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

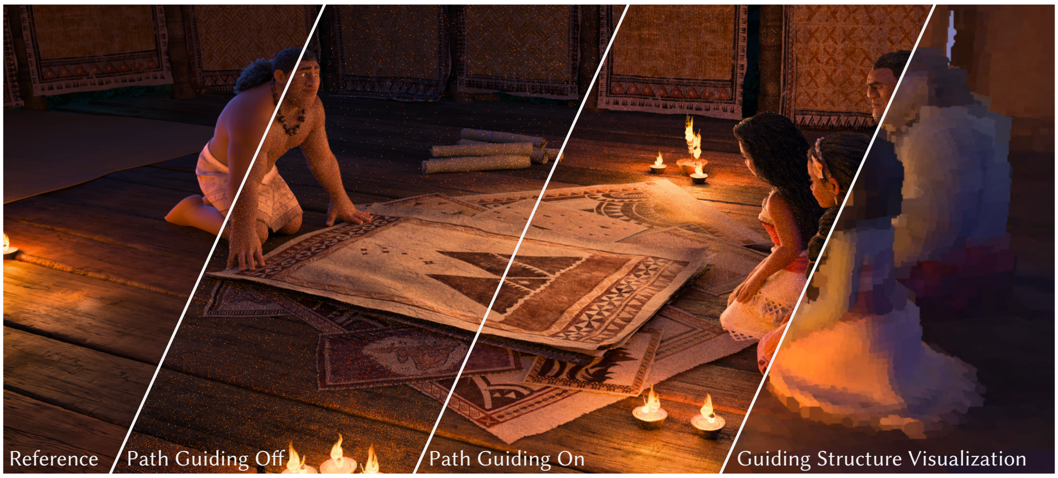


Fig. 1. A production scene from *Moana 2*, rendered using path guiding in Disney's Hyperion Renderer. From left to right: reference baseline, 64 SPP without path guiding, 64 SPP with path guiding, and visualization of the path guiding spatio-directional field at 256 SPP. © 2025 Disney

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.

Authors' Contact Information: Lea Reichardt, lea.reichardt@disneyanimation.com, Walt Disney Animation Studios, Vancouver, Canada; 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; Marco Manzi, marco.manzi@disneyresearch.com, DisneyResearch|Studios, Zürich, Switzerland.

Contents

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

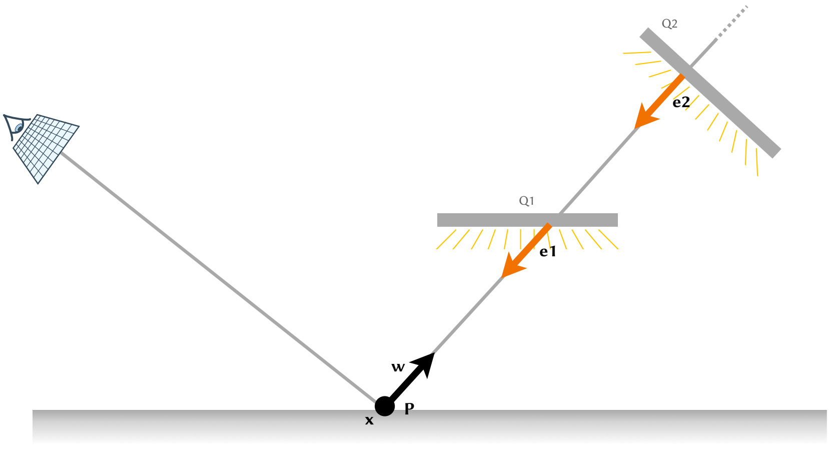


Fig. 2. Simple direct lighting setup. The goal is to compute the incident radiance at path vertex **P** at location **x** in direction **w** due to two virtual lights Q1 and Q2. **e1** and **e2** are the emission values at the light intersection points in direction -**w**.

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 **e1**
- Wave 2: $\mathbf{P_{ray}}$ hits Q2 producing emission **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 **e1**: record $\mathbf{P_{rad}} = \frac{\mathbf{e1}}{\mathbf{P_{tp}}}$
- Wave 2: $\mathbf{P_{ray}}$ hits Q2 producing emission **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 **P**, both of which are likely too small. The correct, single training value to record for this path is

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

This means we really can't record any value for **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 **e1**: record nothing, but "remember" **e1**
- Wave 2: $\mathbf{P_{ray}}$ hits Q2 producing emission **e2**: record nothing, but "remember" **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)

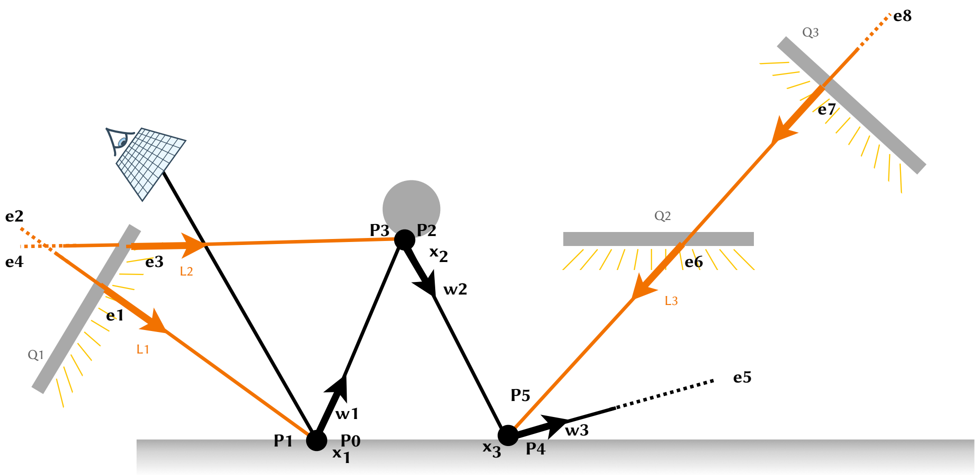


Fig. 3. A more complex lighting setup with direct and indirect illumination. We show a single path that includes both NEE rays (orange) and path continuation rays (black). The goal is to compute the full incident radiance at each path vertex **P0** - **P5**.

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 $\mathbf{P1}_{\text{rad}} = \frac{\mathbf{e1+e2}}{\mathbf{P1}_{\text{tp}}}$
  $\mathbf{P2}_{\text{ray}}$ 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 $\mathbf{P3}_{\text{rad}} = \frac{\mathbf{e3+e4}}{\mathbf{P3}_{\text{tp}}}$
  $\mathbf{P4}_{\text{ray}}$ 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 $\mathbf{P5}_{\text{rad}} = \frac{\mathbf{e6+e7+e8}}{\mathbf{P5}_{\text{tp}}}$
  record $\mathbf{P2}_{\text{rad}} = \frac{\mathbf{e5+e6+e7+e8}}{\mathbf{P2}_{\text{tp}}}$
  record $\mathbf{P0}_{\text{rad}} = \frac{\mathbf{e3+e4+e5+e6+e7+e8}}{\mathbf{P0}_{\text{tp}}}$

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*.

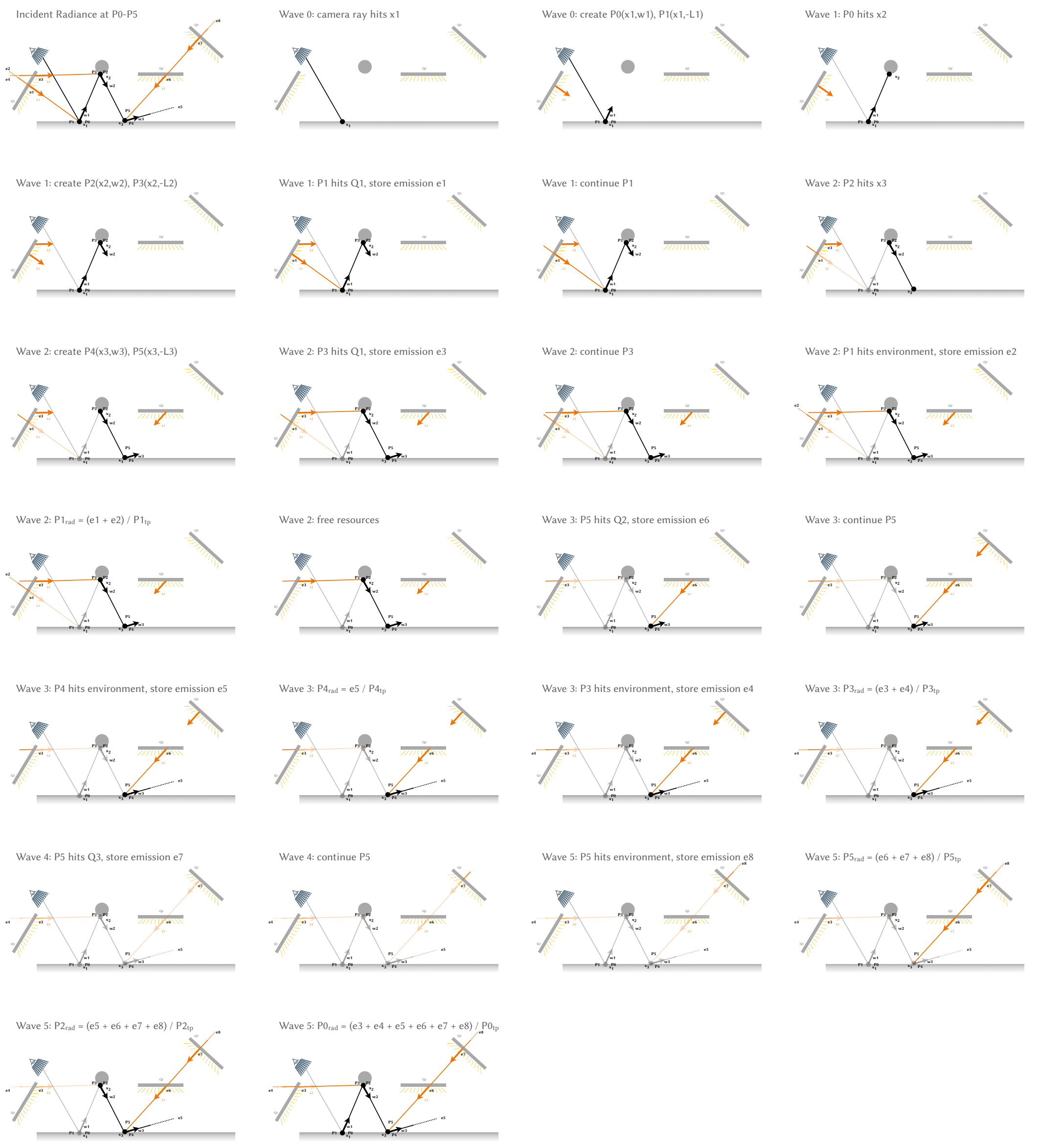

Incident Radiance at P0-P5
Wave 0: camera ray hits x1
Wave 0: create P0(x1,w1), P1(x1,-L1)
Wave 1: P0 hits x2
Wave 1: create P2(x2,w2), P3(x2,-L2)
Wave 1: P1 hits Q1, store emission e1
Wave 1: continue P1
Wave 2: P2 hits x3
Wave 2: create P4(x3,w3), P5(x3,-L3)
Wave 2: P3 hits Q1, store emission e3
Wave 2: continue P3
Wave 2: P1 hits environment, store emission e2
Wave 2: $P1_{rad}$ = (e1 + e2) / $P1_{tp}$
Wave 2: free resources
Wave 3: P5 hits Q2, store emission e6
Wave 3: continue P5
Wave 3: P4 hits environment, store emission e5
Wave 3: $P4_{rad}$ = e5 / $P4_{tp}$
Wave 3: P3 hits environment, store emission e4
Wave 3: $P3_{rad}$ = (e3 + e4) / $P3_{tp}$
Wave 4: P5 hits Q3, store emission e7
Wave 4: continue P5
Wave 5: P5 hits environment, store emission e8
Wave 5: $P5_{rad}$ = (e6 + e7 + e8) / $P5_{tp}$
Wave 5: $P2_{rad}$ = (e5 + e6 + e7 + e8) / $P2_{tp}$
Wave 5: $P0_{rad}$ = (e3 + e4 + e5 + e6 + e7 + e8) / $P0_{tp}$

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.

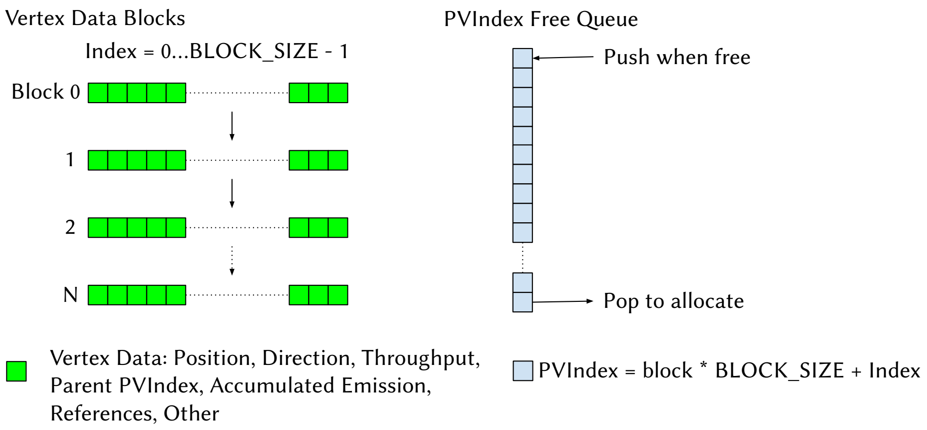

Vertex Data Blocks
Index = 0...BLOCK_SIZE - 1
Block 0
1
2
N
PVIndex Free Queue
Push when free
Pop to allocate
Vertex Data: Position, Direction, Throughput, Parent PVIndex, Accumulated Emission, References, Other
PVIndex = block * BLOCK_SIZE + Index

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:

```
1:  Results results ← shade_scatter_event(rayData)
2:  PVIndex pv ← rayData.getPV()
3:  if results.haveEmission() then
4:      pv.addEmission(results.emission())
5:  for all newRayData in results.scattered_or_nee_ray() do
6:      // Increment the reference count on pv
7:      // childPV reference count is 1
8:      PVIndex childPV ← pv.newPV(newRayData)
9:      newRayData.setPV(childPV)
10: if results.continueRay() then
11:     pv.continue() // Increment reference count on pv
12: 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:

```
1:  function BackpropagateRecursive(pv)
2:      Assert (pv.references == 0)
3:      incidentRadiance ← pv.emission / pv.throughput
4:      RecordTrainingSample(pv, incidentRadiance)
5:      parentPV ← pv.parent()
6:      if parentPV.notCamera() then
7:          parentPV.addEmission(pv.emission)
8:          Decrement(parentPV.references)
9:          if parentPV.references == 0 then
10:             BackpropagateRecursive(parentPV)
11:     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.

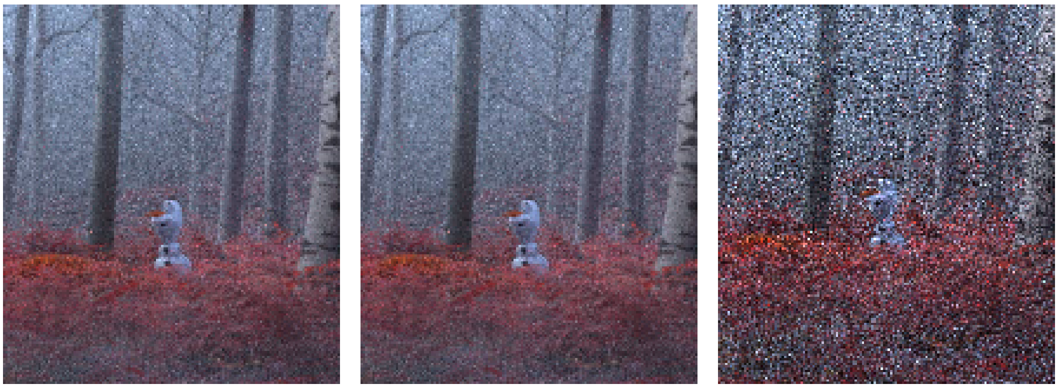

(a) Rendered image (b) Debug output (no skipping) (c) Debug output (skipping 90%)

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.

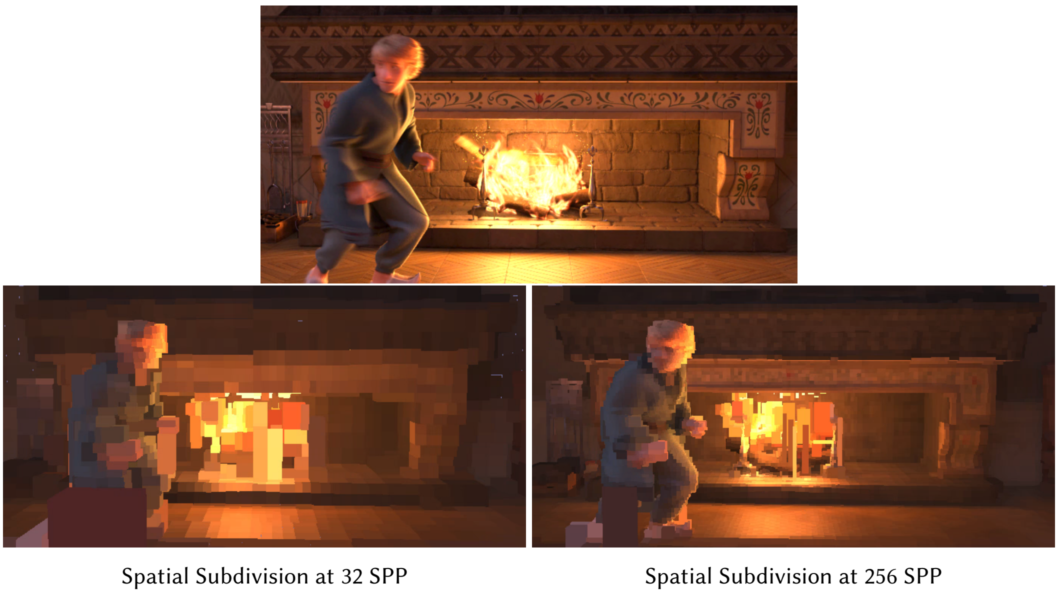


Fig. 7. A rendering from *Frozen 2* and the corresponding voxel representation of the spatial subdivision at 32 and 256 samples per pixel. © 2025 Disney

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

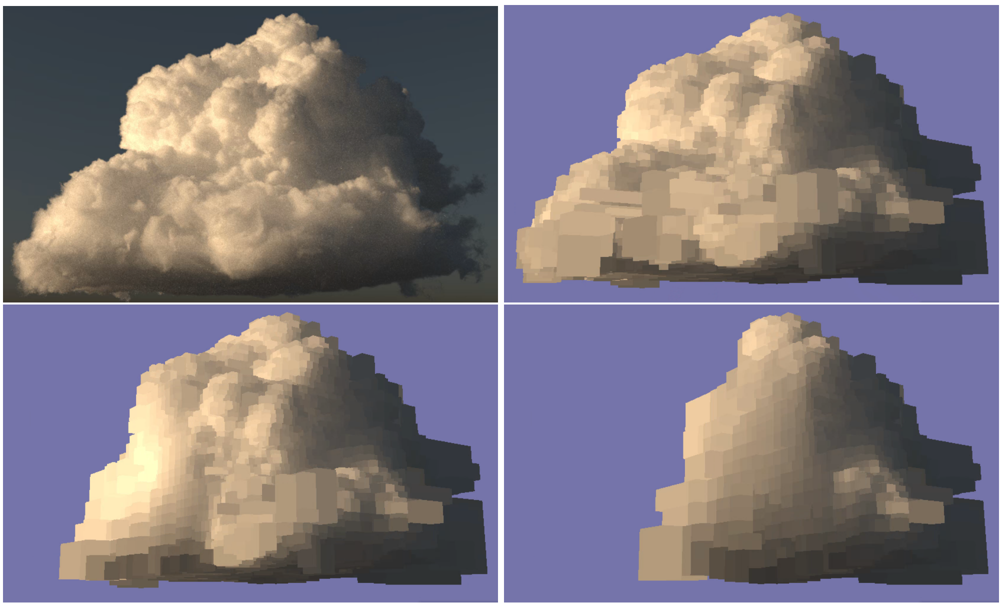
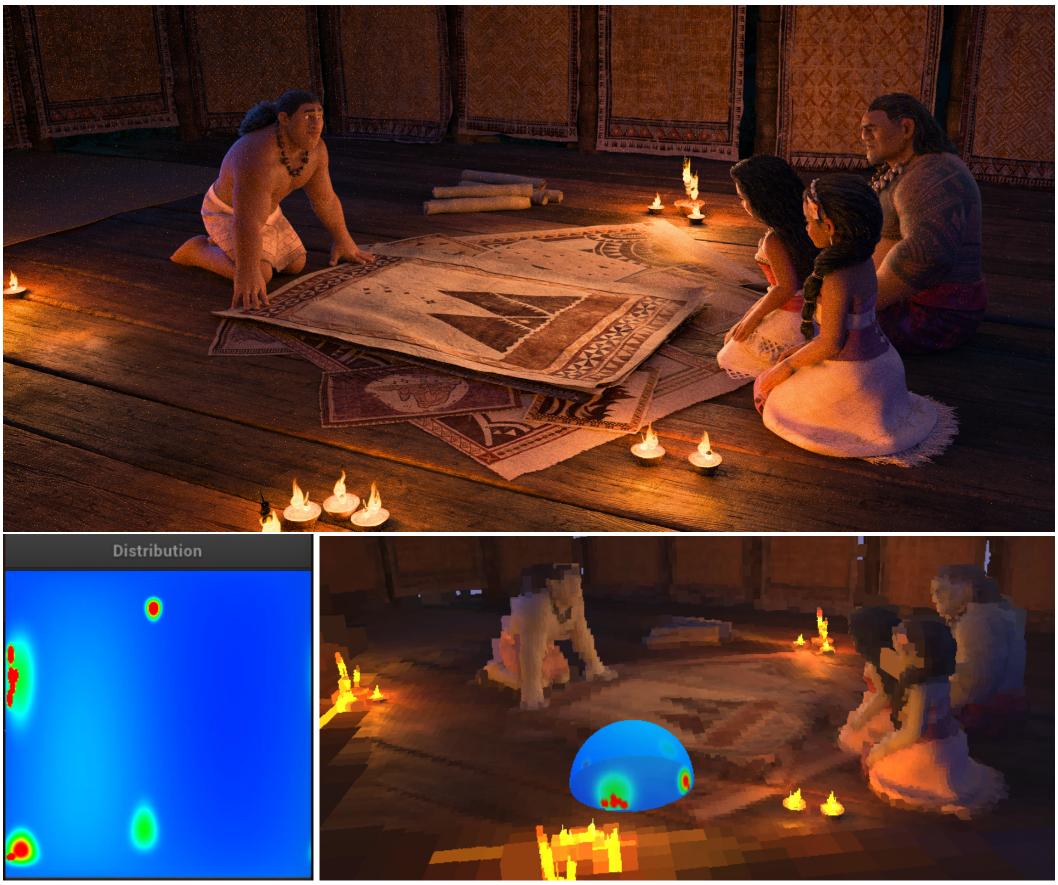


Fig. 8. A rendering of a heterogeneous cloud (top left) and its corresponding voxel based visualization showing the full volume (top right), and two different cuts through the voxel representation of the volume (bottom). © 2025 Disney

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,



Cylindrical representation Spherical representation

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.

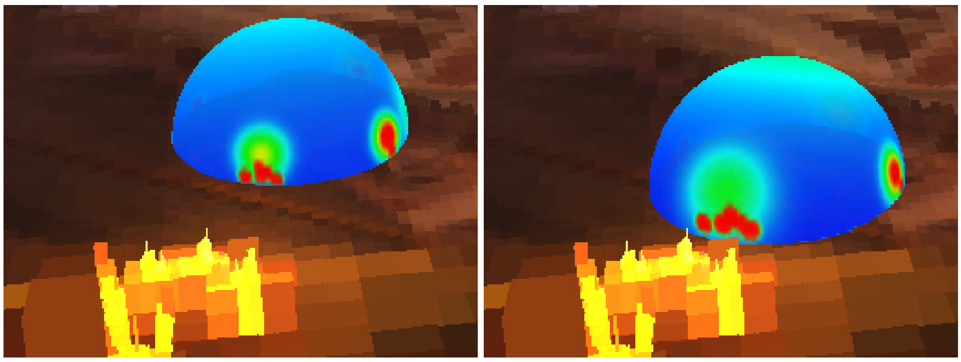


Fig. 10. Offsetting the querying location within or around a voxel to examine the effect of the parallax compensation on a closeup of a scene from *Moana 2*. Notice how the frontal peak of the distribution moves as we change the offset to the query position, such that it always points towards the light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

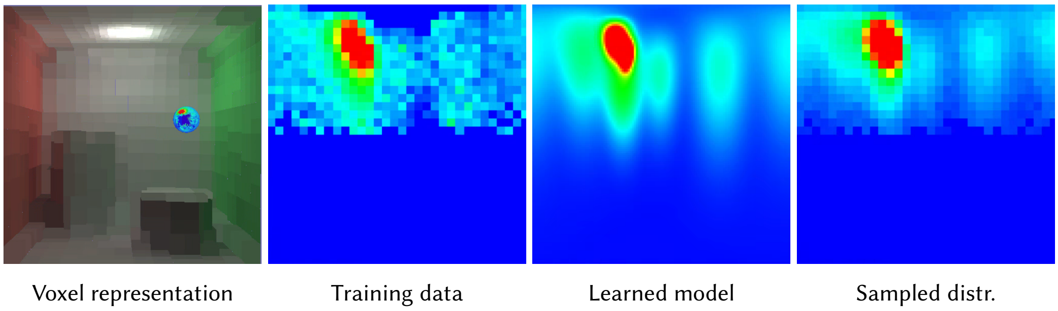


Fig. 11. Example of the recorded incident radiance in a Cornell Box (2nd from left), the corresponding learned model (2nd from right) and a 2D histogram of guiding samples drawn from this model in the renderer (far right).

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.

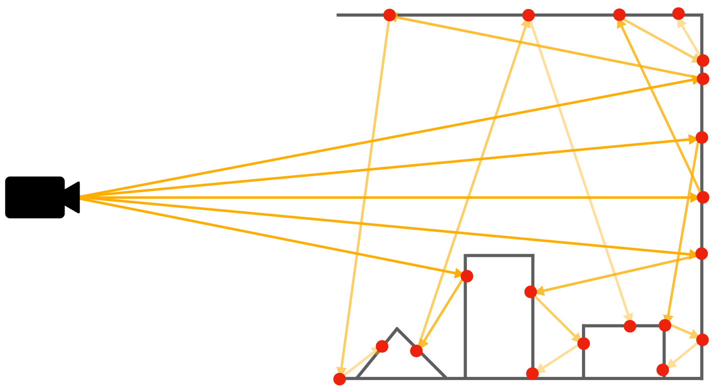


Fig. 12. Illustrative example of restricting number of recorded radiance estimates to the first N=4 vertices of a path. The first 4 vertices can have decent spatial coverage in simple scenes.

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*.

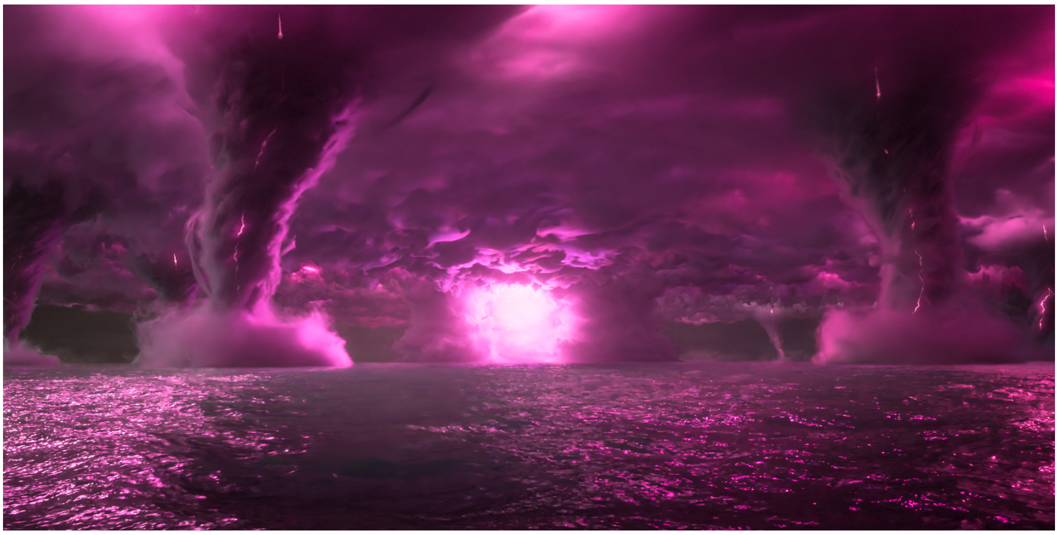


Fig. 13. A volume-heavy shot from *Moana 2*. © 2025 Disney

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.

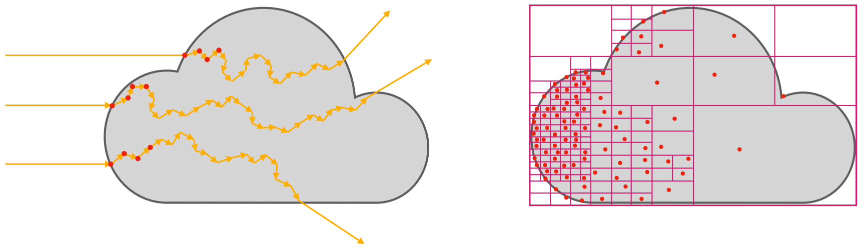


Fig. 14. Illustration of paths scattering through a dense volume. When recording the radiance estimate only at the first N=4 vertices, the resulting SD-tree data structure can have imbalanced spatial coverage of data inside the volume.

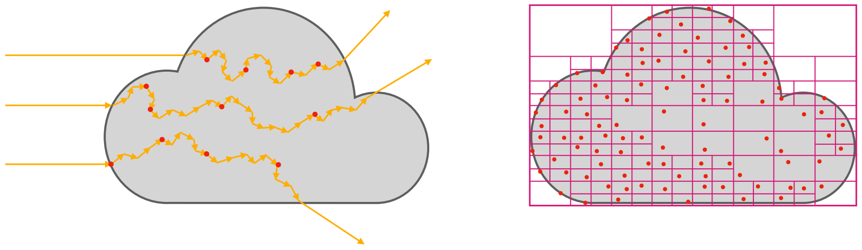


Fig. 15. Illustration of stochastic vertex skipping with a soft vertex budget of 4 vertices per path. Skipping vertices earlier on the path leads to better spatial distribution of data inside the volume.

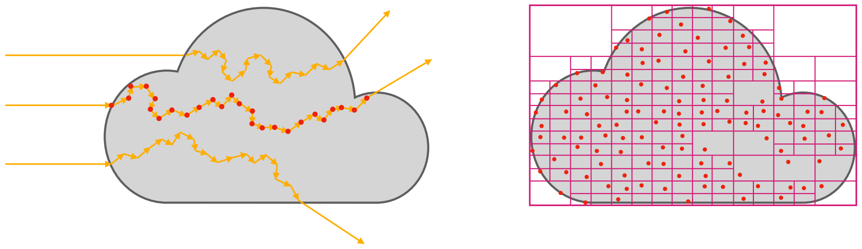


Fig. 16. Illustration of stochastic path skipping with a soft vertex budget of 4 vertices per path. We allow selected paths to record more vertices than the soft budget, allowing recording data deeper into the volume.

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.

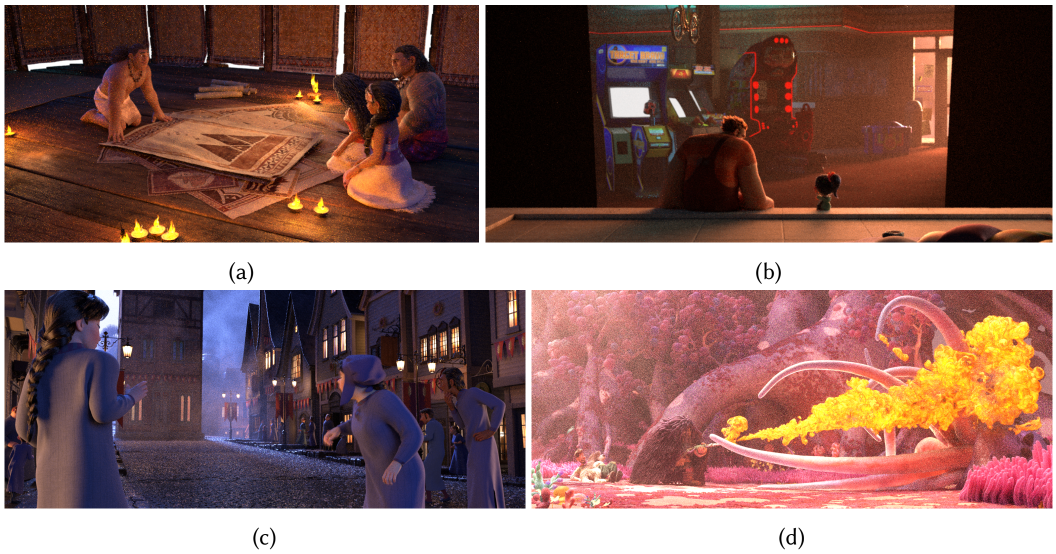


Fig. 17. Renders used in Table 2, from a) *Moana 2*, b) *Ralph Breaks the Internet*, c) *Frozen 2*, and d) *Strange World*. All rendered with path guiding at 64 SPP. © 2025 Disney

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]

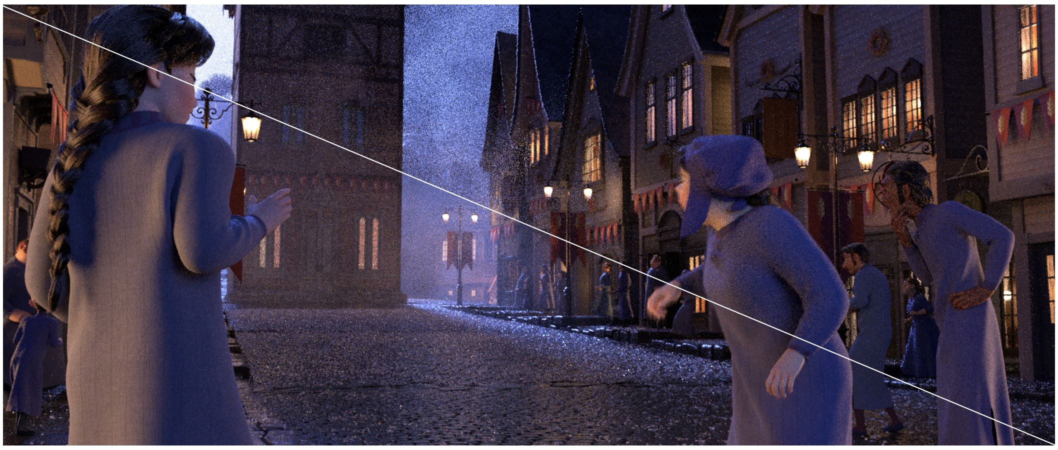
Therefore, the baseline, rendered in a production renderer, can be much harder to beat than a baseline rendered with a research renderer such as Mitsuba [Jakob 2010] or PBRT [Pharr et al. 2023]. Consequently, we cannot expect gains observed in a research setting to translate directly into production. 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.

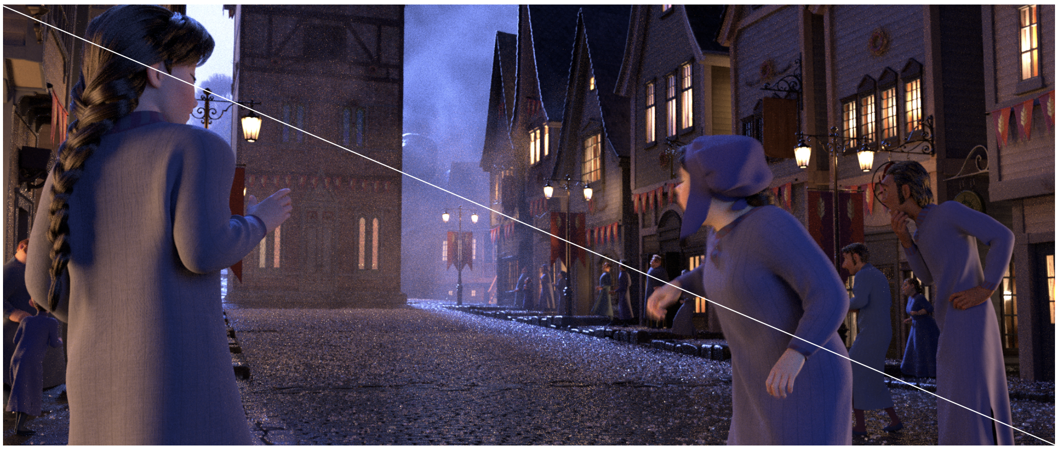
Figure 18a shows an equal SPP comparison at 128 SPP. For both renders, we disabled Hyperion's cache points optimization, to illustrate path guiding's gains without the optimization. We see significant noise reduction at equal SPP in the render that uses path guiding. However, if we leave our cache points 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

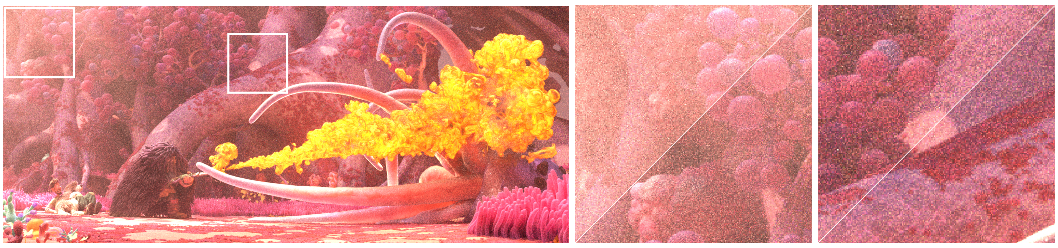


Fig. 19. Shot from *Strange World* showing brightness difference when using path guiding. The crops on the right of the figure show the result with path guiding (top left triangle) and without path guiding (bottom right triangle). © 2025 Disney

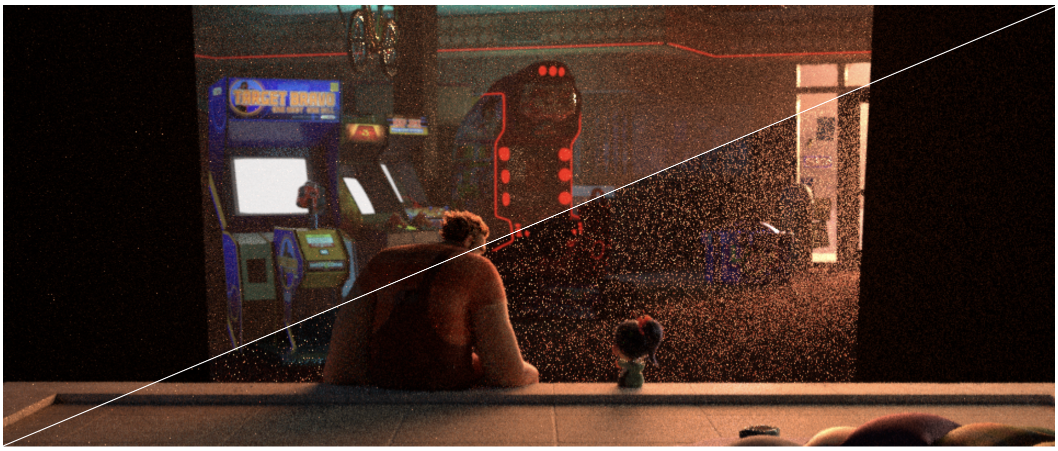
from *Ralph Breaks the Internet*, filled with a forward-scattering participating media illuminated from outside through a glass door. Without path simplification enabled (bottom right), most illumination seen in this scene is from caustic paths, which together with the forward-scattering volume, leads 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.

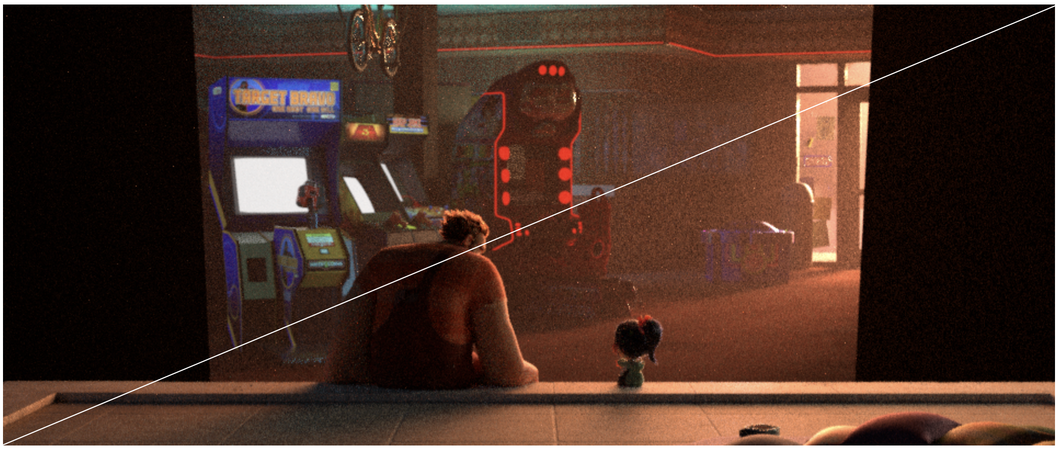
5.3.3 Evaluation Criteria in Production The decision to adopt a new method in a production renderer and enable it for production shots is more involved than evaluating image comparisons; we also look at considerations specific to our production environment, which may be less relevant in research environments.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

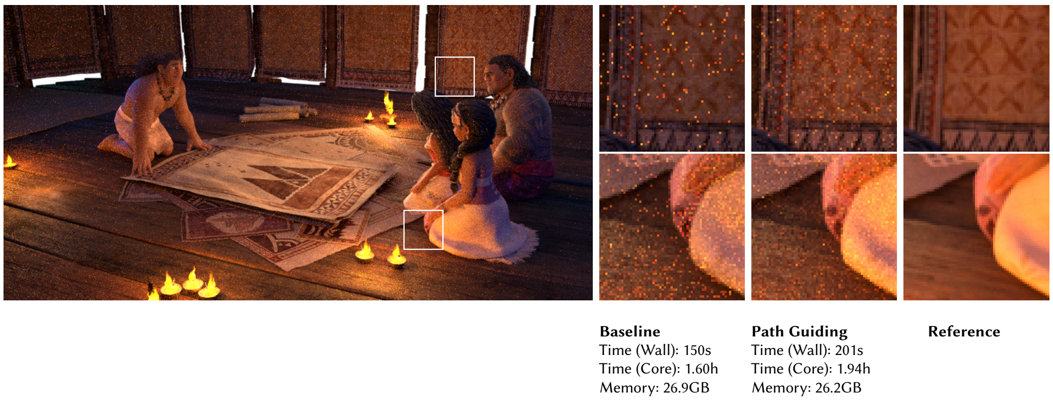


Fig. 21. A frame from *Moana 2*. Equal SPP comparison rendered with 64 SPP. © 2025 Disney

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.

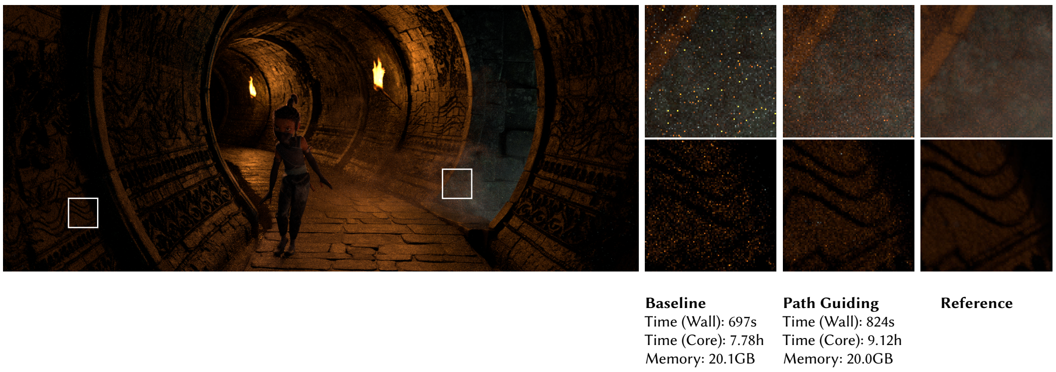


Fig. 22. A frame from *Raya and the Last Dragon*. Equal SPP comparison rendered with 64 SPP. © 2025 Disney

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.

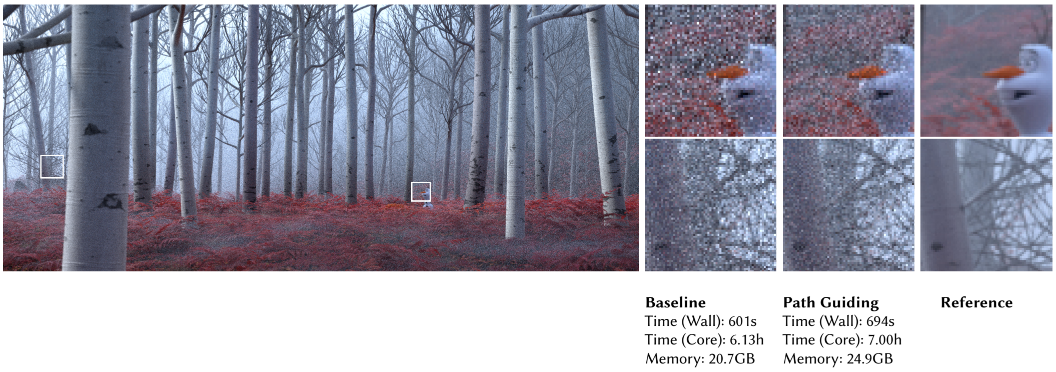


Fig. 23. A frame from *Frozen 2*. Equal SPP comparison rendered with 64 SPP. © 2025 Disney

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.

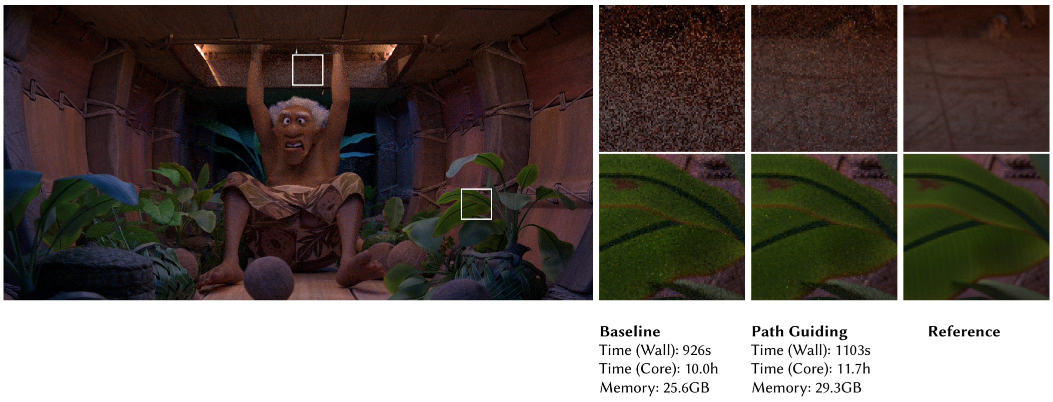


Fig. 24. A frame from *Moana 2*. Equal SPP comparison rendered with 64 SPP. © 2025 Disney

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 [Herholz 2025, Sec. 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 [Herholz 2025, Sec. 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. 2025. Integrating Path Guiding into a Production Render: The Nitty Gritty Details. In *ACM SIGGRAPH 2025 Courses: Path Guiding in Production and Recent Advancements, Chapter 1* (Vancouver, Canada) *(SIGGRAPH '25)*. Association for Computing Machinery, New York, NY, USA.

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

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