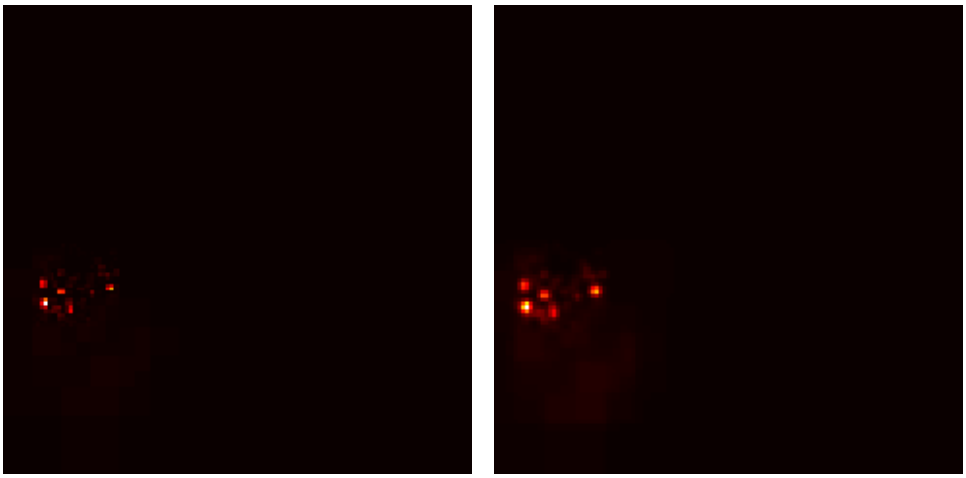


Hierarchical Reconstruction of Quadtree-Based Approximations of Incident Radiance

Master's thesis in Computer science and engineering

JOHANNES NILSSON

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

MASTER'S THESIS 2023

Hierarchical Reconstruction of Quadtree-Based Approximations of Incident Radiance

JOHANNES NILSSON

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

Hierarchical Reconstruction of Quadtree-Based Approximations of Incident Radiance
JOHANNES NILSSON

Supervisor: Erik Sintorn, Department of Computer Science and Engineering
Examiner: Ulf Assarsson, Department of Computer Science and Engineering

Master's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A quadtree before and after being reconstructed by the implemented algorithm.

Typeset in LaTeX
Gothenburg, Sweden 2023

Hierarchical Reconstruction of Quadtree-Based Approximations of Incident Radiance
JOHANNES NILSSON
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This thesis presents a method of hierarchically reconstructing quadtree-based approximations of incident radiance for path guiding. To that end, Gaussian denoising is applied by mapping quadtree data to matrices, performing regular matrix convolutions, and then mapping the data back to the quadtree. The reconstructed guiding distributions are shown to outperform previous work in most cases, especially when the number of path samples is limited. Additionally, using a simple target distribution in the beginning of the learning process before switching to the full version is shown to speed up the learning. To limit the overhead of the reconstruction procedure, an automatic workload budgeting algorithm is presented. While the improved quality of the guiding distributions seems promising, the implemented reconstruction algorithm imposes a large enough overhead to nearly cancel out the benefits.

Keywords: Path, Tracing, Light, Transport, Guiding, Reconstruction, Denoising.

Acknowledgements

I would first of all like to thank Alexander Rath, for helping me curate the idea that led to this thesis. Secondly, Erik Sintorn, for his continued guidance, and the valuable discussions we had that led to some of the performance optimizations made in the implementation. Thirdly, I would like to thank my computer for, despite being over 11 years old, being able to survive the evaluation of this project.

Johannes Nilsson, Gothenburg, June 2023

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

AVX	Advanced Vector (E)xtensions
BRDF	Bi-directional Reflectance Distribution Function
DS	Distribution Switching
D-tree	Directional tree
MC	Monte Carlo
PDF	Probability Density Function
relMSE	Relative Mean Squared Error
SD-tree	Spatio-Directional tree
SIMD	Single Data Multiple Instruction
S-tree	Spatial tree
WL	Workload Limiting

Contents

List of Figures

List of Tables

1
Introduction

Photorealistic image rendering has been a challenge in industry and academia for decades. Product visualization, architectural design, and movie production are all examples of where physically based rendering is commonly used to achieve highly realistic images. *Path tracing* is a simple yet reliable rendering algorithm that achieves high levels of realism due to its unbiased way of simulating how light propagates and interacts with objects of different materials within a scene [11]. Simulating the physics of light propagation naturally results in realistic global illumination while also producing various effects such as caustics, soft shadows, and depth of field. Due to its physical accuracy and ease of implementation, path tracing has become the de facto standard in the VFX industry [3, 2, 4, 5].

A major challenge with the standard path tracing algorithm is that it inherently suffers from diminishing returns. Images can therefore take an extremely long time to converge to a realistic result free of noise, especially in difficult lighting conditions. Image noise is a noticeable and quite disturbing effect to the average observer, which is why efforts must be made to minimize it. Image denoising algorithms are commonly used to get around this problem. Pixar, for example, make use of such algorithms in movie production, as discussed by Christensen and Jarosz [3]. Denoising can help produce acceptable results, but it is not an ideal solution if the ultimate goal is to produce realistic images, as some details are lost in the denoising process.

A substantial amount of research has been devoted to the design of algorithms that simulate light transport more efficiently by prioritizing the construction of high-energy light paths [20, 12]. These sophisticated algorithms can be very useful in scenarios where standard path tracing is impractical due to difficult lighting conditions. They can, however, under-perform in other scenarios where the lighting conditions are simpler. Another issue with these methods is their complexity, which poses engineering challenges when implementing and integrating them into production systems.

Another approach to reduce the variance, i.e noise, in path tracing is to store information about how light interacts with the scene during the rendering process. This information can then be used to guide the construction of light paths. Lafortune and Willems [13] were among the first to introduce the concept of caching illumination information to reduce the variance of path tracing. One of the benefits of this technique, which is now commonly referred to as *path guiding*, is that it is relatively simple to extend a standard path tracing algorithm to support guiding. With a standard path tracing algorithm at its core, this method can perform well in scenes with simple lighting conditions, but it can also perform well in scenes featuring more indirect illumination thanks to the cached illumination information. The robustness offered by path guiding, and the advances made in more recent

research [21, 6, 14, 18, 19, 23] make path guiding an interesting area of research for the purpose of further driving rendering technology forward.

State-of-the-art path guiding still has a few remaining issues to be solved, one of them being that gathering the necessary illumination information is a rather time-consuming task. As mentioned by Rath et al. [18], the technique struggles to compete with more sophisticated algorithms for short renderings, because approximating the radiance field with few path samples is very difficult. Thus, the question posed in this work is the following: how can path guiding be made more efficient for short renders?

1.1 Thesis aim

The overarching purpose of this work is to explore ways in which state-of-the-art path guiding can be improved, and the goal is to find a way to make the method more robust for short renderings, which is relevant for preview renders in product visualization and architectural design, for example. The distributions that guide the construction of paths are essentially two-dimensional functions that suffer from noise when the number of path samples is limited. The initial hypothesis is therefore that traditional denoising techniques, such as Gaussian filtering, can be applied to improve the quality of the guiding distributions for short renders.

1.2 Outline

The structure of the report can be described as follows:

Chapter 2 - Preliminaries This chapter covers concepts that are essential for understanding this thesis. The reader will be introduced to Monte Carlo integration, path tracing, and recent related work.

Chapter 3 - Problem Definition
A detailed description of the problem at hand.

Chapter 4 - Implementation
Chapter 4 presents the implementation of the various algorithms.

Chapter 5 - Evaluation
Chapter 5 contains the performance results with comparisons against previous work.

Chapter 6 - Discussion
Chapter 6 discusses the results and proposes areas of future work.

Chapter 7 - Conclusion

2

Preliminaries

This chapter will cover concepts that are fundamental to the main topics of this thesis. Readers who are familiar with the term *ray tracing* may wonder what similarities it shares with path tracing. This will be explained in section 2.1. Section 2.2 will cover modern path guiding and related research.

2.1 Path Tracing

Designing an algorithm that is able to render photorealistic images, capturing various lighting phenomena such as global illumination, caustics, refractions, lens flares etc., is a challenging task. In reality, even weak emitters will emit an extremely large number of photons every second that interact with different objects and materials in various ways, giving rise to these lighting effects. To get an idea of the quantity of photons, Planck's hypothesis can be used:

$$E = nh\left(\frac{c}{\lambda}\right),$$

where E is the energy, n is the number of photons, h is Planck's constant, c is the speed of light and λ is the wavelength. As an example, consider a 100% efficient light source emitting 10 watts of monochromatic yellow light with a wavelength of 600 nm. The number of photons emitted per second would thus be:

$$n = \frac{E\lambda}{hc} = \frac{10 \cdot 6 \cdot 10^{-7}}{6.626 \cdot 10^{-34} \cdot 2.998 \cdot 10^{8}} = 3.021 \cdot 10^{19}.$$

A naive solution to capture all lighting effects would be to simulate each photon emitted by the light source, which is commonly known as *forward ray tracing*. As indicated by the example above, however, tracing each emitted photon would be computationally expensive. Furthermore, only a small fraction of the emitted light ends up at the camera, which makes forward ray tracing impractical in most cases. *Backward ray tracing*, i.e tracing from the camera to the emitter, is more efficient and is therefore the standard approach in most ray-tracing-like algorithms. The intuition behind both methods is the same, however: complex lighting phenomena can be simulated by tracing enough rays.

To determine the color of a certain pixel, a primary ray is created using the camera's origin point and the pixel's point on the image plane. This primary ray is then traced throughout the scene, as illustrated by Figure 2.1. Figure 2.2 illustrates how global illumination is achieved using this technique.

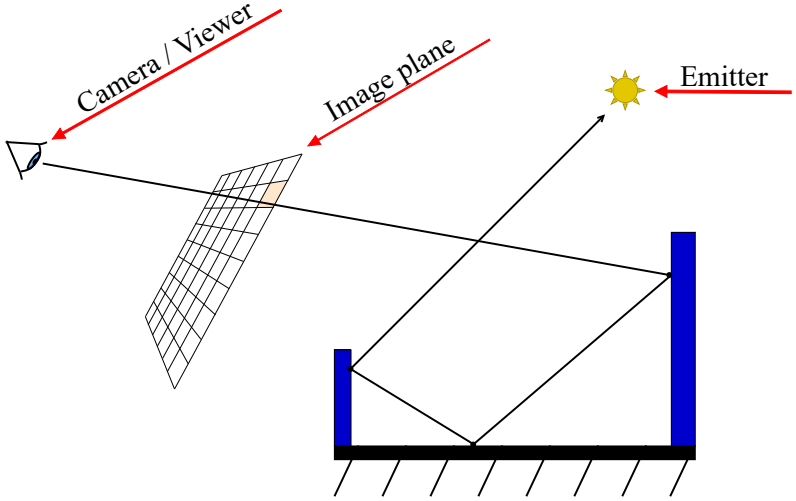


Figure 2.1: Illustration of a ray being traced from the camera through a pixel on the image plane.

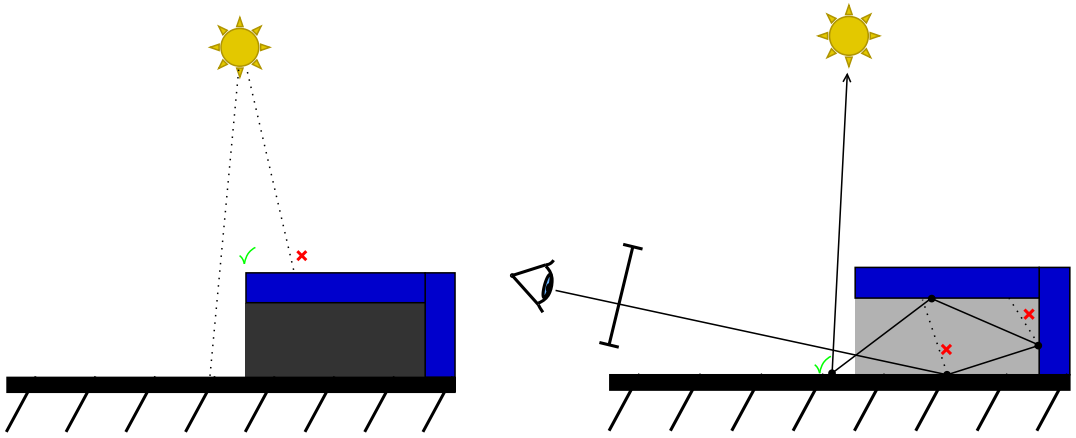


Figure 2.2: Using direct illumination only, no light would make its way under the blue object (**a**). By following a whole light path, indirect illumination can be added, making the points under the blue object dimly lit (in light shadow) (**b**).

The following sections will delve deeper into the tools required for a practical approach to ray tracing. Monte Carlo (MC) integration (2.1.1) and the *rendering equation* (2.1.2) are the pillars of what makes path (and ray) tracing work. The rendering equation does not have an analytical solution, which is why some tool for numerical approximation, such as MC integration, must be used. It is important to understand the motivation behind importance sampling (2.1.3), as much of the thesis is built upon this concept.

2.1.1 Monte Carlo Integration

Monte Carlo integration is a method of numerically approximating the value of arbitrary integrals by means of sampling the function in question using random sampling points that follow a certain distribution [17]. As an example, let us study how the one dimensional integral $\int_a^b f(x)\mathrm{d}x$ can be evaluated using MC integration. Let F_N be the following MC estimator:

$$F_N = \frac{b-a}{N}\sum_{j=1}^{N} f(X_j). \tag{2.1}$$

According to the MC estimator F_N, given a set of N uniformly distributed random variables $X_j \in [a, b]$, the value of the integral is equal to the expected value of the estimator:

$$\int_a^b f(x)\mathrm{d}x = \mathbb{E}\left[F_N\right] = \mathbb{E}\left[\frac{b-a}{N}\sum_{j=1}^{N} f(X_j)\right]. \tag{2.2}$$

The expected value $\mathbb{E}\left[g(x)\right]$ of a function $g(x)$ is defined as the average value of $g(x)$ over some distribution $p(x)$ over its domain D. Thus:

$$\mathbb{E}\left[g(x)\right] = \int_D g(x)p(x)\mathrm{d}x. \tag{2.3}$$

Since the random variables X_j are uniformly distributed, the *Probability Density Function* (PDF) $p(x)$, i.e the probability of selecting sample x, is equal to $\frac{1}{b-a}$. This is because p must be constant and p must integrate to 1 over the interval $[a, b]$. It can therefore be shown algebraically that equation 2.2 holds

$$
\begin{aligned}
\mathbb{E}\left[F_N\right] &= \mathbb{E}\left[\frac{b-a}{N}\sum_{j=1}^{N} f(X_j)\right] = \\
&= \frac{b-a}{N}\sum_{j=1}^{N} \mathbb{E}\left[f(X_j)\right] = && \text{(expand using eq. 2.3)} \\
&= \frac{b-a}{N}\sum_{j=1}^{N} \int_a^b f(x)p(x)\mathrm{d}x = && \left(p(x) = \frac{1}{b-a}\right) \\
&= \frac{1}{N}\sum_{j=1}^{N} \int_a^b f(x)\mathrm{d}x = \\
&= \int_a^b f(x)\mathrm{d}x.
\end{aligned}
$$

This can be generalized further to scenarios where the samples are not necessarily drawn from a uniform distribution. If the estimator

$$
F_N = \frac{1}{N}\sum_{j=1}^{N} \frac{f(X_j)}{p(X_j)} \tag{2.4}
$$

is used instead, the integral can be approximated by drawing the samples X_j from an arbitrary PDF $p(x)$. This can be shown in a similar manner as above:

$$
\begin{aligned}
\mathbb{E}\left[F_N\right] &= \mathbb{E}\left[\frac{1}{N}\sum_{j=1}^{N} \frac{f(X_j)}{p(X_j)}\right] = \\
&= \frac{1}{N}\sum_{j=1}^{N} \mathbb{E}\left[\frac{f(X_j)}{p(X_j)}\right] = \\
&= \frac{1}{N}\sum_{j=1}^{N} \int_a^b \frac{f(x)}{p(x)}p(x)\mathrm{d}x = \\
&= \int_a^b f(x)\mathrm{d}x,
\end{aligned}
$$

which means that this is a valid estimator too. However, this estimator puts an additional constraint on $p(x)$, namely that it is non-zero for all x where $|f(x)| > 0$.

The generalization step that leads to estimator 2.4 is very important, since drawing samples from an arbitrary PDF is essential for a practical path tracing algorithm. The shape of the PDF generally has a big impact on the noise in rendered images. Why this is so will be further explained in the following sections.

2.1.2 Monte Carlo Ray Tracing

In 1986, James T. Kajiya [11] presented the *rendering equation* and laid the groundwork of modern ray tracing. The rendering equation can be simplified as

$$L_o(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + L_r(\mathbf{x}, \omega_o),$$

where $L_o(\mathbf{x}, \omega_o)$, $L_e(\mathbf{x}, \omega_o)$ and $L_r(\mathbf{x}, \omega_o)$ are the total outgoing radiance, emitted radiance and reflected radiance, respectively, at point $\mathbf{x}$ and in direction ω_o.

Readers who are unfamiliar with the term radiance may benefit from knowing that it is, in simple terms, a measure of the intensity of light. More specifically, it is power per unit projected area per unit solid angle, giving it the unit $\frac{W}{\text{sr}\cdot m^2}$. However, in the context of ray tracing it may help the reader to think of it as related to the light's color and its intensity.

The rendering equation thus states that the radiance leaving point $\mathbf{x}$ in direction ω_o is equal to the radiance emitted by the surface of wherever $\mathbf{x}$ is located, plus the radiance that is reflected by the surface in direction ω_o. The L_r term can be expanded to describe the reflected radiance in terms of incoming radiance, yielding

$$L_o(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + \int_\Omega L(\mathbf{x}, \omega_i) f(\mathbf{x}, \omega_o, \omega_i) \cos\theta \mathrm{d}\omega_i, \tag{2.5}$$

where $L(\mathbf{x}, \omega_i)$ is the incoming radiance at point $\mathbf{x}$ from direction ω_i, $f(\mathbf{x}, \omega_o, \omega_i)$ is the *Bidirectional Reflectance Distribution Function* (BRDF) and θ is the angle between the incident ray and the surface normal, as illustrated by Figure 2.3. Note that the domain of the integral, Ω, is the hemisphere.

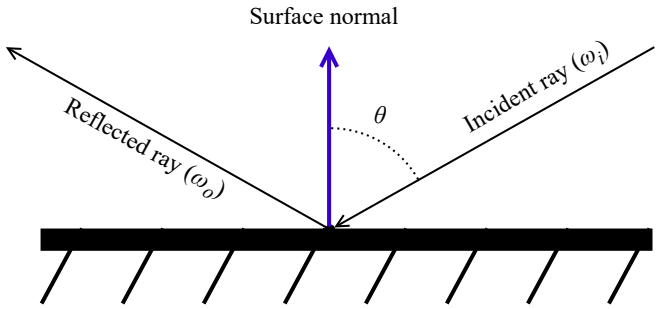
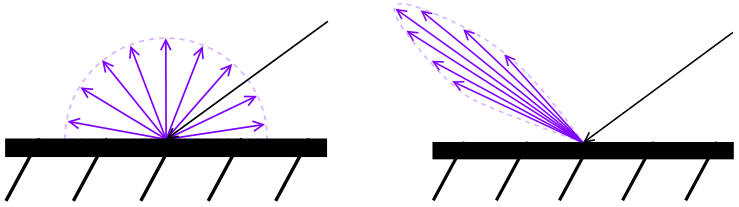


Figure 2.3: θ denotes the angle between the surface normal and the incident ray at the point of intersection.

The BRDF essentially describes how much of the light coming from direction ω_i is reflected in direction ω_o at point $\mathbf{x}$. As such, the BRDF can take many shapes depending on the material of the surface at the ray intersection point. Figure 2.4a shows a BRDF of a perfectly diffuse surface, which means that incoming light is reflected uniformly across the hemisphere. In contrast, Figure 2.4b shows a BRDF of a glossy surface where most of the incoming light is reflected such that the outgoing light's angle to the normal is similar to the incoming light's angle to the normal.



(a) **(b)**

Figure 2.4: Illustration of outgoing directions when the BRDF represents **a)** a perfectly diffuse surface and **b)** a somewhat glossy surface.

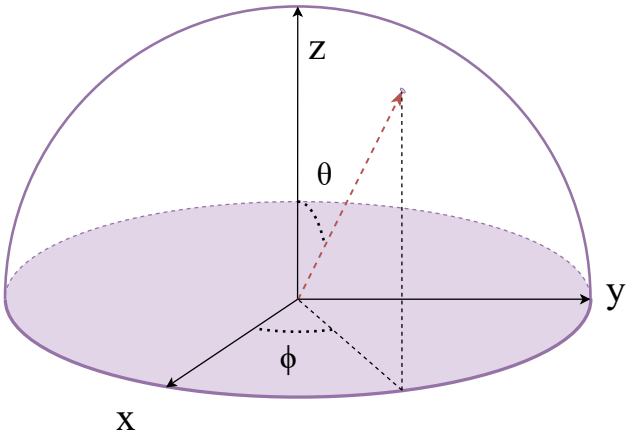


Figure 2.5: Illustration of spherical coordinates in the top hemisphere.

The rendering equation (eq. 2.5) does not have an analytical solution, but by applying the MC estimator introduced above (eq. 2.1), it would be possible to accurately approximate the integral. To account for the difference $b-a$ in the numerator, it is necessary to first calculate the PDF of uniform hemisphere sampling. The PDF is constant and must integrate to one over its domain, therefore

$$\int_\Omega p(\omega)\mathrm{d}\omega = 1 \implies c\int_\Omega \mathrm{d}\omega = 1 \implies c = \frac{1}{\int_\Omega \mathrm{d}\omega}.$$

Instead of integrating over the hemisphere using solid angles, this can be expressed using spherical coordinates, making it simple to evaluate:

$$c = \frac{1}{\int_0^{2\pi}\int_0^{\frac{\pi}{2}} \sin\theta \mathrm{d}\theta\mathrm{d}\phi} = \frac{1}{\int_0^{2\pi} 1\mathrm{d}\phi} = \frac{1}{2\pi},$$

where θ and ϕ are angles that represent the sample. It may help to look at Figure 2.5 to realize why the intervals of θ and ϕ are $[0, \frac{\pi}{2}]$ and $[0, 2\pi]$, respectively. Since only the top hemisphere is sampled, θ can be at most $\frac{\pi}{2}$ radians.

Putting the pieces together yields the estimator in eq. 2.6, which can be used to approximate the reflected light using uniform hemisphere sampling.

$$\langle L_{\mathrm{r}} \rangle = \frac{2\pi}{N} \sum_{j=1}^{N} L(\mathbf{x}, \omega_j) f(\mathbf{x}, \omega_{\mathrm{o}}, \omega_j) \cos\theta j. \tag{2.6}$$

With this, it is now possible to formulate a naive Monte Carlo ray tracing algorithm, which can be seen in pseudo-code in listing 2.1 below. To estimate the reflected light in direction ω_{o} at point $\mathbf{x}$, a number of incoming directions ω_{j} are sampled. To find the amount of incoming light from direction ω_{j}, a new ray is recursively traced in that direction. This recursive process repeats until a ray directly intersects a pure light source (which does not reflect any rays), or until a ray does not intersect anything, which means that it was reflected in a direction such that the ray left the scene entirely. Figure 2.6 illustrates a light path created using this recursive method.

While this algorithm would work, it has some inherent inefficiencies:

1. The algorithm is recursive and the branching factor is equal to the number of sampled directions. This generally results in an exploding tree as seen in Figure 2.7 (assuming no light sources are hit, which would stop the recursion). Additionally, since the color of each pixel is evaluated only once, the visual feedback during the rendering process is poor. The color of the pixels cannot be evaluated until the algorithm has finished completely.

2. Rays generated after several bounces along a path will contribute less to the variance of the image due to the passivity of surfaces. This means that the vast majority of the execution time will be spent evaluating the contribution of the least important rays (due to the high branching factor).

3. The sampling is uniform, which works well for perfectly diffuse surfaces, but not as well for glossy surfaces. By looking at Figure 2.4b, it is quite clear that it would be inefficient to sample uniformly across the hemisphere when only a small portion of the sampled directions would lead to a meaningful contribution. This problem will be explored further in section 2.1.3.

Algorithm 2.1: Naive MC ray tracing

```
1  function traceRay(scene, o, ω_o, S):
2      // find intersection point and surface normal
3      (object, x, N, hit) ← intersect(scene, o, ω_o)
4      if not hit:
5          /* if the ray does not intersect with anything,
6           * i.e leaves the scene, the light from the
7           * environment can be returned
8           */
9          return environmentLight(o, ω_o)
10     else if object is PureLightSource:
11         return emittedLightOf(object)
12     end if
13
14     (D, pdf) ← sampleDirectionsUniform(N, S)
15     L_r ← 0
16     for ω_j in D:
17         L_r ← L_r + traceRay(scene, x, ω_j, S) ·
18               brdf(x, ω_o, ω_j) · cos(N, ω_j)
19     done
20     L_e ← emittedLightOf(object)
21     L_o ← L_e + (2·π / S) · L_r
22     return L_o
23 end function
24
25 function traceImage(scene, S):
26     // S: number of samples
27     // scene: the scene to be rendered
28     P ← CameraOrigin
29     for each (i, j) in pixels:
30         PX ← pixelPoint(i, j)
31         // calculate direction from camera to pixel
32         d ← (PX − P)/||PX − P||
33         Image(i, j) ← traceRay(scene, P, d, S)
34     done
35 end function
```

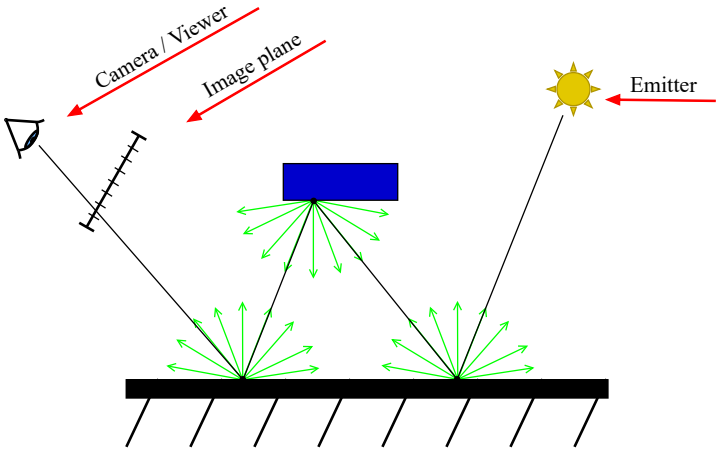


Figure 2.6: Naive MC ray tracing. For each pixel, one primary ray is traced from the camera which then bounces around in the scene. Green arrows represent the uniform sampling distribution of the hemisphere.

The algorithm in listing 2.1 can be modified to avoid issues 1 and 2 listed above. In fact, Kajiya [11] identified this in his paper presenting the rendering equation. The main insight was that instead of estimating the color of each pixel only once using multiple samples of ω_i at each bounce, it would be possible estimate the color of each pixel multiple times using only one sample of ω_i at each bounce, as this would still converge to the correct pixel color. This method is referred to as *path tracing*. The assignment of a certain pixel's color for the n:th sample can thus be described as follows

$$\langle I_\mathrm{px} \rangle_0 = 0,$$

and

$$\langle I_\mathrm{px} \rangle_\mathrm{n} = \frac{1}{n+1} \left(n \cdot \langle I_\mathrm{px} \rangle_{\mathrm{n}-1} + L_\mathrm{i}(\mathbf{c}, \mathbf{p}_\mathrm{px} - \mathbf{c}) \right), \quad 0 < n \leq N,$$

where N is the number of paths per pixel to sample, L_i is the incoming light, $\mathbf{c}$ is the camera origin, and $\mathbf{p}_\mathrm{px}$ is the point of the pixel on the image plane.

The path tracing algorithm has a branching factor of only one, as Figure 2.8 illustrates, which means that the computation time is evenly divided across each generation of rays. As such, this algorithm converges much faster than naive MC ray tracing. Furthermore, by essentially rendering the whole image by iterating over all pixels each time a new path is to be added to the pixel estimates, the visual feedback becomes more immediate. Pseudo-code for the path tracing algorithm can be found in Appendix A, along with the modifications required for importance sampling.

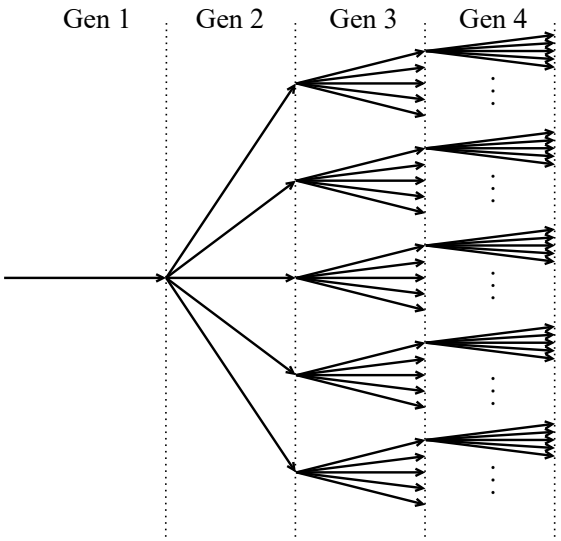


Figure 2.7: The exploding recurrence tree that naive MC ray tracing gives rise to. The first generation is a primary ray originating at the camera, and the following generations of rays are created as rays are reflected.

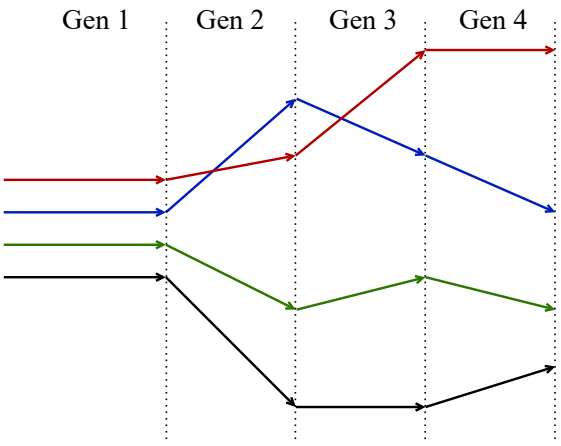


Figure 2.8: Branching of the path tracing algorithm. Note that this figure has multiple rays in generation 1, unlike naive ray tracing in Figure 2.7

2.1.3 Importance Sampling

Importance sampling is a technique used to decrease the variance of MC integration [17]. Consider again the generalized MC estimator (eq. 2.4). By selecting a PDF $p(x)$ that resembles the shape of the integrand $f(x)$, the variance is minimized. The intuition behind this is that a combination of a high value of the integrand and a low value of the PDF leads to extreme values that will require many more samples to balance out. In the context of path tracing, these extreme outliers are usually referred to as ”fireflies” and appear as very bright pixels that stand out in an image. Figure 2.9 illustrates this phenomenon.

Figure 2.9: Fireflies appearing as bright white pixels.

Importance sampling the BRDF is perhaps the most common approach to reduce the variance since the modifications required are few. However, the details of how to model materials in order to appropriately sample incoming directions ω_i depending on the surface roughness (or shininess) are out of scope for this thesis.

In some scenarios, importance sampling the BRDF is not enough to avoid high variance, as is illustrated by Figure 2.10. In this scenario, there are two light sources. Light source B can be sampled directly from the intersection point, while A is occluded. When sampling ω_i from the BRDF, there is a very low probability that the next ray will follow the surface normal along the red path. Since the mirror is reflecting all light from A, the integrand would have a high value due to the incoming light from ω_i. Again, if the numerator is much larger than the denominator, high variance occurs.

The BRDF is not the only candidate however. Consider the following estimator, derived using the generalized MC estimator (eq. 2.4) and the rendering equation (eq. 2.5)

$$\langle L_\mathrm{r} \rangle = \frac{1}{N} \sum_{j=1}^{N} \frac{L(\mathbf{x}, \omega_j) f(\mathbf{x}, \omega_\mathrm{o}, \omega_j) \cos\theta_j}{p(\omega_j | \mathbf{x}, \omega_\mathrm{o})}. \tag{2.7}$$

The incoming light $L(\mathbf{x}, \omega_j)$ could also be approximated and used for importance sampling, further making the shapes of the nominator and denominator more similar. How to do this efficiently remains an open research problem, but *path guiding*, which will be covered in the next section, is one possible solution to this problem.

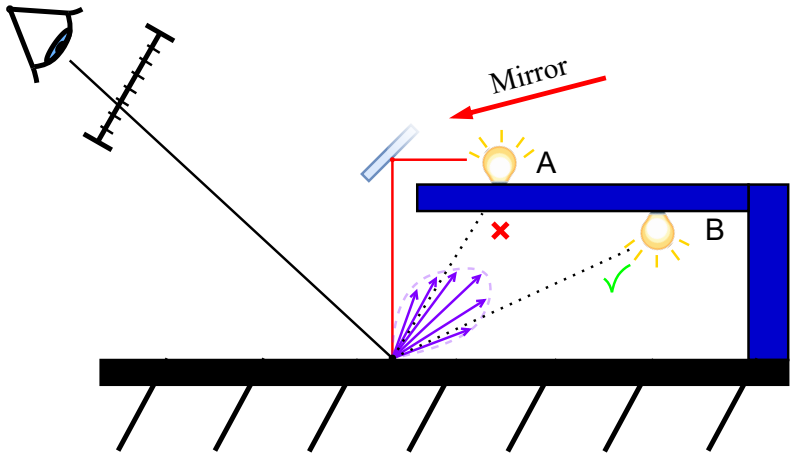


Figure 2.10: A scenario where a firefly could appear in the image due to the intense light being reflected by the mirror.

2.2 Path Guiding

Efficiently exploring the ways in which light paths can be constructed between the camera and emitter is a challenging task. Aside from directly sampling the light sources in a scene, a regular path tracer does not have any information about incoming radiance at any given point in the scene. This can become a problem in many scenes that model the complexities of the real world, Figure 2.10 being one such example. If there existed an accurate model of the *incident radiance field* $L(\mathbf{x}, \omega_i)$, i.e information about the incoming illumination, it would be possible to achieve low variance in the scenario in Figure 2.10. By looking up the incident radiance at the intersection point, it would be revealed that intense light is coming from the direction of the mirror, and it would therefore be important to sample in that direction.

The idea behind path guiding is to gather and store information about the incident radiance, either before or during the rendering process, yielding a statistical approximation of $L(\mathbf{x}, \omega_i)$. Several solutions to this problem have been proposed in previous research. Lafortune and Willems [13] make use of a 5-dimensional tree, similar to what will be discussed below. Jensen [10] makes use of a *photon map*, i.e information from particles traced from the light source, to create hemispherical histograms to approximate incident radiance. Hey and Purgathofer [7] use a similar approach with a photon map, but instead of histograms they create particle footprints on the hemisphere using cones centered around incident photon directions, the width of which are determined by the particle density of the direction in question. Vorba et al. [21] and Ruppert et al. [19] use *Mixture Models*, which approximate the distribution of incoming light using a set of parametric distributions (e.g Gaussian). Müller et al. have even used neural networks to learn the incident radiance [15, 16].

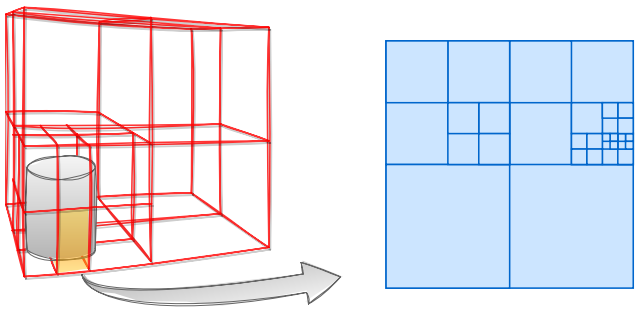
The aim of all methods above is, as previously mentioned, to increase the number of samples taken from directions of high incident radiance for a given point in the scene. While there are several promising data structures presented in previous research, state-of-the-art

can be considered to be the 5-dimensional *SD-tree* proposed by Müller et al. [14]. This data structure, and the iterative learning algorithm that drives the progressive refinement of the light field approximation will be described in the following sections.

2.2.1 The SD-tree

The SD-tree consists of an upper part that divides the spatial domain in three dimensions (S-tree), and a lower part that divides the directional domain in two dimensions (D-tree) [14]. More specifically, the upper part is a spatial binary tree, while the lower part is a quadtree. The terms ”D-tree” and ”quadtree” are used interchangeably throughout this thesis. The subdivision of the spatial binary tree is done by splitting a leaf node in half, alternating the splitting plane between the x, y and z axes. The quadtree is subdivided by splitting a leaf node into four equal parts. The conditions for splitting will be covered in 2.2.2.

This data structure allows for a cheap approximation of the incident radiance field, but there are some obvious sacrifices that have to be made when using a structure like this. Firstly, several points within the same volume (leaf node) of the S-tree will share the same D-tree, even though the distance between these points can sometimes be relatively large. Secondly, the approximation of the incident light in the directional domain is a piecewise-constant function due to the nature of quadtrees. Thus, the approximation can have some rather steep jumps in estimated radiance from two neighboring ranges of directions. This would rarely be the case in reality. Despite these structural drawbacks, however, the SD-tree works quite well in practice.



(a) **(b)**

Figure 2.11: Each leaf node in the upper part (**a**) of the SD-tree contains a quadtree (**b**) that approximates the incoming light from the full sphere of directions.

2.2.2 Iterative Learning Algorithm

The guiding tree is iteratively improved in a training phase at the beginning of the rendering process. The algorithm performs multiple training passes, and the number of path samples created in each pass grows geometrically with the number of iterations. In iteration k, 2^k path samples are constructed for each pixel.

The learning algorithm maintains two SD-trees, one of which is used for *guiding* the construction of light paths, while the other is used for collecting MC estimates of incident radiance. For brevity, these trees will be referred to as $\hat{L}^{k-1}$and $\hat{L}^{k}$, respectively, where k denotes the k:th training iteration. When a light path has been constructed, radiance estimates $L(\mathbf{x}_\mathrm{v}, \omega_\mathrm{v})$ are added to the corresponding leaf node in $\hat{L}^{k}$for each vertex in the path.

At the end of each iteration, $\hat{L}^{k-1}$is set to be a copy of $\hat{L}^{k}$, and $\hat{L}^{k}$is refined in preparation for the next training pass. The decision to split a leaf node in the S-tree is determined only by the total number of path vertices that were recorded in that volume. This ensures that the S-tree adapts to the geometry of the scene. The condition for subdividing a leaf node in the quadtrees is as follows: if the flux flowing through a leaf node (Φ_n) is greater than a certain threshold fraction (ρ) of the total flux flowing through the whole quadtree (Φ), the node is subdivided recursively. When the node is subdivided, its collected radiance is distributed evenly among its new children. Note that one of the properties of $\hat{L}^{k-1}$is that the value of any node is the sum of its four children. Put differently, if $\frac{\Phi_n}{\Phi} > \rho$, the leaf node gets four new children, each with a flux of $\frac{\Phi_n}{4}$. Before the next training pass begins, the data in $\hat{L}^{k}$is reset to zero, so that it does not become biased by old samples over time.

The quadtree subdivision threshold ρ has an impact on the memory footprint of the algorithm, as a lower threshold leads to more refined trees. Müller et al. [14] suggest a threshold of $\rho = 0.01$ to strike a balance between performance and memory footprint. This may not be an appropriate threshold for every use case, however, which is why the subdivision threshold is left as an input parameter in the authors' implementation, giving the user more control. In addition to suggesting an appropriate subdivision threshold, the authors also propose an automatic budgeting algorithm that aims to limit the training phase to approximately 15% of the total rendering budget. According to the authors' data, this budget partitioning minimizes the variance of the resulting image.

2.2.3 Importance Sampling the Incident Radiance

Remember that the goal of path guiding is to approximate $L(\mathbf{x}, \omega_\mathrm{i})$ and use it for importance sampling in order to reduce the variance of the estimator in eq. 2.4. The general procedure can be described as follows:

1. Trace a ray to an intersection point $\mathbf{x}_\mathrm{v}$.
2. Descend spatially in the S-tree to find the leaf node containing $\mathbf{x}_\mathrm{v}$.
3. Sample an incoming direction ω_v from the D-tree contained in the same spatial leaf node.
4. Compute the PDF $p(\omega_\mathrm{v}|\mathbf{x}_\mathrm{v}, \omega_\mathrm{o})$ using algorithm 2.2.
5. Compute the incident radiance and use the PDF to compute the MC estimate $\langle L_\mathrm{r} \rangle$.

The probability of sampling a particular direction ω_i can be computed recursively using the D-tree. The pseudo-code below provides an outline for the computation presented by Müller et al. [14].

Algorithm 2.2: Computing the probability of sampling ω from a D-tree [14].

```
1 function pdfDTree(node, ω):
2     if isLeaf(node):
3         return 1/4π
4     else:
5         child ← getChild(ω)
6         α ← 4 · flux(child) / flux(node)
7         return α · pdfDTree(child)
8     end if
9 end function
```

2.2.4 Other Related Work

Closely related to this thesis is the work by Zhu et al. [23, 22], who use neural networks to reconstruct incident radiance approximations. In [23], the authors use fixed low-resolution sampling maps as a replacement of the quadtrees mentioned above. In [22], they feed quadtrees to the neural network, which outputs a reconstructed quadtree. These methods show promising results and further motivate the need to investigate whether more traditional methods of denoising can be applied to commonly used radiance approximations.

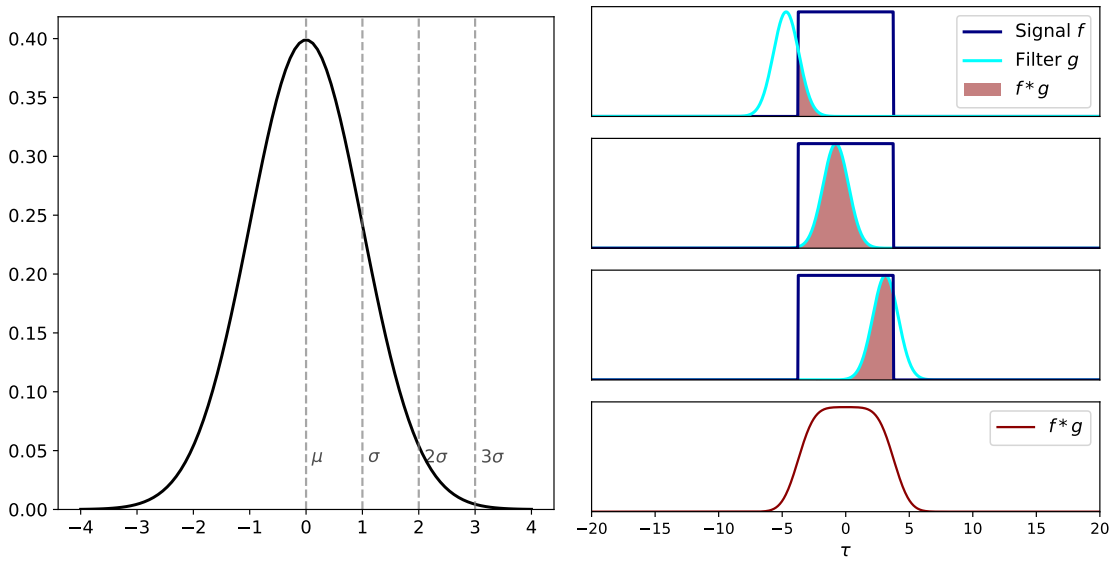
Rath et al. [18] propose alternative target distributions to the one used by Müller et al, which is solely radiance based. The alternatives proposed are referred to as the *variance-aware* and *simple* distributions. As discussed by the authors, some target distributions outperform others depending on how scarce the samples are [18]. For example, the simple distribution, and the radiance-based distribution used by Müller et al, outperforms the variance-aware distribution when the number of samples are still very low, even though the opposite is true when more samples are available. The mathematical details of these distributions are out of scope for this thesis but curious readers are referred to the paper by Rath et al. [18].

2.3 Signal Denoising by Convolution

Noisy signals can be smoothed, or *blurred*, by convolving the input signal with a certain *filter* (or *kernel*). The choice of filter may differ between use cases, but a common choice for blurring a signal is one that follows a *Gaussian*, or *Normal*, distribution, illustrated in Figure 2.12a. A convolution of the functions f and g is defined as the integral of the product of the two functions:

$$(f * g)(t) := \int_{-\infty}^{\infty} f(\tau)g(t-\tau)\mathrm{d}\tau,$$

where g is reflected about the y-axis and shifted along the τ-axis. The mathematical definition may be somewhat more difficult to grasp than it really is conceptually. One way to think about this operation is that $(f * g)(t)$ is the area of the intersection between f and g at time t, as illustrated by Figure 2.12b. The input signal in this example has rather sharp edges that can be smoothed using a Gaussian type filter.



(a) **(b)**

Figure 2.12: **(a)**: A Gaussian distribution with standard deviation σ, centered around the mean $\mu = 0$. **(b)**: The output of the convolution of f and g is equal to the area of the intersection as g slides across across the τ-axis.

It is worth noting that even though the variable t is used in the definition, convolutions can be performed in some other domain. Furthermore, the functions f and g can be discretized and extended to multiple dimensions. Figure 2.13 shows a two-dimensional Gaussian filter, which can be used to denoise two-dimensional input matrices. Points 1-4 below outline the procedure of matrix convolutions.

1. For each element (i, j) in the input matrix, position the filter such that the center overlaps with the input element (i, j) (the movement of the filter is illustrated by Figure 2.14).

2. For each element in the filter (positioned at (i, j)), multiply its value with the corresponding input matrix element it overlaps.
3. If an element of the filter is outside the boundary of the input matrix, the product at that location is zero.
4. Sum all products computed in step 2 and write the sum to the output matrix at (i, j).

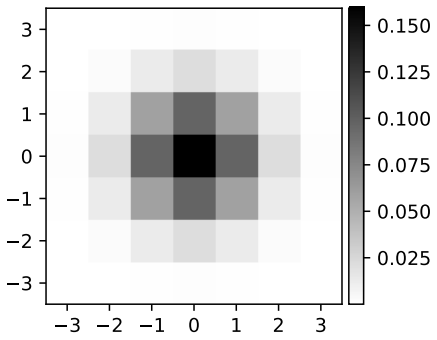


Figure 2.13: A 7x7 Gaussian filter with the standard deviation $\sigma = 1$.

One implication of point 1 is that the size of the filter must be odd to give it a clear midpoint. Note also that, as Figure 2.12a shows, points beyond three standard deviations from the mean of a Gaussian distribution are practically insignificant. Filters can thus be truncated beyond this point in order to reduce the number of computations needed to perform the convolution. These two points motivate the choice of 7x7 as an appropriate size for the Gaussian filter presented in Figure 2.13.

Regarding point 3, there are different ways of dealing with the situation where the filter has elements outside the boundary of the input matrix. A different approach is to reflect rows or columns of the filter that fall outside the input matrix. This way, these elements will still contribute to the overall sum, instead of just being products of zero. How to deal with this issue may depend on the use case. In this thesis, however, elements outside the input matrix will be treated as zeros.

Figure 2.14: A 3x3 filter moving across a matrix in the convolution prodcedure.

3

Problem Definition

The subdivision scheme of the quadtrees presented by Müller et al. [14] (also used in other recent research [18, 22]) is sometimes sub-optimal, as we shall see by example in this chapter. When the approximated distribution is insufficiently refined, it is possible to end up in a situation where the incident radiance at some point $\mathbf{x}$ from direction ω_{i} is actually high, but according to the learned distribution, the probability of sampling direction ω_{i} is low. These situations generally lead to increased variance.

An example of when the subdivision can go wrong is provided in Figures 3.1 and 3.2. Figure 3.1 shows the progressive refinement across iterations to accommodate the yellow region of high incident radiance at some point in a scene. In iteration k, one of the sampled directions is taken from this intense region, leading to the corresponding leaf being subdivided recursively. The number of subdivisions that occur depends on the subdivision threshold factor, covered in section 2.2.2. Iteration $k + 2$ happens to be the final iteration in this example, and the structure of the quadtree is therefore kept and used for the guiding of the rendering phase. Figure 3.2 thus shows the quadtree that ends up being used when guiding.

Figure 3.3 illustrates a problematic situation with the final subdivision. If a direction is sampled from where the red marker indicates, the incident radiance is high, but the PDF according to algorithm 2.2 is low (but non-zero assuming that the flux in the sampled leaf node is non-zero). A strategy that could prove to be useful in dealing with this problem is to somehow reconstruct the quadtrees such that directions near the intense region are explored more. Figure 3.4 illustrates the desired outcome of such an operation. The quadtree is a hierarchical data structure, and the reconstruction procedure is therefore non-trivial. How the reconstruction procedure can be implemented will be explored in this thesis.

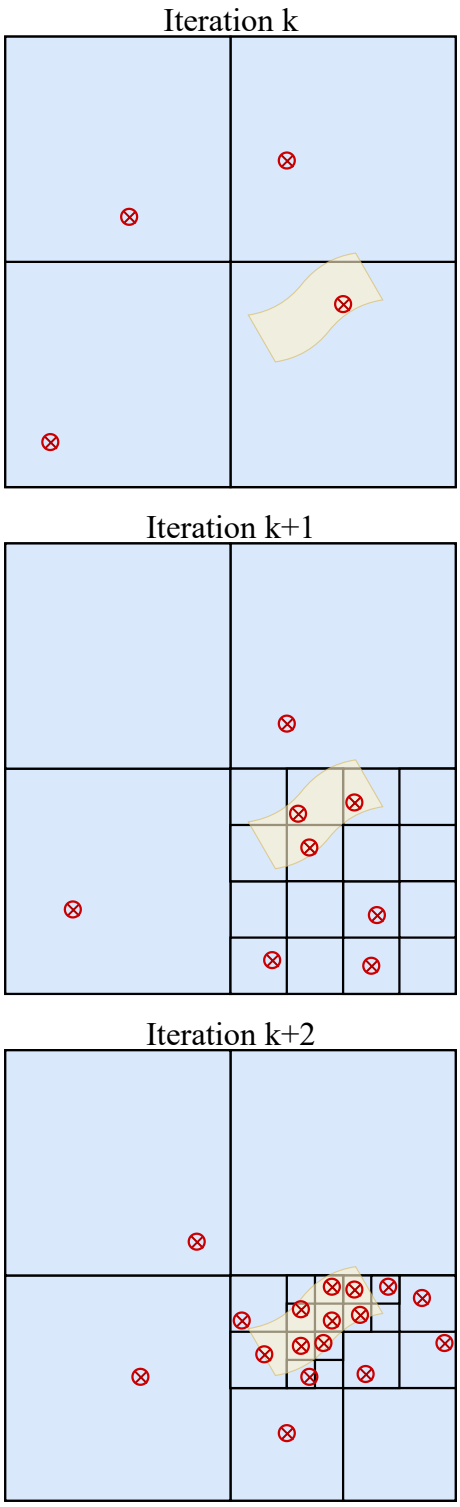


Figure 3.1: A D-tree being progressively refined in iterations according to the samples (red crosses) taken. The yellow band represents a region of high incident radiance that the D-tree should approximate.

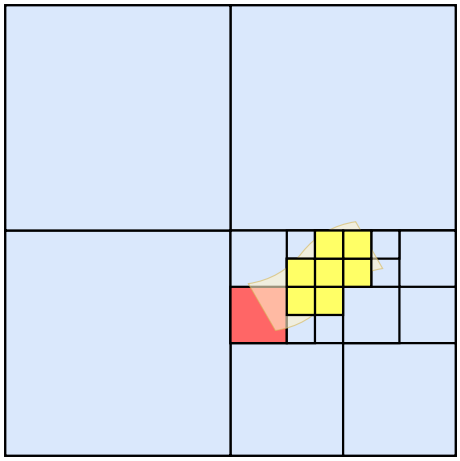


Figure 3.2: The resulting D-tree after taking the samples in iteration $k + 2$ in Figure 3.1.

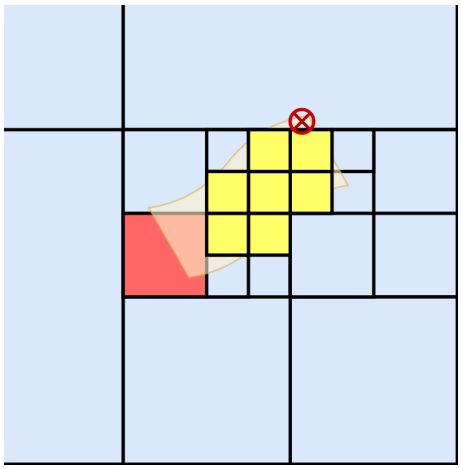


Figure 3.3: Sampling the direction indicated by the red marker would lead to increased variance.

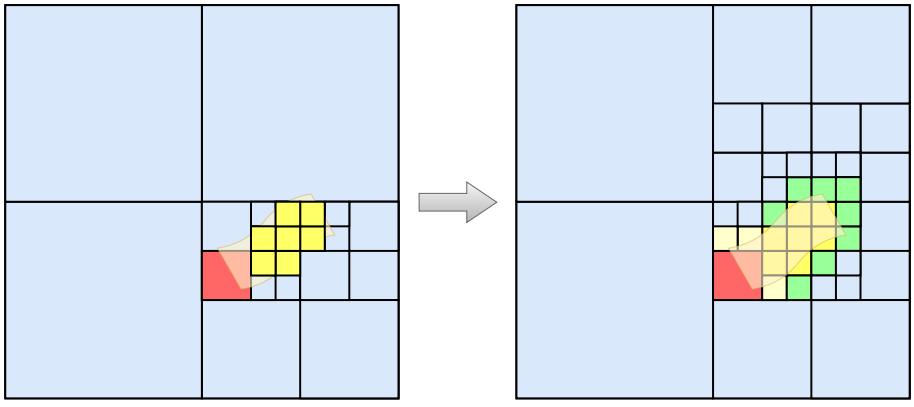


Figure 3.4: Desired outcome of a D-tree reconstruction operation.

4

Implementation

The goal of reconstructing the quadtree is essentially to smooth out the two dimensional function approximation contained within it. As discussed in chapter 2, the steep jumps in the approximated function are rarely an accurate representation of reality. What will be attempted with my implementation is therefore to smoothen, or blur, the incident radiance approximation using a Gaussian denoising procedure.

Gaussian denoising is straight-forward to apply to a one- or two-dimensional function, but it is not obvious how it could be performed when the data is structured hierarchically, as in the quadtree. This chapter will cover one approach, which is to extract data from the quadtrees and map it to a two-dimensional matrix, which can then be convolved with a Gaussian filter, and then be mapped back. Section 4.3 covers the implementation of a convolution algorithm and certain performance optimizations that can be made to minimize the overhead of the overall reconstruction procedure.

4.1 Flattening the Quadtrees

The leaves in a quadtree exist at different depths. When performing Gaussian denoising on a quadtree, one must therefore first select at which depth, or *layer*, to denoise. Leaves deeper in the tree represent a smaller range of directions. It would therefore not be appropriate to simply blur data from different layers directly.

Consider again the quadtree in Figure 3.2. The deepest level of this tree is equal to 4, as the root is considered to be at depth 1. Assume that we wish to blur the data at the deepest level, d_{max}. Each leaf at depth d_{max} would thus correspond to one element in the two-dimensional matrix. In other words, the size of the matrix input for the convolution operation is 2^{d_t} x 2^{d_t}, where d_t is the depth of the target layer we wish to blur.

Leaf nodes at depth $d < d_t$ would cover more than one element in the matrix and must therefore be split when mapping the data to a two-dimensional matrix, as Figure 4.1 shows. Splitting does not mean subdividing the node. It simply means that the flux in the leaves residing in layers above the target depth must be divided by the area they cover in the matrix.

By the same token, if the target layer of the denoising operation is not the deepest layer, i.e $d_t < d_{\text{max}}$, then it would take multiple leaves from levels at depth $d > d_t$ to cover one element of the matrix. Extra care must be taken in this instance, however. Due to the subdivision scheme presented by Müller et al. [14] (discussed in 2.2.2), the flux flowing

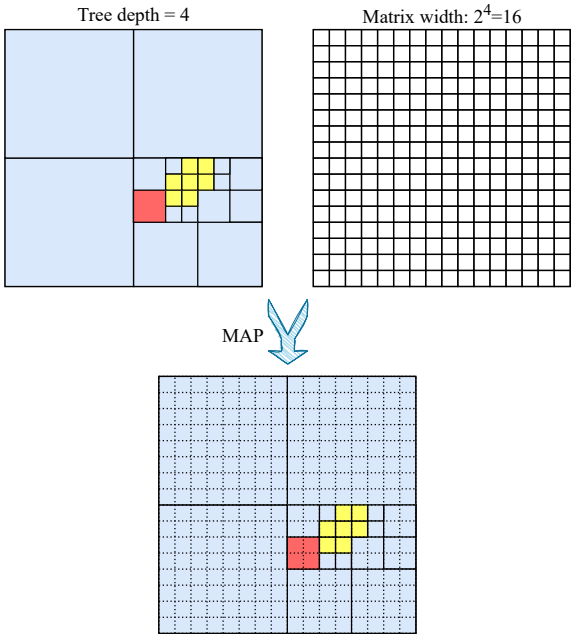


Figure 4.1: Mapping depth 4 of the quadtree to a matrix. Note how certain leaves high in the tree must be split.

through subtrees with significant depths beginning at the target level d_t is disproportionately higher than the flux flowing through leaves at d_t. For this reason, the matrix entry corresponding to the root of a subtree is set to be the sum of the subtree multiplied by the scaling factor $\frac{1}{4^{d_s}}$, where d_s is the depth of the subtree.

The layer-wise approach essentially enables denoising of the data at different resolutions. Some branches of the quadtree may not be very deep, but may still contain noisy information that would benefit from being denoised. If the denoising procedure is only applied at the highest resolution, i.e a matrix of size $2^{d_{max}}$ x $2^{d_{max}}$, these branches would remain largely unaffected. However, while denoising multiple layers is beneficial, the resolution can also be too low. This would lead to blurring of information from directions too far apart to be of any realistic benefit. When implementing the reconstruction algorithm, a minimum depth of 6 was found to work well.

The size of the input matrix of the convolution algorithm grows exponentially with the target depth. This is thus a limiting factor, as it has a big impact on the overhead of the overall reconstruction procedure. Even with a fast convolution algorithm, exponential input growth is hard to overcome. Therefore, quadtrees deeper than 10 are not considered as candidates for reconstruction in this implementation when *workload limiting* (Section 4.5) is enabled.

4.2 Rebuilding and Mapping Back

When the matrix representations of the different layers have been denoised, the results are merged into a single matrix, which is then used to rebuild the quadtree. Merging the matrices is done by expanding smaller ones such that all matrices get the dimensions $2^{d_{\max}}$ x $2^{d_{\max}}$. Then, an element-wise *max*-operation is applied between the matrices.

When rebuilding the quadtree from the resulting matrix, a similar approach to the one presented by Müller et al. [14] was adopted, meaning that no leaf contains more than 1% of the total flux flowing through the quadtree. However, it is possible to adjust the subdivision threshold of the reconstruction algorithm separately, which enables a higher resolution of the guiding quadtrees if desired. This is left as a parameter for the user.

4.3 Efficient Convolutions

All code for the reconstruction algorithm was written in C++, including the convolution algorithm. The starting point of the design was a naive implementation of matrix convolutions, Source Code 4.1, which was then optimized with the various techniques covered in the remainder of this section.

4.3.1 Matrix Padding

One of the inefficiencies of the algorithm in Source Code 4.1 is the excessive index checking in the innermost loop at line 27. Instead of checking indices, it is possible to add rows and columns to the input matrix, creating a margin consisting only of zeros. This way, even if the filter is at the edge of the input signal, it will not move out of bounds of the input matrix. The `if`-statement on line 27 can thus be removed. This does increase the number of floating point operations, but it is still more efficient nevertheless. The benefits of this optimization may depend on the computer architecture. Without branch prediction, for example, this optimization would have a big impact.

4.3.2 Advanced Vector Extensions

Advanced Vector Extensions (AVX) is a set of instructions that extend the x86 architecture and enable *Single Instruction Multiple Data* (SIMD) operations on 256-bit vectors [8]. SIMD instructions are very useful when implementing an efficient convolution algorithm, as multiple elements need to be multiplied and summed each time the filter is moved. 256-bit vectors can fit 8 32-bit floating point numbers. If the filter size is less than 8, for example 7, the filter can be loaded into 7 AVX vectors, where the last element of each vector is simply not used.

Below is an outline of the approach to using AVX in Source Code 4.2, which assumes a filter size of 7. Detailed information about what the AVX-related functions do can be found in Appendix B:

1. The kernel vectors can be pre-loaded before beginning the convolution.

2. For each kernel stride, load 7 AVX vectors with values from the input matrix with respect to the kernel position.
3. For each kernel vector i, multiply it with the input vector i.
4. Sum all the vector products.
5. Reduce the resulting vector sum to a single floating point value.

Unfortunately, the CPU used when implementing Source Code 4.2 did not support *fused multiply-add*, an instruction that both multiplies and adds the intermediate result. Lines 55-57 could be replaced with such an operation, potentially improving the performance [8].

Source Code 4.1: Naive matrix convolution in C++.

```
1  void naiveC2D(const SquareMat<float>& input,
2                SquareMat<float>& output,
3                const SquareMat<float>& kernel) {
4      const int N = input.size();
5      // assuming a square filter
6      const int kRows = kernel.size();
7      const int kCols = kRows;
8      const int kCenter = kRows / 2;
9
10     // rows of input
11     for(int i = 0; i < N; ++i) {
12         // columns of input
13         for(int j = 0; j < N; ++j) {
14             float sum = 0.f;
15             // kernel rows
16             for(int m = 0; m < kRows; ++m) {
17                 int mm = kRows - 1 - m;
18                 // kernel columns
19                 for(int n = 0; n < kCols; ++n) {
20                     int nn = kCols - 1 - n;
21                     // Input matrix indices corresponding
22                     // to filter elements
23                     int ii = i + (kCenter - mm);
24                     int jj = j + (kCenter - nn);
25                     // Check indices
26                     if(ii >= 0 && ii < N && jj >= 0 && jj < N)
27                         sum += input[ii][jj] * kernel[mm][nn];
28                 }
29             }
30             output[i][j] = sum;
31         }
32     }
33 }
```

4.3.3 Loop Tiling

Loop tiling, or *loop blocking*, is a powerful technique to increase the performance of loops by using the CPU cache hierarchy more efficiently. The goal of this loop transformation is to break down the matrices into blocks that fit in the cache, and completing all computations for that block before advancing to the next and loading new values into the cache. An appropriate tile size can usually be calculated by looking up the cache size of the CPU. For the CPU used in this work, a tile size of 32 was found to work well. Line 30, 32, 34, and 36 in Source Code 4.2 are the result of transforming the two loops below.

```
for(int i = kCenter; i < N - kCenter; ++i) {
    for(int j = kCenter; j < N - kCenter; ++j) {
        {...}
    }
}
```

Worth noting, however, is that this particular implementation of loop tiling makes assumptions about the inputs. Since the algorithm in Source Code 4.2 will only convolve matrices where the dimensions are powers of 2, some of the index checking that would otherwise be necessary, if the tile size is not a divisor of the matrix size, could be optimized away.

4.3.4 Matrix Masking

This optimization is grounded in the observation that, quite often, large parts of the input matrices contain identical values. For example, the elements of the upper left quadrant of the matrix in Figure 4.1 is one such region. If the filter is small enough to fit inside this region, there is no purpose in performing any computations for this filter position, because the output element will be the same as the input element. Instead, the filter position can be advanced until it covers a region where the elements of the input are not the same.

This functionality was implemented by creating a mask for each input matrix where the values are either 0 or 1, indicating whether the corresponding element can be skipped or not. As it turns out, this optimization has a large impact on the efficiency of the convolutions in this implementation.

Source Code 4.2: Optimized matrix convolution in C++.

```
void fastConv(const SquareMat<float>& input,
              SquareMat<float>& output,
              const SquareMat<float>& kernel,
              const SquareMat<char>& mask) {
  const int N = input.size();
  // 7 is the largest filter size where rows can fit
  // inside 256 bit AVX vectors (filter size must be odd)
  const int kRows = 7;
  const int kCols = 7;
  const int kCenter = 3;

  // Preload AVX filter vectors
  __m256 kernelVectors[7];
  for(int i = 0; i < kRows; ++i) {
    __attribute__ ((aligned (32))) float kVecRow[8] = { };
    for(int j = 0; j < kCols; ++j)
      kVecRow[j] = kernel[i][j];
    kernelVectors[i] = _mm256_load_ps(kVecRow);
  }

  // Used to load AVX input vectors (must be aligned)
  __attribute__ ((aligned (32))) float inputV[8] = {};
  __m256 inputVectors[7];

  /* Note: this can break for inappropriately
   * sized matrices. The assumption is that the
   * matrix is square and the dimension is a power of 2
   */
  // Row tile
  for(int i = kCenter; i < N - kCenter; i += TILESIZE) {
    // Row within tile
    for(int it = i; it < i + TILESIZE; ++it) {
      // Column tile
      for(int j = kCenter; j < N - kCenter; j += TILESIZE) {
        // Column within tile
        for(int jt = j; jt < j + TILESIZE; ++jt) {
          // Check mask
          if(!mask[it - kCenter][jt - kCenter]) {
            output[it - kCenter][jt - kCenter] = input[it][jt];
            continue;
          }
          // Load AVX input vectors
          int rowIndex = kRows - 1;
          for(int m = it - kCenter; m <= it + kCenter; ++m) {
```

```
            int vectorIndex = kCols - 1;
            for(int n = jt - kCenter; n <= jt + kCenter; ++n) {
              inputV[vectorIndex--] = input[m][n];
            }
            inputVectors[rowIndex--] = _mm256_load_ps(inputV);
          }

          // Multiply input with kernel vectors
          __m256 sumV = _mm256_setzero_ps();
          for(int k = 0; k < kRows; ++k) {
            __m256 product = _mm256_mul_ps(inputVectors[k],
                                           kernelVectors[k]);
            sumV = _mm256_add_ps(product, sumV);
          }

          // Reduce result
          // Reduce sum vector to a single float
          const __m128 x128 =
            _mm_add_ps(_mm256_extractf128_ps(sumV, 1),
                       _mm256_castps256_ps128(sumV));
          const __m128 x64 =
            _mm_add_ps(x128, _mm_movehl_ps(x128, x128));
          const __m128 x32 =
            _mm_add_ss(x64, _mm_shuffle_ps(x64, x64, 0x55));

          output[(it - kCenterY)][(jt - kCenterX)] =
            _mm_cvtss_f32(x32);
        }
      }
    }
  }
}
```

4.4 Switching Target Distribution

The implementation mainly uses the *variance-aware* guiding distribution introduced by Rath et al. [18]. However, as mentioned earlier, this target distribution does not perform very well when there are not enough samples to approximate the distribution. Therefore, this implementation attempts to adopt a strategy of switching between different target distributions depending on the state of the training process. The method referred to as *DS* in the evaluation uses the *simple* target distribution from the beginning, and switches to the variance-aware distribution in the fourth training iteration, in which 16 path samples per pixel are created.

4.5 Workload Limiting

The reconstruction introduces some computational overhead to the overall rendering algorithm in order to improve the guiding distributions. Sometimes, however, the overhead may be too large, and the result would be better if the algorithm had simply kept rendering instead of reconstructing. To avoid this, an automatic workload limiting algorithm was implemented. The decision whether to reconstruct or not is guided by the number of quadtrees that should be reconstructed, their depth, and what iteration the training process is in. The number of samples (and therefore computation time) grows geometrically with the training iterations. Therefore, the time budget that can be allotted to reconstruction also grows geometrically. In essence, the workload is defined as the accumulated number of elements of all the matrices that would have to be convolved. The base threshold, i.e the maximum workload of iteration 0, was set to be $1.5 \cdot 10^7$. The maximum workload in iteration k is thus $2^k \cdot 1.5 \cdot 10^7$.

5

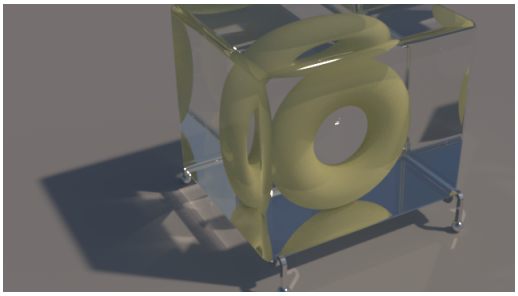
Evaluation

The reconstruction algorithm was integrated into the Mitsuba renderer [9], and results are compared with previous work [14, 18]. The rendering algorithm is evaluated using the four scenes found in Figure 5.1, all of which feature difficult lighting scenarios. As proposed by Rath et al. [18], the metric used to evaluate the performance is the *relMSE*, or *relative mean squared error*, which is the mean of the squared error between a reference image and the image to be evaluated, divided by the squared reference value. To find the best filter for each scene, the average relMSE is plotted against filters of varying standard deviations, as seen in Figure 5.2. The lower limit is set to 0.3 because a standard deviation of less than this makes the denoising insignificant. Figure 5.3 gives a visual example of how the implemented denoising algorithm impacts a certain guiding distribution.

(a) KITCHEN - Also known as ”Country Kitchen”, by Jay-Artist (CC BY 3.0).

(b) LIVING ROOM - Also known as ”The White Room”, by Jay-Artist (CC BY 3.0).

(c) VEACH DOOR - Also known as ”Veach, Ajar”, by Benedikt Bitterli (CC0 1.0).



(d) TORUS - By Olesya Jakob. Available for download from the Mitsuba website [9].

Figure 5.1: The scenes used in the evaluation. Scenes **a)**-**c)** were sourced from Benedikt Bitterli’s website [1].

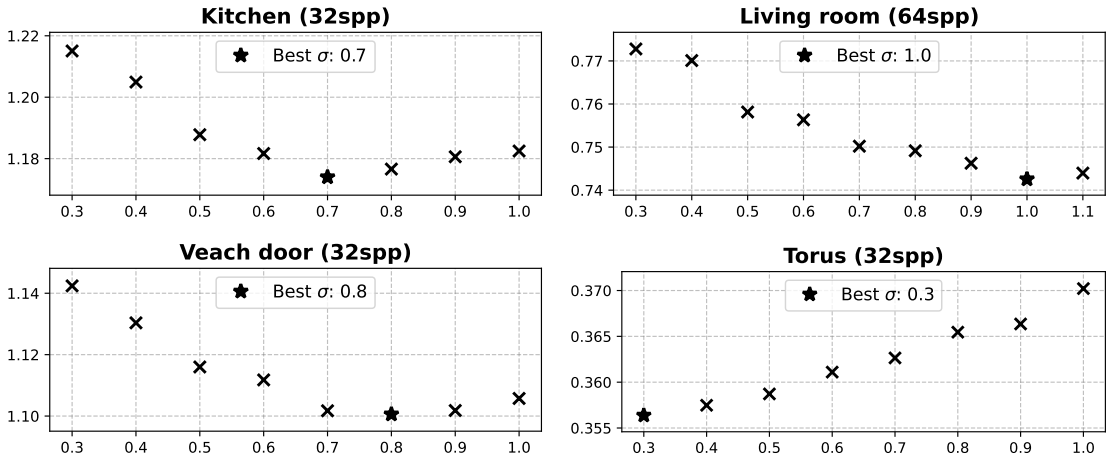


Figure 5.2: Average relMSE plotted against the standard deviation of the Gaussian filter for various scenes. The training budget is given in the parenthesis of the title of the corresponding plot. The rendering budget was 512spp for all scenes.

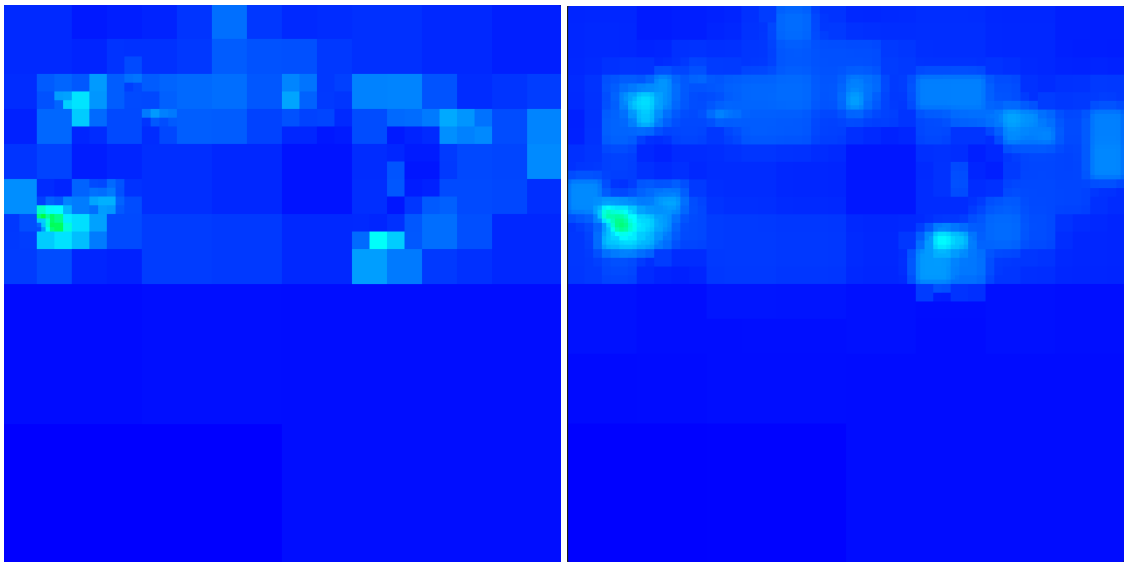


Figure 5.3: A visual example of the implemented reconstruction algorithm using $\sigma = 0.8$

Two approaches to the reconstruction of the guiding quadtrees are evaluated, and results are compared with previous work [14, 18]. In one approach, reconstruction is applied only after the last iteration of the training algorithm has finished. The motivation behind this approach is that it enables a direct comparison between path guiding with reconstruction and previous work using the same guiding distribution.

The iterative learning algorithm used in previous work gives rise to learning inconsistencies due to inherent randomness, which means that the learned incident radiance field differs in quality between different runs of the algorithm. An example of this can be seen in figure 5.4. If reconstruction is not enabled during training, the final SD-tree can be used to render 1) a version of the image with reconstruction enabled, and 2) one where it is disabled. This means that the learning process is identical for images 1) and 2), and differences in outcome can be accredited to the reconstruction procedure, rather than learning inconsistencies.

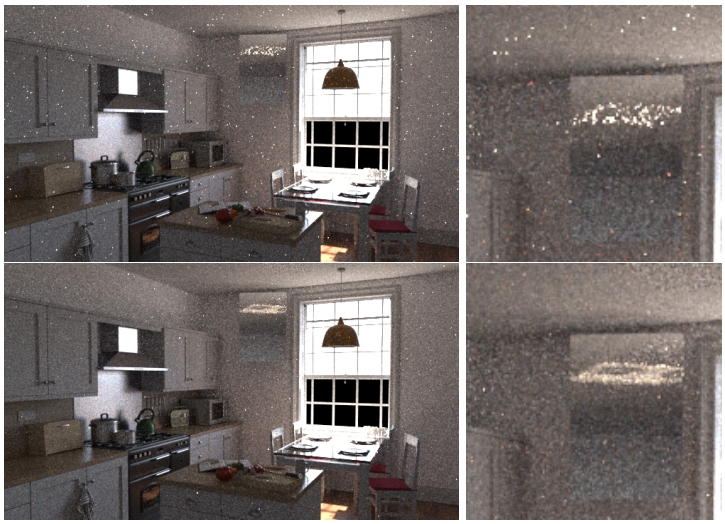


Figure 5.4: Two images rendered with identical training and rendering budgets using previous work [18]. Due to learning inconsistencies, the caustic seen in the mirror is not captured as well in the first image as in the second.

In the second approach, reconstruction is enabled during the training. This means that each iteration of the training algorithm is guided by the reconstructed tree of the previous iteration. Since this affects the learning process, this approach is somewhat more difficult to evaluate, and results need to be averaged over multiple runs of the algorithm to mitigate the effects of learning inconsistencies. Before the impact of the reconstruction is covered, however, Section 5.1 gives an overview of the performance of the convolution algorithm and the computational overhead it introduces.

The evaluation of the reconstruction algorithm was done using the most appropriate filter for each scene, as indicated by Figure 5.2. The machine used was equipped with an Intel Core i7 3770k (4 cores, 8 threads at 3.5 GHz) and 24 GB of memory. The resolution of the images rendered was 640x360.

5.1 Performance of the Convolution Algorithm

Table 5.1 summarizes the impact of the various optimizations discussed in 4.3 in an additive manner. Adding matrix masking alone leads to a speedup of 17x compared to the naive implementation when applied to a randomly selected quadtree of depth 11 from the KITCHEN scene.

Table 5.1: Single thread matrix convolution performance with various optimizations applied. The input matrix was 2048x2048 and the execution time was averaged over 100 runs.

Optimization	Time (ms)
Naive (baseline)	250
+ masking	14.7
+ padding	13.4
+ AVX	12.5
+ tiling	12.2

5.2 Disregarding Reconstruction Overhead

To ascertain how effective the reconstruction could be hypothetically, the performance is first compared with previous work without counting the overhead of the reconstruction procedure. This is done by only regarding the relative MSE as a function of the training budget given in samples per pixel. Furthermore, no upper limit to the depth of the quadtrees is set, and the budgeting algorithm introduced in Section 4.5 is disabled, meaning that reconstruction occurs regardless of the workload.

5.2.1 Reconstructing After Training

Figure 5.5 compares path guiding with reconstruction of the guiding distributions with Rath et al.'s [18] implementation. Note that, in this particular experiment, no target distribution switching takes place, since the purpose here is to discern only how reconstruction affects the quality of the distribution. Müller et al.'s [14] implementation is excluded from this experiment since they use a radiance-based target distribution, which would not be directly comparable in this case.

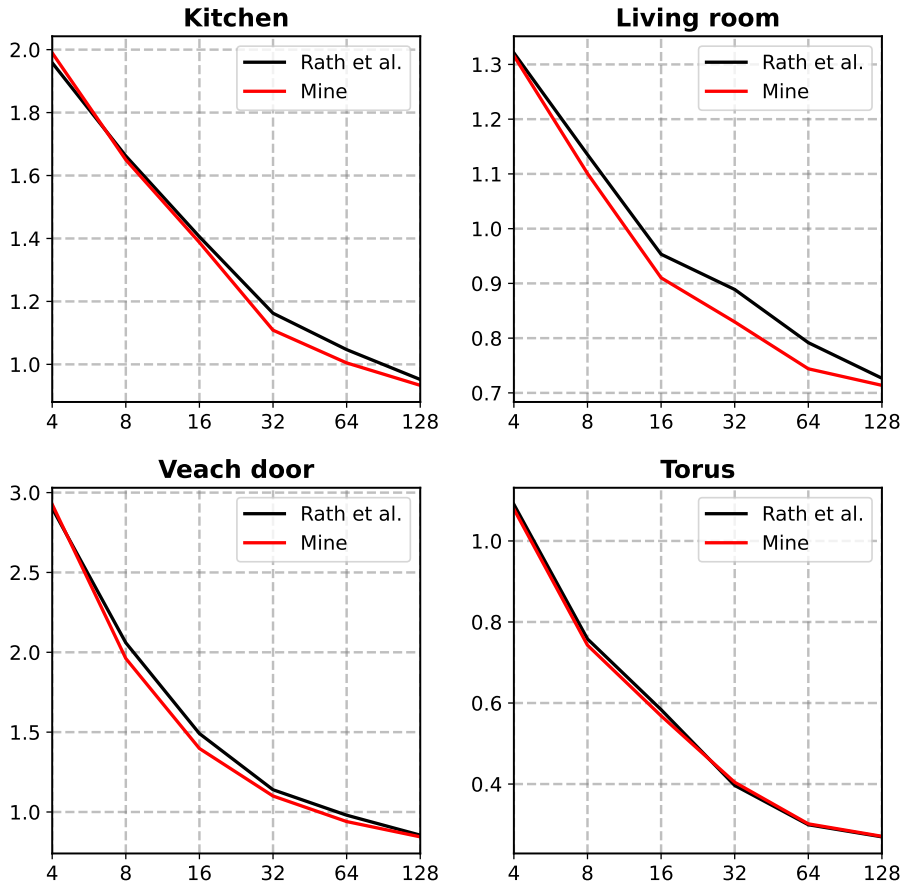


Figure 5.5: Relative MSE plotted against various training budgets using last iteration reconstruction only. The rendering budget was 512spp for all scenes.

Table 5.2: Statistics of Rath et al.'s method and mine using reconstruction after the last training iteration.

Scene	Method	Training	Rendering	relMSE
Kitchen	Rath et al.	32spp	512spp	1.162
	Mine	32spp	512spp	**1.109**
Living Room	Rath et al.	32spp	512spp	0.889
	Mine	32spp	512spp	**0.830**
Veach Door	Rath et al.	32spp	512spp	1.139
	Mine	32spp	512spp	**1.100**
Torus	Rath et al.	32spp	512spp	**0.397**
	Mine	32spp	512spp	0.404

5.2.2 Reconstructing During Training

In this experiment, the impact of reconstruction between the training iterations is evaluated. The method labeled *DS* uses target distribution switching. Workload limiting is still disabled, however, since the purpose of the experiment is to gauge what the upper limits of the improvement can be. To mitigate the effects of learning inconsistencies, the results are averaged over 3 separate runs. Figure 5.6 shows relMSE as a function of the training budget, and Table 5.3 summarizes the result.

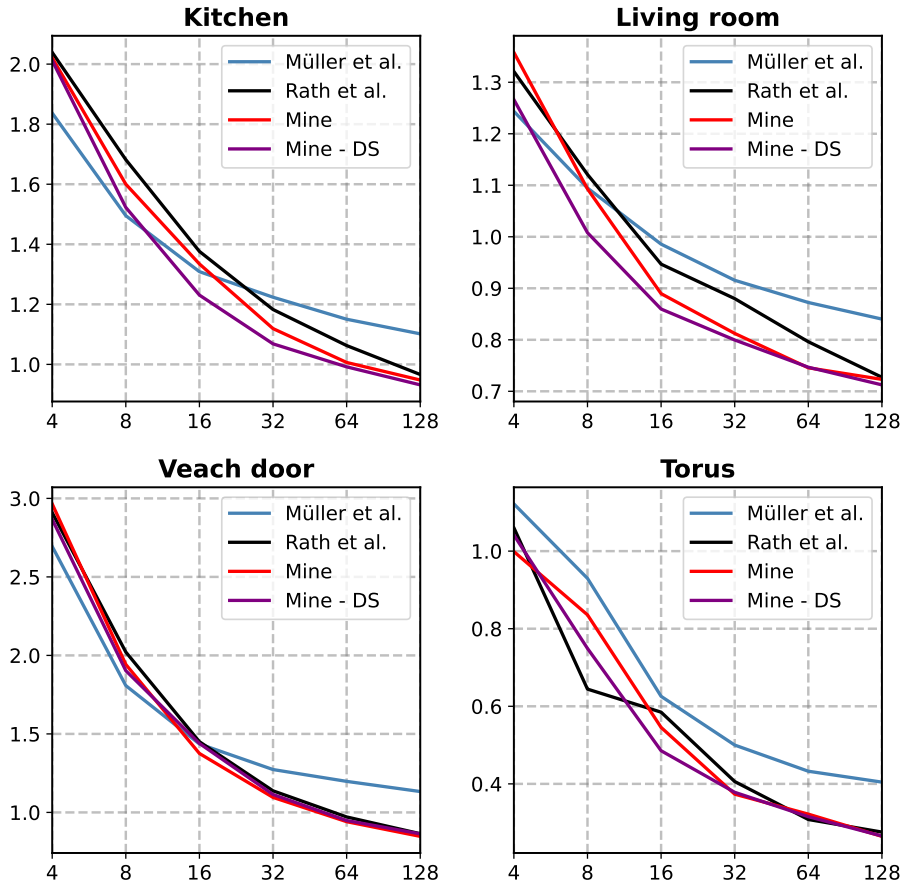


Figure 5.6: Relative MSE plotted against various training budgets using reconstruction between training iterations. The rendering budget was 512spp for all scenes.

Table 5.3: The relMSE of the various algorithms given a training budget of 32spp.

Scene	Method	Training	Rendering	relMSE
KITCHEN	Müller et al.	32spp	512spp	1.224
	Rath et al.	32spp	512spp	1.182
	Mine	32spp	512spp	1.119
	Mine + DS	32spp	512spp	**1.068**
LIVING ROOM	Müller et al.	32spp	512spp	0.916
	Rath et al.	32spp	512spp	0.880
	Mine	32spp	512spp	0.812
	Mine + DS	32spp	512spp	**0.800**
VEACH DOOR	Müller et al.	32spp	512spp	1.273
	Rath et al.	32spp	512spp	1.138
	Mine	32spp	512spp	**1.095**
	Mine + DS	32spp	512spp	1.114
TORUS	Müller et al.	32spp	512spp	0.500
	Rath et al.	32spp	512spp	0.406
	Mine	32spp	512spp	**0.374**
	Mine + DS	32spp	512spp	0.378

5.3 Taking Overhead Into Consideration

In this section, the various algorithms render images using equal time budgets. The training to rendering ratio follows Müller et al.'s proposed scheme, i.e 15% of the time is dedicated to training. The reconstruction workload grows particularly fast in the TORUS and KITCHEN scenes, and the effects of this can be seen in the table below. The method labeled *WL* uses workload limiting. Table 5.5 summarizes the performance of each rendering algorithm with the given time budgets. To mitigate the effects of learning inconsistencies, the results were generated by averaging the relMSE over 5 separate runs of each algorithm.

Table 5.4: The two scenes included in this table exemplify the importance of limiting the overhead of reconstruction.

Scene	Method	Budget	relMSE
KITCHEN	Rath et al.	3.5 min	1.715
	Mine	3.5 min	1.736
	Mine + WL	3.5 min	**1.679**
TORUS	Rath et al.	3 min	**0.263**
	Mine	3 min	0.296
	Mine + WL	3 min	0.270

Table 5.5: Statistics of the various rendering algorithms given a certain time budget.

Scene	Method	Budget	relMSE
KITCHEN	Müller et al.	5 min	1.559
	Rath et al.	5 min	1.434
	Mine + WL	5 min	1.430
	Mine + WL + DS	5 min	**1.395**
LIVING ROOM	Müller et al.	7 min	1.104
	Rath et al.	7 min	0.869
	Mine + WL	7 min	0.867
	Mine + WL + DS	7 min	**0.846**
VEACH DOOR	Müller et al.	7 min	1.316
	Rath et al.	7 min	1.107
	Mine + WL	7 min	1.115
	Mine + WL + DS	7 min	**1.078**
TORUS	Müller et al.	3 min	0.394
	Rath et al.	3 min	**0.255**
	Mine + WL	3 min	0.304
	Mine + WL + DS	3 min	0.280

6

Discussion

This chapter discusses the results obtained during the evaluation, and potential areas of future research.

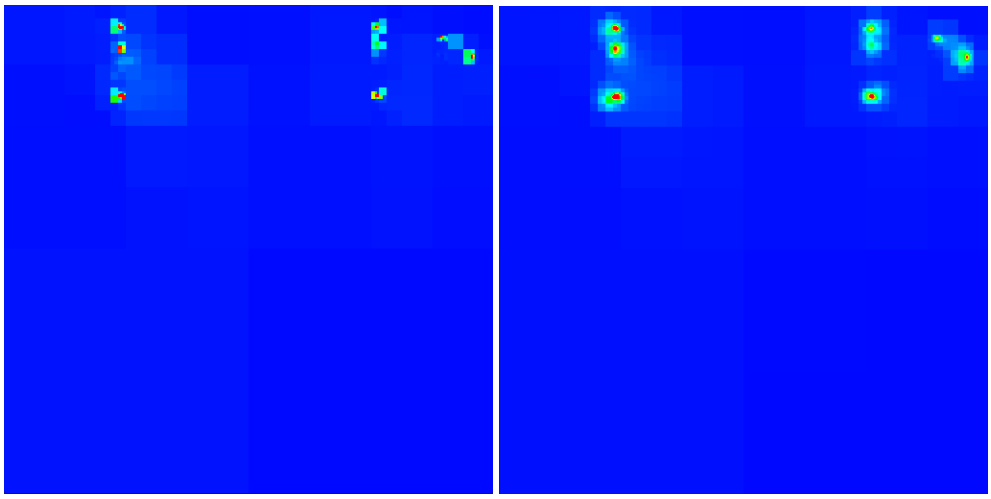
6.1 Discussion of Results

The results in Section 5.2.1 and 5.3 strongly suggest that reconstructing the guiding distributions using Gaussian denoising has benefits for some scenes. The reduction in variance in the rendered images in section 5.2.1 cannot be attributed to learning inconsistencies, because the base SD-tree was identical for both algorithms. The speedup in relMSE for the LIVING ROOM scene is approximately 8%, as indicated by Table 5.3 (without DS). This speaks to the improved quality of the guiding distributions.

Using a simpler target distribution early in the training process and then switching to the variance-aware target distribution seems to yield better results short-term, and potentially long-term as well, as indicated by the LIVING ROOM and KITCHEN scenes, where the DS-method yields a lower error even for longer renders.

Not all scenes seem to benefit from the reconstruction, however. Something that is clear throughout the whole evaluation is that the TORUS scene does not take well to reconstructed D-trees. The reason for this is twofold. The denoising workload for this scene is typically very high, because the trees tend to get deep very quickly. This is because most of the collected radiance estimates are focused in a very small range of directions, as shown in Figure 6.1a. The second reason is somewhat connected to the first: if most of the flux is focused in a small range of directions, applying hierarchical denoising could just decrease the quality of the distribution, because the high intensity areas are quite well approximated already. Therefore, in the case of the TORUS scene, overhead is added while making the guiding distributions worse! A potential solution to this problem could be to gather some statistics from the quadtree to deduce whether it is the type of tree that would benefit from denoising, and making decisions accordingly. For a quadtree like the one in Figure 6.1a, denoising only the deepest layer could be more appropriate, for example.

Choosing the most appropriate filter for the matrix convolutions remains an unsolved problem. For this evaluation, the filter was chosen on a scene-by-scene basis, but for each scene, the same filter was used for all quadtrees for all levels that were selected for denoising. This is probably not an ideal approach, for the reasons discussed above. Layer depth and variance in leaf nodes are some examples of parameters that could be important to



(a) (b)

Figure 6.1: A quadtree, taken from a point on the torus in the TORUS scene, before and after denoising with $\sigma = 0.8$.

consider when generating the filter.

While the optimizations applied to the convolution algorithm increased the performance considerably, it is still not fast enough to afford denoising between every training iteration. For example, if training iteration X takes Y seconds to complete, but the time needed to denoise the resulting guiding distributions is comparable to Y, then denoising is definitely not worth it, because continuing with the next training iteration, taking twice the number of samples in just $2Y$ seconds, would be a much better use of that time. However, if denoising can be applied in some small fraction of the time Y, then it can be considered worth it. The purpose of the workload limiting algorithm is therefore to ensure that the denoising time does not exceed a certain threshold for each training iteration.

In practice, any improvements made to the guiding distributions are almost entirely cancelled out by the overhead, as Table 5.5 shows. Two ways of making Gaussian denoising a more viable strategy for improving the guiding distributions can thus be immediately identified. The first is to reduce the overhead of the algorithm further, which would lead to an actual performance closer to Figures 5.5 and 5.6. The second is to find a better filter selection strategy that would change what the upper limits of the quality improvement of the quadtrees could be. Doing just the latter could be enough to justify the amount of overhead the convolution algorithm imposes in its current state.

Since nothing is added to the data structure of previous work, the memory footprint of the implementation is negligible. The only memory cost imposed by the implementation are the matrices used in-between training iterations, but these are limited to approximately 4 MB per thread. In contrast, the related work by Zhu et al. [23], which uses machine learning for the reconstruction, and a flat data structure instead of quadtrees, adds a significant amount of memory overhead. Their follow-up work [22] does use quadtrees, but there is still some added memory overhead due to the parameters required for the neural network. However, the quality of their reconstructed quadtrees seems to easily justify the memory overhead.

6.2 Future Work

As discussed above, the two approaches to improving the technique further is to either decrease the execution time of the overall reconstruction algorithm, or to increase the quality of the of the reconstruction by making better decisions regarding the filter based on the quadtree in question. Some strategies to reduce the overhead of the algorithm further are, for example, performing the convolutions on the GPU, or descending in quadtrees that are too deep to reconstruct completely. The idea behind the latter is based on the observation that the quadtrees are often very unbalanced, meaning that all parts of the quadtree may not benefit equally from denoising. It would then be possible to descend in the quadtree, selecting a different node as the root of the reconstruction procedure, which would mean an exponential reduction in the workload for every level descended. Figure 6.2 illustrates this.

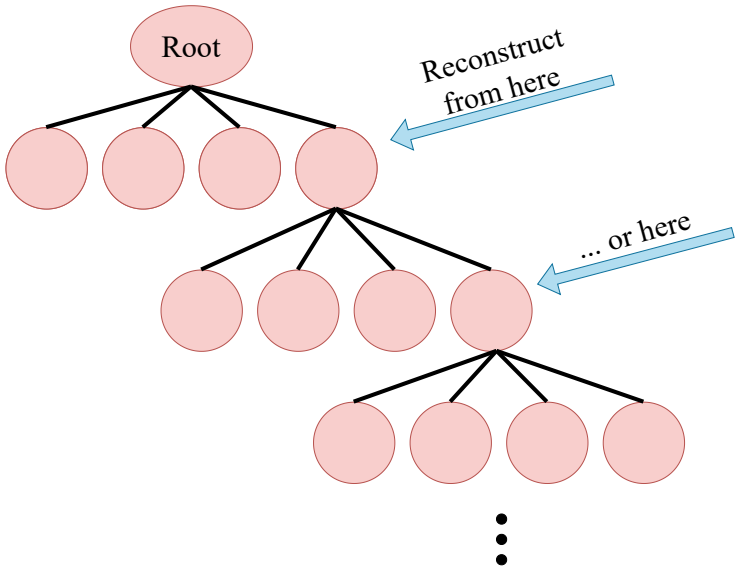


Figure 6.2: When a quadtree is unbalanced, selecting a different root node of the reconstruction could reduce the workload.

6.3 Risk Analysis and Ethical Considerations

The topic of realistic image synthesis is sometimes debated since it can be used for the unethical purposes of spreading misinformation or committing acts of fraud. While these are valid concerns, it is unlikely that this work will create new possibilities in that area because it is focused on such a small aspect of the rendering procedure, and the results do not indicate any groundbreaking improvements.

7

Conclusion

This thesis presents an approach to hierarchically reconstruct quadtree-based incident radiance approximations. The method differs from recent research in that it does not require machine learning, and it does not add to the data structure used in previous work, which keeps the memory footprint low. For some scenes, the implemented algorithm is able to improve the quality of the guiding distributions significantly when the number of path samples is limited. However, the algorithm imposes some computational overhead, which prevents the method from reaching its full potential. The expectation is that this work can be further improved to reduce the overhead, and to increase the quality of the reconstruction by using other filter selection strategies.

Bibliography

[1] Benedikt Bitterli. *Rendering Resources*. 2016. URL: https://benedikt-bitterli.me/resources/ (visited on 05/28/2023).

[2] Brent Burley et al. "The Design and Evolution of Disney's Hyperion Renderer". In: *ACM Transactions on Graphics* 37.3 (July 2018), 33:1–33:22. ISSN: 0730-0301. DOI: 10.1145/3182159. URL: https://dl.acm.org/doi/10.1145/3182159 (visited on 05/29/2023).

[3] Per H. Christensen and Wojciech Jarosz. "The Path to Path-Traced Movies". In: *Foundations and Trends® in Computer Graphics and Vision* 10.2 (Oct. 2016), pp. 103–175. ISSN: 1572-2740. DOI: 10.1561/0600000073. URL: https://doi.org/10.1561/0600000073 (visited on 04/25/2023).

[4] Luca Fascione et al. "Manuka: A Batch-Shading Architecture for Spectral Path Tracing in Movie Production". In: *ACM Transactions on Graphics* 37.3 (Aug. 2018), 31:1–31:18. ISSN: 0730-0301. DOI: 10.1145/3182161. URL: https://dl.acm.org/doi/10.1145/3182161 (visited on 05/29/2023).

[5] Iliyan Georgiev et al. "Arnold: A Brute-Force Production Path Tracer". In: *ACM Transactions on Graphics* 37.3 (Aug. 2018), 32:1–32:12. ISSN: 0730-0301. DOI: 10.1145/3182160. URL: https://dl.acm.org/doi/10.1145/3182160 (visited on 05/29/2023).

[6] Sebastian Herholz et al. "Product Importance Sampling for Light Transport Path Guiding". In: *Computer Graphics Forum* 35.4 (July 2016), pp. 67–77. ISSN: 0167-7055.

[7] Heinrich Hey and Werner Purgathofer. "Importance sampling with hemispherical particle footprints". In: *Proceedings of the 18th Spring Conference on Computer Graphics*. SCCG '02. New York, NY, USA: Association for Computing Machinery, Apr. 2002, pp. 107–114. ISBN: 978-1-58113-608-1. DOI: 10.1145/584458.584476. URL: https://dl.acm.org/doi/10.1145/584458.584476 (visited on 05/29/2023).

[8] Intel. *Intel® C++ Compiler Classic Developer Guide and Reference*. URL: https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-9/overview.html (visited on 05/23/2023).

[9] Wenzel Jakob. *Mitsuba*. 2010. URL: https://www.mitsuba-renderer.org/index_old.html.

[10] Henrik Wann Jensen. “Importance Driven Path Tracing using the Photon Map”. In: *Rendering Techniques ’95*. Ed. by Patrick M. Hanrahan and Werner Purgathofer. Vienna: Springer Vienna, 1995, pp. 326–335. ISBN: 978-3-7091-9430-0.

[11] James T. Kajiya. “The rendering equation”. In: *ACM SIGGRAPH Computer Graphics* 20.4 (Aug. 1986), pp. 143–150. ISSN: 0097-8930. DOI: 10.1145/15886.15902. URL: https://dl.acm.org/doi/10.1145/15886.15902 (visited on 04/25/2023).

[12] Eric Lafortune and Yves Willems. “Bi-Directional Path Tracing”. In: *Proceedings of Third International Conference on Computational Graphics and Visualization Techniques (Compugraphics’* 93 (1993).

[13] Eric P. Lafortune and Yves D. Willems. “A 5D Tree to Reduce the Variance of Monte Carlo Ray Tracing”. en. In: *Rendering Techniques ’95*. Ed. by Patrick M. Hanrahan and Werner Purgathofer. Eurographics. Vienna: Springer, 1995, pp. 11–20. ISBN: 978-3-7091-9430-0. DOI: 10.1007/978-3-7091-9430-0_2.

[14] Thomas Müller, Markus Gross, and Jan Novák. “Practical Path Guiding for Efficient Light-Transport Simulation”. In: *Computer Graphics Forum* 36 (July 2017), pp. 91–100. DOI: 10.1111/cgf.13227.

[15] Thomas Müller et al. “Neural Importance Sampling”. In: *ACM Transactions on Graphics* 38.5 (Oct. 2019), 145:1–145:19. ISSN: 0730-0301. DOI: 10.1145/3341156. URL: https://dl.acm.org/doi/10.1145/3341156 (visited on 05/29/2023).

[16] Thomas Müller et al. “Real-time neural radiance caching for path tracing”. In: *ACM Transactions on Graphics* 40.4 (July 2021), 36:1–36:16. ISSN: 0730-0301. DOI: 10.1145/3450626.3459812. URL: https://dl.acm.org/doi/10.1145/3450626.3459812 (visited on 05/29/2023).

[17] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering: From Theory To Implementation*. English. 3rd ed. Morgan Kaufmann, Nov. 2016. ISBN: 978-0-12-800645-0. URL: https://pbr-book.org/ (visited on 04/28/2023).

[18] Alexander Rath et al. “Variance-aware path guiding”. In: *ACM Transactions on Graphics* 39.4 (Aug. 2020), 151:151:1–151:151:12. ISSN: 0730-0301. DOI: 10.1145/3386569.3392441. URL: https://doi.org/10.1145/3386569.3392441 (visited on 04/25/2023).

[19] Lukas Ruppert, Sebastian Herholz, and Hendrik P. A. Lensch. “Robust fitting of parallax-aware mixtures for path guiding”. In: *ACM Transactions on Graphics* 39.4 (Aug. 2020), 147:147:1–147:147:15. ISSN: 0730-0301. DOI: 10.1145/3386569.3392421. URL: https://dl.acm.org/doi/10.1145/3386569.3392421 (visited on 05/29/2023).

[20] Eric Veach and Leonidas J. Guibas. “Metropolis light transport”. In: *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*. SIGGRAPH ’97. USA: ACM Press/Addison-Wesley Publishing Co., Aug. 1997, pp. 65–76. ISBN: 978-0-89791-896-1. DOI: 10.1145/258734.258775. URL: https://dl.acm.org/doi/10.1145/258734.258775 (visited on 04/25/2023).

[21] Jiří Vorba et al. “On-line learning of parametric mixture models for light transport simulation”. In: *ACM Transactions on Graphics* 33.4 (July 2014), 101:1–101:11. ISSN: 0730-0301. DOI: 10.1145/2601097.2601203. URL: https://doi.org/10.1145/2601097.2601203 (visited on 04/25/2023).

[22] Shilin Zhu et al. “Hierarchical neural reconstruction for path guiding using hybrid path and photon samples”. In: *ACM Transactions on Graphics* 40.4 (July 2021), 35:1–35:16. ISSN: 0730-0301. DOI: 10.1145/3450626.3459810. URL: https://dl.acm.org/doi/10.1145/3450626.3459810 (visited on 04/25/2023).

[23] Shilin Zhu et al. “Photon-Driven Neural Reconstruction for Path Guiding”. In: *ACM Transactions on Graphics* 41.1 (Nov. 2021), 7:1–7:15. ISSN: 0730-0301. DOI: 10.1145/3476828. URL: https://dl.acm.org/doi/10.1145/3476828 (visited on 04/25/2023).

A

Path Tracing Algorithm

Algorithm A.1: Path tracing psuedo-code

```
1  function traceRay(scene, o, ω_o, S):
2      // find intersection point and surface normal
3      (object, x, N, hit) ← intersect(scene, o, ω_o)
4      if not hit:
5          return environmentLight(o, ω_o)
6      // if the object is a non-reflective light source,
7      // the path can be terminated
8      else if object is PureLightSource:
9          return emittedLightOf(object)
10     (D, pdf) ← sampleDirection(x, N)
11     L_r ← 0
12     for ω_i in D:
13         L_r ← L_r + traceRay(scene, x, ω_i, S) ·
14             brdf(x, ω_o, ω_i) · cos(N, ω_i) / pdf
15     done
16     L_e ← emittedLightOf(object)
17     L_o ← L_e + L_r
18     return L_o
19 end function
20
21 function traceImage(scene, S):
22     // S: number of samples
23     // scene: the scene to be rendered
24     P ← CameraOrigin
25     for n in [0, PathsPerPixel − 1]:
26         for each (i, j) in Pixels:
27             PX ← PixelPoint
28             // calculate direction from camera to pixel
29             d ← (PX − P)/||PX − P||
30             Image(i, j) ← (1 / n+1)(n·Image(i, j) +
31                                 traceRay(scene, P, d, S))
32         done
33     done
34 end function
```

B

Intrinsics

The following is a description of the intrinsics functions used in the implementation [8].

- `__m256 _mm256_load_ps(float const * mem_addr)`
  - Loads 8 single-precision floating point numbers from a memory block beginning at `mem_addr` into a vector.
  - Returns a 256-bit `__m256` AVX vector.
- `void _mm256_store_ps(float * mem_addr, __m256 a)`
  - Stores the 8 single-precision floating point numbers of vector `a` to a memory block beginning at `mem_addr`.
- `__m256 _mm256_mul_ps(__m256 a, __m256 b)`
  - Performs element-wise multiplication of two vectors `a` and `b`, each containing 8 single-precision floating point numbers.
  - Returns a 256-bit vector containing the result.
- `__m256 _mm256_add_ps(__m256 a, __m256 b)`
  - Performs element-wise addition of two vectors `a` and `b`, each containing 8 single-precision floating point numbers.
  - Returns a 256-bit vector containing the result.
- `__m128 _mm256_extractf128_ps(__m256 a, const int imm8)`
  - Shuffle single-precision (32-bit) floating-point elements in a using the control in `imm8`, returning the result.
- `__m128 _mm256_castps256_ps128(__m256 a)`
  - Cast vector of type `__m256` to type `__m128`. Only used for compilation.
- `__m128 _mm_movehl_ps(__m128 a, __m128 b)`
  - Creates a vector `dst`, moves the upper 2 single-precision (32-bit) floating-point elements from `b` to the lower 2 elements of `dst`, and copies the upper 2 elements from `a` to the upper 2 elements of `dst`.
- `__m128 _mm_shuffle_ps(__m128 a, __m128 b, unsigned int imm8)`
  - Shuffles single-precision (32-bit) floating-point elements in `a` using the control in `imm8`, returns the result.