Practical Path Guiding

Gustavo Silvera
gsilvera@andrew.cmu.edu
https://github.com/gustavosilvera/mitsuba3/tree/pathguide
Carnegie Mellon University, Pittsburgh, PA, USA

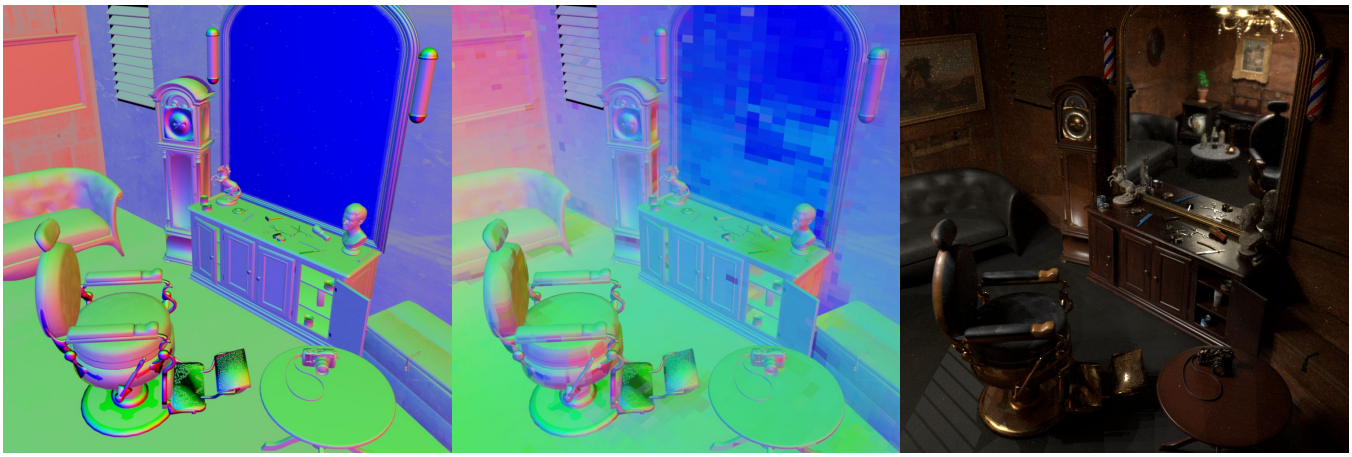

Figure 1: Sampled scatter ω_o distribution for BSDF (left), and pathguide (center), and final rendered scene (right) using MIS.

ABSTRACT

In the realm of physically based rendering one of the fundamental tasks is to accurately simulate the way light interacts with the virtual scene. The traditional approach, known as path tracing, traces rays of light from the camera and samples the reflective properties of the surfaces they encounter vis BSDF sampling. However, path tracing can be incredibly computationally expensive, especially for scenes with complex indirect lighting and materials. Path guiding is an advanced technique that optimizes the path tracing algorithm by guiding the rays towards the most significant lighting components in the scene to maximize each path's energy contribution. Path guiding has only recently become practical [8] through advancements in the data structures used to implement it. This final project explores the concept of practical path guiding (PPG), investigates PPG improvements from [7], and discusses implementation details in both Dirt (our course renderer) and the popular open source renderer Mitsuba3 [3].

KEYWORDS

Rendering, Path Guiding, Reinforcement Learning, Mitsuba3

PBR Reference Format:
Gustavo Silvera. 2023. Practical Path Guiding. In *Practical Path Guiding.* , 8 pages.

PBR, Spring 2023, Pittsburgh, PA, USA

1 INTRODUCTION

"Practical" path guiding gets its name from the fast nature of the querying operations needed to run the entire rendering + pathguiding process. This is critical for use in production environments where performance, flexibility, and simplicity are desired. Many complex scenes include difficult areas where indirect lighting is dominant and unidirectional path tracing with emitter sampling (next event estimation) alone is not very effective.

The ultimate goal of path guiding is to importance sample the rendering equation [4]

$$L_s(\mathbf{x}, \omega_o) = \int_S L_i(\mathbf{x}, \omega_i) f_s(\mathbf{x}, \omega_i, \omega_o) \cos\theta \mathrm{d}\omega_i$$

By perfectly importance sampling the incident radiance term L_i, the Monte-Carlo estimate for this integral can have zero variance (noise). This is very difficult to achieve because L_i is the primary quantity of interest that involves shooting a ray and recursing on the equation. Instead, the authors [8] takes an approach to construct an approximation of L_i everywhere and use this for importance sampling.

The authors' formulation for this involves a reinforcement learning approach to path guiding that iteratively learns how to construct high-energy light paths. In order to accomplish this in an efficient manner, they store a discrete approximation of the scene's 5D light-field (consisting of 3D positional and 2D directional domains) using an optimized adaptive Spatio-Directional tree (SD-tree) as shown in Fig. 2. The SD-tree consists of an upper KD tree to subdivide the 3D spatial domain, and then a lower quadtree to partition the 2D directional domain.

Due to its minimal hyperparameter tuning, ease of implementation, and low overhead, this work is well-suited for modern production environments.

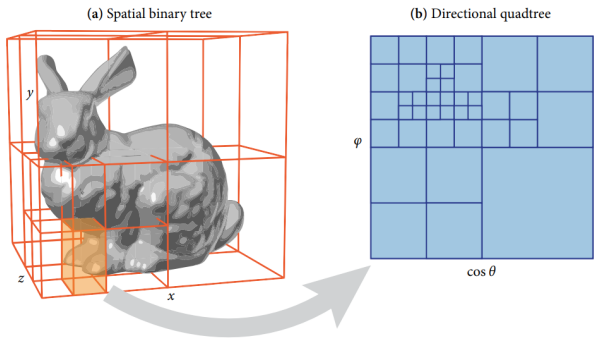

Figure 2: Visualization of spatial-directional tree. Left demonstrates the spatial subdivisions by a binary tree. Right showcases a quadtree partitioning the directional domain.

2 METHODOLOGY

The process for this project was to first implement the practical path guiding system in our *Dartmouth Introductory Ray Tracer* (Dirt), the class renderer used for 15-468 programming assignments. This consists of implementing three primary components: 1. the `PathGuide` data structure class; 2. the training process; and 3. the inference process (sampling) in our path tracer integral. My choice to use Dirt for writing the baseline code was primarily for ease of implementation and familiarity for the codebase.

For implementation reference and inspiration, I was able to resource the authors' original implementation [6] in Mitsuba0.6 [2] as well as the recently open-sourced Dreamworks Animation Moonray renderer [5] which has its own implementation of PPG.

2.1 The main `PathGuide` class

The path guider class object can effectively be broken down into three important parts: 1. The user-facing API used for query methods and training; 2. the spatial tree for partitioning the spatial domain; and 3. the quadtree used for partitioning the directional domain.

2.1.1 Public API. The user-facing API of the path guider primarily consists of initialization, recording/training, status querying, and sampling (direction and pdf). The path guider class is intended to be included as part of the scene or the integrator and the user must provide an ability to perform multiple render passes that can be used for pathguider training. After initialization, the various training render passes need to incorporate recording bounce radiance for "training" the path guider, then once this (training) render pass is complete, a refinement step is performed on the internals (SD-tree) to further optimize the path guider.

All the while, the path guider must provide some way to indicate whether it still needs data for training or whether it is done training and can be used for inference (sampling direction and pdf).

Once the path guider internally flips its flag indicating that it is ready for inference, the user only needs to query the path guider for sampling ω_o given a position $\mathbf{x}$, and calculating the path-guider pdf given a position $\mathbf{x}$ and direction ω from BSDF sampling. Both of these querying operations are required since the new sampling scheme still needs the pdfs from the BSDF and PG direction in order to incorporate MIS sampling.

2.1.2 Spatial Tree. The upper part of the SD-tree consists of a KD-tree that is used to partition the spatial domain. Nearly all queries on this data structure can be performed in $O(\log(n))$ asymptotic complexity because the tree is able to disregard fractions of the search space at a time. Every axis of the spatial dimension is subdivided iteratively for subsequent generations in the tree. Upon refinement, the individual nodes also need to support subdivision into two children each with half the number of samples as the (ex-leaf) new parent node. This enables efficient and adaptable restructuring around the high-frequency regions of the scene without spending resources on inexpensive regions with few samples. A visualization can be found on the left side of Fig. 2.

The entire spatial tree is implemented as a chunk of contiguous memory (`std::vector<Node>`) of nodes where nodes contain indices in the vector to their children. This is primarily to enable fast constant-time access of nodes in the tree and should be cache-friendly as the memory layout is regular and predictable.

2.1.3 Directional Tree. The lower part of the SD-tree consists of a quad-tree that is used to partition the directional domain. Implementation-wise however, there is first a thin-wrapper around the directional trees that contains two directional trees per each spatial tree leaf node. This is because while the training process only considers two SD-trees at a time (the previous SD-tree guiding the current SD-tree) the authors note that because the Spatial-tree in the current SD-tree is a more refined version of the Spatial tree in the previous SD-tree, it is equivalent to use the more-refined SD-tree for spatial-queries in both SD-trees. Therefore, it is more compute and memory efficient to use the same (more-refined) Spatial tree for both the previous and current SD-tree and have two different directional trees at the leaf nodes, removing the need to query two spatial trees to reach the same location.

The individual direction trees is implemented similarly to the spatial tree but the nodes contain four children with data corresponding to the "weight" of the radiance samples in this node. Notably, many of the storage types in these trees need to be atomic in order to avoid race conditions while recording concurrent radiance values in the training phase (which performs multithreaded rendering). There is a similar subdivision process for the directional tree to adaptively refine the domain to focus resources where they are most needed. A visualization can be found on the right side of Fig. 2.

Furthermore, the authors [7] discuss their reasoning behind using *world-space-aligned cylindrical coordinates* for parameterizing their directional guiding distribution. Instead of a *surface-aligned* representation, having world-space aligned distribution allows for the same distribution even when encountering high-frequency normal variance. Then, rather than spherical coordinates, cylindrical coordinates have a desirable area-preserving correspondence to the surface of the solid sphere.

2.2 The training process

Training the path guider involves a series of additional render passes that can be considered "preprocessing" because the samples used for this training are not typically used in the final render because they are of higher variance. In [7] the authors discuss ways to utilize these samples in the final render rather than completely discarding them, but I did not investigate this in my project. The "preprocessing" render passes start out with small number of samples per pixel (spp) and increase geometrically [8] up to some threshold that can be set by the user. There is considerable discussion in the original paper [8] on how to automatically set this threshold given a sample-count or time budget, but I mostly let this be a user-made decision.

For my implementation, I set this up as a loop over the `render` function for some number of iterations, and on each subsequent iteration I doubled the number of spp for the `render()` call. Inside the `render()` method there is a call to `Li()` which computes the incident radiance recursively starting from a pixel in the camera sensor. Inside the `Li()` method, I can then introduce the pathguider specific code that accumulates the (per-bounce) ray origin, direction, and incident radiance right after the recursion was calculated. This involves a very minor change to the overall `Li()` method since obtaining the incident radiance at every bounce is free when using this recursive `Li()` approach.

After every `render()` call in the preprocess step is completed, the pathguider has accumulated a new L_i distribution approximation that it can then refine to help guide the next pass. Notably, the `render` method is highly parallelizeable and computes incident raidiance on multiple processors, the path guider can accumulate the samples atomically (hence not introducing race conditions). However, the `refine` step after the render process is not thread-safe in my (or the authors') implementations for simplicity so this is done sequentially after the `render` method is called. While it is not impossible to parallelize this refinement method, it is mostly unnecessary as it has been fast enough in my testing and supporting parallel tree subdivisions would be much more complicated to implement.

2.3 The final rendering inference

Once the path guider has been sufficiently trained, a flag is enabled to indicate that there is no more need to record radiance values and instead begin querying the pathguider for sample directions of ω_o. This is also a relatively minor change in the integrator's `Li` method but also includes calculating the multiple-importance-sample (MIS) weight of taking this sample in order to preserve unbiasedness. Note that since the pathguider is still a data-driven discrete approximation of L_i there could be directions that were never sampled in the training process which might lead to zero probability of sampling this direction with solely pathguiding. Therefore, we combine the pathguider sampling with the standard BSDF sampling with MIS selection. In order to support this MIS weights, the inference process flips an α-weighted coin to choose whether to sample the path guider (PPG) or BSDF for ω_o then divides by the weighted pdf combination. The authors use an $\alpha = 0.5$ but have also discussed an improvement in [7] that uses a dynamic α and another optimization process to compute it. This was not explored in this project so I will refer to the paper for its implementation details.

A visualization of these individual scattering distributions is shown in Fig. 3, note that standalone BSDF sampling produces directions that look similar to simple scene normals, while the pathguider distribution is clearly more scene aware and can direct bounces in particular regions of the scene towards the light source.

3 DIRT RESULTS

The resulting pathguider on the simple Cornell box scene in dirt (seen in Fig. 4) shows significant noise reduction compared to the reference solution.

3.1 Next event estimation

Notably, next-event-estimation (NEE) is disabled in both renders because pathguiding is known to help *less* than NEE, so to better visualize the improvement it is disabled. In many complex scenes predominantly lit with indirect lighting, NEE is ineffective so pathguiding would shine regardless.

3.2 Performance

Although I did not have much time to take a deep-dive into this, I will mention a bit about pathguide performance. With the same number of samples, the pathguider implementation is slightly slower because there are more advanced (pathguider) queries required per path bounce (logarithmic complexity) compared to the constant-time BSDF sampling. However, in very simple scenes such as Dirt CBox in Fig. 4 the pathguider was actually slightly faster, likely because the paths were able to terminate (hitting the emitter) in fewer bounces on average while contributing more energy. This behavior was not consistent on other scenes with more complex light paths however. Regardless, the main promise of pathguiding is that fewer samples can be used than in the purely-pathtracing render, so there should be performance gains to be had if this was tuned more. Perhaps as future work.

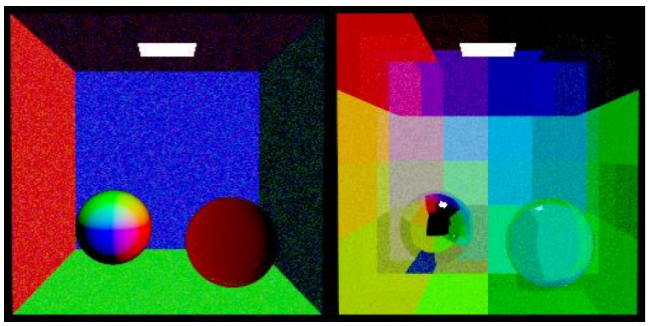

Figure 3: Comparison of scattering distribution from Dirt Cbox (Fig. 4) with BSDF sampling (left) and pathguider sampling (right).

4 ADDITIONAL IMPROVEMENT

In the SIGGRAPH course [7] by one of the original authors, there is discussion on three areas of improvement to make PPG even more

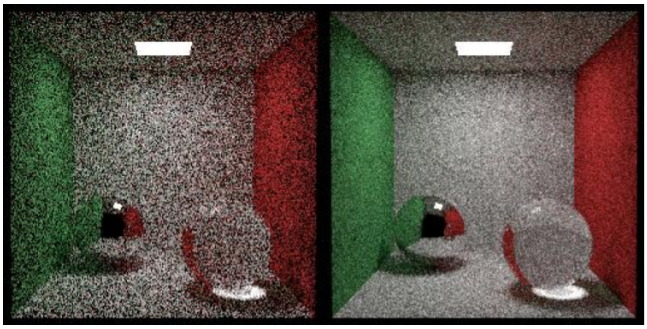

Figure 4: Comparison of rendering between reference (left) and mine with pathguiding (right). Both images were rendered with 64 samples per pixel.

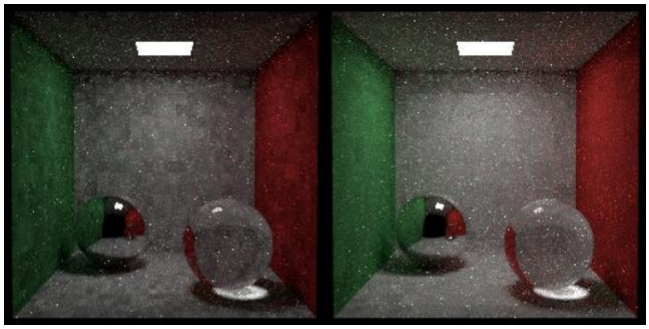

Figure 5: Demonstration of Dirt Cbox (Fig. 4) render without any filtering and extreme hyperparameter settings leading to visually unappealing error (left). Same scene and render but with stoachastic-spatial filter implemented in pathguider (right).

practical, especially for simpler scenes where it seems to be mostly ineffective. The first area of improvement involves a technique to not discard the samples used for training as mentioned earlier, the last technique involves a learning approach to dynamically select the MIS weight threshold α, also mentioned earlier. The second technique is what I mostly implemented for my final project (see 4.1).

4.1 Spatio-Directional Filter

Unfortunately, the hyperparameters for the PPG training process can be tuned to extreme levels to learn a better approximation of the scene, but may cause nonuniform and visually unappealing noise artifacts in the image. This can be seen in Fig. 5. To address this issue, the authors introduce an additional filtering step to the SD-tree splatting process by recording samples in not only the individual (spatial and directional) leaf node but also a small neighbourhood. Naively performing this filtered splatting process is significantly more expensive because the SD tree needs to be queried multiple times for a single sample.

Due to time constraints I was only able to implement a variant of this filtering process that only queries the SD tree twice and introduces additional randomness. By randomly jittering the recorded sample within its neighbourhood, this is a form of stochastic filtering that achieves significantly lower overhead at a slight quality loss. The SD tree needs to be queries twice for these samples: once to find the size of the neighbourhood to jitter the sample (without recording anything), then once more to record the newly-jittered sample.

With stochastic-spatial filtering on only the upper-half (spatial domain) of my SD tree implementation, I was already able to get significantly better results as seen in Fig. 5. While some artifacts can still be visualized in the improved render, this demonstrates a contrived example with extreme hyperparameter tuning that is strongly discouraged by the authors. Still there may be cases where learning the radiance field very precisely is paramount so these artifacts should be mostly mitigated with this filtering.

5 INTEGRATING INTO MITSUBA3

As an additional aspect of this final project, I wanted to integrate my implementation into the popular open source academic renderer Mitsuba3 [3]. I had this in mind for a few reasons: I wanted to experience development in a proper research renderer, I wanted to add this new feature and experiment with more advanced integrators, I wanted to leverage its larger support for other popular tools such as blender, and importantly I wanted to release my code publicly which I could not do in Dirt (which has assignment solutions for 15-468).

5.1 Challenges

In addition to its highly optimized and heavily templated C++ architecture, the Mitsuba codebase is much more complicated and cumbersome to develop in than Dirt (which is very student-friendly). Additionally, since Mitsuba3 (not 0.6) is both a forward and inverse renderer, my code would need to still work for the differentiable rendering aspect of Mitsuba in order to be compliant with their render structure. This is subject for future work at time of writing since I was only able to implement the forward rendering implementation for this project.

5.2 Throughput accumulation

Additionally, another challenge with the Mitsuba integration is how they calculate incident radiance along a path. Unlike Dirt, which uses a recursive `Li` method to calculate radiance along each bounce, Mitsuba uses a "forward" for loop that accumulates the throughput along a path via emitter sampling as seen in Fig. 6.

Throughput can be defined mathematically as:

$$T(\mathbf{x}_k) = G(\mathbf{x}_0, \mathbf{x}_1) \prod_{j=1}^{k-1} \left(f(\mathbf{x}_j, \mathbf{x}_{j+1}, \mathbf{x}_{j-1}) G(\mathbf{x}_j, \mathbf{x}_{j-1}) \right)$$

where $T(\mathbf{x}_k)$ denotes the throughput of path $\mathbf{x}_0 \rightarrow \mathbf{x}_{k-1}$, $G(\mathbf{x}_i, \mathbf{x}_j)$ denotes the geometry term between $\mathbf{x}_i$ and $\mathbf{x}_j$ (includes foreshortening and visibility), and $f(\mathbf{x}_j, \mathbf{x}_{j+1}, \mathbf{x}_{j-1})$ denotes the BSDF sampling at $\mathbf{x}_j$ with the incoming direction from $\omega_i = \mathbf{x}_{j-1} \rightarrow \mathbf{x}_j$ and outgoing direction $\omega_o = \mathbf{x}_j \rightarrow \mathbf{x}_{j+1}$. The explanation of this throughput path formulation can be found in more detail from [1].

While this is mathematically equivalent to the recursive `Li` method as we've implemented in Dirt, there is no straightforward

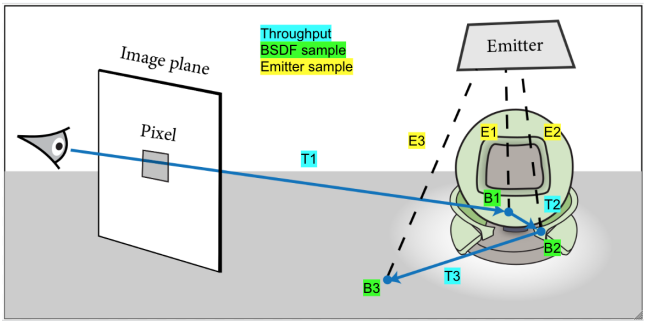

Figure 6: Mitsuba's Path integrator schematic demonstrating throughput accumulation and emitter sampling. From Mitsuba documentation: `mitsuba.readthedocs.io`

way to get the incident radiance at every bounce point for free as was possible before (due to the recursion). Now, several intermediate variables need to be kept track of per bounce in order to post-hoc compute all the radiance values of every bounce as is needed to train the path guider.

In my formulation, the primary elements I store are the ray (bounce) origin $\mathbf{x}_i$, direction $\omega_o = d_i$, total radiance of path without NEE $L(\mathbf{x}_0 \rightarrow \mathbf{x}_i)$, total radiance of path with NEE $L_{\text{total}}(\mathbf{x}_0 \rightarrow \mathbf{x}_i)$, and total throughput $T(\mathbf{x}_i)$ so far. Since path guiding is best trained on indirect radiance (as opposed to direct illumination which would dominate the distribution) the resulting computation should provide the non-NEE radiance at $\mathbf{x}_i$ from the rest of the path: $L(\mathbf{x}_i \rightarrow \mathbf{x}_k)$ where k is the light source.

In order to get the current radiance without NEE, I subtract the previous path (with NEE) radiance $L_{\text{total}}(\mathbf{x}_0 \rightarrow \mathbf{x}_{i-1})$ from the total radiance sum $L_{\text{total}}(\mathbf{x}_0 \rightarrow \mathbf{x}_i)$ that is accumulated at the end of loop iteration i (which accumulates the sum of all previous paths). This delta provides the total radiance of the path only up to this point (without the NEE from previous bounces) which represents the indirect lighting throughput $L(\mathbf{x}_0 \rightarrow \mathbf{x}_i)$ up to this point. Finally, since the intermediate throughput is being calculated as the loop progresses, the $T(\mathbf{x})$ is stored directly.

Then I iterate through this collection backwards to start at the end of the path to find the latest bounce k with non-zero radiance $L(\mathbf{x}_0 \rightarrow \mathbf{x}_k) > 0$ (representing the latest bounce forming a complete path between the eye and a light source) and use this quantity $L(\mathbf{x}_0 \rightarrow \mathbf{x}_k)$ as the total energy for the path (without NEE).

With this final energy $L(\mathbf{x}_0 \rightarrow \mathbf{x}_k)$ since I don't want to train on direct illumination I skip recording this bounce and continue iterating (backwards) through the previous bounces to compute the radiance of the path from $L(\mathbf{x}_i \rightarrow \mathbf{x}_k)\ \forall i < k \in \mathcal{N}$. It is also important to remember that this total radiance up to $\mathbf{x}_k$ included several geometry terms and BSDF evaluations from the eye that happen before the bounce at $\mathbf{x}_i$, so these need to be cancelled out. Thankfully, this quantity is encoded directly in the stored throughput accumulation $T(\mathbf{x}_i)$, which can be divided out of the expression as follows:

$$L(\mathbf{x}_i \rightarrow \mathbf{x}_k) = \frac{L(\mathbf{x}_0 \rightarrow \mathbf{x}_k)}{T(\mathbf{x}_i)} = \frac{T(\mathbf{x}_0 \rightarrow \mathbf{x}_i)T(\mathbf{x}_i \rightarrow \mathbf{x}_k)}{T(\mathbf{x}_0 \rightarrow \mathbf{x}_i)} = T(\mathbf{x}_i \rightarrow \mathbf{x}_k)$$

Which gives the final radiance value that can be used for training by recording $\mathbf{x}, \omega_o,$ and $L(\mathbf{x}_i \rightarrow \mathbf{x}_k)$.

6 MITSUBA RESULTS

The other implementation details of PPG (adding the `PathGuide` class and final rendering inference with MIS) were mostly straightforward and very similar to the Dirt implementation. With this pathguider complete in Mitsuba I was able to successfully render some famous scenes and create a fancy scene of my own.

6.1 Veach BDPT scene

The well known Veach BDPT scene [9] is notorious for its difficulty to render efficiently because of the complex lighting effects in the scene. Notably there is a caustic effect from the glass egg on the table and the main light sources are blocked by being hidden inside their light fixtures, making direct connections for next event estimation difficult and mostly ineffective. The scattering distributions (see Fig. 7) also highlight guiding behavior of light bounces to maximize radiance.

Since NEE is not helpful in this scene, the pathguider render reduces noise considerably as compared to the reference render Fig. 8. Notably there is much less drastic noise highlights in the image, especially around the areas in the scene with high indirect illumination.

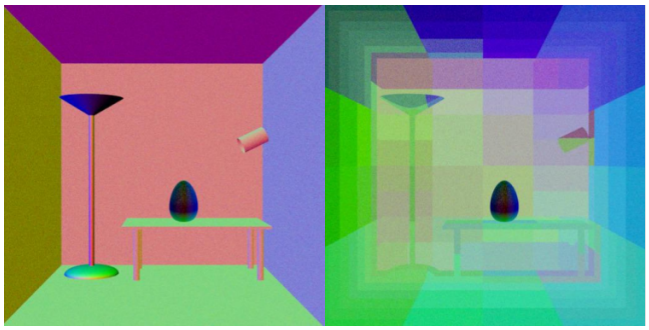

Figure 7: Veach BDPT scene [9] rendering out the scattering distribution with BSDF sampling (left) and with pathguiding sampling (right).

6.2 Barbershop scene

For my final scene and rendering competition submission (see Fig. 10) I decided to have some fun in blender and model a barbershop with fancy models and complex lighting. The scene itself required nearly 30 Gb of memory to render and was rendered overnight at 20k samples per pixel. The scene is designed to suffer from similar indirect dominant lighting such as from the chandelier and the dim sunlight blocked by the shutters. The light sources from each of the chandelier candles are housed inside the chandelier lamp enclosures, so next event estimation struggles to create visible direct connections for the majority of the scene.

I also rendered the same scene in the reference Mitsuba3 renderer with the same number of samples per pixel to compare the noise levels (see Fig. 10) and found that the resulting noise in the difficult

Figure 8: Veach BDPT scene [9] rendered with path guiding (left). Close up crop view on noise with path guider (top-right) and without (bottom-right)

areas of the scene was greater compared to my pathguider-assisted implementation. This makes sense as path guiding is supposed to help with difficult indirect lighting paths which are harder to sample purely with BSDF sampling.

Finally, I also provide a comparison of the scenes' scattering distribution with path guiding (Fig. 9), highlighting certain areas where the light paths are encouraged to follow a particular direction to maximize their energy and contribution.

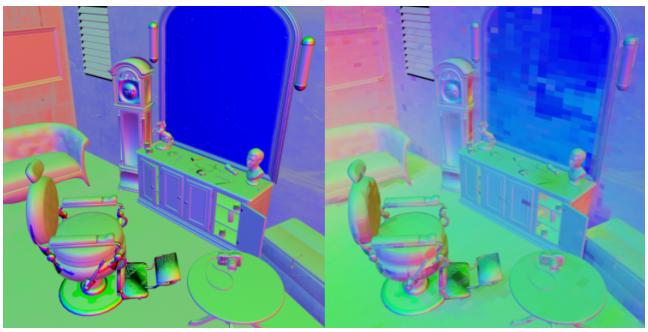

Figure 9: Custom barbershop scene rendering the scattering distribution with BSDF sampling (left) and path guiding (right)

7 CONCLUSION

Path guiding provides a means to approximately importance sample the incident radiance term L_i of the recursive rendering equation [4] which is useful to reduce variance in an unbiased fashion. By encouraging paths to bounce in a more optimal way to maximize their energy contribution, complex lighting effects such as indirect-dominant illumination are more easily sampled than with standard unidirectional path tracing techniques.

This project explores an implementation of "Practical Path Guiding" (PPG) primarily from the discussions in [8] and improves on it with discussions from [7]. This implementation focuses on the spatio-directional tree as the primary data structure and provides positive results on Dirt, our in-class renderer. The implementation was then ported to the Mitsuba3 [3] renderer and achieved better noise levels than the reference solution as desired.

8 FUTURE WORK

Ideally, I will be able to improve upon this implementation (primarily performance concerns and implementation simplicity) to eventually merge it into the master branch of Mitsuba3. This would be a good contribution to the rendering community as it has shown promise to reduce noise and theoretically improve performance of scenes in Mitsuba. In order to be compliant with Mitsuba3, this entire process would need to support the backwards (differentiable) rendering pipeline that Mitsuba has in place, which is also grounds for future work.

Additionally, implementing the two other improvements discussed in [7] would be beneficial to the path guiding algorithm as the performance and quality should be improved yet again.

ACKNOWLEDGMENTS

This project is primarily based on the work done in *Practical Path Guiding for Efficient Light-Transport Simulation* [8], and the follow-up work on improvements from *"Practical Path Guiding" in Production* [7]. I'd also like to acknowledge the Dreamworks Animation team for releasing their implementation in their Moonray Renderer [5] which was an invaluable resource for implementation details and inspiration. Guidance for this project was provided by the course professor Dr. Ioannis Gkioulekas.

REFERENCES

[1] Ioannis Gkioulekas. 2023. Course notes on Path Integral Formulation and BDPT. http://graphics.cs.cmu.edu/courses/15-468/lectures/lecture_13.pdf
[2] Wenzel Jakob. 2010. Mitsuba renderer. https://www.mitsuba-renderer.org
[3] Wenzel Jakob, Sébastien Speierer, Nicolas Roussel, and Delio Vicini. 2022. Dr.Jit: A Just-In-Time Compiler for Differentiable Rendering. *Transactions on Graphics (Proceedings of SIGGRAPH)* 41, 4 (July 2022). https://doi.org/10.1145/3528223.3530099
[4] James T. Kajiya. 1986. The Rendering Equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '86)*. Association for Computing Machinery, New York, NY, USA, 143–150. https://doi.org/10.1145/15922.15902
[5] Mark Lee, Brian Green, Feng Xie, and Eric Tabellion. 2017. Vectorized Production Path Tracing. In *Proceedings of High Performance Graphics* (Los Angeles, California) *(HPG '17)*. Association for Computing Machinery, New York, NY, USA, Article 10, 11 pages. https://doi.org/10.1145/3105762.3105768
[6] Thomas Müller. 2017. GitHub repository for PPG implementation. https://github.com/Tom94/practical-path-guiding
[7] Thomas Müller. 2019. "Practical Path Guiding" in Production. In *ACM SIGGRAPH Courses: Path Guiding in Production, Chapter 10* (Los Angeles, California). ACM, New York, NY, USA, 18:35–18:48. https://doi.org/10.1145/3305366.3328091
[8] Thomas Müller, Markus Gross, and Jan Novák. 2017. Practical Path Guiding for Efficient Light-Transport Simulation. *Computer Graphics Forum (Proceedings of EGSR)* 36, 4 (June 2017), 91–100. https://doi.org/10.1111/cgf.13227
[9] Eric Veach. 1998. *Robust Monte Carlo Methods for Light Transport Simulation*. Ph. D. Dissertation. Stanford, CA, USA. Advisor(s) Guibas, Leonidas J. AAI9837162.

Figure 10: Custom barbershop scene rendered without pathguiding (left) and with path guiding (right)

Figure 11: Full resolution version of the pathguide-rendered final scene for my rendering competition submission. The raw (.exr) and processed (.png) image can be found on my GitHub code repository at https://github.com/gustavosilvera/mitsuba3/tree/pathguide