wavefront

renderer. In fact, the efficiency of a wavefront architecture is designed specifically around avoiding the need to do this!

The improved algorithm requires a secondary data structure to accumulate emission results and *backpropagate* radiance results as training data as paths complete. We call this data structure the *Radiance Recorder*.

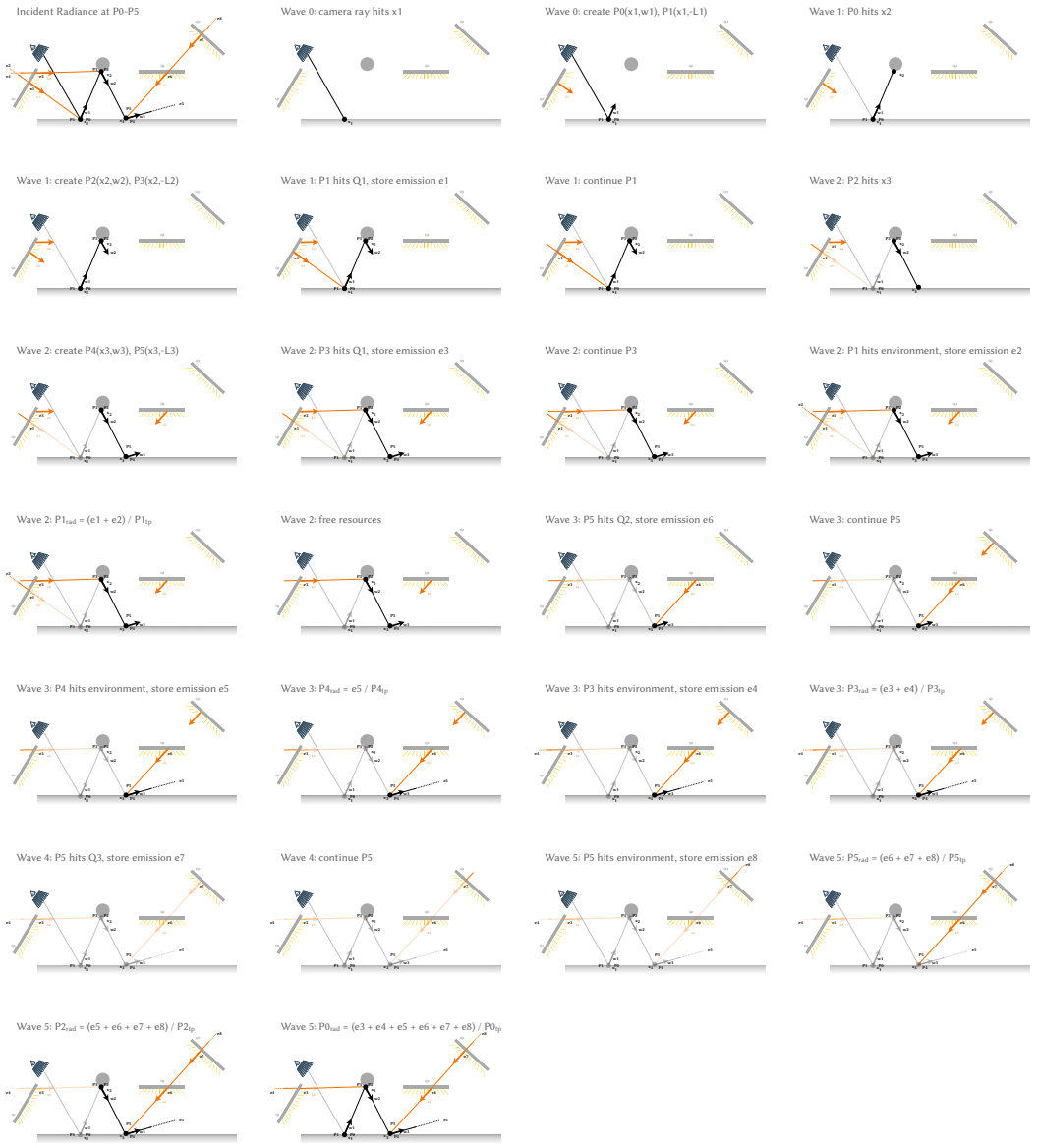


Fig. 4. The algorithm from Section 3.1 and Figure 3 illustrated per wave in a wavefront context.

3.3 Radiance Recording Compared to a Depth-First Renderer

Recording incident radiance in a depth-first renderer is seemingly trivial when compared to a wavefront render. The secondary storage requirement needed by our improved algorithm is completely satisfied by the ray trace stack.

Writing out each step in Figure 4 yields a natural depth-first implementation:

Camera ray hits **x1**
  Create **P0**, $P0_{em} = 0$
    $P0_{ray}$ hits **x2**
      Create **P2**, $P2_{em} = 0$
        $P2_{ray}$ hits **x3**
          Create **P4**, $P4_{em} = 0$
            $P4_{ray}$ hits environment
            $P4_{em}$+= e5
          Record $\mathbf{P4_{rad} = P4_{em}/P4_{tp}}$
          $P2_{em}$ += $P4_{em}$
          Create **P5**, $P5_{em} = 0$
            $P5_{ray}$ hits Q2
            $P5_{em}$ += e6
            $P5_{ray}$ continues to Q3
              $P5_{em}$ += e7
              $P5_{ray}$ continues to the environment
                $P5_{em}$ += e8
          Record $\mathbf{P5_{rad} = P5_{em}/P5_{tp}}$
          $P2_{em}$ += $P5_{em}$
        Record $\mathbf{P2_{rad} = P2_{em}/P2_{tp}}$
        $P0_{em}$+ = $P2_{em}$
        Create **P3**, $P3_{em} = 0$
          $P3_{ray}$ hits Q1
          $P3_{em}$ += e3
          $P3_{ray}$ continues to environment
            $P3_{em}$ += e2
        Record $\mathbf{P3_{rad} = P3_{em}/P3_{tp}}$
        $P0_{em}$+ = $P3_{em}$
  Record $\mathbf{P0_{rad} = P0_{em}/P0_{tp}}$
  Create **P1**, $P1_{em} = 0$
    $P1_{ray}$ hits Q1
    $P1_{em}$ += e1
    $P1_{ray}$ continues and hits environment
      $P1_{em}$ += e2
  Record $\mathbf{P1_{rad} = P1_{em}/P1_{tp}}$

Although its recursive nature makes the notation a bit awkward, we see that the essential values (local emission value and local path throughput) are readily available whenever paths complete. All we need to do is pass accumulated emission results back up the stack.

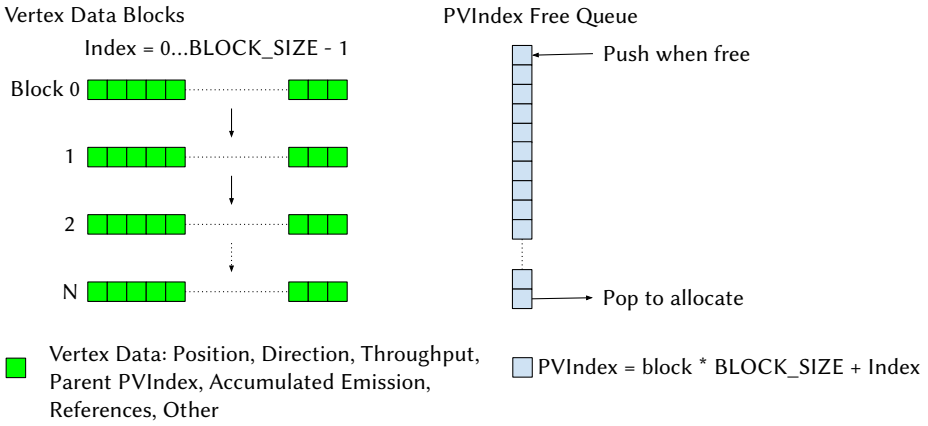


Fig. 5. Block diagram of the Radiance Recorder data structure.

3.4 Radiance Recorder Data Structure

Per ray data in a wavefront renderer is expensive. This data is stored en-masse (millions of rays) as batches that are sorted by various criteria prior to traversal and shading. There really isn't a need, nor the memory to carry around the ray history from previous waves within a path, unless you want to do something fancy like path guiding! In fact, the underlying assumption of the wavefront architecture is that anything we need to do can be done from local data on the current path ray segment itself. To some extent, this makes a wavefront architecture a poor choice if path guiding is high on your priority list of features.

Space for path vertex data is allocated in BLOCK_SIZE sized blocks (currently space for 4096 vertices). Each addressable location is computed as a unique 32-bit integer via the obvious block * BLOCK_SIZE + index_within_the_block. This value is referred to as a *PVIndex*. It is stored in our ray data, providing a mechanism to reference off-ray dynamic memory storage. PVIndices are allocated on demand at scattering events from a queue of available PVIndices and pushed into the queue when they are freed.

3.4.1 Vertex Data For any given path vertex, we store a variety of data members that are useful for path guiding and debugging. The most important (and obvious) ones are position, direction, and path throughput. Several members deserve more special explanation: parent PVIndex, reference count, and accumulated emission.

By storing the parent PVIndex for any path vertex, we create a single-link list of path vertices that define a path. Given any PVIndex, we are able to walk the entire path from it back to the original camera ray.

When a path vertex is created, its reference count member is set to 1. When or if a child is created from it as a parent, or the path vertex is continued, the reference count is increased. When an interaction is marked as "done", the reference count decreases. When a path vertex's reference count drops to zero, a process we call *backpropagation* is triggered and the PVIndex is freed.

When a path vertex is created, its accumulated emission member is set to 0. If it interacts with an emissive geometry or an emissive volume, this emission is added to its emission. When a child vertex's reference count drops to 0, but before it is deleted, its emission is added to its parent.

3.4.2 At Scattering Events At any scattering event (i.e. surface hit or volume interaction) the current PVIndex is retrieved from the ray data and acted on in the following ways:

- Add any emission (i.e. we hit an emissive geometry or an emissive volume).
- Create new child path vertices for scattered rays and NEE rays.
- Continue the PVIndex if needed (e.g. ray continues after passing through a virtual light).
- Mark the interaction as done.

Illustrated as pseudo-code:

```
Results results ← shade_scatter_event(rayData)
PVIndex pv ← rayData.getPV()
if results.haveEmission() then
    pv.addEmission(results.emission())
for all newRayData in results.scattered_or_nee_ray() do
    // Increment the reference count on pv
    // childPV reference count is 1
    PVIndex childPV ← pv.newPV(newRayData)
    newRayData.setPV(childPV)
if results.continueRay() then
    pv.continue() // Increment reference count on pv
pv.setDone() // Done with interaction, decrement pv's reference count
```

3.4.3 Backpropagation When a path vertex reference count drops to 0, no further action will take place on it. The path vertex will have no more direct emission added, it will not be continued, and it will accumulate no further emission from its children. The following actions now take place:

- The incident radiance is computed from its accumulated emission and the sample is recorded as training data for path guiding.
- Its emission is added to its parent's emission.
- The parent's reference count is reduced by 1.
- If the parent's reference count drops to 0, repeat this process recursively.

We call this entire process *backpropagation*. Illustrated as pseudo-code:

```
function BackpropagateRecursive(pv)
    Assert (pv.references == 0)
    incidentRadiance ← pv.emission / pv.throughput
    RecordTrainingSample(pv, incidentRadiance)
    parentPV ← pv.parent()
    if parentPV.notCamera() then
        parentPV.addEmission(pv.emission)
        Decrement(parentPV.references)
        if parentPV.references == 0 then
            BackpropagateRecursive(parentPV)
    Delete(pv)
```

3.4.4 Controlling Memory It is often impractical as well as unnecessary to record incident radiance values for every path and/or at all scattering locations within a path. Accordingly, our Radiance Recorder supports the ability to skip whole paths and/or skip vertices within a path.

By default, we set a target budget of N=4 vertices per path. On each iteration, we determine a rate at which we will skip paths based on the average path length in the previous iteration and our target budget. For example, if on the last iteration we had a total of 8 paths with 8 path vertices each, we had a total of 64 path vertices. Our goal is to use the equivalent number of vertices based on our target, in this case 8 * 4 = 32. So our skip rate for the next iteration will be 0.5. When a camera ray is generated, we sample a random value and choose to skip the path based on this random value and the rejection rate. A path is marked as skipped simply by using a special PVIndex value of "SKIPPED". A skipped PVIndex never allocates memory, nor ever has any children.

Heavy volumes have a tendency to produce a large number of very closely located scattering events. To address this, we establish a minimum distance between path vertices. When attempting to create a child vertex that is very close to the parent, we simply CONTINUE the parent path vertex instead of allocating memory for a new path vertex.

4 DEBUGGING PATH GUIDING IN HYPERION

The need for additional debugging tools arose quickly during the process of integrating path guiding into our production renderer, both to gain confidence in the correctness of our implementation and to better understand the limitations of the approach used.

During our integration effort, we aimed to validate three aspects of path guiding separately:

(1) Is the sample data that is passed from the renderer to the path guiding training routines correct?
(2) Is the model learning a reasonable model given the training data?
(3) Is sampling from the model done in such a way that variance in the final rendering is reduced without introducing bias?

For the first point, we augmented our Radiance Recorder data structure (Section 3) with additional debugging capabilities. And for the second and third points, we developed a renderer-agnostic debugging tool, the *Path Guiding Visualizer*, which can visualize the learned spatio-directional model, and can optionally show additional histograms of arbitrary data that can be gathered during rendering (e.g., 2D histograms of the sampled incident data).

4.1 Radiance Recorder Debug Output

The wavefront-based architecture of Hyperion required us to develop a Radiance Recorder, i.e., a subsystem to gather training data for path guiding, as described in Section 3. We realized that we can instrument this subsystem to validate the gathered training data against the framebuffers. To that end, we added the capability to the Radiance Recorder to splat certain training samples into an auxiliary debug buffer. For instance, for every primary hit training sample, we can compute the color that this path would splat to the beauty buffer. We do so by taking the sample's incident radiance, and multiplying it with the throughput between the camera and the current vertex and local hit point's material response. All these quantities are available to the Radiance Recorder. We can then splat this quantity into a debug buffer, and compare the resulting image to the beauty image. If they match, we know that we gathered the data at primary hit locations correctly. If path- or vertex-skipping was used during training data aggregation, the debug image would appear to be noisier than the beauty image, but both images should still converge towards the same image.

This approach became our go-to tool to verify the correctness of the Radiance Recorder during development, mainly due to its very low overhead and simplicity.



(a) Rendered image (b) Debug output (no skipping) (c) Debug output (skipping 90%)

Fig. 6. If implemented correctly, the Radiance Recorder's debug output should perfectly match the rendered beauty image if no path skipping is used. If path skipping is used, the debug image appears noisier, but should still converge to the same solution as the beauty image. © 2025 Disney

The approach above can tell us if the total radiance gathered up to the primary hit location is correct. However, it does not rule out the possibility that the wrong parts of the throughput get factored out along later bounces. We can generalize the above approach to validate the data gathered at the Nth bounce. To that end, at every Nth hit sample, we splat the product of the incident radiance and the local material response, but don't multiply it with the prefix throughput. If that debug image matches a (non-beauty) output image from the renderer in which the throughput between the camera and the Nth bounce is assumed to be 1, then we should again get a match or images that converge towards the same solution in case of vertex- or path-skipping. This approach was more complex to implement and maintain, so it was used only intermittently. Since much of the gathering code and logic between primary and secondary hit point samples is shared, it's unlikely to have a failure at the Nth bounce which does not show up in the first bounce already.

4.2 Path Guiding Visualizer

Our most powerful debugging tool is a visualizer that loads a navigable representation of the spatio-directional field that was created by the path guiding method. We specifically tailored the version of our tool shown in this course to OpenPGL. In principle, however, adjusting the tool to other path guiding methods is straightforward as long as the method learns a spatial subdivision tree that can be used to create a cheap, navigable representation of the scene. Given the serialized output from OpenPGL, our goal was to design a tool that can load the field from disk, deserialize it, and create a navigable 3D representation of the scene, where the user can examine the learned model and inspect guiding statistics per 3D location in the scene.

4.2.1 Navigable Representation of the Scene We designed our tools to be independent of the renderer that created the path guiding field, hence we completely avoid the need to parse the scene descriptions, as their format might differ significantly between different renderers. Therefore, our representation of the scene can be created from the compact guiding field data alone, which typically is just a few megabytes in size. The field stores a spatial subdivision tree where leaf nodes

correspond to voxels in the scene that are associated with a learned distribution of the incident radiance. For each leaf node, OpenPGL stores additional information such as the extent of the spatial region represented by the node, as well as the bounding box of all training samples that fall within a node's region. Leaf nodes are visualized as voxels. The voxels' extent corresponds to the bounding box of the training samples used to train that leaf node's model since the last spatial subdivision was triggered. If we were to use the full extent of the leaf node's region instead, the voxels would include a lot of empty space, making interpretation of the scene representation difficult. For ease of navigation in the scene, we augmented OpenPGL to store the mean reflected radiance towards the camera as a statistic per leaf node, which we can use to assign a meaningful color to each drawn voxel in the visualizer (Figure 7). We store additional information about the camera into a separate metadata file. With that, we can correctly set the initial camera view when loading the scene representation, such that it matches the view in the rendered image. The user can then navigate freely through the scene using simple controls.



Spatial Subdivision at 32 SPP Spatial Subdivision at 256 SPP

Fig. 7. A rendering from *Frozen 2* and the corresponding voxel representation of the spatial subdivision at 32 and 256 samples per pixel. © 2025 Disney

4.2.2 Representing Volumes Exploring volumes with this voxel representation poses a bit of a challenge, as voxels will fill out the full extent of the volume. A useful tool seemed to give the user the option to "slice" the scene at a specific distance from the camera, or from a specific location in space to examine the volume at certain depths (Figure 8). In the future we'd like to augment our viewer by the option to make voxels corresponding to volumetric content transparent. However, this would require the renderer to pass a transmittance estimate per voxel to the field, which our production renders do not provide right now.

4.3 Visualizing the Directional Models

In our voxelized scene representation, the user can select a voxel by clicking on it, and the corresponding directional distribution at the center of the voxel will be shown. Our tool directly uses the

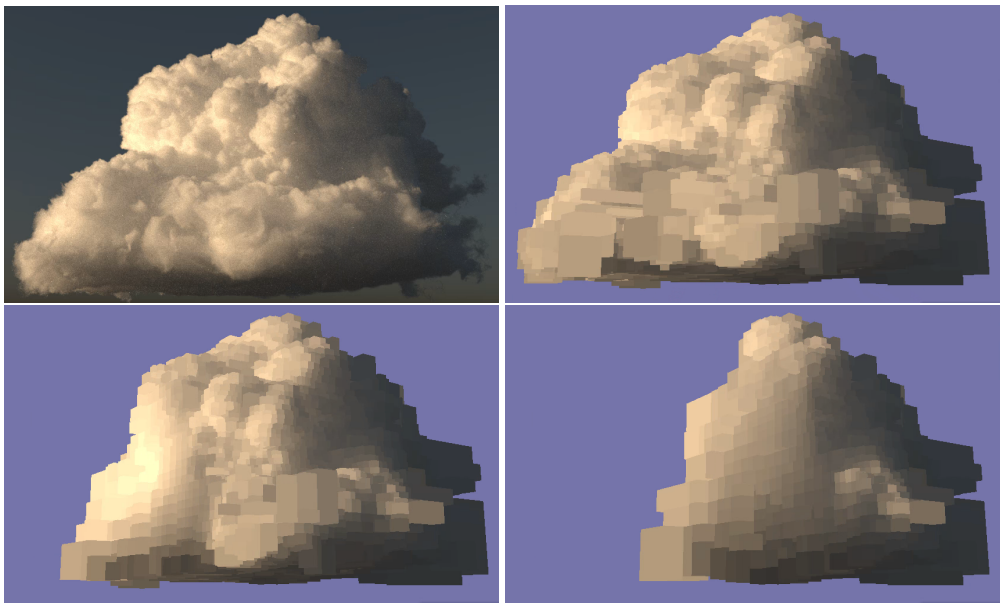


Fig. 8. A rendering of a heterogeneous cloud (top left) and its corresponding voxel based visualization showing the full volume (top right), and two different cuts through the voxel representation of the volume (bottom). © 2025 Disney

OpenPGL API to query the loaded spatio-directional field, and can evaluate the learned directional model on the fly for any location in space. Given a selected location, we initialize the distribution for that location and then query the density over an azimuth-altitude grid covering the whole sphere of directions. We achieve interactive framerates, since the (expensive) initialization of the directional distribution with OpenPGL has to happen only once per location, and the directional queries over the azimuth-altitude grid can be parallelized trivially. Once the azimuth-altitude datapoints have been acquired, we visualize the density in two ways: As a 2D cylindrical projection of the sphere of directions in a separate window, and on a sphere projection in the main scene representation window located at the selected location in the scene (Figure 9). We found that both views are useful: The former makes the entire sphere of directions visible at one glance, while the latter helps the user associate peaks in the distribution with corresponding scene features.

4.3.1 Visualizing the Parallax Compensation The mixture-based model from OpenPGL uses parallax compensation as described in Ruppert et al. [Ruppert et al. 2020]. This is a powerful feature in OpenPGL that allows in many cases to represent the radiance across larger regions of space more accurately. However, it requires passing distances to the emitter as part of the training data, which in a production renderer can be rather difficult to provide in a consistent fashion (e.g. what distance should we use for direct incident light arriving from multiple virtual lights located at different distances along the ray?). Hence, visualizing the parallax compensation's effect on the distribution is important so that we can detect issues that might arise from it. To check the effect of parallax compensation on the learned distributions, we implemented additional sliders to specify positional offsets in XYZ direction with respect to the center of the selected voxel, such that movement within a voxel becomes possible. The offset gets applied on the central location of the voxel before querying the distributions. In combination with the spherical distribution view,



Fig. 9. Showcase of directional distribution visualizations on a frame from *Moana 2*. 2D cylindrical projection of a directional distribution (bottom left). Projection of the same distribution onto a sphere, placed inside the voxel representation of the scene (bottom right). © 2025 Disney

one can then perturb the position of the distribution and assess if the compensation performs as expected (Figure 10). To make changes due to parallax compensation visible for small voxels, we made the limits of the offset 4x larger than the extent of the voxel. By going beyond the limits of the voxel and observing if the parallax compensation is able to still track features in the scene, it becomes easier to judge if it indeed has the expected effect.

4.3.2 Visualizing the Stochastic Lookup OpenPGL performs a stochastic lookup across neighbors during inference. We found that ignoring this effect in the visualized distribution can lead to misleading conclusions. We thus add the option to "simulate" such stochastic lookups in our viewer. To that end, while selecting a location, the viewer continuously queries the model with different stochastic inputs (this means querying a fixed location from a randomly selected neighboring model). The visualized distribution is updated in real time showing the integral across all these distributions that were looked up so far. After a few seconds, the shown distribution corresponds to the expected distribution that the renderer will query from a specific location in space.

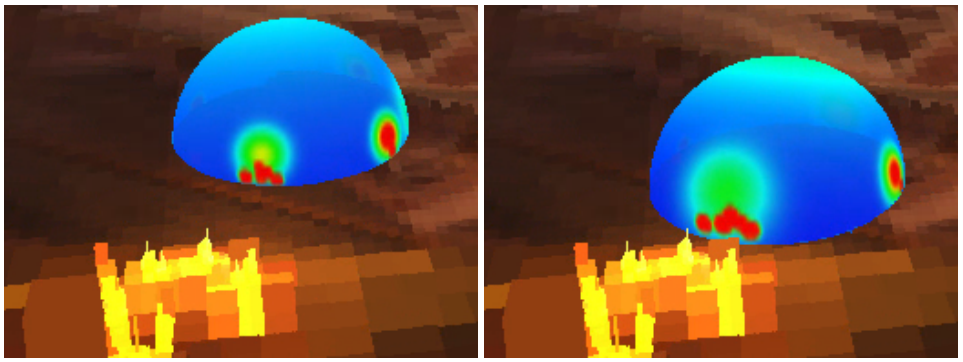


Fig. 10. Offsetting the querying location within or around a voxel to examine the effect of the parallax compensation on a closeup of a scene from *Moana 2*. Notice how the frontal peak of the distribution moves as we change the offset to the query position, such that it always points towards the light source in the front. © 2025 Disney

4.4 Visualizing Additional Statistics

Sometimes it is hard to assess whether a model really makes sense, e.g. whether the brightness of a light source as seen in the model is correct. During development, we stored low resolution 2D images per spatial tree leaf node that store the incident radiance training data. This would not identify issues with the training data itself but could be used to assess whether the learned model matches the training data. We later extended this simple approach to store additional data; by far the most useful additional data was a 2D histogram of the empirical distribution of samples. We could check if the actual empirical distribution of samples would match the learned distribution. We used this as a sanity check to see if our integration of the learned model into the directional sampling routines would be correct (see item 3 in Section 4). Differences would still be visible due to the empirical distribution being marginalized over the region of space corresponding to a leaf node of the spatial SD-tree, whereas the shown models are for a specific location in space. So in practice the empirical distributions were blurrier than the theoretical ones. Despite this, this visualization proved to be useful in identifying defects in the sampling routines.

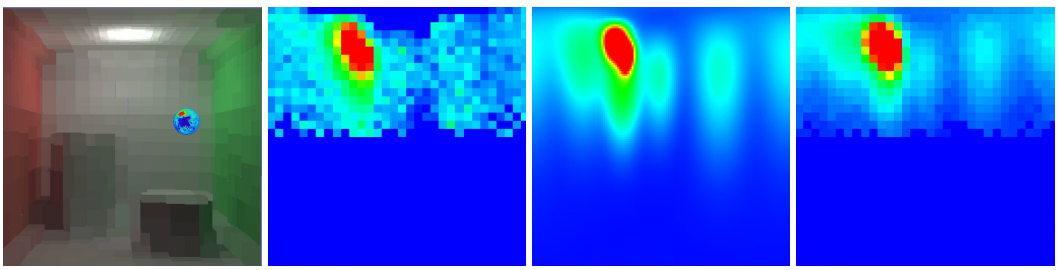


Fig. 11. Example of the recorded incident radiance in a Cornell Box (2nd from left), the corresponding learned model (2nd from right) and a 2D histogram of guiding samples drawn from this model in the renderer (far right).

5 LESSONS FROM PATH GUIDING IN PRODUCTION

In this section, we share our experience of bringing a research project into a production environment. We highlight aspects of production scenes and workflows that pose challenges that may be overlooked in a research environment.

Sections 5.1 and 5.2 discuss two technical details of our second-generation path guiding implementation that were influenced by insights we gained during testing on shots in our production environment. The technical details discussed are specific to Hyperion and may not apply directly to other renderers. Nevertheless, we believe these insights help provide intuition into how the integration of a new method can affect a production renderer in many ways. In Section 5.3, we give an overview of the practical realities of production rendering, and we discuss why experiments conducted in research environments might not directly translate to production environments. And lastly, Section 5.4 gives insights into a selection of production shots and discusses path guiding's performance on those shots.

5.1 Vertex Splatting Depth

In Section 3.4.4, we discuss that the Radiance Recorder data structure supports stochastic skipping of vertices or entire paths when recording incident radiance. This section presents our motivation for this functionality in more detail on a practical use case from production. In fact, we initially did not plan on adding vertex and path skipping functionality, and the need for it only became clear during production testing on exceptionally volume-heavy shots on *Moana 2*.

5.1.1 Motivation As discussed in Section 3.4.4, the purpose of the vertex and path skipping functionality is memory control. As paths can become very long on complex production shots, recording radiance estimates for every single path vertex becomes impractical. In production shots with dense volumes, paths can have hundreds of bounces. As the radiance estimates are only splatted into the path guiding radiance cache at the end of a render iteration, peak memory usage would be excessive in such cases.

In the following, we motivate our choice of stochastic vertex and path skipping, as opposed to a naive vertex restriction method.

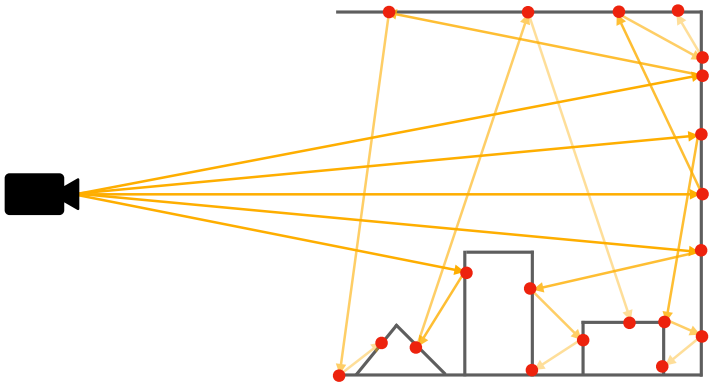


Fig. 12. Illustrative example of restricting number of recorded radiance estimates to the first N=4 vertices of a path. The first 4 vertices can have decent spatial coverage in simple scenes.

5.1.2 Naive Approach Naively, we can restrict the number of recorded radiance estimates per path, by storing only the radiance estimates at the first N vertices. This way, the memory cost per path is fixed and controllable. Moreover, vertices earlier on a path are likely to cover important areas of the scene, while later vertices tend to have lower importance. In a simple scene with mainly surface interactions, this strategy may work very well. Figure 12 illustrates this thought: With only the first N=4 vertices, we achieve decent spatial coverage of the entire scene.

Unfortunately, this strategy fails for scenes with very dense volumes. During production testing on *Moana 2*, we encountered extremely volume-heavy shots, with volumes of very high density, much denser than what we had seen in past shows and in research scenes. Figure 13 shows such a shot from *Moana 2*.



Fig. 13. A volume-heavy shot from *Moana 2*. © 2025 Disney

If we record only the radiance estimates at the first N vertices, the spatial distribution of the recorded data can be insufficient and only "scratch the surface" of a dense volume, as illustrated in Figure 14. The spatial subdivision of the resulting SD-tree data structure will look imbalanced as a result. And more importantly, the training data available deeper inside the volume is too sparse to learn meaningful guiding distributions.

5.1.3 Improving Spatial Distribution of Training Data in Dense Volumes The vertex and path skipping feature discussed in Section 3.4.4 solves the problem of imbalanced spatial distribution of training data in dense volumes. If we stochastically skip vertices or entire paths when recording radiance estimates, we can record data at later bounces while maintaining a soft vertex budget.

Let's take a closer look at how our two skipping strategies, vertex skipping and path skipping, improve the above example with a very dense volume. In Figure 15, we illustrate how stochastic vertex skipping can improve the spatial distribution of training data inside the volume, while maintaining a soft vertex budget of 4 vertices per path, using a suitable skipping probability learned in an earlier iteration.

The following mathematical example further motivates the use of the skipping functionality: With the naive approach from Section 5.1.2, and a vertex budget of 4, the maximum path depth at which we record data is always 4. In contrast, with stochastic vertex skipping with a soft vertex

budget of 4, and skipping probability 0.25, for example, the maximum path depth at which we record data is 16 on average. This ignores early path termination for simplicity.

Similar to vertex skipping, we can skip entire paths stochastically, and in turn allow more vertices per path to be splatted. This way, we can also reach deeper into the volume, while maintaining a soft average vertex budget per path. Figure 16 illustrates this concept.

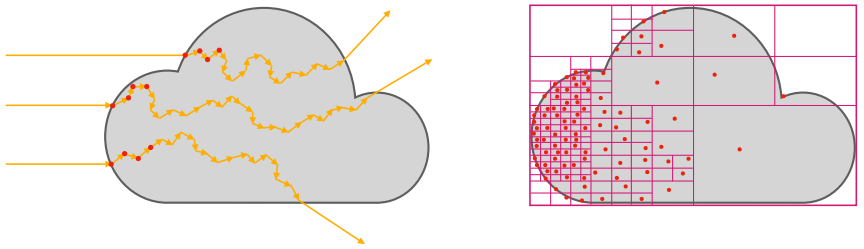


Fig. 14. Illustration of paths scattering through a dense volume. When recording the radiance estimate only at the first N=4 vertices, the resulting SD-tree data structure can have imbalanced spatial coverage of data inside the volume.

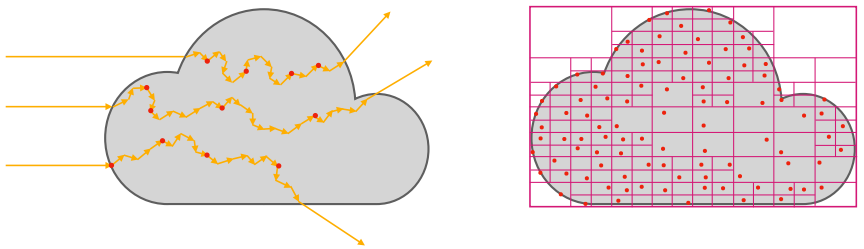


Fig. 15. Illustration of stochastic vertex skipping with a soft vertex budget of 4 vertices per path. Skipping vertices earlier on the path leads to better spatial distribution of data inside the volume.

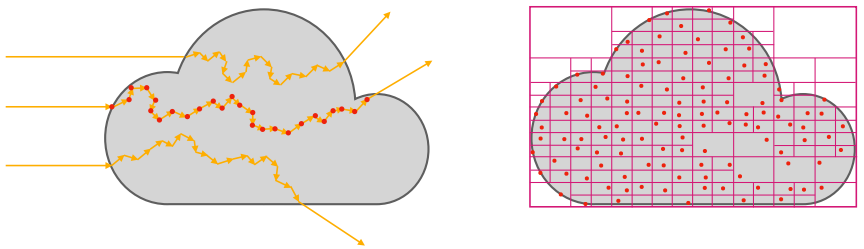


Fig. 16. Illustration of stochastic path skipping with a soft vertex budget of 4 vertices per path. We allow selected paths to record more vertices than the soft budget, allowing recording data deeper into the volume.

5.2 Render Iteration Schedule

In this section, we show two implementation details that relate to the render iteration schedule. The adoption of a path guiding iteration schedule in Hyperion demanded implementation changes to prevent undesirable impacts on render time and memory usage. We discuss two examples and discuss how those implementation changes affected rendering performance.

5.2.1 Render Iteration Schedule Recap We first recap how we define a render iteration in Hyperion. For more details, see Burley et al. [Burley et al. 2018]. Hyperion divides a render into render iterations. In each iteration, we render a subset of the total SPP. The first iteration is small (usually either 16 SPP for batch renders or 4 SPP for interactive renders), and the iteration sizes double each round until reaching a max iteration size. The following could be a typical sequence of iteration *sizes* in a Hyperion render:

16	16	32	64	128	256	256	...	256

In between iterations, no rays are in flight, which makes it a good time to do computations without concurrency issues. Hyperion does tasks, such as:

- Checkpointing
- BVH Updates
- Adaptive Sampler Updates
- Cache Points Updates
- Photon Mapping Updates
- Path Guiding Updates

For path guiding, we in theory desire small iterations, especially in the beginning. We want to update the guiding distributions frequently, so that subsequent iterations can benefit from better distributions as quickly as possible. Therefore, our path guiding iteration schedule differs from our regular iteration schedule shown earlier. These are the key differences:

- We start with a smaller iteration size than in the regular iteration scheme.
- We repeatedly use the smallest iteration size in the beginning, before transitioning to geometrically increasing increments.
- The maximum iteration size is smaller than in the regular iteration scheme.

Depending on the renderer architecture and features, the path guiding iteration schedule can strongly influence the performance and result of path guided renders. The following considerations specific to Hyperion played into our schedule:

(1) **Peak Memory Usage** of path guiding data is lower with smaller iterations: The radiance estimates recorded with the Radiance Recorder get stored for the duration of an iteration. After each iteration, the data is used to update the learned guiding distributions, and gets cleared afterwards. The longer the iterations, the more path guiding data gets accumulated at the same time, which increases memory usage. For this reason, we set our maximum iteration size for path-guided renders lower than for non-path-guided renders.

(2) **Quality of Image Result** improves with smaller iterations: Since we only update the guiding distribution after every iteration, it is beneficial to have smaller iterations to allow more frequent updates; this way, the guiding distribution converges faster and the samples will have lower variance earlier on. The final image will look less noisy. Small iterations are mostly beneficial early in a render, before the guiding distributions have converged.

(3) **Render Time** per SPP may increase with smaller iterations: Smaller iterations can lead to longer render time as ray batching in our wavefront renderer becomes less effective, and

the computations between iterations add overhead. Section 5.2.2 discusses an example of such an overhead in detail.

5.2.2 Checkpointing A path guiding iteration schedule generally has smaller iterations than our regular iteration schedule in Hyperion, because of the frequent training updates and the memory considerations mentioned in Section 5.2.1. And as mentioned earlier, smaller iterations can lead to higher average render times per SPP, because of the additional overhead between iterations. With small iteration sizes, the checkpointing at the end of every iteration can make up a sizeable portion of the total render time, which is not surprising: The number of image buffer outputs in a production shot can be large to allow flexibility in compositing. This often includes outputs such as: mattes, per-light emission, albedo, depth, variance.

The amount of checkpointing overhead is scene dependent; in a scene with complex light transport where traversal and shading take very long, the overhead of frequent checkpointing may be insignificant. A simple scene, on the other hand, may have very fast light transport computations, and the overhead of checkpointing can be significant relative to the total render time. Moreover, the absolute cost of a single checkpoint can be scene dependent; for example, the number of matte and emission buffers we write out can vary. If we make heavy use of separating emission into different buffers, the number of buffers can become very large for scenes with many light sources.

Disabling checkpoints to save render time is not an option in production. In a production environment that makes use of a renderfarm, render jobs get bumped from one machine to another frequently. Checkpointing relevant data structures and image buffers is crucial to ensure efficient resumption of render jobs. It's generally a good practice to produce a checkpoint in geometrically increasing intervals, aligning with the regular Hyperion iteration schedule.

Therefore, we employ a checkpoint skipping logic for path guiding iteration schedules. The skipping logic ensures that we are producing the same number of checkpoints in a path guiding iteration schedule, as in a regular Hyperion iteration schedule. The skipping logic works with arbitrary path guiding schedules, assuming the path guiding iteration schedule generally has smaller or equal sized iterations compared to the regular iteration schedule. This is always true in our case.

Skipping Logic Our skipping logic is conceptually simple. Ideally, we write checkpoints at the same cumulative SPP counts as our non-path-guided renders, which use geometrically increasing intervals. For example, for a 128 SPP render, we ideally write a checkpoint after iterations with the following *cumulative* SPP counts, and skip checkpointing for all other iterations:

4	8	16	32	64	128

However, the cumulative SPP counts at each iteration can be irregular in a path guiding iteration schedule. The cumulative SPP count at each iteration could be as follows:

1	2	3	4	5	6	7	8	10	14	22	38	54	70	86	102	118	128

Therefore, we write checkpoints at iterations whose cumulative SPP count is closest to the ideal cumulative SPP count. More precisely, it's the iteration with the next smaller or equal cumulative SPP count. In the above example, we would perform checkpoint skipping as follows:

~~1~~	~~2~~	~~3~~	4	~~5~~	~~6~~	~~7~~	8	~~10~~	14	22	~~38~~	54	~~70~~	~~86~~	~~102~~	~~118~~	128

	Checkpoint after every iteration	Checkpoint skipping logic
Render a)	17.91%	3.19%
Render b)	16.02%	2.58%
Render c)	9.82%	1.47%
Render d)	6.85%	0.95%

Table 2. Percentage of render time (wall) spent on checkpointing when using a path guiding iteration schedule. The left column shows the percentage if we create a checkpoint at the end of every iteration. The right column shows the percentage if we employ the skipping logic discussed in this section. The percentage is calculated as: time (wall) used for checkpointing, divided by total render time (wall). The percentages are averaged over three runs.

Checkpoint Overhead Savings In Table 2 we report the checkpoint overhead relative to the total render time, for four different shots from different shows, using path guiding. It illustrates how the relative checkpointing overhead varies between shots. The left column shows the relative checkpoint overhead if we perform a checkpoint at the end of every iteration. The right column shows the reduced values as a result of the checkpoint skipping logic we employ in Hyperion. Checkpoint skipping may not always lead to significant time savings, but there are no notable downsides. Therefore, we apply checkpoint skipping by default in renders using path guiding.

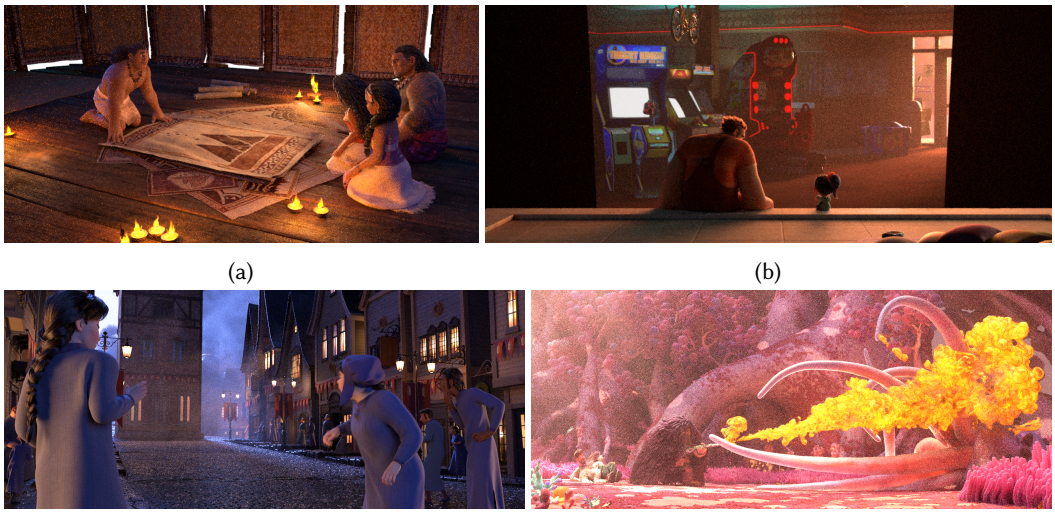


Fig. 17. Renders used in Table 2, from a) *Moana 2*, b) *Ralph Breaks the Internet*, c) *Frozen 2*, and d) *Strange World*. All rendered with path guiding at 64 SPP. © 2025 Disney

5.2.3 Consistent Alpha Here we present a somewhat esoteric and unexpected issue that we ran into with adjusting Hyperion's iteration schedule for path guiding. We include a discussion of this problem here not so much because we expect any other implementers to run into the exact same problem, but more so as an example of the kind of cascading secondary effects that implementing

path guiding into a complex production system can have and therefore the care that must be taken when considering how to fit path guiding into a production renderer.

Hyperion makes extensive use of adaptive sampling. With adaptive sampling, Hyperion interprets the requested SPP as an overall sample budget instead of a per-pixel budget, meaning that on average over the entire framebuffer, the requested SPP level should be reached, but individual pixels may be allocated SPP values potentially significantly below or above the requested SPP level. How many SPP an individual pixel receives is guided by a running estimate of variance for that pixel. We refer the reader to Burley et al. [Burley et al. 2018] for the specific details of this scheme.

One practical challenge that adaptive sampling raises is alpha inconsistency between render passes. In production rendering workflows, lighters often break out different elements in a scene onto separate render passes to allow for both more creative choices in lighting and more flexibility in compositing. If a pixel exists on the boundary between two elements on separate passes that must be added together in compositing, then that pixel in each pass may end up with a fractional alpha value, with the expectation that alpha values for the same pixel across all render passes should add up to 1.0. However, adaptive sampling can distribute different SPP counts to the same pixel across different render passes, resulting in the total alpha value adding up to not exactly 1.0; whenever this discrepancy occurs, compositing artifacts can appear along the edge of geometry.

Our solution to this problem is consistent alpha, in which adaptive sampling is applied to color channels but the alpha channel is rendered using a fixed SPP number per pixel that is identical across all render passes. In practice, the way consistent alpha is implemented is that the renderer traces alpha and color on the same ray and keeps a running tally of how many samples each has received; once the target alpha SPP number for a given pixel has been reached, additional rays for that pixel only contribute to color. If the adaptive sampler decides that no more color samples are needed for a given pixel but the alpha SPP target has not been reached, then Hyperion adds additional alpha-only rays for that pixel to the last iteration of the render until the target alpha SPP is reached.

In wavefront renderers, a common design pattern is for camera ray generation to be interleaved with the main ray processing system in some way [Laine et al. 2013; Lee et al. 2017], and Hyperion is no exception here. The reason for this pattern is because camera rays are required to "prime" the rest of the system, and as paths are completed, maximum occupancy can only be maintained if new paths are started. Hyperion previously implemented this pattern by running camera ray generation on a separate thread in parallel with the main ray processing system. Generated camera rays were placed into a first-in-first-out (FIFO) waiting queue, from which the main ray processing system pulled work off as needed. In order to ensure maximum occupancy of all processor cores, the sum of camera ray generation threads and main processing threads on a net oversubscribes the number of underlying physical cores, and the renderer relies on the underlying operating system's pre-emptive multitasking to "naturally" balance these two systems.

Normally, the balance of work was weighted heavily in favor of main ray processing because 1. normally in the last iteration there are many more secondary rays than camera rays and 2. main ray processing is a much more expensive operation than camera ray generation. As a result, camera ray generation was typically effectively rate limited, which meant that in practice the rate of camera rays going onto the queue of waiting work typically was roughly balanced by the rate of work being pulled off of the queue for processing. When these two rates were balanced, the size of the FIFO queue between the two systems was relatively small. When Hyperion is calculating consistent alpha, the number of additional alpha-only camera rays that need to be generated can be very large if adaptive sampling stopped sampling large swaths of the image plane at SPP levels well below the target average. However, because in a non-path-guided render the last iteration contains half of the total SPP of the entire render, the large number of alpha-only rays needed for consistent

alpha is almost always still small relative to the total quantity of rays to be processed in the last iteration, allowing the balance between camera ray generation and main ray processing rates to be maintained.

As discussed in Section 5.2.1, for path guiding, Hyperion's iteration schedule is adjusted to make iterations contain fewer SPP to allow for more frequent guiding distribution updates. One of the adjustments mentioned earlier is that the maximum iteration size is set to be smaller for path guiding. One effect of this change is that the last iteration of a path-guided render is considerably smaller than the last iteration of a non-path-guided render. This change had an unexpected secondary effect when coupled with consistent alpha. Because additional alpha-only rays for consistent alpha are only generated in the last iteration, on scenes where a very large number of alpha-only rays are generated, in path-guided renders the number of alpha-only rays could sometimes vastly eclipse the total number of regular rays to be processed in the last iteration. When this happened, the balance between camera ray generation and main ray processing would sometimes shift rapidly in favor of camera ray generation, which led to the size of the FIFO queue of camera rays to balloon in size, which in turn resulted in exploding memory usage by the renderer. Before we implemented the Radiance Recorder data structure, all rays of all types had to pre-allocate enough memory to carry all potentially needed path guiding information, which further exacerbated the memory usage spike caused by camera rays piling up in the FIFO queue.

Our solution to this problem was to move camera ray generation from running in parallel to main ray processing to instead run alternating with main ray processing. Now, as Hyperion completes work in main ray processing, if occupancy drops below a certain threshold, Hyperion will pause starting any additional work in main ray processing, run camera ray generation until the minimum occupancy threshold has been reached, and only then proceed with beginning new work in main ray processing.

5.3 Practical Realities of Production Rendering

This section discusses production rendering conditions that may differ from the conditions in a research environment, and can influence the perceived gain of path guiding. In our production tests, we have observed cases where path guiding brings significant improvement, and we will discuss a selection of shots in more detail later in Section 5.4. But before looking at image comparisons, we provide more context into what it takes from a new method, such as path guiding, to bring significant benefit to production. In fact, it can be difficult for a method that proved effective in a research context, to bring the same gains in production.

5.3.1 Scene Complexity Path guiding is a powerful noise reduction technique, but it adds computational cost. It is not equally effective in all scenes, and care needs to be taken to employ it only in situations where its overhead will amortize. It mostly brings benefits in scenes where a non-path-guided render has significant noise levels as a result of suboptimal BxDF or phase function sampling. But production scenes can be surprisingly simple in terms of lighting setups, since they are optimized to render efficiently using standard sampling techniques.

Smart choice of light source type and placement can greatly help improve noise, while achieving the desired look. During our testing, we have found that often scenes can have surprisingly simple lighting setups with a lot of direct lighting. Hyperion provides features for more artistic control over the lighting in the scene, some of which reduce the need for elaborate indirect lighting setups. For example virtual lights, which are non-occluding and therefore invisible to the camera, can be placed within the view frustum and directed at regions that need more illumination. The shot we discuss in Section 5.4.5 illustrates this well. Moreover, our artists have learned to author scenes

that work well for unidirectional path tracing. In such scenarios, the noise reduction due to path guiding can be small, which makes it more difficult to amortize path guiding's cost.

5.3.2 Production Renderer Baseline There are still plenty of complex shots where the non-guided sampling techniques will lead to a lot of variance in the final image, and path guiding has potential to bring significant gains. However, production renderers such as Hyperion already employ numerous optimization techniques by default to handle complex shots, most notably:

- Cache Points for Direct Illumination [Li et al. 2024]
- Outlier Rejection
- Path Simplification
- Adaptive Sampling
- Denoising [Dahlberg et al. 2019; Vogels et al. 2018]

Therefore, the baseline, rendered in a production renderer, can be much harder to beat than a baseline rendered with a research renderer such as Mitsuba [Jakob 2010] or PBRT [Pharr et al. 2023]. Consequently, we cannot expect gains observed in a research setting to translate directly into production. The following examples discuss three optimizations in Hyperion that can affect path guiding's perceived gain.

Example: Cache Points for Direct Illumination Figure 18 shows a shot from *Frozen 2* that has a complex lighting scenario: many light sources, both directly and indirectly illuminating the scene, as well as volumes. If rendered with standard light transport algorithms, this scene would offer plenty of inefficiency for path guiding to leverage and improve upon. In Hyperion, however, our cache points optimization for sampling direct illumination [Li et al. 2024], which is enabled by default, already heavily improves this scene and leaves less room for improvement for path guiding.

Figure 18a shows an equal SPP comparison at 128 SPP. For both renders, we disabled Hyperion's cache points optimization, to illustrate path guiding's gains without the optimization. We see significant noise reduction at equal SPP in the render that uses path guiding. However, if we leave our cache points optimization enabled, shown in Figure 18b, path guiding brings less noise reduction at equal SPP. We still see some noise reduction in some areas thanks to path guiding, but the perceived gain is much smaller than in the example without the cache points optimization.

Example: Throughput Clamping Seeing path guiding reduce noise at equal SPP, as in the previous example above, is a good indicator that supports the use of path guiding for a given shot. However, as this next example demonstrates, image comparisons can be more difficult to interpret in cases where the use of path guiding alters the image appearance. Hyperion employs certain biased optimization techniques, such as throughput clamping. Throughput clamping is a commonly used noise reduction technique in which we clamp the path throughput if it reaches a specific maximum value. This helps reduce fireflies that occur if sampling distributions are not proportional to the integrand and lead to an imbalanced numerator and denominator in the throughput, at the cost of adding a small bias. This noise reduction technique was especially important before the era of ML-based denoisers, which can easily remove fireflies as a post-processing step. Nevertheless, throughput clamping is still used today in Hyperion, but much more conservatively with higher clamping thresholds.

Path guiding reduces the use of throughput clamping, because learning better sampling distributions leads to more balanced throughputs. As a result, we end up with cases where a path-guided render is brighter than the baseline, because less energy gets removed by outlier rejection; path guiding acts as an unbiased, but expensive alternative to throughput clamping. In such cases, it's not trivial which image is preferred, since we do not strictly require our production renderer to produce unbiased results.



(a) Rendered without cache points optimization. With path guiding (bottom left) and without path guiding enabled (top right).



(b) Rendered with cache points optimization. With path guiding (bottom left) and without path guiding enabled (top right).

Fig. 18. Equal SPP comparison of path guided vs. baseline renders on a shot from *Frozen 2*. In a) our cache points optimization is disabled and the noise reduction due to path guiding is more significant than in comparison b), where our cache points optimization is enabled. © 2025 Disney

Figure 19 shows a shot from *Strange World*, where path guiding leads to brightening in some areas. Atmospheric volumes and subsurface scattering add complexity to the light transport in this shot. The crops on the right of the figure show the result with (top left triangle) and without path guiding (bottom right triangle). When rendering this shot without path guiding, the complex light transport would lead to a large amount of fireflies in some areas, which is what throughput clamping prevents at the cost of a slight darkening. Path guiding leads to a visibly brighter, less biased result.

Example: Path Simplification We describe our approach to path simplification in detail in [Burley et al. 2018]. In short, path simplification employs material simplification and roughening at secondary bounces to simplify light transport at the price of bias. Figure 20 shows an interior scene

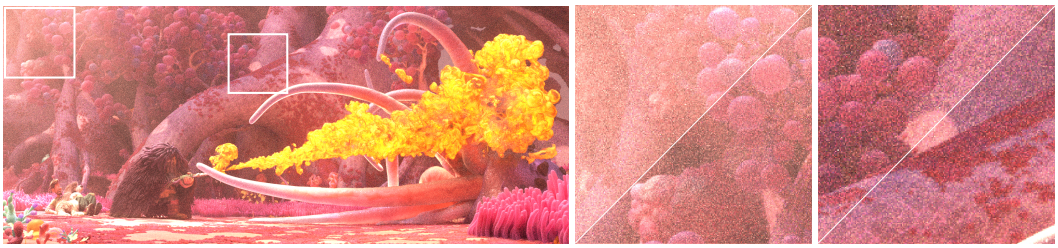


Fig. 19. Shot from *Strange World* showing brightness difference when using path guiding. The crops on the right of the figure show the result with path guiding (top left triangle) and without path guiding (bottom right triangle). © 2025 Disney

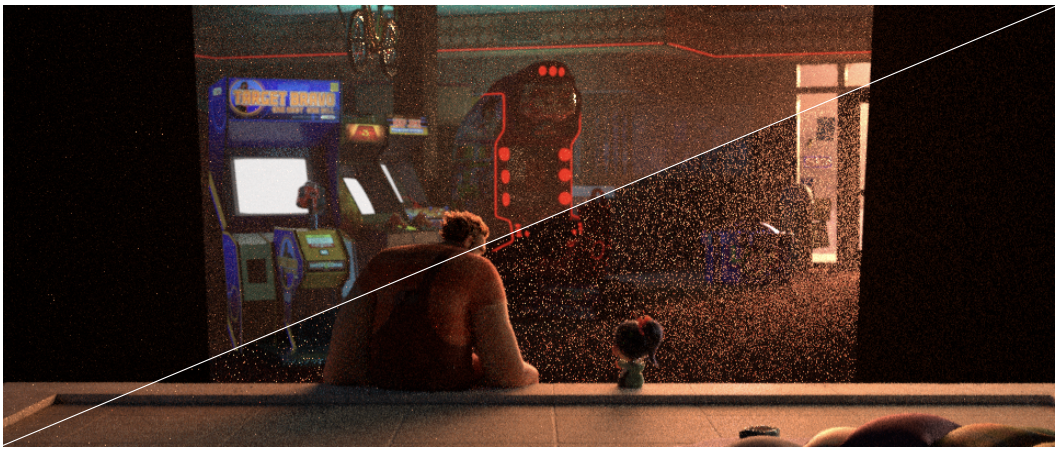
from *Ralph Breaks the Internet*, filled with a forward-scattering participating media illuminated from outside through a glass door. Without path simplification enabled (bottom right), most illumination seen in this scene is from caustic paths, which together with the forward-scattering volume, leads to excessive noise. With path simplification enabled (top left), the volume gets closer to isotropic after a few bounces, and the glass door's material gets simplified to a light-attenuating pass-through material, making shadow ray connections to the outside light source trivial. However, path simplification can impact the rendered scenes in unpredictable ways, and is heavily dependent on heuristics that may or may not work well in particular scenes.

Path simplification is enabled by default, and using path guiding on top of that in this scene seems to yield only minor benefits, as shown in Figure 20b. However, with path simplification disabled, as shown in Figure 20a, path guiding yields massive benefits over the baseline; it nearly allows us to breach the gap between the renderings with and without path simplification enabled.

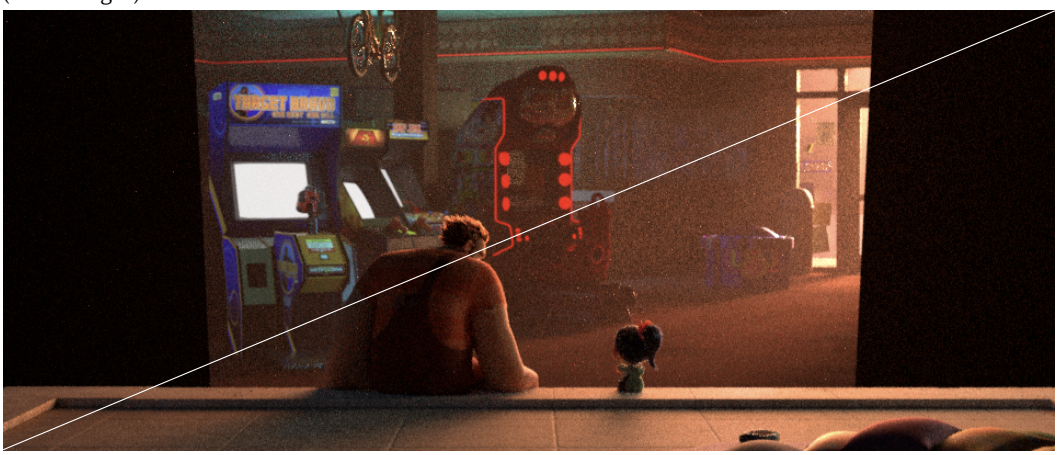
5.3.3 Evaluation Criteria in Production The decision to adopt a new method in a production renderer and enable it for production shots is more involved than evaluating image comparisons; we also look at considerations specific to our production environment, which may be less relevant in research environments. In the following, we give three such examples.

Amortization Rate: Path guiding usually takes more time to render a single path sample, but can amortize its additional cost by providing faster convergence. In the beginning of the render, path guiding is most certainly performing worse than the baseline, as its learned distributions are not optimal yet and don't provide much benefit over BxDF sampling. How many SPP it takes to amortize its cost can vary for each scene. If a baseline render only requires very low SPP counts to get to an acceptable noise level, there might not be enough room for path guiding to amortize its cost. This is an important factor to consider in production environments, which make use of a denoiser to reduce SPP counts even further. As an extreme example, consider a render that reaches an acceptable noise level after just 16 SPP without using path guiding. With our path guiding configuration, we use the first 8 SPP of the render to learn guiding distributions, and only start using the distributions after these first 8 SPP. This then only leaves us with 8 remaining SPP to make use of the better distributions and bring significant noise reduction in the final image. For non-volume-heavy scenes, SPP counts of 32 to 64 are often sufficient, while volume-heavy scenes may use more; 256 to even 1024 SPP are not unheard of. Therefore, it's important to choose suitable SPP counts when doing evaluations, as path guiding gains are subject to the SPP budget.

Interactivity: When assessing the performance of a method or feature, we consider how it affects artist iteration time. *Time-to-first-pixel* is a relevant metric in production: A production



(a) Rendered without path simplification. With path guiding (top left) and without path guiding enabled (bottom right).



(b) Rendered with path simplification. With path guiding (top left) and without path guiding enabled (bottom right).

Fig. 20. Equal SPP comparison of path guided vs. baseline renders on a shot from *Ralph Breaks the Internet*, rendered with 64 SPP. a) Compares the rendered result if path simplification is disabled, and b) compares the result if path simplification is enabled, which is our default in Hyperion. © 2025 Disney

renderer not only renders final frames on the renderfarm, it is also a tool for making artistic decisions quickly. Some artistic decisions can be made already after seeing a handful of SPP, long before the render has finished. For example, we might already see if a light is too bright or too dim, or the tone is off, even if the image is still very noisy. In these cases, every second added at the beginning of the render affects artist iteration time much more than seconds added at later stages in a render. Time-to-first-pixel is a limiting factor for artist iteration time, which is one of the primary metrics we want to minimize. Therefore, changes in time-to-first-pixel also affect how we perceive the gain of a new method.

Physical Correctness: In research experiments, we often use error as an evaluation metric, and we calculate it by comparing the result with a groundtruth image rendered at extremely high

SPP. Bias increases this error and is often considered undesirable. In production, we also prefer unbiasedness if possible. Nevertheless, a small and consistent bias or physical inaccuracy is not necessarily a problem as long as it still provides sufficient predictability to allow the artists to render the scene they envision. As mentioned in Section 5.3.2, Hyperion employs certain biased techniques, such as throughput clamping, path simplification and denoising, that offer noise reduction, at the expense of a small bias or physical inaccuracy. We have also seen in Section 5.3.2 that path guiding can provide less biased results in cases where our baseline Hyperion version does not. Nevertheless, unbiasedness is not our main motivation for using path guiding in such cases, and we fall back to other evaluation metrics to justify its use.

5.4 Discussion of Production Shots Rendered with Path Guiding

This section presents the results of testing path guiding on a selection of shots from Disney Animation's past shows that we have not discussed in earlier sections. In addition to render statistics data, we provide an analysis of path guiding's strengths and weaknesses on given shots, which requires understanding of the scene and lighting setup that is often not visible in the final images.

5.4.1 Experiment Setup and Comparison Metric The baseline renders we use for comparison reflect the default Hyperion configuration, which has several optimization techniques enabled, as mentioned in Section 5.3.2. For the path guided renders, we enable path guiding on top of the already existing optimizations. The reference renders are higher SPP renders of the given scene.

For the following results, we show equal SPP comparisons and provide relevant statistics for each image, such as render time and memory usage. While *equal time* comparisons or *time-to-unit-variance* comparisons are often used in research evaluations, the results depend on various factors influencing render time. Differences in machine, thread count, and asset loading time can influence render time. And different renderer architectures and production environments might react differently to a path guiding implementation in terms of render time overhead. Therefore, we provide equal SPP comparisons, which may be more transferable to other environments.

For the following tests, we ran each render 3 times and discarded the first of the three runs to eliminate inconsistencies due to loading assets into the cache in the first run. The time and memory statistics shown are therefore the average of 2 runs. We report peak memory usage, which is subject to slight variations between runs due to small differences in the order and timing of memory allocations and deallocations during the parallel execution of the render jobs. Therefore, in cases where path guiding does not affect peak memory usage significantly, it is possible that path-guided renders can occasionally have slightly lower memory usage than a non-path-guided renders.

5.4.2 Shot 1 The shot from *Moana 2* in Figure 21 shows an indoor scene with most illumination seemingly coming only from candles. The candle flames are set up as emissive geometry. In reality, there are many more virtual light sources invisible to the camera: several quad lights are directed at the characters. Moreover, small spherical light sources are placed around the candle flames, to further support the candle-like illumination. The main source of noise are the candle flames made of emissive geometry.

Our cache point optimization for direct light sampling does not consider emissive geometry by default; in this scene, it will instead favor sampling all of the virtual lights. As a result, the probability of a path hitting such a small emissive geometry is very small, and leads to fireflies. For path guiding, on the other hand, the type of light source does not matter for learning the guiding distributions. Paths will be directed towards the emissive geometry and its areas of indirect

contribution. This leads to reduced noise at equal SPP, as shown in Figure 21. It's important to note that we also guide towards direct illumination with path guiding, which may also help in this scene.

Hyperion offers an option to turn on emissive geometry sampling for our cache points optimization, which also drastically reduces the noise in this scene without the use of path guiding. The emissive geometry sampling render option is turned off by default, however, since it incurs a large overhead at the beginning of the render during the generation of the cache points: we have to evaluate the emission function for every single triangle of the emissive geometry. This overhead only pays off in selected scenes, such as the one discussed here, because the increase in render startup time can adversely affect artist iteration time. As discussed in Section 5.3.3, interactivity is of high importance, and we therefore disable emissive geometry sampling by default.

This scene is a great example that demonstrates how path guiding can be an alternative to other expensive optimization techniques. Moreover, path guiding reduces the need for manual finetuning of render options affecting light sampling; guiding distributions simply guide the rays towards directions of high illumination, regardless of the origin of such illumination. This also makes it robust to handle novel sources of emission we might add in the future, which might need special handling in other optimization methods we employ.



Fig. 21. A frame from *Moana 2*. Equal SPP comparison rendered with 64 SPP. © 2025 Disney

5.4.3 Shot 2 Figure 22 shows a shot from *Raya and the Last Dragon*. It appears similar to the shot we discussed in Section 5.4.2, as it appears mainly illuminated by flames. The scene setup and type of light sources, however, differ very much from the previous shot. In this shot, the flames are made of emissive volumes, and not emissive geometry. Therefore, the emissive geometry sampling render option discussed in Section 5.4.2 will not improve the non-path-guided render. In addition, this shot contains atmospheric volumes, visible on the right side of the image near the floor. This atmospheric volume is a main source of noise in the non-guided render, since next event estimation from inside the volume proves difficult in this shot. Our cache points optimization in fact supports emissive volume sampling, which is enabled by default. But in this example shot, direct sampling of the emissive volume, i.e. the flames, is not always effective: In Hyperion, we use a simple delta tracking transmittance estimator on shadow rays. This means that we get a binary transmittance estimate that is either 0 or 1, which is a noisy estimate. In addition, the emissive volume from the flame is surrounded by non-emissive smoke, which makes it even harder for a shadow ray to reach the emissive flame without encountering a scattering event. Therefore, sampling direct illumination

is difficult in this shot. Moreover, paths that are lucky enough to reach the emission indirectly through BxDF and phase function sampling may suffer from imbalanced throughputs, leading to fireflies. Path guiding can help in this situation since it provides guiding distributions that lead the path towards the emissive volume, eliminating the need for effective next event estimation from inside the volume.

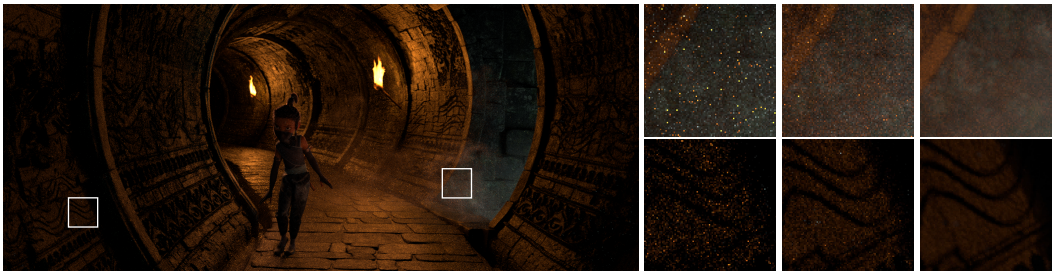


Fig. 22. A frame from *Raya and the Last Dragon*. Equal SPP comparison rendered with 64 SPP. © 2025 Disney

5.4.4 Shot 3 Figure 23 shows a shot from *Frozen 2* that is permeated by a strongly forward-scattering volume (fog) but illuminated mainly by the sky/sun. When scattering according to BxDF sampling, we tend to scatter forward, parallel to the floor. When doing light sampling, we tend to sample orthogonally to the floor. In the former case we don't reach the light, and in the second case the phase function evaluates to a very low value. Ideally, paths would follow a trajectory that slowly increases inclination towards the sky after every scattering event, such that all scattering events evaluate to a reasonably large throughput, while we eventually reach an inclination that makes light sampling possible. Product path guiding, which OpenPGL supports, naturally results in such paths and significantly reduces variance in this scene.



Fig. 23. A frame from *Frozen 2*. Equal SPP comparison rendered with 64 SPP. © 2025 Disney

5.4.5 Shot 4 In the shot from *Moana 2* shown in Figure 24, the camera is inside a canoe, and sunlight is coming through a slightly opened deck hatch. As the sun- and skylight mostly illuminate this shot indirectly, this shot offers itself for path guiding. We indeed observe noise reduction at equal SPP when using path guiding. The noise reduction is clearly visible near the hatch door and objects close by, but less in the areas further away, for example the area around the large plant in the foreground on the right side.

The reason for the spatially varying degree of noise reduction is the lighting setup using virtual lights: in addition to the sunlight coming through the side of the hatch door, several virtual quad lights are placed close to the ceiling, pointing downwards. In fact, these virtual lights make up most of the illumination visible in the foreground of this scene. Virtual quad lights can be sampled directly, and lead to much faster convergence, compared to a setup where all light comes from outside the canoe. Unsurprisingly, the areas where the virtual light contribution dominates are also the areas where path guiding shows less improvement. And areas unaffected by the virtual quad lights show great noise improvements at equal SPP.



Fig. 24. A frame from *Moana 2*. Equal SPP comparison rendered with 64 SPP. © 2025 Disney

6 FUTURE WORK

This section gives an overview of our ongoing efforts to improve our path guiding system in Hyperion. Some of the efforts mentioned are already available for experimentation in Hyperion today, but may need further refinement based on insights gained from ongoing testing. At the same time, we are considering novel approaches that require more development and which will hopefully make it into our production environment in the future.

6.1 Improving Sampling Techniques

For combining sampled directions drawn from path guiding with directions drawn from BxDF sampling, we currently implement both traditional multiple importance sampling (MIS) [Veach 1998] and resampled importance sampling (RIS) [Talbot 2005], for both surfaces and volumes. See also Chapter 1 Section 3 for more details on guided sampling based on MIS and RIS. For RIS, we further allow users to tune the number of samples considered. However, the results presented in these course notes are with RIS disabled. We are continuing to evaluate RIS versus MIS for combining results and, with the goal of preventing artists from needing to worry about deeply technical details like the choice of strategy for combining samples, we are aiming to find reasonable

defaults that generally work well for a majority of our production scenes. Finding good default settings continues to be a work in progress.

We have found experimentally that combining path guiding with adjoint-driven Russian roulette and splitting [Vorba and Křivánek 2016] is generally beneficial, but this work remains experimental and has yet to be fully incorporated back into our main production branch. See Chapter 1 Section 4.2 for more details on guided Russian roulette strategies. In a similar vein, we have experimented with using a purely spatial form of reservoir resampling [Bitterli et al. 2020] for improving direct lighting samples with encouraging results, but figuring out how to incorporate this idea with path guiding remains a work in progress.

6.2 Sample Combination Scheme

The discussion of our path guiding iteration schedule in Section 5.2 leaves out details on how we weight the image results of different iterations. With path guiding, earlier iterations' results often have higher sample variance than later results. Therefore, there exist techniques that aim to optimize combining the results of different iterations to reduce the variance of the final image, e.g. an inverse-variance-based weighting scheme [Müller 2019], or schemes that discard earlier results entirely [Müller et al. 2017]. Our implementation currently does not optimize combining the per-iteration results and weighs them all equally. While this conservative approach ensures that path-guided renders are never noisier than non-path-guided renders at equal SPP, it can also limit the amount of improvement that path guiding can provide. We aim to explore weighting schemes in the future to fully leverage the potential of path guiding.

6.3 Estimating Efficiency

The overhead of path guiding in our current integration is significant. As discussed earlier, not all scenes benefit from guiding equally, and in some cases enabling path guiding negatively impacts overall rendering efficiency. Therefore, having path guiding as an always-on feature in our renderer is not desirable. To help users determine whether path guiding should be enabled in a given frame or sequence of frames, we provide statistics during path guiding that estimate the expected efficiency gain from path guiding (or gain in time-to-unit-variance) from low SPP renderings. We exploit the fact that the very first rendering iteration does not use path guiding, since no guiding model has been learned yet, and estimate the efficiency of the rendering based on crude heuristics. We repeat the same procedure for a later render iteration that uses path guiding, and compare the two efficiencies. If the efficiency went up by a large enough margin between the non-path-guided and path-guided iteration, we recommend the usage of path guiding in this scene, otherwise we don't. Ideally, the later path-guided iteration for this measurement is taken late enough in the rendering process, such that the guiding model is already yielding most of its benefit, and early enough in the rendering, such that the user does not get blocked too long to wait for the recommendation. The efficiency is the inverse product of the error and the cost. We approximate the error by the sample variance of the pixel color, averaged over the entire image, and approximate the cost by the average path length of all paths created during the measured iteration. While the cost estimate is especially crude since it ignores the increased sampling costs arising from more complex sampling routines, and the overhead from gathering training data, shorter iteration size and training cost, we found that efficiency gain estimates based on these heuristics are usually precise enough for our purpose. We believe such systems, and potentially fully automated versions thereof, in the spirit of various other variance and efficiency aware systems [Grittmann et al. 2022; Rath et al. 2022, 2020] to be crucial components. They help increase the acceptance of sophisticated sampling methods that might not always amortize, such as path guiding, into production environments.

6.4 Going Neural

The path guiding methods we experimented with so far used classical machine learning techniques and do not rely on special hardware, e.g., PPG [Müller et al. 2017] and PAVMM [Ruppert et al. 2020]. However, with the increased availability of GPU hardware on workstations and render clusters, sampling methods relying on neural networks to learn the distributions become attractive. Such methods promise to learn more accurate models than their classical counterparts, and hence might reduce variance in our renderings even more. Especially given that many production renderers, such as Hyperion, are purely CPU-based, requiring GPU resources to train such models might be an attractive way of making use of such hardware for rendering without the need to completely overhaul the existing rendering systems. We recently started examining such methods in the context of production rendering with promising initial results [Rath et al. 2025].

6.5 Guiding Other Sampling Decisions

So far, we have used path guiding only to improve directional sampling. Much of the noise in production rendering originates from other sources, though, for instance, from undersampling thin volumes. While Hyperion provides a sophisticated system to improve direct light sampling with some limited application to volumes [Li et al. 2024], no comprehensive system for better *general* volume sampling has been implemented yet. Methods for distance guiding in volumes or scatter probability guiding [Herholz et al. 2019; Xu et al. 2024] seem like promising avenues to further reduce variance, especially since some of these methods can piggy-back on the existing data structures that are already used for directional path guiding.

7 CONCLUSION

We presented the second-generation path guiding system in use today in Disney's Hyperion Renderer. While we have come a long way since our earlier first-generation path guiding system, we consider the modern system to still be a work-in-progress, and these course notes represent a snapshot of the progress we have made and of the myriad of problems we have had to solve. We had to resolve architectural incompatibilities between wavefront path tracing and path guiding by developing new data structures. Developing powerful visualizations and debugging tools was crucial to validating our implementation and helping us solve further production problems. In order to educate artists and TDs on how to use path guiding and give production confidence, we ourselves had to put considerable effort into analyzing how path guiding behaves in complex production scenes and how various practical production rendering factors influence the relative performance of path guiding. We are very excited that, after all of this work, we are now finally at a point where we are embarking on the first large-scale deployment of path guiding into active productions, and we still have further improvements to make to the current system. We hope that everything we have learned will be useful to others interested in path guiding in both production and research.

ACKNOWLEDGMENTS

We thank our partners at DisneyResearch|Studios for their work on this project; specifically, we thank Alexander Rath, Lento Manickathan, Marios Papas, and Tiziano Portenier for their contributions. We are also grateful to Sebastian Herholz from Intel for his assistance with OpenPGL, and to Wei-Feng Wayne Huang for authoring the original project proposal that precipitated this entire effort. We also thank our partners from production who helped us conduct tests on active production data during several previous shows; in particular, we thank Alex Nijmeh, Bernie Wong, Diana LeVangie, Kendall Litaker, Kenji Endo, Laura Grondahl, Mohit Kallianpur, and Sujil

Sukumaran. We also thank Brent Burley, Daniel Teece, Mackenzie Thompson, Charlotte Zhu, Sergio Sancho and Xianyao Zhang for their detailed feedback on and suggestions for these course notes. Finally, we are indebted to the rest of the Hyperion development team and to Disney Animation and DisneyResearch|Studios' tech leadership and managers for their support for this project.

REFERENCES

Benedikt Bitterli, Chris Wyman, Matt Pharr, Peter Shirley, Aaron Lefohn, and Wojciech Jarosz. 2020. Spatiotemporal Reservoir Sampling for Real-Time Ray Tracing with Dynamic Direct Lighting. *ACM Transactions on Graphics (Proc. of SIGGRAPH)* 39, 4, Article 148 (July 2020). doi:10.1145/3386569.3392481

Brent Burley, David Adler, Matt Jen-Yuan Chiang, Hank Driskill, Ralf Habel, Patrick Kelly, Peter Kutz, Yining Karl Li, and Dan Teece. 2018. The Design and Evolution of Disney's Hyperion Renderer. *ACM Transactions on Graphics* 37, 3, Article 33 (Aug. 2018). doi:10.1145/3182159

Matt Jen-Yuan Chiang, Peter Kutz, and Brent Burley. 2016. Practical and Controllable Subsurface Scattering for Production Path Tracing. In *ACM SIGGRAPH 2016 Talks*. Article 49. doi:10.1145/2897839.2927433

Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. 2009. GigaVoxels: Ray-Guided Streaming for Efficient and Detailed Voxel Rendering. In *Proc. of Symposium on Interactive 3D Graphics and Games (I3D '09)*. 15–22. doi:10.1145/1507149.1507152

Henrik Dahlberg, David Adler, and Jeremy Newlin. 2019. Machine-Learning Denoising in Feature Film Production. In *ACM SIGGRAPH 2019 Talks*. Article 21. doi:10.1145/3306307.3328150

Christian Eisenacher, Gregory Nichols, Andrew Selle, and Brent Burley. 2013. Sorted Deferred Shading for Production Path Tracing. *Computer Graphics Forum (Proc. of Eurographics Symposium on Rendering)* 32, 4 (July 2013), 125–132. doi:10.1111/cgf.12158

Pascal Grittmann, Ömercan Yazici, Iliyan Georgiev, and Philipp Slusallek. 2022. Efficiency-Aware Multiple Importance Sampling for Bidirectional Rendering Algorithms. *ACM Transactions on Graphics (Proc. of SIGGRAPH)* 41, 4, Article 80 (July 2022). doi:10.1145/3528223.3530126

Sebastian Herholz and Addis Dittebrandt. 2022. Intel® Open Path Guiding Library. http://www.openpgl.org.

Sebastian Herholz, Yangyang Zhao, Oskar Elek, Derek Nowrouzezahrai, Hendrik P.A. Lensch, and Jaroslav Křivánek. 2019. Volume Path Guiding Based on Zero-Variance Random Walk Theory. *ACM Transactions on Graphics (Proc. of SIGGRAPH)* 38, 3, Article 25 (June 2019). doi:10.1145/3230635

Wei-Feng Wayne Huang, Peter Kutz, Yining Karl Li, and Matt Jen-Yuan Chiang. 2021. Unbiased Emission and Scattering Importance Sampling For Heterogeneous Volumes. In *ACM SIGGRAPH 2021 Talks*. Article 3. doi:10.1145/3450623.3464644

Wenzel Jakob. 2010. Mitsuba renderer. http://www.mitsuba-renderer.org.

Peter Kutz, Ralf Habel, Yining Karl Li, and Jan Novák. 2017. Spectral and Decomposition Tracking for Rendering Heterogeneous Volumes. *ACM Transactions on Graphics (Proc. of SIGGRAPH)* 36, 4, Article 111 (July 2017). doi:10.1145/3072959.3073665

Samuli Laine and Tero Karras. 2010. Efficient Sparse Voxel Octrees. In *Proc. of Symposium on Interactive 3D Graphics and Games (I3D '10)*. 55–63. doi:10.1145/1730804.1730814

Samuli Laine, Tero Karras, and Timo Aila. 2013. Megakernels Considered Harmful: Wavefront Path Tracing on GPUs. In *Proc. of High Performance Graphics (HPG '13)*. 137–143. doi:10.1145/2492045.2492060

Mark Lee, Brian Green, Feng Xie, and Eric Tabellion. 2017. Vectorized Production Path Tracing. In *Proc. of High Performance Graphics (HPG '17)*. Article 10. doi:10.1145/3105762.3105768

Yining Karl Li, Charlotte Zhu, Gregory Nichols, Peter Kutz, Wei-Feng Wayne Huang, David Adler, Brent Burley, and Daniel Teece. 2024. Cache Points for Production-Scale Occlusion-Aware Many-Lights Sampling and Volumetric Scattering. In *Proc. of Digital Production Symposium (DigiPro '24)*. Article 6. doi:10.1145/3665320.367099

Thomas Müller. 2019. Practical Path Guiding in Production. *Path Guiding in Production, SIGGRAPH 2019 Course Notes*, Article 18 (July 2019), 37-49 pages. doi:10.1145/3305366.3328091

Thomas Müller, Markus Gross, and Jan Novák. 2017. Practical Path Guiding for Efficient Light-Transport Simulation. *Computer Graphics Forum (Proc. of Eurographics Symposium on Rendering)* 36, 4 (June 2017), 91–100. doi:10.111/cgf.13227

Matt Pharr, Wenzel Jakob, and Greg Humphreys. 2023. *Physically Based Rendering: From Theory to Implementation* (4th ed.). MIT Press. https://www.pbr-book.org/4ed

Alexander Rath, Pascal Grittman, Sebastian Herholz, Philippe Weier, and Philipp Slusallek. 2022. EARS: Efficiency-Aware Russian Roulette and Splitting. *ACM Transactions on Graphics (Proc. of SIGGRAPH)* 41, 4, Article 81 (July 2022). doi:10.1145/3528223.3530168

Alexander Rath, Pascal Grittmann, Sebastian Herholz, Petr Vévoda, Philipp Slusallek, and Jaroslav Křivánek. 2020. Variance-Aware Path Guiding. *ACM Transactions on Graphics (Proc. of SIGGRAPH)* 39, 4, Article 151 (July 2020). doi:10.1145/

3386569.3392441

Alexander Rath, Marco Manzi, Farnood Salehi, Sebastian Weiss, Tiziano Portenier, Saeed Hadadan, and Marios Papas. 2025. Neural Resampling with Optimized Candidate Allocation. In *Proc. of Eurographics Symposium on Rendering (EGSR '25)*. Article 20251181. doi:10.2312/sr.20251181

Lukas Ruppert, Sebastian Herholz, and Hendrik P. A. Lensch. 2020. Robust Fitting of Parallax-Aware Mixtures for Path Guiding. *ACM Transactions on Graphics (Proc. of SIGGRAPH)* 39, 4, Article 147 (July 2020). doi:10.1145/3386569.3392421

Justin Talbot. 2005. *Importance Resampling for Global Illumination.* Master's thesis. Brigham Young University, Provo, UT, USA. Advisor(s) Egbert, Parris. https://scholarsarchive.byu.edu/etd/663/

Eric Veach. 1998. *Robust Monte Carlo Methods for Light Transport Simulation.* Ph. D. Dissertation. Stanford University, Stanford, CA, USA. Advisor(s) Guibas, Leonidas J. https://graphics.stanford.edu/papers/veach_thesis

Thijs Vogels, Fabrice Rouselle, Brian McWilliams, Gerhard Rothlin, Alex Harvill, David Adler, Mark Meyer, and Jan Novák. 2018. Denoising with Kernel Prediction and Asymmetric Loss Functions. *ACM Transactions on Graphics (Proc. of SIGGRAPH)* 37, 4, Article 124 (Aug. 2018). doi:10.1145/3197517.3201388

Jiří Vorba and Jaroslav Křivánek. 2016. Adjoint-Driven Russian Roulette and Splitting in Light Transport Simulation. *ACM Transactions on Graphics (Proc. of SIGGRAPH)* 35, 4, Article 42 (July 2016). doi:10.1145/2897824.2925912

Kehan Xu, Sebastian Herholz, Marco Manzi, Marios Papas, and Markus Gross. 2024. Volume Scattering Probability Guiding. *ACM Transactions on Graphics (Proc. of SIGGRAPH Asia)* 43, 6, Article 184 (Dec. 2024). doi:10.1145/3687982

Bora Yalçıner and Ahmet Oğuz Akyüz. 2024. Path Guiding For Wavefront Path Tracing: A Memory Efficient Approach for GPU Path Tracers. *Computers & Graphics* 121, Article 103945 (June 2024). doi:10.1016/j.cag.2024.103945

Efficient Rendering of Caustics: Photon Guiding at Corona

MARTIN ŠIK, Chaos, Czech Republic



Fig. 1. Corona's caustics solver automatically renders directly and indirectly visible caustics without requiring the user to set any technical parameters. In these course notes, we discuss the latest improvements to our caustics solver.

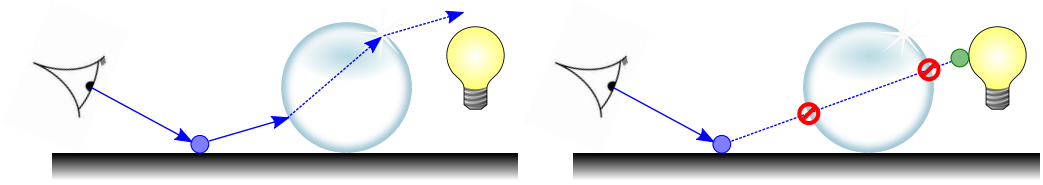
1 INTRODUCTION

In this part of the course notes, we will describe an algorithm used in the Corona Renderer[9] for efficient rendering of caustics. Although this algorithm was already presented at EGSR some time ago [Šik and Křivánek 2019a], in this course we will mainly discuss the improvements made to the solver since.

Caustics play a crucial role in enhancing the perceived realism of scenes that feature highly glossy reflective and refractive surfaces. They significantly impact rendering in various fields, including visual effects (VFX), product design, the automotive industry, and, most notably for our purposes, architectural visualization. Unfortunately, caustics are notoriously hard to render with path tracing, the most commonly used rendering algorithm in production including Corona renderer. The reason is that none of the two common sampling techniques utilized in path tracing, bidirectional scattering distribution function (BSDF) sampling and direct illumination sampling, can efficiently sample caustics paths (see Figure 2).

Several methods for rendering caustics within a Monte Carlo framework have been suggested over the years, including photon mapping [Hachisuka and Jensen 2009; Hachisuka et al. 2008; Jensen 1996], bidirectional path tracing [Lafortune and Willems 1993; Veach and Guibas 1994], and hybrid algorithms that enhance robustness by integrating both techniques [Georgiev et al. 2012; Hachisuka et al. 2012; Křivánek et al. 2014]. However, these combined solutions often introduce considerable implementation complexity and increase run-time overhead compared to simpler unidirectional path tracing. Consequently, they are seldom utilized in production rendering [Christensen et al. 2018; Křivánek et al. 2018]. For that reason, we were looking for a more lightweight solution that could be applied in Corona renderer.

[9] www.corona-renderer.com



a) BSDF sampling b) Direct illumination

Fig. 2. Neither BSDF sampling nor direct illumination sampling can efficiently sample caustic paths. a) BSDF sampling has low probability of sampling the direction that leads to the light source - once the direction from the shading point (blue) is sampled, rest of the path (dashed) is deterministic due to delta refractions. c) Direct illumination can't make the connection (dashed) from the shading point (blue) through the refractive object to the sample (green) on the light source.

1.1 Path guiding vs Corona's caustics solver

One possible solution to efficient caustics rendering is path guiding. Unlike in uninformed unidirectional path-tracer, path guiding can efficiently generate directions from the shading point that leads to a caustic path. We wanted to use path guiding in Corona to render caustics, so we would not have to use bidirectional methods with their shortcomings. Unfortunately, our experiments have shown us that path guiding was not always sufficient to handle all caustic paths. While path guiding was a significant improvement over vanilla path tracing, in many scenes the caustics were still discovered after many rendering passes, the guiding distributions were not learning fast enough, and thus the image convergence was slow. Figure 3 shows a comparison of a scene with caustics rendered using vanilla path tracer, path guiding provided by the OpenPGL library [10], and the Corona's caustics solver.



a) Path tracing (10 min) b) Path guiding (10 min) c) Our approach (10 min)

Fig. 3. A same-time comparison of renderings of the bathroom scene, which is lit from the outside and from the bulbs through glass, so all the illumination in the scene is due to caustics. On the left, vanilla path tracing completely fails to capture the illumination from caustics. In the middle, using path guiding provided by the OpenPGL library improves the result significantly. On the right, the Corona's caustics solver delivers the cleanest result.

Since we found that path guiding was not sufficient for rendering caustics, we have opted to base our caustics solver on bidirectional methods. In Section 2 we introduce the initial version of

[10] https://github.com/RenderKit/openpgl

our caustics solver and describe how we have tackled the disadvantages of bidirectional methods. In Section 3 we discuss the improvements we have made to our solver since its introduction to Corona. Section 4 covers the latest addition to our caustics solver, rendering of volumetric caustics. Finally, in Section 5 we shortly talk about future work.

2 CORONA'S CAUSTICS SOLVER - INITIAL IMPLEMENTATION

In this section we briefly introduce the original Corona's solver before we will discuss the additional improvements in the next section. For a more comprehensive overview of the initial version of our solver, see our previous work [Šik and Křivánek 2019a]. Our objective was to provide Corona Renderer users with a caustics solver that efficiently handles all types of caustics, both reflective and refractive, whether viewed directly or indirectly. To promote user-friendliness, we wanted for this solver to require minimal parameters, ideally just a single checkbox to toggle it on and off. Since our solver is based on bidirectional methods, we acknowledged that the solver will introduce some overhead compared to a standard path-tracer, but we tried to keep this overhead to a minimum.

Our implementation is based on the bidirectional algorithm known as Vertex Connection and Merging (VCM) [Georgiev et al. 2012; Hachisuka et al. 2012]. VCM combines the path sampling strategies of bidirectional path tracing with radiance estimates from photon mapping [Hachisuka and Jensen 2009; Hachisuka et al. 2008; Jensen 1996]. As a path is traced from the camera, photon map lookups are performed at each camera path vertex. This photon mapping component is crucial for accurately rendering indirectly visible caustics, such as those found at the bottom of a swimming pool. The full VCM algorithm effectively handles all types of caustics and provides a strong foundation for our work. Unfortunately, the baseline VCM has three main issues:

Computational cost. VCM can be several times slower than path tracing. It has to trace paths from both the lights and the camera, calculate weights for their combinations, and perform photon lookups. In scenes where path tracing works well, using VCM is overkill.

Photon guiding. If photon paths are not guided toward important parts of the scene, convergence can be very slow. This is a major issue in indoor architectural scenes, where skylight enters through small openings like narrow windows.

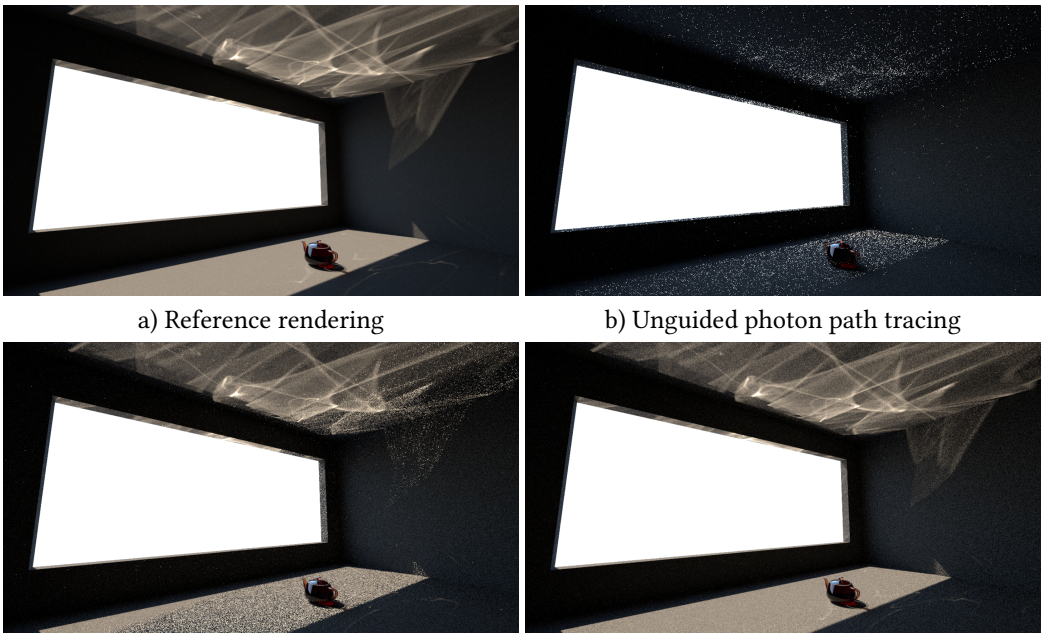
Number of photon paths. Users usually have to tweak the number of photon paths traced in each algorithm iteration, and this interferes with good usability.

In the following text, we discuss how we addressed these issues in our caustics solver.

2.1 Simplified Vertex Connection And Merging

To reduce overhead as much as possible, we removed all bidirectional connection path sampling techniques from VCM. Without these techniques, we can still efficiently render all types of caustics. Furthermore, we do not use pure light tracing (e.g. paths from the light sources connected to the camera). While this technique is effective at handling directly visible caustics, it also requires additional buffer for each render element (aka render pass), which can be prohibitively memory expensive for large image resolutions. The only techniques that remain are original path tracing techniques and photon lookups.

We further limit where we utilize photon lookups. We don't perform photon lookups for paths that have high enough probability with path tracing techniques (an idea borrowed from Lightweight Photon Mapping [Grittmann et al. 2018]). We also store photons only after interaction with highly glossy materials, and we stop tracing photon paths after two consecutive diffuse bounces.



a) Reference rendering b) Unguided photon path tracing

c) Adaptive emission map d) Our Metropolis guiding

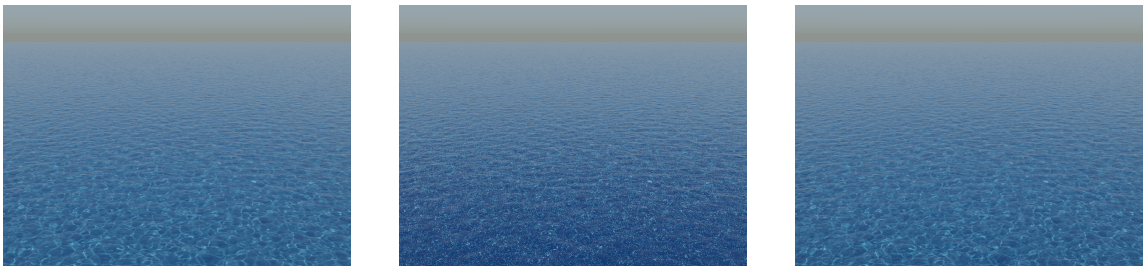
Fig. 4. Photon path guiding. a) Reference rendering. b) No photon path guiding. c) Guiding based on an adaptive emission map [Grittmann et al. 2018]. d) Our Metropolis-based guiding.

Finally, we modify the emission of photons from environment illumination maps, so that only high-frequency environment illumination is considered. This also means that we do not trace any photons from uniform environment maps.

2.2 Photon guiding

To guide photons we have tried both adaptive emission of photons [Grittmann et al. 2018] and Markov chain Monte Carlo (MCMC) [Metropolis et al. 1953; Šik and Křivánek 2019b]. MCMC generates samples by randomly searching for a sample with a high value of *target function* and once it finds it, it will slightly modify (or 'mutate') this sample to create several different samples of high value. More specifically, new samples are proposed by mutation kernels based on the current sample, and a proposed sample is then randomly accepted based on its target function value. In the case of rejection, the current sample is repeated. Figure 4 shows the comparison of the two different guiding approaches.

In our experiments, MCMC could even in difficult scenes find the important caustic paths more quickly compared to the adaptive emission, and thus we have decided to use it in our solver. For the target function that drives the sampling, we have chosen a combination of visibility [Hachisuka and Jensen 2011] and the inverse of the square distance to the camera. The visibility is a function stored in a spatial subdivision structure that has a value of one in places where previous photon lookup was successful and zero otherwise (initially we set it to one in places where camera paths have their vertices). In this way, we ensure that photons are generated in the important regions of the scene. Including the inverse of the square distance to the camera ensures that we generate more photons closer to the camera and thus efficiently handle scenes with large caustics (see Figure 5).



a) Reference rendering b) Binary visibility Metropolis sampling c) Our solution

Fig. 5. Handling of large caustics. a) Reference. b) Binary visibility based importance produces suboptimal photon distribution: too many photons are concentrated far away from the camera, while the region near camera is underpopulated. c) Our solution modulates the importance by the inverse squared distance from the camera, producing a more equalized photon distribution.

2.3 Number of photon paths

Originally, we addressed the issue of determining the number of photon paths by detecting for how many pixels are photons useful - i.e. they contribution is significant for the given pixel. We then took this number multiplied by an experimentally verified constant and got the number of photon paths. This requires some additional handling for render regions; see our previous work [Šik and Křivánek 2019a]. Unfortunately, this is not always enough, and thus we have developed a more robust method (see Section 3.3).

2.4 Additional details

To further improve how our caustics solver worked in cases of multiple light sources, we have developed an adaptive algorithm for the selection of photon emitters based on their caustic contribution (see our previous work [Šik and Křivánek 2019a] for more details). Although this algorithm worked well in many cases, we have since improved upon it as discussed in the next section.

Another important detail is how we handle motion blur and dispersion. In the scene shown in Figure 6, a moving car is being tracked by the camera. To correctly render caustics with motion blur, we randomly choose separate times for the camera path and the photon path. During photon lookup, we only use photons with a time similar to the camera path. Dispersion uses the same approach; each photon has a selected wavelength, and a photon lookup then considers only photons with wavelengths similar to the wavelength of the camera path. Note that using motion blur and/or



a) Incorrectly handled motion blur b) Correct motion blur c) Dispersion

Fig. 6. Handling of motion blur/dispersion in our caustics solver. a) Wrong solution, which smears caustics reflected from a car tracked by the camera. b) Correct solution that takes into account time associated with both a camera path and a photon path. c) We handle dispersion in similar fashion to motion blur. Image courtesy of Rakete GmbH Munich.

dispersion in the scene means that many photons will not be used, which limits the efficiency of our old automatic detection of the number of photon paths, and a user often has to manually tweak the number of photon paths.

3 CAUSTICS SOLVER IMPROVEMENTS

In this section we discuss various changes that we applied to our caustics solver since its initial release. Their main goal is to improve the convergence of caustics rendering.

3.1 Light sources stratification

The first improvement we made to our original solver was to improve light sources stratification. Although the previous version of our solver had an algorithm for adaptive sampling of light sources, the fact that it was built on top of a Markov chain severely decreased its efficiency. It often happened that the chain got "stuck" (i.e. was repeating the same sample or generating samples in its small neighborhood) and kept generating photon paths just from the most caustic-contributing light source.

To avoid this issue, we started to use multiple Markov chains instead of one. Each chain generates photon paths just from a given light source or a group of light sources. The grouping is based on the light sources hierarchical structure used for direct illumination sampling in Corona (see [Vévoda et al. 2018] for more details). This allows us to use ordinary stratified samples to select light sources for photon emission. As in the original version of the solver, light sources are sampled adaptively based on their caustic contribution, but this time outside the MCMC framework. Once a light source is sampled, the exact photon emission position and direction (as well as following photon scattering directions) are generated using the associated Markov chain. Figure 7 demonstrates this process.

Using separate Markov chains for light source groups means that we can also improve MCMC adaptivity. As stated above, MCMC generates samples by proposing mutations of the current sample and accepting/rejecting the proposal. The proposal/mutations kernel can be adapted based on the acceptance ratio - this allows us to increase or decrease the ratio based on all the previously generated samples and achieve an optimal acceptance rate [Haario et al. 2001]. However, adapting single Markov chain that generates photon paths from many different light sources is not accurate enough - each light source might need a differently tuned mutation kernel. Using a separate Markov chain for each light source group allows us to adapt each chain more precisely.

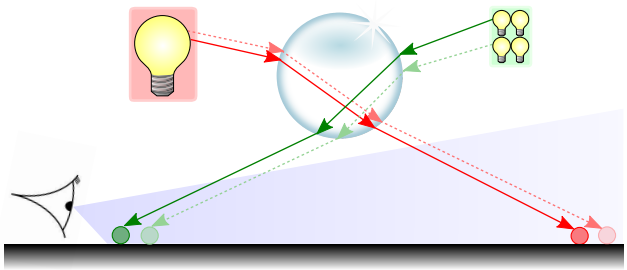


Fig. 7. In this schematic view we have two light groups (red, green). Our algorithm starts by sampling one of the light groups and then the associated Markov chain is mutated. Its current sample (full line) is mutated using a kernel adapted to this light group and a new proposal is generated (dotted line). Since each light group has its own chain, each chain can focus on caustics generated by a single light group and thus the whole state space is better explored.



a) Old solver (30 min) b) New solver (30 min)

Fig. 8. A same-time comparison of renderings of the room scene with caustics formed on the back wall by 3 light sources and by the light coming from the outside through the glass bricks. The photon paths generated from the internal light sources are almost always accepted, however, photon paths generated by the outside light source (environment) have much lower acceptance rate, their require more guiding. Due to this disproportion a single-chain algorithm (a) will generate too many samples for the internal light sources and not enough for the external one. By using multiple chains we can mitigate this issue (b).

We compare our previous version of the solver (with a single chain) and the new multi chain variant in Figure 8. Our multi chain variant avoids the issue of generating too few samples from an important light source that has lower acceptance rate compared to other light sources.

3.2 Adaptive emission

While using multiple chains allows us to better focus on each light group, there is still one source of inefficiency. MCMC typically uses two types of mutations: small and large steps. The small step mutates the previous sample using the adapted mutation kernel, its goal being to better explore the caustic locally. The large step then generates a sample independent of the previous sample to handle discovery of potential new contributing parts of the state space. While the small step adapts during the rendering, the large step remains the same and does not learn from previous samples.

To address this inefficiency, we combine adaptive emission with MCMC sampling. More specifically, once we select a Markov chain (i.e. sample a light source group), we use it to generate two 2D random numbers (ξ_p, ξ_d) - these would be previously used to sample a photon position on the light source (group) and its direction. Now, instead of using these directly to generate position/direction, we first transform (ξ_p, ξ_d) into (ξ'_p, ξ'_d) using our new adaptive emission tree.

Building the emission tree. The emission tree divides the four-dimensional space of positions and directions. At each tree depth, we divide one of the dimensions into two halves. The split dimension depends solely on the depth (so at depth = 1 the first dimension is split, at depth = 2 the second dimension is split, and so on). At the rendering start, only the root node exists that contains the whole 4D space. We then continuously split the nodes after they have accumulated enough contributing photon paths (64 in the current implementation). We say that a photon path is accumulated in a node if it contributes to the image and its transformed sample (ξ'_p, ξ'_d) lies in this node. Figure 9 illustrates the building of the emission tree.

Sampling from the emission tree. For sampling purposes, each node also stores its *node value*. Given a set of all the photon paths accumulated in a node, *node value* is computed as the sum over this set of inverse probabilities $P(\xi'_p, \xi'_d)$ of generating the final sample (ξ'_p, ξ'_d). When sampling from the emission tree, we start at the root of the tree and scale its child nodes based on their *node*

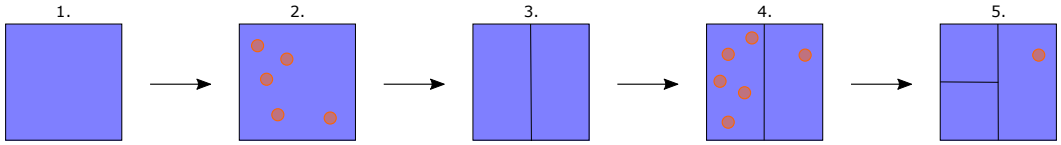


Fig. 9. Here we show the update of the adaptive emission tree, for illustration purpose only in 2D instead of 4D. The tree starts as a single root node (1), when it accumulates enough photon paths (2) it splits in the middle into two nodes (3). After more photon paths have been accumulated (4), the node with enough photon paths is split again (5) - this time according to the other dimension. A node always remember its number of photon paths until it is split.

value. We select the child node in which the random number (ξ_p, ξ_d) falls and continue until we reach a leaf node. Once in the leaf node, we generate the final (ξ'_p, ξ'_d) uniformly inside this leaf node. The simplified illustration of the sampling process is shown in Figure 10.

Preventing chain getting stuck. It might happen that the distribution of positions/directions contains high frequencies (with one or two strong peaks), which may cause the MCMC chain to get stuck in these peaks. To avoid this and promote better state space exploration, we modify the nodes in the emission tree to have a given minimum probability of sampling - therefore ensuring each point in the state space has non-zero probability of being sampled. Furthermore, when accumulating the contributing photon paths to nodes we slightly jitter their transformed sample (ξ'_p, ξ'_d) and thus we might accumulate them to neighboring node instead. This smoothens the distribution of positions/directions.

The results of this improvement are shown in Figure 11. We can see that adding the emission tree further improves the guiding of the photons to the important regions of the scene.

3.3 Photon count estimation

We have further improved how we estimate the ideal number of emitted photons. We start with an initial value of the emitted photons N and once we have rendered the first pass containing photon lookups, we compute a new photon count estimate.

First, we determine which pixels have a significant contribution from photon lookups. We do this by comparing the contribution from path tracing and contribution from photon lookups in each pixel. When photon lookups contribute to at least 1% of the total pixel value, we consider the contribution significant. We compute an image mask C based on this information and blur it to increase robustness.

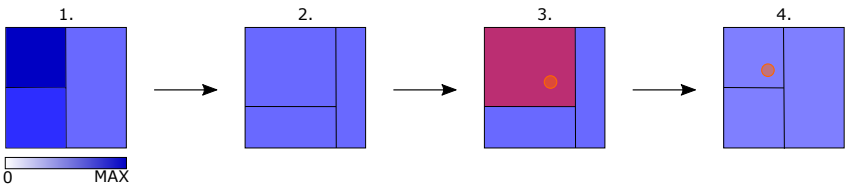
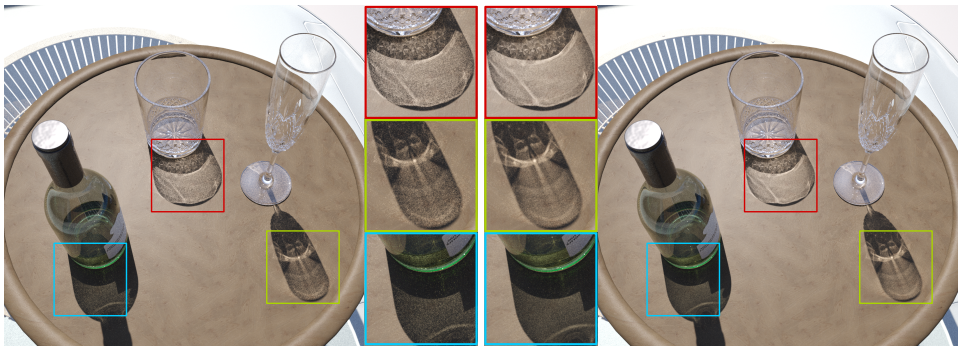


Fig. 10. Here we show the process of sampling a photon direction and position from the adaptive emission tree, for illustration purpose only in 2D instead of 4D. The probability of generating new sample in a given tree node is given by its *node value* - here displayed in the shades of blue (1). One can assume the nodes are scaled based on their value (2). Given an initial sample we determine in which scaled node it lies (3). The final sample is computed by projecting the initial sample into the unscaled node (4).



a) Without adaptive emission tree (10 min) b) With adaptive emission tree (10 min)

Fig. 11. A same-time comparison of renderings of the glasses scene. The adaptive emission tree makes are photon guiding more efficient and thus the result of our new caustics solver (b) is less noisy compared to the old version (a).

For every pixel p in the mask C, we compute the value of the number of photons actually used in the pixel's photon lookups (excluding those discarded by motion blur/dispersion radius) divided by the number of the pixel's photon lookups. This ratio R_p is the number of photons used per one photon lookup. We further clamp this value to avoid pixels with too many photons to influence our estimation.

We can then compute the photon count scaling as the ratio of pixels in the mask C divided by the sum of all R_p. The new number of emitted photons N' is computed as follows:

$$N' = P\frac{N|C|}{\sum_{p\in C}\min\left(R_p, P\right)} \tag{26}$$

where P is the ideal number of photons per photon lookup. We set it by default to 1.5. Figure 12 demonstrates the entire estimation process.

We compare the old photon count estimation and the new estimation in Figure 13. Especially in the scenes with motion blur or dispersion (as in the Figure 13) the newer algorithm is able to estimate the photon count better and thus allow faster caustics convergence in the same amount of time.

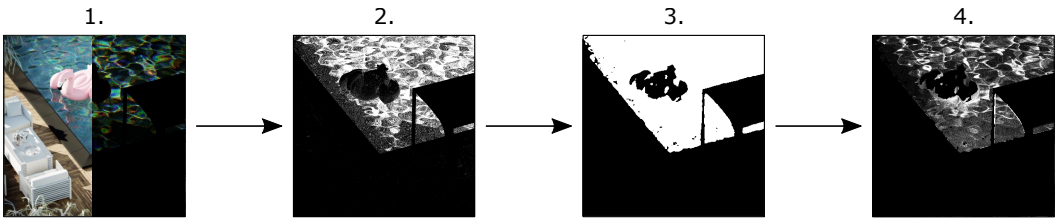
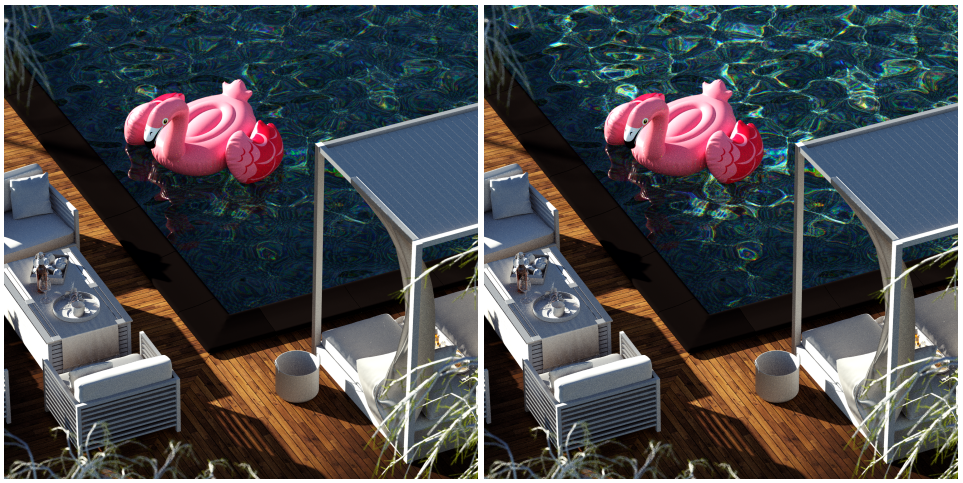


Fig. 12. We estimate the number of photons by taking the beauty image (1-left), photon look-up contribution (1-right) and computing a scaled difference between photon lookups contribution and path tracing contribution (2), where photon lookups are multiplied by 100. From the contribution difference we compute the mask C (3). Finally for every pixel in the mask, we compute the number of photons per photon lookup (4). From this we can estimate how much more photons we have to trace using Equation 26.



a) Old solver (8 min) b) New solver (8 min)

Fig. 13. A same-time comparison of renderings of the pool scene with dispersive caustics on the bottom of the pool. Our new photon count estimation can more accurately predict how many photons are necessary (especially in scenes with dispersion/motion blur) and thus the rendering has correct bright caustics, while the old solver underestimates the number of photons and the caustics are not properly resolved.

3.4 Misc

Besides the improvements that influence how fast can our new solver resolve caustics, we have added other features that allow the users to modify the results as their require.

Repetitions limit. If the Markov chain gets stuck, i.e. it repeats the same sample/photon path several times, we only store the contribution of given number of photon repetitions. To avoid energy loss in the whole image, we redistribute the contribution of the over-limit repetitions to the following generated photon paths. This allows the users to limit the fireflies without darkening the image.

Caustic multiplier. The users often want to tweak the strength of the caustics without changing the rest of the scene, for that purpose, we have added a multiplier that changes the intensity of caustics.

Light/receivers selection. We also allow users to select only a subset of lights in the scene that will generate caustics. In the same way, they can exclude some objects from receiving caustics. Therefore, users can easily remove unwanted caustics and focus the solver on the caustics they want to present in their images.

Render passes. The caustics can be also stored in separate render passes based on whether they are created by reflection or refraction.

4 VOLUMETRIC CAUSTICS

The first published version of our caustics solver could only resolve caustics on surfaces. However, caustics in volumes can significantly improve perceived realism in scenes, as demonstrated in Figure 14. In our latest version, we have added support for volumetric caustics. In this section, we discuss the changes to our solver needed to support volume caustics.



a) Pool b) Car

Fig. 14. Example renderings containing volumetric caustics rendered using our caustics solver.

4.1 Beam radiance estimator

Photon lookups in volumes can be performed using various estimators. They differ by their variance in media with different densities and by their computational overhead [Deng et al. 2019; Jarosz et al. 2011, 2008] . One can use a single estimator or combine multiple estimators [Křivánek et al. 2014]. For more information on this topic, see other sources [Novák et al. 2018].

From the variety of options, we have chosen a simple beam radiance estimator [Jarosz et al. 2008]. Although other estimators offer lower variance, they have higher complexity and are more computationally expensive (especially in heterogeneous volumes, since they require recomputing attenuation along photon paths). The beam radiance estimator works by collecting all volume photons that are found in the volume at a given distance from a segment of a camera path. Note that volume photons are the vertices of the photon path that correspond to scattering events inside a volume. The photons corresponding to scattering events on a surface are ignored by the beam estimator. Figure 15 demonstrates this estimator.

For surface photon lookups we compute the lookup radius based on ray-differentials [Igehy 1999]. For our beam estimator, we follow the same logic. Instead of cylindrical beams with fixed radius, we trace conical beams where the radius increases according to the ray differentials. Since the photon paths are accepted based on the inverse of the squared distance to the camera, using such conical beams allows us to reduce variance at the regions of the scene further from the camera.

We have also experimented with so-called "short" beams, where we do not collect photons along the whole segment in the volume, but instead sample a maximum distance based on the transmittance at which we stop collecting photons. However, the speed up was not sufficient given the increased variance.

4.2 Volume photons.

We trace volumetric photons together with surface photons, however we store them in a separate KD-tree, which we have optimized for a conical ray queries. To limit the number of volume photons, we only store volume photons at the first scatter event inside a volume (and if there has been a preceding vertex at a glossy surface). After that we terminate the photon path.

For guiding of the photon paths we use a target function that is a maximum of target function values at each photon generated by a single photon path. For volumetric photons, we use a separate data structure to store *photon visibility*. While we use an octree to store surface photons' visibility,

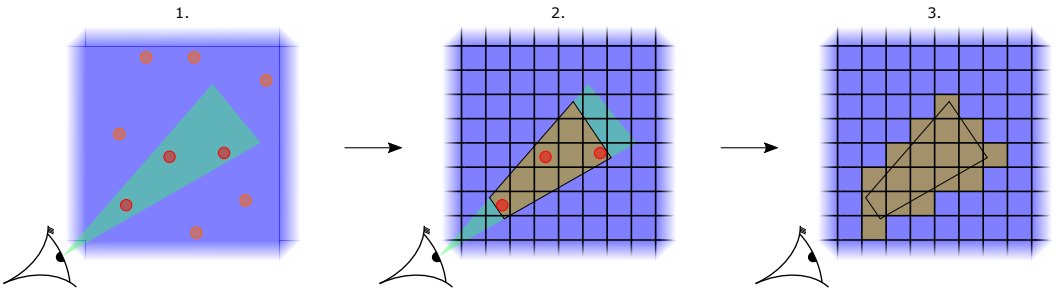


Fig. 15. 1. Beam radiance estimator - the photons (red) in the conical beam shot from the camera are considered in the estimator. 2. To upgrade the *photon visibility* voxel grid we consider the segment of the beam between the first and the last contributing photon. 3. Voxel cells intersecting the segment of the beam are marked - volume photons in these voxels will have non-zero target function.

we have found out that such a structure is inefficient for volumetric photons. The issue is updating the structure based on conical queries, which is quite computationally expensive. Therefore, we have decided to use a voxel grid instead. Figure 15 shows how we update the volumetric photons' visibility. As in the previous version of our caustics solver, we compute the target function value for each photon by multiplying the photon visibility with the inverse of the squared distance from the camera.

5 SUMMARY AND FUTURE WORK

In this part of the course we have presented our solution to resolving caustics in Corona renderer. Our solver is based on photon lookups, since we have found it to be more efficient than path guiding. Since our initial implementation presented at EGSR [Šik and Křivánek 2019a], we have improved our caustics solver by

- Improving stratification over light sources by introducing multiple chains.
- Adding adaptive emission on top of the MCMC guiding of photons.
- Making a more robust photon count estimation.
- Adding support for volumetric caustics.

Although we have managed to significantly improve our caustics solver, it still has few limitations that we would like to address in the future.

Render regions. Our users often use render regions, rendering just part of the image. This does not work well with our caustics solver, which usually needs to trace many photons even for small regions containing caustics. While we try to amortize the cost by reusing the same set of photons for multiple rendering passes, rendering regions with caustics is still inefficient.

Non-uniform convergence. While our guiding can deliver photons to the important regions in the scene, some caustics still receive more photons than others and thus converge faster. We have experimented with modifying our target function to include the information about the number of photons per merge lookup (see Section 3.3), however, so far we did not manage to improve the uniformness.

Low-density volumes. Our volumetric estimator is quite efficient in handling volumes with medium or higher density; however, its efficiency drops in sparse volumes. We would like to try combining it with different estimators to improve the solver efficiency.

REFERENCES

Per Christensen, Julian Fong, Jonathan Shade, Wayne Wooten, Brenden Schubert, Andrew Kensler, Stephen Friedman, Charlie Kilpatrick, Cliff Ramshaw, Marc Bannister, Brenton Rayner, Jonathan Brouillat, and Max Liani. 2018. Renderman: An advanced path-tracing architecture for movie rendering. *ACM Transactions on Graphics* 37, 3, Article 30 (July 2018). doi:10.1145/3182162

Xi Deng, Shaojie Jiao, Benedikt Bitterli, and Wojciech Jarosz. 2019. Photon Surfaces for Robust, Unbiased Volumetric Density Estimation. *ACM Transactions on Graphics (Proc. of SIGGRAPH)* 38, 4, Article 46 (July 2019). doi:10.1145/3306346.3323041

Iliyan Georgiev, Jaroslav Křivánek, Tomáš Davidovič, and Philipp Slusallek. 2012. Light transport simulation with vertex connection and merging. *ACM Transactions on Graphics (Proc. of SIGGRAPH Asia)* 31, 6, Article 192 (Nov. 2012). doi:10.1145/2366145.2366211

Pascal Grittmann, Arsène Pérard-Gayot, Philipp Slusallek, and Jaroslav Křivánek. 2018. Efficient Caustic Rendering with Lightweight Photon Mapping. *Computer Graphics Forum (Proc. of Eurographics Symposium on Rendering)* 37, 4 (July 2018), 133–142. doi:10.1111/cgf.13481

Heikki Haario, Eero Saksman, and Johanna Tamminen. 2001. An adaptive Metropolis algorithm. *Bernoulli* 7, 2 (April 2001), 223–242. doi:10.2307/3318737

Toshiya Hachisuka and Henrik W. Jensen. 2009. Stochastic Progressive Photon Mapping. *ACM Transactions on Graphics (Proc. of SIGGRAPH Asia)* 28, 5, Article 141 (Dec. 2009). doi:10.1145/1618452.1618487

Toshiya Hachisuka and Henrik W. Jensen. 2011. Robust Adaptive Photon Tracing Using Photon Path Visibility. *ACM Transactions on Graphics* 30, 5, Article 114 (Oct. 2011). doi:10.1145/2019627.2019633

Toshiya Hachisuka, Shinji Ogaki, and Henrik W. Jensen. 2008. Progressive Photon Mapping. *ACM Transactions on Graphics (Proc. of SIGGRAPH Asia)* 27, 5, Article 130 (Dec. 2008). doi:10.1145/1409060.1409083

Toshiya Hachisuka, Jacopo Pantaleoni, and Henrik W. Jensen. 2012. A path space extension for robust light transport simulation. *ACM Transactions on Graphics (Proc. of SIGGRAPH Asia)* 31, 6, Article 191 (Nov. 2012). doi:10.1145/2366145.2366210

Homan Igehy. 1999. Tracing ray differentials. In *Proc. of SIGGRAPH (SIGGRAPH '99)*. 179–186. doi:10.1145/311535.311555

Wojciech Jarosz, Derek Nowrouzezahrai, Robert Thomas, Peter-Pike Sloan, and Matthias Zwicker. 2011. Progressive Photon Beams. *ACM Transactions on Graphics (Proc. of SIGGRAPH Asia)* 30, 6, Article 181 (Dec. 2011). doi:10/fn5xzj

Wojciech Jarosz, Matthias Zwicker, and Henrik Wann Jensen. 2008. The Beam Radiance Estimate for Volumetric Photon Mapping. *Computer Graphics Forum (Proc. of Eurographics)* 27, 2 (April 2008), 557–566. doi:10/bjsfsx

Henrik Wann Jensen. 1996. Global Illumination using Photon Maps. In *Rendering Techniques (Proc. of Eurographics Workshop on Rendering)*. 21–30. doi:10.1007/978-3-7091-7484-5_3

Jaroslav Křivánek, Iliyan Georgiev, Toshiya Hachisuka, Petr Vévoda, Martin Šik, Derek Nowrouzezahrai, and Wojciech Jarosz. 2014. Unifying points, beams, and paths in volumetric light transport simulation. *ACM Transactions on Graphics (Proc. of SIGGRAPH)* 33, 4, Article 103 (July 2014). doi:10.1145/2601097.2601219

Jaroslav Křivánek, Ondřej Karlík, Vladimir Koylazov, Henrik Wann Jensen, Thomas Ludwig, and Christophe Chevallier. 2018. Realistic Rendering in Architecture and Product Visualization. In *ACM SIGGRAPH 2018 Courses*. Article 10. doi:10.1145/3214834.3214872

Eric P. Lafortune and Yves D. Willems. 1993. Bi-Directional Path Tracing. In *Proc. of Compugraphics (Compugraphics '93)*. 145–153.

Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. 1953. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics* 21, 6 (June 1953), 1087–1092. doi:10.1063/1.1699114

Jan Novák, Iliyan Georgiev, Johannes Hanika, and Wojciech Jarosz. 2018. Monte Carlo methods for volumetric light transport simulation. *Computer Graphics Forum (Proc. of Eurographics - State of the Art Reports)* 37, 2 (May 2018), 551–576. doi:10/gd2jqq

Martin Šik and Jaroslav Křivánek. 2019a. Implementing One-Click Caustics in Corona Renderer. In *Eurographics Symposium on Rendering - DL-only and Industry Track (EGSR '19)*, Tamy Boubekeur and Pradeep Sen (Eds.). 61–67. doi:10.2312/sr.20191221

Martin Šik and Jaroslav Křivánek. 2019b. Survey of Markov Chain Monte Carlo Methods in Light Transport Simulation. *IEEE Transactions on Visualization and Computer Graphics* 26, 4 (April 2019), 1821–1840. doi:10.1109/TVCG.2018.2880455

Eric Veach and Leonidas Guibas. 1994. Bidirectional Estimators for Light Transport. In *Rendering Techniques (Proc. of Eurographics Workshop on Rendering)*. 147–162. doi:10.1007/978-3-642-87825-1_11

Petr Vévoda, Ivo Kondapaneni, and Jaroslav Křivánek. 2018. Bayesian online regression for adaptive direct illumination sampling. *ACM Transactions on Graphics (Proc. of SIGGRAPH)* 37, 4, Article 125 (July 2018). doi:10.1145/3197517.3201340

Received 20 February 2025; revised 12 March 2025; accepted 5 June 2025