Marvin Reza

Efficient Sample Reusage in Path Space for Real-Time Light Transport

Master's thesis in Computer Science
Supervisor: Theoharis Theoharis
June 2021

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of Science and Technology

Marvin Reza

Efficient Sample Reusage in Path Space for Real-Time Light Transport

Master's thesis in Computer Science
Supervisor: Theoharis Theoharis
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Abstract

In this thesis, we investigate the current state of real-time global illumination techniques. Specifically, we look at the problem of computing the direct lighting in virtual scenes with thousands, and even millions, of emissive objects. Efficient many-light sampling is an intricate problem that remains even in offline rendering—that is, without the real-time constraints. Many of the problems stem from the difficulties of efficiently determining which lights that contribute the most at given points.

To address this, we propose an algorithm that extends the current state of the art, ReSTIR. In contrast to ReSTIR, our algorithm, Path Space Importance Resampling (PSIR), additionally samples its candidates from a hash table that stores light samples in path space. The hash table is constructed by discretizing the path space vertices into voxels of dynamic size. Subsequently, we insert lights stochastically into the hash table based on each light's expected contribution. Accordingly, the data structure performs an unordered grouping of lights into local pools, which facilitates effective filtering of distant and weak lights.

We test our algorithm and compare it against ReSTIR. The results show that our algorithm consistently outperforms or equals ReSTIR, both qualitatively and quantitatively. However, in most cases, we see that the differences between the results are relatively minimal. Nonetheless, we also see that PSIR comes at a performance cost depending on the scene complexity, screen resolution, and the number of lights in the scene.

Sammendrag

I denne oppgaven undersøker vi ulike eksisterende løsninger for sanntidssimulering av global belysning. Nærmere bestemt, ser vi på problemstillingen rundt det å beregne mengden av direkte belysning i virtuelle scener med tusenvis, og til og med millioner, av lysemitterende objekter. Effektiv utvelging av mange lys er et komplekst problem som også oppstår i offline rendering—det vil si, uten hensyn til sanntidsbegrensninger. Mange av problemene som oppstår kommer av vanskelighetene rundt det å effektivt bestemme hvilke lys som bidrar mest til gitte områder av scenen.

For å ta tak i disse problemene foreslår vi i denne oppgaven en utvidelse til den nåværende state-of-the-art algoritmen, ReSTIR. I motsetning til ReSTIR, vil vår algoritme, Path Space Importance Resampling (PSIR), i tillegg trekke et tilfeldig utvalg av lyskandidater fra en hashtabell som lagrer dem i "path space". Hashtabellen er konstruert ved å diskretisere punktene i path space til "voxler" av dynamiske størrelser. Dermed kan vi utføre stokastisk innsetting av lysdata, med sannsynligheter basert på lysenes forventede bidrag. Følgelig kan det betraktes at datastrukturen utfører en uordnet gruppering av lysene inn i lokale grupper, hvilket fasiliterer for effektiv filtrering av fjerne og svake lys.

Vi tester vår algoritme og sammenlikner den med ReSTIR. Resultatene viser at vår algoritme konsekvent oppnår bedre eller like resultater som ReSTIR—både kvantitativt og kvalitativt. Imidlertid ser vi i de fleste tilfeller at forskjellene mellom de to algoritmene er relativt minimale. Riktignok ser vi også at PSIR har en ytelseskostnad som, blant annet, avhenger av scene kompleksiteten, skjermstørrelsen og antallet lys i scenen.

Acknowledgements

First and foremost, I would like to express my sincerest gratitude to my advisor, Theoharis Theoharis, for his continuous support, supervision, motivation, and advice during this thesis. It was mainly because of the discussions we had before the start of this thesis and the invaluable guidance he gave that I could work on something I genuinely had an interest in.

Finally, none of this would have been possible without the unwavering love and patience of my family, friends, and loved ones, and my heartfelt gratitude goes out to them.

List of Abbreviations

BRDF	Bidirectional reflectance distribution function
BVH	Bounding volume hierarchy
GPU	Graphics processing unit
NEE	Next-event estimation
PDF	Probability density function
PSIR	Path space importance resampling
ReSTIR	Reservoir-based spatiotemporal importance resampling
RIS	Resampled importance sampling
RMSE	Relative mean squared error
spp	Samples per pixel
TAA	Temporal anti-aliasing

Nomenclature

$\mathbf{n}$	Normal vector
ω	Direction vector
Φ	Radiant flux
E	Irradiance
f_r	BRDF
L	Incident or exitant radiance
M	Radiant exitance

Contents

CHAPTER 1

Introduction

Photorealistic rendering of virtual scenes has a long-standing tradition in the field of computer graphics. With an ever-increasing demand for visual fidelity and realism of these renderings and widespread use in the entertainment industry, it remains an important research topic to this day.

One aspect of achieving photorealism in rendering is *light transport*, which studies light and its interactions with the virtual world. Ideally, the light transport algorithms should capture the *global illumination* in a scene. The rendering equation [Kajiya, 1986] provides a framework that facilitates simulation of global illumination. Interestingly, many of the light transport techniques can be viewed as methods of approximation of this framework [Arvo and Kirk, 1990]. Informally, the rendering equation expresses that the outgoing light from any point depends on the reflected and emitted light at every other point in the scene. However, simulating many light bounces quickly becomes computationally infeasible. Accordingly, many of the most popular global illumination algorithms are based on *Monte Carlo methods*, where random sampling forms a natural part of the process that generates the image. These methods do not necessarily differ in the visual fidelity of the images they can compute. In fact, for a given scene, most of them will produce the same solution if run long enough. Instead, they mainly differ in the *time* required to produce a visually acceptable result. This is of practical significance since we only ever have a finite amount of available time to render an image.

In this thesis, we are concerned with the problem of *sampling many lights*. When there are many light sources in a scene, it becomes infeasible to trace shadow rays to all of them. Furthermore, finding the lights that contribute the most at a given point depends on various factors, for instance, each light's visibility to that point, the reflectance distribution function at the point, and the light source's emission characteristics. Our objective, as such, is to find and understand the limitations of the current state of the art for sampling many light sources and propose a solution to these problems.

For these purposes, we present an extension to the algorithm which, at the time of writing, is considered the current state of the art, namely, ReSTIR. Our proposed algorithm extends ReSTIR by facilitating for it to perform resampling in path space and, thus, enable it to sample higher quality light samples. Accordingly, we conduct an experiment and present the corresponding results to investigate the changes in the rendering quality when applying our proposed version of ReSTIR. Moreover, we accompany the results with an analysis to understand and evaluate the costs and benefits of our presented algorithm. Finally, note that, in this work, we only account for direct lighting on primary surfaces. However, our approach is not inherently limited in this regard and may be extended to higher-order bounces and indirect lighting in future work.

1.1 Notation and Conventions

Throughout the text, the author (Marvin Reza) uses the author's "we" and, for the reader's convenience, we have included a nomenclature listing symbols and abbreviations used in this work. It precedes the table of contents and this introduction. We have tried to keep the notation as standard as possible while unifying it across the entire work. For mathematical notations regarding rendering algorithms and equations, we try to follow the notation of [Pharr et al., 2016].

1.2 Thesis Overview

The rest of the thesis is structured as follows:

- Chapter 2 contains background material on Monte Carlo rendering, ray tracing, physically based rendering, and global illumination. The main focus is on understanding why global illumination is difficult, and why its results can be so noisy.
- Chapter 3 presents the related work, with a focus on others' work on denoising and many-light sampling.
- Chapter 4 presents our proposed algorithm and gives an in-depth explanation of the algorithms and data structure it is built upon.
- Chapter 5 starts by explaining the experiment conducted to evaluate the proposed algorithm and, accordingly, presents the corresponding results and accompanies it with a discussion.

- Finally, Chapter 6 concludes the work and the results in this thesis and mentions exciting avenues of future work.

CHAPTER 2

Theory

This chapter contains the material to back up our work in this thesis. In order to keep this chapter succinct, some of the material is briefly touched upon for readers with little to no background in Monte Carlo rendering, ray tracing, or physically based rendering. Some of the more complex topics and proofs are outside the scope of this thesis. Nonetheless, we include the corresponding references, so the reader can look them up if so desired.

2.1 Monte Carlo Integration

Monte Carlo integration is a probabilistic method for numerically integrating functions. Rendering methods based on Monte Carlo integration were first introduced to graphics by Cook et al. [Cook et al., 1984] in order to render distributed effects such as motion blur, depth of field, and soft shadows. Kajiya later applied the Monte Carlo method to the rendering equation to render global illumination effects [Kajiya, 1986].

2.1.1 Basic Monte Carlo Integration

Monte Carlo integration is based on the fact that the integral $I = \int_\Omega f(x)dx$, for integrand f and domain Ω, can be approximated with the following estimator:

$$\hat{I} = \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{p(x_i)} \tag{2.1}$$

Where N random samples $\{x_1, \ldots, x_N\}$ are drawn from a *sampling distribution* with probability density function (PDF) p. This makes the estimator $\hat{I}$ a random variable and, from this, we have that its *expected value* can be computed as:

$$E\left[\hat{I}\right] = E\left[\frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{p(x_i)}\right] \overset{i.i.d}{=} \frac{1}{N}\sum_{i=1}^{N}\int_{\Omega}\frac{f(x)}{p(x)}p(x)dx = \int f(x)dx \tag{2.2}$$

Where the second equality is due to the assumption that the samples are independent and identically distributed (i.i.d) random variables, and that the expected value of a random variable X with PDF $g(x)$ is defined as:

$$E[X] = \int_{\mathbb{R}} xg(x)dx$$

The derivation in Equation 2.2 shows that the estimator $\hat{I}$ yields the correct result on average—or, in other words, that the estimator is *unbiased*. Moreover, by the law of large numbers, we have that the estimator is guaranteed to converge to the expected value as $N \to \infty$, as long as $p(x_i) > 0$ whenever $f(x_i) \neq 0$.

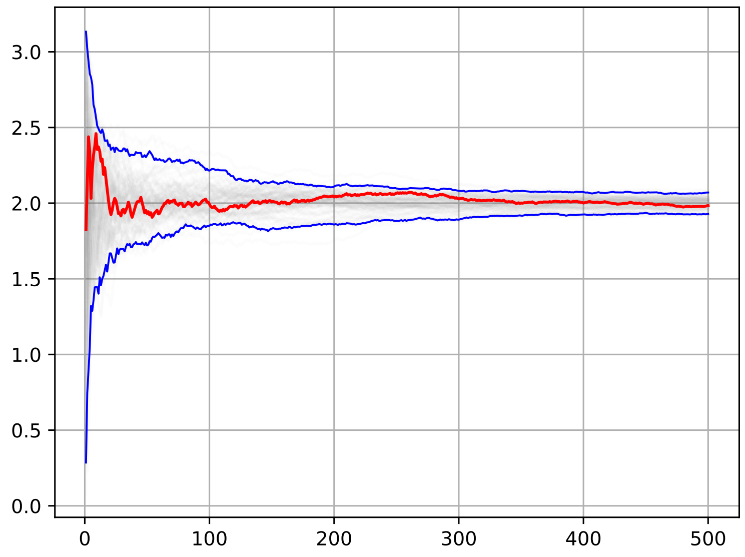


Figure 2.1: Estimating $\int_0^{\pi}\sin(x)dx$ with the Monte Carlo method. The x-axis represents the number of samples N, and the y-axis represents the estimated value of the integral $\hat{I}$. We plot 1000 runs of Monte Carlo integration in grey and the 2.5th and 97.5th percentile in blue to see how the variation in y changes with sample size. The red line denotes a sample path. Note that all estimation runs converge towards the correct result and that the variance decreases as the sample size increases.

In Figure 2.1, multiple runs of Monte Carlo estimation of the integral $I = \int_0^\pi \sin(x)dx$ using Equation 2.1 are shown. In fact, this integral can be solved analytically:

$$I = \int_0^\pi \sin(x)dx = -\cos(x)\Big|_0^\pi = 2$$

The figure illustrates a large amount of variation for all the runs when the sample count is low. In addition, it shows how all estimates converge towards the correct result $I = 2$ as the sample count is increased. Moreover, it also illustrates that when the sample count is high, newer samples appear to have less and less effect on the final result. From closer inspections of the Monte Carlo estimator (Equation 2.1), we see this happens because each sample is scaled by $1/N$; hence, each sample will have less of an effect on the overall value when the sample count N is high.

Finally, it is worth mentioning that Monte Carlo integration is rather popular nowadays due to its ability to handle general, non-continuous, high dimensional integrals [Kroese et al., 2014]—it only requires that f can be evaluated at the sample points. This generality and simplicity make Monte Carlo integration ideal for computer graphics, where we often need to calculate multi-dimensional integrals with integrands that are seldom well-behaved.

2.2 Variance of the Monte Carlo Estimator

The variance of the Monte Carlo estimator in Equation 2.1 is:

$$Var\left[\hat{I}\right] = Var\left[\frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{p(x_i)}\right] = \frac{1}{N}Var\left[\frac{f(x_i)}{p(x_i)}\right] \tag{2.3}$$

This has multiple implications for the estimator $\hat{I}$. For instance, as long as $Var[f(x_i)/p(x_i)]$ is finite, then the estimator $\hat{I}$ will be consistent. Moreover, Equation 2.3 shows that the variance will approach zero as $N \to \infty$. This condition is met if $p(x_i) > 0$ whenever $f(x_i) \neq 0$.

However, Equation 2.3 also shows an inherent limitation of Monte Carlo integration: The variance of the estimator $\hat{I}$ only decreases linearly with respect to N, and therefore the standard deviation only decreases proportionally to $\sqrt{N}$. Consequently, to decrease the expected integration error by, say, a factor of two, we would have to increase the number of samples N by a factor of four—a relatively poor convergence rate. This slow convergence is illustrated in Figure 2.1, where we see that the rate of variance reduction is decreased for increasing sample counts.

2.2.1 Importance Sampling

Importance sampling is one of the most important variance reduction tools in Monte Carlo integration, mainly due to the generality it provides. We saw earlier from Equation 2.3 that we can decrease the variance by increasing the sample count N. However, from closer inspection of the mentioned equation, we notice that the variance of the Monte Carlo estimator $\hat{I}$ also depends on the ratio $\frac{f(x_i)}{p(x_i)}$. Thus, if the variance of this ratio is decreased, then the overall variance can also be reduced without increasing the number of samples N. In other words, the closer the sampling density $p(x_i)$ approximate the integrand $f(x_i)$, the lower the variance.

Importance sampling refers to the technique of choosing the sampling density p to minimize the variance of this ratio. Ideally, we want $p \propto f$, that is, with $p(x) = \frac{f(x)}{I} = \frac{f(x)}{\int f(x)dx}$. This gives us a constant ratio for all x_i and a variance of:

$$Var\left[\frac{f(x)}{p(x)}\right] = Var\left[\frac{f(x)}{f(x)/I}\right] = Var\left[I\right] = 0 \tag{2.4}$$

Unfortunately, building such a zero-variance estimator is most of the time not practical as it would require knowledge of f in order to normalize the distribution p. This also dramatically limits our sampling distribution choices, as the distributions must be normalized and should be easy to generate samples from. Consequently, many algorithms rely on building useful approximations to f.

One such practical approximation heuristic is to sample from parts of the integrand f. In particular, if the integrand is a product function, the distribution p is typically chosen to match a few of the terms, but seldom all of them. This is because the complexity of the distribution can quickly grow in the number of terms [Pharr et al., 2016].

Note that we still need to be careful when selecting our sampling distributions, as it is also possible to increase the estimator's variance if we choose p poorly. Indeed, this is what often happens in global illumination applications since the exact form of the integrand f is unknown a priori [Talbot et al., 2005].

2.2.2 Resampled Importance Sampling

As discussed in the previous subsection, the sampling distribution p should be normalized and easy to sample from (e.g., using rejection sampling or inversion sampling) for the Monte Carlo estimator $\hat{I}$ to be well-behaved. To overcome the restriction of the sampling distribution p, Talbot et al. modify the Monte Carlo estimator [Talbot et al., 2005]. In particular, $M \geq 1$ candidate samples $\mathbf{x} = \{x_1, \ldots, x_M\}$ are instead generated from a source distribution q that may be sub-optimal (e.g., q can be unnormalized), but easy to sample from. For each candidate sample x_i a weight $w(x_i)$ is generated:

$$w(x_i) = \frac{\hat{p}(x_i)}{q(x_i)} \tag{2.5}$$

Where $\hat{p}(x)$ is the desired target PDF, for which no practical sampling algorithm may exist.

Accordingly, with the generated candidates $\mathbf{x}$ their and corresponding weights $w(x_i)$, we *resample* these candidates by drawing $N \ll M$ samples $\mathbf{y} = \{y_1, \ldots, y_N\}$ *with replacement* from $\mathbf{x}$, with probabilities proportional to the proposal weights $w(x_i)$. The *resampled importance sampling* (RIS) estimator with *RIS weights* $w(x_i)$ is defined as:

$$\hat{L}_{\text{ris}}^{N,M} = \frac{1}{N} \sum_{i=1}^{N} \left(\frac{f(y_i)}{\hat{p}(y_i)} \right) \cdot \frac{1}{M} \sum_{j=1}^{M} w(x_j) \tag{2.6}$$

When $M = 1$, the samples are marginally distributed according to q. As $M \to \infty$, the distribution of each sample approaches $\hat{p}$. To illustrate this, Figure 2.2 shows the distribution of the RIS-generated samples for different values of M when the candidate and target distributions, respectively, are given as $q \sim \mathcal{U}(0, 1.5)$ and $\hat{p} = \cos(\theta) + \sin^4(6\theta)$.

Talbot et al. give proof of the algorithm's unbiasedness in their work [Talbot et al., 2005]. In short, the RIS estimator is unbiased as long as $M, N \geq 1$, and the candidate density q and target density $\hat{p}$ are positive wherever the integrand f is non-zero.

Moreover, Talbot et al. show that the RIS estimator may reduce the estimator variance under the conditions that: First, $\hat{p}$ is a better importance sampling density than q (i.e., $\hat{p}$ mimics f better)—otherwise, there will be no advantage to performing the resampling step; second, generating candidate samples is computationally more efficient than evaluating the samples. Otherwise, we would be better off simply computing more samples, rather than wasting time generating candidates.

Intuitively, the RIS estimator uses the generated candidate samples y_i as if they were drawn from the desired distribution $\hat{p}$, and then uses the latter factor of Equation 2.6 (i.e., the average RIS weight) to correct for the fact that the true distribution of y_i only approximates $\hat{p}$.

2.2.3 Multiple Importance Sampling

Multiple importance sampling (MIS), as coined by Veach and Guibas [Veach and Guibas, 1995], refers to the family of techniques that extends importance sampling to the case where more than one sampling technique is used.

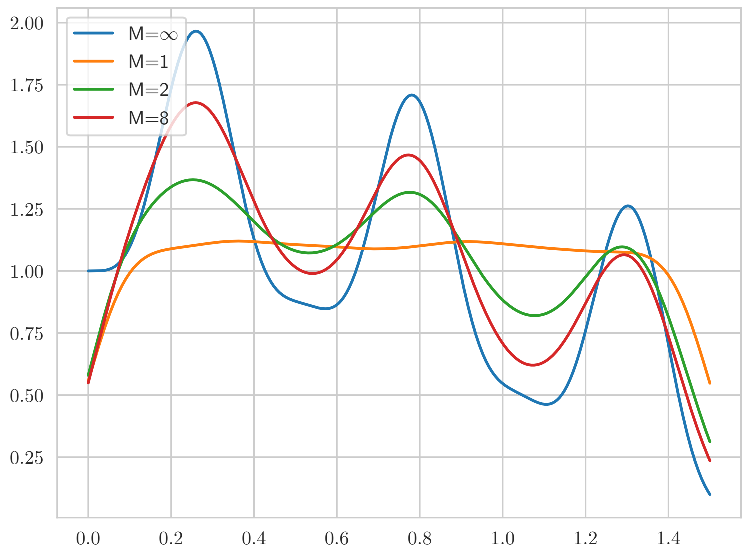


Figure 2.2: The distribution of the samples that were output from RIS for different values of M. The samples are distributed according to the uniform candidate density q when $M = 1$. As $M \to \infty$, the sample distribution approaches the target distribution $\hat{p}$. Note that the low estimates on the lower left for $M = 1$, $M = 2$, and $M = 8$ are artifacts of the density estimation method. Adapted from [Talbot et al., 2005]

When applying MIS, the samples are robustly weighted using heuristics that are designed to reduce the variance.

Veach and Guibas noted that we often know that the integrand f can be well-mimicked by any one of a set of K sampling distributions $\{p_1, \dots, p_K\}$ which we take $\{n_1, \dots, n_K\}$ number of samples from. However, we do not know *a priori* which p_i that minimizes the variance the most. In that case, Veach and Guibas propose using a multi-sample estimator F of the function f which combines all the sampling distributions into a single density:

$$F = \sum_{i=1}^{K} w_i(x_i) \frac{f(x_i)}{p_i(x_i)} \tag{2.7}$$

Where w_i are the *MIS weights* of distribution i. As long as $\sum_i w_i = 1$ and $w_i(x_i) > 0$ only when $p_i(x_i) > 0$, F will converge to the correct result. Veach and Guibas suggest using the *balance heuristic*—a heuristic that provably results in a variance that is smaller than any other unbiased combination strategy, to within a small additive term [Veach and Guibas, 1995]:

$$\hat{w}_i(x) = \frac{n_i p_i(x)}{\sum_k n_k p_k(x)} \tag{2.8}$$

Which is equivalent to sampling from the mixture of the PDFs. With this in mind, we see that MIS provides a practical procedure for making Monte Carlo rendering more robust: Whenever there is some situation that is not handled well by a single sampling technique, we can simply add other sampling techniques that are designed better for those situations alone, and weight the distributions according to Equation 2.8.

2.3 Ray Tracing

Ray tracing is a technique for generating images where *rays* are generated from a user-defined camera and traced into a virtual scene. The image is generated by simulating how these rays interact with objects in the scene. Usually, ray tracing is seen as a recursive algorithm: For each ray-surface collision, several new rays can be spawned; for instance, reflection rays can be spawned on reflective surfaces, or refractive rays can be spawned through transmissive materials. In contrast, methods, such as rasterization, that project geometry onto the image plane, have problems with accurately capturing global effects such as reflections and shadows because they are principally designed for rendering the primary surfaces (i.e., the visible surfaces). Since one can interpret the ray tracing model as a light transport model, it can be understood that ray tracing inherently captures these effects.

The first ray tracing algorithm was introduced by Arthur Appel [Appel, 1968] in 1968. His idea was to cast a ray through a pixel of an image plane from the eye and follow it until it collided with geometry in the scene. Accordingly, the algorithm would stop when the ray hit the closest object—this form of ray tracing was dubbed *ray casting*. The illumination values could then be computed by spawning *light rays* from the collision point to each light source. The final illumination for the corresponding pixel was computed based on whether the light sources were occluded or not. In 1979, Turner Whitted [Whitted, 1979] took the next step and described what we would today recognize as *traditional ray tracing*. Specifically, he made the ray casting algorithm a recursive process and, consequently, made it possible to robustly render global effects such as reflections, refractions, and shadows. In particular, at surface hits, up to three rays of different types could be spawned: reflection rays, refraction rays, or shadow rays. *Shadow rays* resembles the light rays defined by Appel and checks for a light's occlusion so that the contribution of light to the surface can be computed. A reflection and refraction ray represent the rays in the mirror direction and the transmitted light direction, respectively.

2.3.1 Acceleration Structures

Naively implementing a ray tracer can result in unacceptable performances since a linear-time intersection algorithm would have to test a ray against every object in the scene. With modern scenes containing millions of triangles and an increasing visual fidelity, this naive approach quickly becomes infeasible. Instead, rays are often traced against *acceleration structures*, which performs hierarchical subdivision of the scene and, consequently, reduces the intersection time substantially. Particularly, the complexity of performing ray-geometry intersections becomes logarithmic. The core idea of such acceleration structures is to cluster nearby geometries into larger entities and trace against these larger entities instead; if a ray does not intersect with the cluster, all the geometry contained in the cluster can be skipped as such. Common acceleration structures include k-d trees and bounding volume hierarchies (BVH) [Meister et al., 2021].

The speed-ups provided by such acceleration structures have proved to be so influential that hardware vendors have started implementing hardware support for traversing these structures. For instance, NVIDIA recently introduced the *RT cores* in their NVIDIA Turing graphics processing unit (GPU) architecture, which accelerates BVH ray traversal and triangle intersections [NVIDIA, 2018]. In this thesis, we assume that a ray is traced against such a hardware-accelerated BVH when we mention that a ray is traced through a virtual scene.

2.4 Radiometry

In order to generate an image of a virtual scene, the amount of reflected light towards the viewer needs to be computed. *Radiometry* provides the theoretical basis for how this works and is the science and technology of the measurement of radiation from all wavelengths within the optical spectrum [Heath and Munson, 1996]. That is, it deals with the measurement of light propagation and reflection. Concretely, radiometry describes light at the geometrical optics level, where macroscopic properties of light suffice to describe how light interacts with objects much larger than its wavelength [Pharr et al., 2016]. Due to this abstraction, effects such as polarization, diffraction, and interference cannot be captured by radiometry without extending the framework. The rendering algorithms presented later in this chapter are based on the radiometric theory presented here.

Textbooks on physically based rendering will typically include exhaustive theory on radiometry. We base this section mainly on [Pharr et al., 2016] and cover only the theory required to understand this thesis. Nonetheless, the reader is highly encouraged to consult the mentioned work, as well as other work, for more information.

2.4.1 Radiant Flux

One of the fundamental units of radiometry is *radiant flux*—also known as radiant power—and is denoted by Φ. It captures the total amount of energy passing through a surface or region of space per unit time and is expressed in watts (i.e., joules/second). Formally, radiant flux can be computed by taking the derivative of the radiant energy Q (received, emitted, or reflected) with respect to the time t:

$$\Phi = \lim_{\Delta \to 0} \frac{\Delta Q}{\Delta t} = \frac{dQ}{dt} \tag{2.9}$$

2.4.2 Irradiance and Radiant Exitance

Given a surface with finite area A, the average density of radiant power Φ over the area is given by:

$$E = \frac{\Phi}{A} \tag{2.10}$$

Thus, the unit is given in watts/m^2. The quantity of E represents either the incident or exitant power on a surface per unit surface area. As such, if we look at the area density of flux *arriving* at a surface the quantity is called *irradiance* (E). Conversely, if we look at the area density of flux *leaving* a surface, the quantity is referred to as *radiant exitance* (M).

2.4.3 Radiance

The most important radiometric quantity in this thesis, and perhaps in general, is *radiance*, L. While irradiance and radiant exitance do not distinguish the directional distribution of radiant power, radiance takes this last step: It measures the irradiance or radiant exitance *with respect to solid angles*. To emphasize, this makes radiance the solid angle density of irradiance or radiant exitance. It is defined by:

$$L = \frac{dE}{d\omega \cos\theta} = \frac{d\Phi}{d\omega \cos\theta dA} \tag{2.11}$$

The quantities involved in this formula are visualized in Figure 2.3: $d\Phi$ is the differential incoming or outgoing radiant flux. $d\omega$ is the differential solid angle which, in the figure, is represented by the red cone. dA is the differential area of the surface, given by the flat rectangle in blue. The projected area is computed by multiplying dA with $\cos\theta$ to consider Lambert's law, that is,

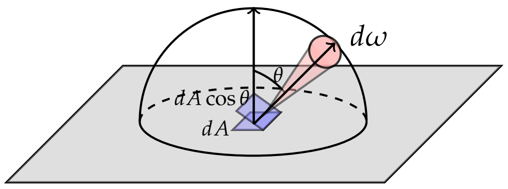


Figure 2.3: Radiance is the solid angle density of irradiance. That is, the energy along a ray. Figure adapted from [Pharr et al., 2016].

the fact that dA becomes ”stretched out” over a larger surface area when $d\omega$ comes from grazing angles (thus, the tilted rectangle in blue).

Put differently, from the second equality in Equation 2.11; radiance can equivalently be understood as the radiant flux per unit *projected area* per unit solid angle. This makes the unit of the measure watts/(m^2·sr). Radiance is a natural quantity to compute with ray tracing since the radiance remains constant as it propagates along rays in vacuum. Moreover, it is a logical quantity to use because it captures how surfaces *appear* with respect to the viewpoint of the observer.

2.5 Lighting Interactions

Light behaves differently depending on the materials it interacts with. For instance, metals and glossy surfaces have mirror-like appearances, while matte surfaces, like painted walls, appear about the same from any viewing angle.

The *bidirectional reflectance distribution function* (BRDF) gives a formalism for describing reflection from a surface. It describes the reflectance of a surface for a given combination of incoming and outgoing light direction. It is defined as the ratio between the reflected differential radiance in exitant direction ω_o, and the differential irradiance incident through a solid angle $d\omega_i$ (i.e., the direction ω_i is considered as a differential cone of directions) at $\mathbf{x}$:

$$f_r(\mathbf{x},\omega_i,\omega_o) = \frac{dL_o(\mathbf{x},\omega_o)}{dE(\mathbf{x},\omega_i)} = \frac{dL_o(\mathbf{x},\omega_o)}{L_i(\mathbf{x},\omega_i)\cos\theta_i d\omega_i} \tag{2.12}$$

Where the last equality comes from the fact that the first equality in Equation 2.11 can be refactored:

$$L = \frac{dE}{d\omega\cos\theta} \implies dE = Ld\omega\cos\theta$$

This means, in other words, that the BRDF captures how much light is reflected in a given direction when a certain amount of light is incident from another direction, depending on the properties of the surface.

For a BRDF to be considered *physically based*, it should have these two qualities [Pharr et al., 2016]:

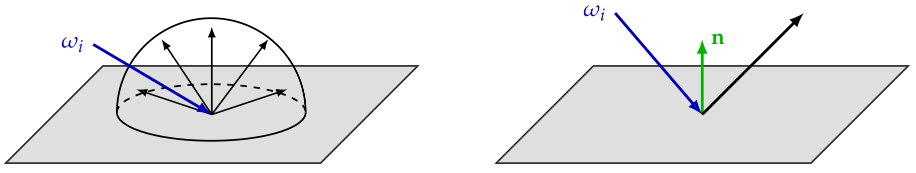
1. The Helmholtz Reciprocity property: The value of the BRDF is unchanged if we swap the incident and exitant directions. That is,

$$f_r(\mathbf{x}, \omega_i, \omega_o) = f_r(\mathbf{x}, \omega_o, \omega_i)$$

2. Conservation of energy: The surface cannot reflect more energy than it recieves,

$$\forall \mathbf{x}, \omega_o \int_{\Omega} f_r(\mathbf{x}, \omega, \omega_o) \cos\theta d\omega \leq 1$$

Finally, it is worth noting that we have, up to now, assumed that light which enters at a point p leaves from the same point. However, in the general case, the light which enters at a point p in some direction ω_i might instead leave at another point q in a new direction ω_o, creating a "subsurface scattering" effect. Reflectance functions that take this into considerations are called *bidirectional surface scattering reflectance distribution functions* (BSSRDF), and compute the ratio between the incident and exitant radiance between (p, ω_i) and (q, ω_o). Nevertheless, in this work, we disregard this phenomenon to simplify our BRDF implementations, and we, consequently, set the exit point equal to the entry point. The following subsections present examples of common material types.



(a) Perfectly diffuse (b) Perfectly specular

Figure 2.4: Exitant directions of perfectly diffuse (a) and perfectly specular (b) surfaces.

2.5.1 Diffuse Surfaces

One of the simplest material types is the perfectly diffuse BRDF and is shown in Figure 2.4a. Such a material type is often called a *Lambertian* material. It assumes that the incident light is scattered in all possible directions uniformly within the hemisphere oriented about the surface normal. Diffuse materials are not view-dependent as such. While not being physically plausible, it is still, nonetheless, a reasonable approximation to many real-world surfaces such as matte paint [Pharr et al., 2016] and is very fast to evaluate as well. The BRDF for a diffuse material is given by,

$$f_r(\mathbf{x}, \omega_i, \omega_o) = \frac{\rho}{\pi} \tag{2.13}$$

where ρ is defined as the reflectivity of the material, which specifies how much of the incident light is reflected in the diffuse lobe. It is typically calculated from a user-defined "base color" parameter [Cook and Torrance, 1982].

2.5.2 Specular Surfaces

A perfectly specular surface will reflect or refract light in one direction only, as shown in Figure 2.4b. A mirror is an example of such a surface. We can compute the reflected direction R for incident direction ω_i and surface normal $\mathbf{n}$ from the law of reflection:

$$R = 2(\mathbf{n} \cdot \omega_i)\mathbf{n} - \omega_i \tag{2.14}$$

How much light reflects away (or scatters into) from the material, given an angle of incidence, is described by the *Fresnel equations*. Moreover, for materials that can both reflect and refract light—such as glass—the direction of refraction can be found using *Snell's law*.

2.5.3 Microfacet Models

Real-world materials are rarely either perfectly diffuse or perfectly specular; instead, they are often a mix of those two. This occurrence can be explained by the assumption that rough surfaces are composed of many tiny facets—each of which acts as a perfect specular reflector. These *microfacets* are assumed to have normals that are distributed about the normal of the actual surface. Particularly, the facet distributions are usually described, statistically, from the degree to which the microfacet normals differ from the normal of the actual surface. In practice, the degree of variance to these distributions is determined by a *roughness* scalar parameter of the surface, where values close to zero correspond to near-perfect specular reflection, and values close to one correspond to diffuse reflections.

More formally, *the microfacet model* postulates that if a surface reflection can occur between an incident direction ω_i and exitant direction ω_o, then there must exist some portion of the surface, or microfacet, with a normal aligned halfway between ω_i and ω_o [Burley, 2012]. This "half-vector", sometimes referred to as the microsurface normal, is defined as $\omega_h = \frac{\omega_i + \omega_o}{||\omega_i + \omega_o||}$. A general form of the BRDF for a microfacet model is the Torrance-Sparrow BRDF (also known as the GGX BRDF):

$$f_r(\mathbf{x}, \omega_i, \omega_o) = \frac{D(\omega_h)G(\omega_o, \omega_i)F(\omega_o)}{4 \cos\theta_o \cos\theta_i} \tag{2.15}$$

Where D is the microfacet distribution function, which tells us the fraction of microfacets that are oriented about ω_h so that incident light from direction ω_i will be reflected in direction ω_o. F is the Fresnel reflection function which describes how much light is reflected off the surface for incident direction ω_i. Finally, G is the geometric attenuation and masking-shadowing function that accounts for mutual shadowing and masking of microfacets.

2.6 Global Illumination

2.6.1 The Rendering Equation

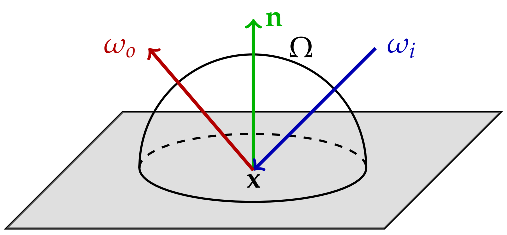


Figure 2.5: A visualization of the quantities involved in the rendering equation. The reflected radiance L_r, from $\mathbf{x}$ in direction ω_o (red), is the integral of the incoming radiance L_i from all incoming directions ω_i (blue) over the hemisphere Ω oriented around the surface normal $\mathbf{n}$ (green). For each direction, radiance is converted to irradiance and scaled by the BRDF f_r. In this figure, one sample incident direction ω_i is shown.

Light transport is described by *the rendering equation* and was first introduced by Jim Kajiya in 1986 [Kajiya, 1986]:

$$\begin{aligned} L_\text{o}(\mathbf{x},\omega_\text{o}) &= L_\text{e}(\mathbf{x},\omega_\text{o}) + L_\text{r}(\mathbf{x},\omega_\text{o}) \\ &= L_\text{e}(\mathbf{x},\omega_\text{o}) + \int_\Omega f_\text{r}(\mathbf{x},\omega_\text{i},\omega_\text{o}) L_\text{i}(\mathbf{x},\omega_\text{i})(\omega_\text{i}\cdot\mathbf{n})\,\mathrm{d}\,\omega_\text{i} \end{aligned} \tag{2.16}$$

In practical terms, the outgoing radiance L_o from the point $\mathbf{x}$ in direction ω_o can be described as the sum of two terms: The *emitted radiance* L_e and the *reflected radiance* L_r. The former describes the self-emission in the direction ω_o on the surface at $\mathbf{x}$. It allows us to model light sources in the scene, including natural emitters such as the sun or artificial emitters such as flashlights or lamps. The latter term, that is, the reflected radiance L_r, describes the amount of light that is received by $\mathbf{x}$ from other surfaces and reradiated towards direction ω_o. It is computed recursively with the following integral:

$$L_\text{r}(\mathbf{x},\omega_\text{o}) = \int_\Omega f_\text{r}(\mathbf{x},\omega_\text{i},\omega_\text{o}) L_\text{i}(\mathbf{x},\omega_\text{i})(\omega_\text{i}\cdot\mathbf{n})\,\mathrm{d}\,\omega_\text{i} \tag{2.17}$$

The quantities involved in this integral are shown in Figure 2.5: L_r is computed as an integral over all incoming hemispherical directions ω_i at $\mathbf{x}$.

Moreover, the three terms of interest are as follows. L_i describes the amount of light that arrives at $\mathbf{x}$ from incoming direction ω_i; f_r is the BRDF of the surface, which describes how much of the received light is reflected towards the outgoing direction ω_o; and $(\omega_i \cdot \mathbf{n})$ is a foreshortening term that scales how much of the incoming radiance that is received by a surface with normal $\mathbf{n}$.

2.6.2 Path Tracing

Path tracing was introduced by Kajiya in 1986 [Kajiya, 1986] and attempts to simulate light transport by estimating the recursive integral for the reflected radiance L_r in Equation 2.17 by using Monte Carlo integration (section 2.1).

The integral in the rendering equation can be estimated by evaluating the integrand for one direction (sample) ω_i oriented around the surface normal $\mathbf{n}$ at shading point $\mathbf{x}$, which is sampled at random according to a PDF $p(\omega)$ [Veach, 1998]. Consequently, the Monte Carlo estimator for L_r becomes:

$$\hat{L}_{\mathrm{r}}(\mathbf{x}, \omega_{\mathrm{o}}) = \frac{1}{N} \sum_{i=1}^{N} \frac{f_{\mathrm{r}}(\mathbf{x}, \omega_{\mathrm{i}}, \omega_{\mathrm{o}}) L_{\mathrm{i}}(\mathbf{x}, \omega_{\mathrm{i}})(\omega_{\mathrm{i}} \cdot \mathbf{n})}{p(\omega_i)} \tag{2.18}$$

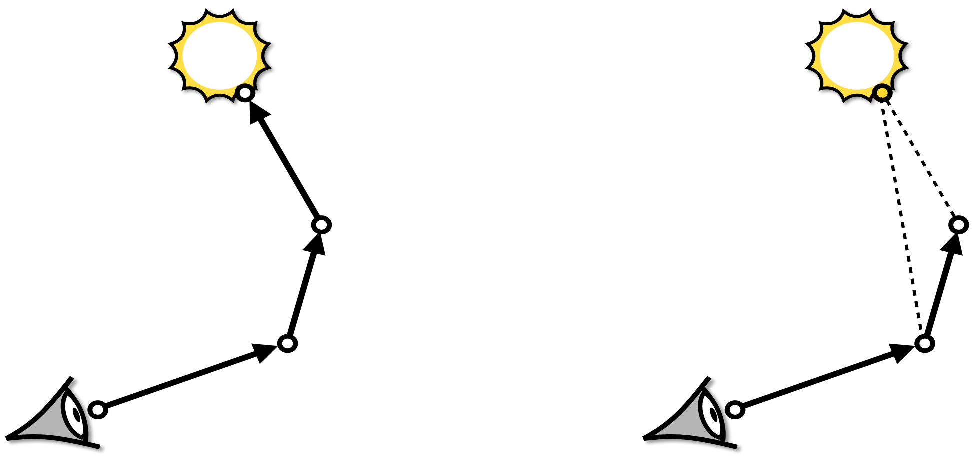
Accordingly, the radiance estimation $L_i(\mathbf{x}, \omega_i)$ in Equation 2.17 is computed by recursively evaluating the equation above. Indeed, the recursive estimations form *paths* that are traced through the scene until light sources are encountered. In practice, *ray tracing* can be used to simulate the light transport of samples that form these traced paths [Whitted, 1979]. That is, the rays will represent light samples that are bounced around the scene at various surface points $\mathbf{x}$ in different directions ω, depending on the surface properties of the materials the samples are bounced on.

A simple choice of the PDF $p(\omega)$ could be based on the uniform area of a hemisphere $p(\omega) = \frac{1}{2\pi}$. However, this would result in inefficient sample usage near the horizon where $(\omega \cdot \mathbf{n}) \approx 0$. Instead, a *cosine-weighted distribution* $p(\omega) = \frac{\cos\theta}{\pi} = \frac{\omega \cdot \mathbf{n}}{\pi}$ is usually used, which directly cancels the foreshortening term in Equation 2.18 and, consequently, lowers the estimator variance [Pharr et al., 2016].

In consequence, we have an algorithm that yields unbiased estimates of the rendering equation (subsection 2.6.1) that will converge to the correct result after a sufficient amount of time and samples. Unfortunately, for practical sample counts, the algorithm results in noisy images due to the variance in the estimator. The following section presents a method that uses importance sampling (subsection 2.2.1) to reduce the variance of the path tracing estimator drastically.

2.6.3 Next Event Estimation

Naively tracing paths based on local importance sampling of the BRDF accounts for all factors contributing to the sampling density, except the emitted radiance L_e. Neglecting the importance sampling of this term can cause excessive variance: We expect L_e to be zero for most surfaces in the scene and be very large for a small subset of the light sources.



(a) BRDF sampling (b) Next-event estimation

Figure 2.6: Different path sampling strategies: BRDF sampling (a) samples directions based on the lobes of the BRDF and can lead to high variance because many of the reflected directions are not directed towards the light sources. In contrast, next-event estimation (b), also called direct light sampling, additionally samples directions towards the light sources directly (in dashed lines) and can significantly reduce the variance.

Naive path tracing relies on BRDF sampling (i.e., sampling according to f_r) to find emissive surfaces—which is inadequate in scenes with more complex placements of light sources. To mitigate this issue, we can employ a technique called *next event estimation (NEE)*. In NEE, we sample for each shading point $\mathbf{x}$ the light source directly by connecting the path from $\mathbf{x}$ to a sampled light position $\mathbf{y}$, as shown in Figure 2.6, where BRDF sampling is compared to NEE. This technique can drastically reduce the variance in a Monte Carlo renderer [Pharr et al., 2016], and an example demonstrating the dramatic effect of applying NEE is shown in Figure 2.7—notice the dramatic decrease in noise. Note that sampling *direct lighting* using a separate ray towards the light source is a form of importance sampling: We effectively ignore many directions on the hemisphere and instead focus on directions where the light sources are visible. Accordingly, we also need a

PDF to sample from in order to weigh the samples correctly. In particular, we want a PDF that is zero for most hemispherical directions and constant over the area of the light source projected on the hemisphere. This is the case if the PDF is $p(\omega_i) = \frac{1}{SA(\omega_i)}$, where SA is the area of the light source projected on the hemisphere—in other words, *the solid angle*:

$$SA(\omega_i) = \frac{A_{\text{light}}(\mathbf{n}_{\text{light}} \cdot \omega_i)}{dist^2}$$

Where A_{light} is the area of the light, $\mathbf{n}_{\text{light}}$ is the surface normal of the light, and *dist* is the distance between the current point and the light.

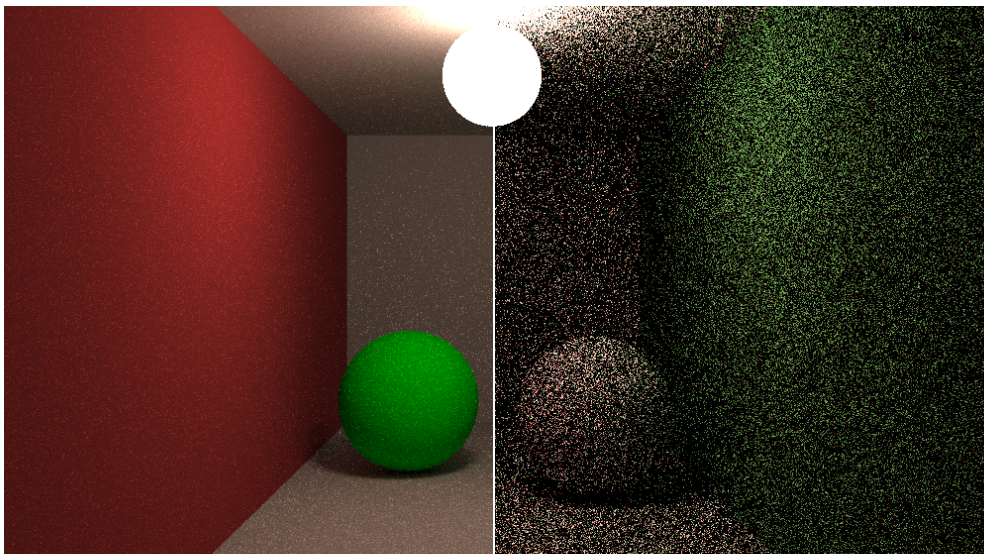


Figure 2.7: NEE (left) vs. without NEE (right) in a 16 spp path traced render. Notice how NEE dramatically reduces the variance and noise in the image.

With this, we have two different techniques to sample lights: By directly sampling the light using NEE, and by ”indirectly” sampling lights by sampling according to the BRDF of the surface at $\mathbf{x}$. We can, therefore, also apply multiple importance sampling (subsection 2.2.3) to the path tracer in order to further reduce the variance. As such, we obtain better estimates by combining the results from NEE and BRDF sampling. In practice, this is done by inserting the computed probability densities of both techniques into Equation 2.8 to compute the MIS weights directly. The weights are, accordingly, used to combine the results of the techniques robustly.

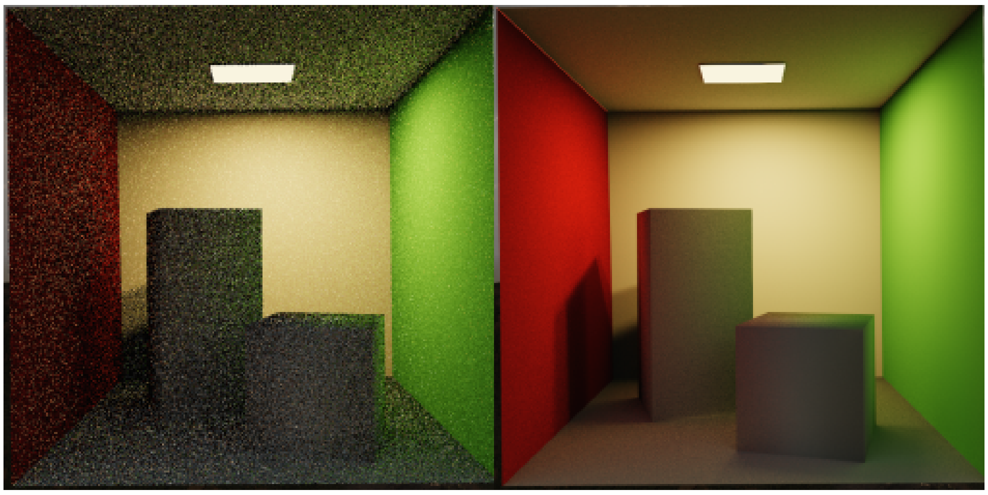


Figure 2.8: 1 spp (a) and 4096 spp (b) path traced renders of the CORNELL Box scene, which took around 3ms and over 3s to render, respectively.

2.7 Image Space Filtering for Monte Carlo Noise

Despite the different variance reduction techniques presented so far in this chapter, it can still take a prohibitively large amount of samples to generate noise-free images with a path tracer. This is exemplified in Figure 2.8, where a 1 sample per pixel (spp) image is compared against a 4096 spp image—both rendered with a path tracer. While the former is rather noisy, it only took 3ms to render. In contrast, the latter is almost noise-free but took over 3 seconds to render. As a result, this still makes path tracing unsuitable for real-time applications—and even in some contexts for offline rendering since convergence can take prohibitive amounts of time. With more complex scenes and more complicated effects, this only becomes a bigger problem.

The purpose of *filtering* in Monte Carlo rendering is to reduce the variance by combining the estimates of multiple Monte Carlo estimators with the assumption that they converge to similar values. This can significantly help to reduce the amount of visually disturbing noise so that noise-free images resembling the converged result can be obtained quickly. A popular and efficient way of achieving this is to work in the screen space (equivalently referred to as image space in this thesis) and do a weighted blending of nearby pixels by exploiting the spatial coherence in images:

$$\hat{\mathbf{c}}_p = \frac{\sum_{q \in N_p} \mathbf{c}_q w(p,q)}{\sum_{q \in N_p} w(p,q)} \tag{2.19}$$

Where the filtered color $\hat{\mathbf{c}}_p$ for pixel p is a weighted sum of the noisy colors

$\mathbf{c}_q$ for each pixel q in a window N_p, which is centered on p. $w(p,q)$ denotes the weight between p and q.

Nevertheless, the assumption that neighboring pixels converge to the same value is usually not satisfied. Consequently, combining neighboring pixels can result in additional *bias*. For instance, if we were to combine samples of pixels on opposite sides of an edge, a large amount of bias would be introduced: As a result, the edge would be attenuated and smeared. In this regard, the goal of the filtering algorithms is to reduce variance while also keeping the introduced bias small. Visible high-frequency noise has to be filtered, while low frequency and sharp features present in the converged render should be retained.

Finally, note that while filtering in image space is currently a popular and practical method, other alternatives exist. For example, it is also possible to perform filtering in path space (equivalently referred to as world space in this thesis) instead of in image space [Hachisuka et al., 2008]. However, filtering in image space remains an attractive choice because of its low computational costs since the complexity of the algorithms is independent of the scene complexity and only depends on the number of pixels. Furthermore, image space filters are often relatively easy to integrate into existing renderers due to them generally working as post-process filters, independent of the rendering pipeline architecture. In fact, image space filters often only need the final image and possibly other auxiliary buffers (such as depth, normal, and color data) as inputs to perform the filtering.

CHAPTER 3

Related Work

This section presents related work to this thesis. The focus will mainly be placed on earlier work on denoising and many-light sampling in the context of Monte Carlo rendering.

3.1 Denoising

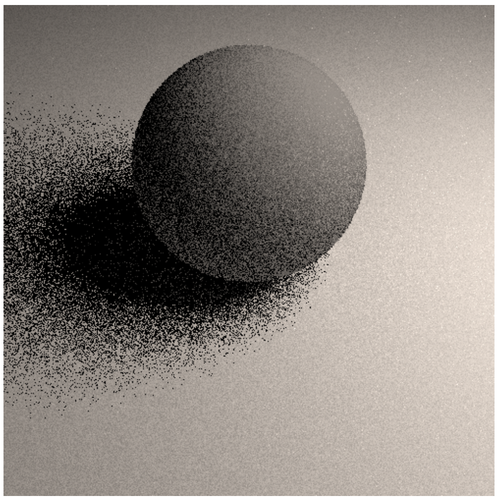


Figure 3.1: 1 spp path-traced render of a simple scene consisting of a diffuse sphere, a diffuse plane, and a spherical light source.

As we have seen so far, noise is an inevitable consequence in Monte Carlo rendering due to the inherent variance in the Monte Carlo estimator $\hat{I}$. Although variance reduction techniques such as importance sampling, MIS, and RIS can help, we will likely end up with visibly perceptible noise for lower sampling rates. This is exemplified in the 1 spp render in Figure 3.1, where there is visible noise even for such a simple scene. Moreover, in the context of real-time rendering, the number of samples per pixel is bound to be low [Keller et al., 2019], which likely means that we have to continue handling noisy renders for a while. As a consequence, *denoising*—also interchangeably referred to as *filtering* in this thesis—is often an integral part of any rendering system based on Monte Carlo rendering.

An essential part of many denoising algorithms, which enables them to filter even at lower sampling rates, is that they increase the effective sampling rate by appropriately *reusing the relevant sample data* (e.g., pixel and path data). Concretely, these algorithms commonly use the insight that spatially and temporally adjacent samples are, in general, related; these samples can thus be reused in order to improve the effective sample rate. Section 2.7 introduced the concept of image space filtering in the context of Monte Carlo rendering, and the recent survey by Zwicker et al. covers much of the work that has been done in this field [Zwicker et al., 2015]. Briefly, Zwicker et al. distinguishes between *a priori* and *a posteriori* methods. In general, a priori methods attempt to form reconstruction filters from analysis of the light transport equations, while a posteriori methods instead attempt to analyze the samples generated by the renderer.

A priori methods often require reconstructing high-dimensional samples and are, consequently, seldom applied in the context of real-time rendering [Hachisuka et al., 2008]. Thus, most of the research done in the past decades has been on a posteriori methods in the form of *post-process filtering*. In practice, a posteriori approaches often rely on image space filtering because of its simplicity and efficiency [Zwicker et al., 2015]. Notably, recent research on *path space filtering* has also shown promising results. The following two subsections (subsection 3.1.1 and subsection 3.1.2) will, respectively, present various image space and path space filter methods.

Interestingly, more recent work [Bitterli et al., 2020, Lin and Yuksel, 2020] has indicated that performing the filtering at earlier stages in the rendering pipeline, as opposed to at the end of the pipeline (as a post-process), can provide better results—even for interactive and real-time use-cases. A weakness of image space filters is that they often solely act on reconstructed pixels: Significant data could be lost from combining noisy samples with little information, and, as such, the existing data might be further corrupted by other outliers. On the contrary, there is much critical information that is only available earlier in the rendering pipeline. Accordingly, this information could potentially be used for further guiding the filtering algorithms with higher quality data. The information could, for instance, be BRDF values, sampling probabilities, directions, or other kinds of higher-order bounce information.

In practice, recent work has been focusing on *filtering light sampling probabilities* to guide the renderer on which lights to perform lighting calculations with. Algorithms that filter these light sampling probabilities are often referred to as *many-light sampling algorithms* and the current state of these are presented in section 3.2.

3.1.1 Screen Space Denoising

Performing the denoising in *screen space* is an attractive choice when designing denoising algorithms. Section 2.7 presented the fundamentals of screen space filtering, explaining that screen space filtering is often preferred due to its low computational costs and ease of integrating into existing rendering systems. The low computational costs mainly come from the fact that the cost of the filtering computation is decoupled from the geometric complexity in the virtual scenes [Mara et al., 2014].

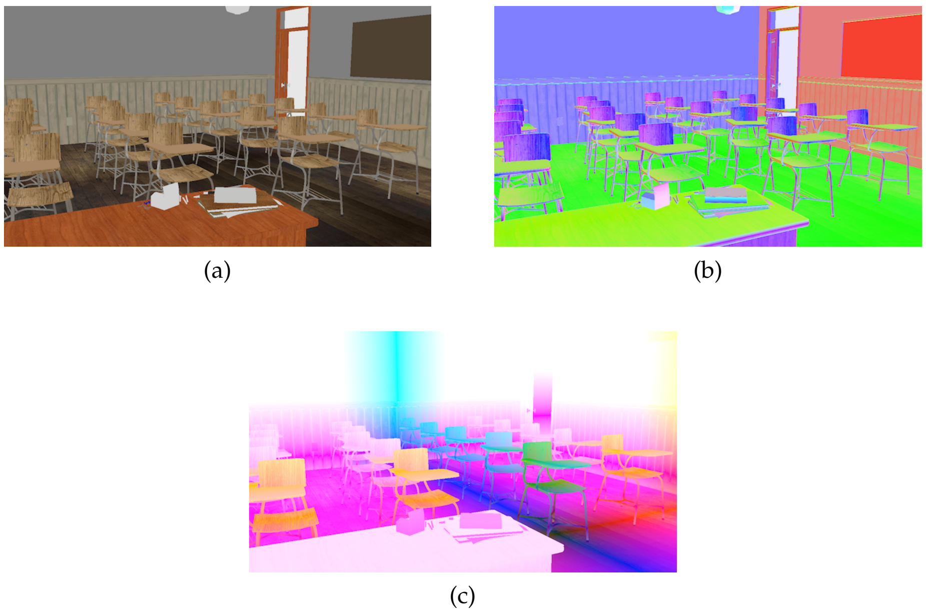


Figure 3.2: An example of a G-buffer consisting of: (a) Surface albedos, (b) surface normals, and (c) world space positions. Each buffer consists of three-dimensional values, which are visualized directly as RGB triplets. (The Classroom is by Christophe Seux under the CC0 license)

Screen space denoising methods can usually assume access to excess information from the rendering pipeline, as it is applied at the end of the pipeline as a post-process. This information may include world positions, depth, surface normals, albedo, material properties, and more. Moreover, the information is usually output from a renderer as textures of the exact dimensions as the actual output image from the renderer. Saito and Takahashi introduced the concept of saving buffer data inferred from the geometry as *G-Buffers* [Saito and Takahashi, 1990]. Figure 3.2 shows an example of a G-buffer consisting of buffers such as normals, world space positions, and albedos, which are standard to be output from any renderer. As we will see later in this section, properly utilizing the information provided by the G-buffer is paramount for performing high-quality image filtering.

Offline Denoising

Offline denoisers are common for movie production. Since there are typically little to no strict time constraints for the renderers in these applications, a higher sample count can be used [Zwicker et al., 2015]. Thus, the amount of noise in the rendered images is usually close to negligible, which simplifies the job of the denoisers. Even with the larger computational budgets, denoising remains crucial due to the denoisers' ability to short-circuit the slow convergence of Monte Carlo integration. Furthermore, it is also possible in offline rendering for the filtering methods to guide the sample generation process in Monte Carlo renderers so that more samples are generated at problematic areas in screen space—also known as adaptive filtering [Li et al., 2012].

Due to their simplicity of integrating with existing renderers and effectiveness, general edge-preserving image filters like guided image filtering [He et al., 2013] or non-local means filtering [Buades et al., 2005], that are guided with G-buffer data, have been applied in these use-cases. Alternatively, instead of handpicking the weights of the filters using heuristics, other works have attempted to fit the G-buffer data to the noisy output images by fitting an online regression-based model [Bitterli et al., 2016, Moon et al., 2014]. Central to these methods is the idea of extracting a weighting kernel and computing the denoised result as a linear combination of kernel-weighted noisy pixels. Finally, another more recent and promising approach that has been shown to outperform regression-based denoisers under specific circumstances has been to use *neural networks* [Kalantari et al., 2015, Vogels et al., 2018, Chaitanya et al., 2017]. The neural networks are first trained on a complete set of frames from a feature-length movie [Bako et al., 2017] and are then used to denoise the noisy frames as needed. This approach has proven effective in offline rendering due to the availability of a large amount of training data and computational resources for the neural networks to be adequately trained. Consequently, employing this type of denoiser can result in more robust denoising results when the (trained) neural networks are used for inference [Dahlberg et al., 2019].

Real-time Denoising

Despite the advancements of hardware-accelerated ray tracing and more efficient sampling algorithms, it has been suggested that it is very likely that a limit of 1-4 spp will persist for a long time [Schied et al., 2017, Koskela et al., 2019]. As a result, we can only afford a handful of samples per pixel under the constraints of real-time rendering. Furthermore, with an ever-increasing demand in the geometric level of detail and display resolution for real-time applications, it seems unlikely that any immediate increase in computational power will benefit the sample counts. Because of the resulting degree of sparsity for the samples, the applied denoising algorithms are often said to *reconstruct* the image rather than filter the noise from it.

Real-time denoisers, such as the ones in [Schied et al., 2017, Mara et al., 2017, Koskela et al., 2019], differ from offline denoisers presented earlier in that they assume that the G-buffer output from the renderer is *noise-free*. This allows the algorithms to safely use the G-buffer data as a ”guide” to, for instance, avoid blurring samples across geometry edges or help reduce smearing the details in the textures. The screen-space denoisers in [Mara et al., 2017, Schied et al., 2017] do this by guiding a cross-bilateral filter [Tomasi and Manduchi, 1998] with the G-buffer. Moreover, Schied et al. takes it a step further and additionally steers the filter with a spatiotemporal estimate of the luminance variance [Schied et al., 2017]. Consequently, their algorithm can filter more aggressively in areas of high variance (e.g., near penumbras) and less in areas with little variance (e.g., in areas of hard shadows). Algorithms such as Blockwise Multi-Order Feature Regression [Koskela et al., 2019] takes an additional step and uses the feature buffers as *covariates* to fit a regression model that predicts the denoised color values.

In addition to filtering the noisy data spatially, a key element of these real-time denoisers is that they often *reproject and temporally accumulate samples* from previous frames. This is critical for reducing the temporal noise that varies between consecutive frames; thus, improving the temporal coherence between the frames. A 2D *motion vector* associated with each color sample C_i for frame i is required to perform the reprojection. Specifically, the motion vectors describe the geometric motion between the current and prior frame and allow the algorithms to project the color sample C_i to its screen space location in the prior frame. By backprojecting C_i to access C_{i-1} from a *color history buffer*, output by the filter in the prior frame, the filtering algorithms can continuously accumulate color samples over multiple frames by blending between consecutive frames. The temporally blended color sample C_i' is generally computed with an exponential moving average:

$$C_i' = \alpha C_i + (1-\alpha)C_{i-1} \tag{3.1}$$

Where α is a blending parameter, often fixed to a value such as $\alpha = 0.2$. However, using a fixed α can result in artifacts such as ghosting and flickering on, for example, disocclusions and fast camera movement. Consequently, it is often proposed [Schied et al., 2017, Koskela et al., 2019] to implement the blending similarly to how it is done in the temporal anti-aliasing (TAA) algorithm [Karis, 2014], with the exception that there is usually no clamping of the temporal neighbors performed, as it is typically done in TAA.

3.1.2 Path Space Filtering

Due to screen space filters being limited to the information only visible information on the screen and the assumption of a noise-free G-buffer, stochastic primary ray effects such as depth of field, motion blur, and transparency becomes incompatible [Schied et al., 2017]. Moreover, visible artifacts in the form of overblurring and smearing can occur during disocclusions.

Many of the aforementioned limitations can be addressed by filtering in *world or path space* instead. Hachisuka et al. store samples in a multidimensional path space, which may include effects such as motion blur, depth of field, and soft shadows [Hachisuka et al., 2008]. Hachisuka et al. first estimate the local contrast incurred by an initial set of samples and adaptively distribute more samples in the multidimensional space where the contrast is the highest. In a second pass, they reconstruct the image by integrating the multidimensional function along all but the image dimensions. The reconstruction is performed by determining the extent of each sample in the multidimensional space using an anisotropic nearest neighbor filter. One big drawback, which limits this method from real-time use-cases, is the fact that a complex high-dimensional data structure has to be maintained for placement and spatial queries of the multidimensional samples. In particular, the algorithm quickly becomes computationally infeasible in higher dimensions due to its dependence on k-d trees, making it challenging to simultaneously render distributed effects such as motion blur, depth of field, and soft shadows.

Recently, *parallel path space filtering* has gained more traction in real-time use-cases through the works of [Binder et al., 2021, Pantaleoni, 2020]. Instead of maintaining complex tree-like data structures, which necessarily do not map well to GPUs, *hash tables* are used for storing and querying the multidimensional samples. By constructing hash keys through discretization of, say, world space positions, surface normals, and other relevant surface properties, one can efficiently store, query, and evict spatial samples in the hash table. Binder et al. use this data structure by efficiently looking up, storing, and computing the average light contributions between similar vertex descriptors in parallel. Instead of filtering per vertex, filtering is incurred per discretized cell (i.e., voxel). Note that querying hash tables, assuming it is sufficiently sized, often requires only a single memory access, resulting in a constant complexity most of the time. Consequently, the costly neighborhood searches can be replaced by averaging contributions directly at the quantized path space descriptors, which is essential for the algorithm proposed by Binder et al. to work in real-time framerates.

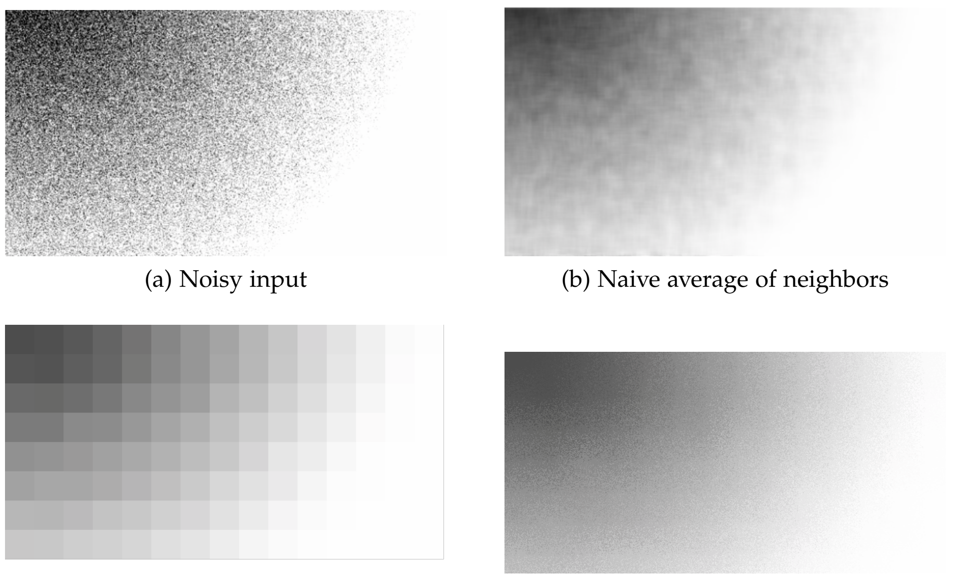

(a) Noisy input (b) Naive average of neighbors (c) Quantized filtering (d) Jittered quantized filtering

Figure 3.3: A 2D Example showing how path space filtering works: Given a noisy input (a), a naive filtering algorithm (b) can result in visible artifacts. Instead, Binder et al. propose to perform quantized filtering (c) by filtering in each cell. Moreover, by jittering before accumulation and look-ups (d), the quantization block artifacts can be effectively resolved in noise. Adapted from [Binder et al., 2021]

Finally, for each selected vertex, its associated averages (stored in the hash table) are used in the corresponding radiance calculations and are accumulated in their respective pixel. In order to mitigate the various discretization artifacts that can occur (due to discontinuities of the quantization), Binder et al. propose to *jitter* the components of the hash keys. Specifically, Binder et al. suggest spatially jittering the world space positions on the tangent plane of the surface normals. Accordingly, the quantization artifacts are traded for noise that is more amenable to the eye and simpler to remove by a secondary filter such as the ones presented in subsection 3.1.1.

A two-dimensional example demonstrating the key ideas of path space filtering is shown in Figure 3.3. Notice how the final result still has some noise in it; however, the noise is far more pleasable to the eye than the residual noise in Figure 3.3c and 3.3b. Moreover, if required, this noise can easily be filtered by one of the presented screen space filters.

The use of hash tables and discretized vertex descriptors to perform filtering is explored further in this thesis. Subsequently, a presentation of this data structure is given in greater detail in section 4.2.

3.2 Many-Light Sampling

Handling many lights in scenes has been a challenge in real-time rendering due to the complexities of accumulating illumination from all light sources for all pixels on the screen. Consequently, most game engines handle scenes with many lights using a mixture of baking (i.e., precomputation of the radiance values for each surface area unit) and tile-based deferred rendering [Olsson and Assarsson, 2011, Olsson et al., 2012] to determine which pixels should be illuminated by which light sources. Furthermore, the current solutions are still limited to use a handful of carefully selected dynamic and shadow-casting lights, while the rest of the lights have to be static. We would ideally like to select light samples with a probability in proportion to each light's contribution. However, the contributions vary spatially and depend on the local surface properties and visibility of the light sources. It is therefore challenging to find a global PDF that works well everywhere.

Methods such as the Light BVH [Moreau et al., 2019] attempt to lift these limitations. The idea is to use a hierarchical acceleration structure, such as BVHs and *k*-d trees—which are built from the scene lights—and use them to guide the sampling process hierarchically. The nodes in these trees represent clusters of lights so that when one traverses the tree from top to bottom, one can at each level estimate how much each cluster contributes. This means that lights are chosen approximately proportional to their expected contributions without explicitly recomputing storing the light PDFs at each shading point.

3.2.1 Lightcuts

The main limitation of algorithms that maintain light hierarchies, such as Light BVH, is that their computational cost limits the number of light samples usable at real-time frame rates, resulting in noisy lighting estimations. Because the current state of real-time renderers are restrained to tracing between 1-4 rays per pixel [Koskela et al., 2019], the cost of constructing and maintaining these hierarchies is higher relative to the time spent rendering.

Work by Lin and Yuksel instead uses a lower quality acceleration structure to lower the cost of maintaining and constructing the hierarchy [Lin and Yuksel, 2020]. Though this may affect the quality of the tree and the light sample distribution, this reduction in sample quality may be exchanged by generating more light samples. In other words, the time saved from tree construction and sample selection is used for more light samples, resulting in better radiance estimations.

To be specific, the authors construct a balanced binary tree in their work (i.e., a tree where all leaf nodes appear at the bottom-most level of the tree). Because of this, there is no need for storing child node pointers since the child node indices can be computed directly from the parent node index. Accordingly, through parallel construction of the tree (in a way that exploits the GPU capabilities), the tree is constructed fast enough to be rebuilt every frame.

Following the light tree construction, a *cut* through the tree is selected. A cut through a tree is defined as a set of nodes (i.e., a cluster of nodes) such that every path from the root of the tree to a leaf contains exactly one node from the cut [Walter et al., 2005]. Lin and Yuksel select the *lightcuts* probabilistically using the *stochastic lightcuts* method [Yuksel, 2019]. The nodes for the lightcuts are sampled based on probabilities generated from weights that are approximately proportional to the expected illumination of the selected nodes in the lightcut. This stochastic sampling, consequently, allows for unbiased sampling of the lights.

Finally, a critical factor of Lin and Yuksel's work, which enables their algorithm to give decent results in real-time contexts, is the introduction of *cut haring*. Similar to the key findings in denoising methods (subsection 3.1.1), the authors observe that neighboring pixels often share the same lightcuts. Based on that observation, they accelerate the cut selection by performing it for a *group of pixels* rather than independently for each pixel. As such, the cut sharing accelerates the light sampling with a smaller additional memory footprint for storing the cut.

3.2.2 Reservoir-Based Spatiotemporal Importance Resampling

Although light BVHs and lightcuts allow for importance sampling lights in sub-linear time, the overhead of traversing the tree structures at low sample counts may become too great. Consequently, we cannot afford photorealistic images at real-time rates. Perhaps more importantly, the current state of these algorithms (based on light hierarchies) does not account for the BRDF and the visibility of the sampled lights. Consequently, these methods may lead to additional estimator variance since they only sample parts of the product in the rendering equation. Finally, another drawback is that these methods require implementing and maintaining complex data structures, which is challenging to do in real-time with fully dynamic environments.

Concurrent work by Bitterli et al. [Bitterli et al., 2020] addresses the aforementioned limitations by introducing an algorithm that can sample lights by repeatedly *resampling* a set of candidate samples. In similarity to real-time lightcuts by Lin and Yuksel [Lin and Yuksel, 2020], Bitterli et al. also accelerate their algorithm through further spatiotemporal resampling to leverage information from relevant adjacent samples. The authors derive an unbiased Monte Carlo estimator that achieves equal-error 6×-60×faster than state-of-the-art methods and a biased estimator that reduces noise further and is 35×-65×faster, at the cost of some energy loss. This thesis will focus on the biased version of the method since it is simpler to implement and more relevant for real-time contexts due to its superior execution times over the unbiased version.

The key idea of the algorithm is to, for each pixel, generate many cheaper light samples that are re-weighted using RIS (explained in subsection 2.2.2) in order to convert them into the desired number of higher quality samples. These high-quality samples are, after that, used in the corresponding lighting calculations. Moreover, to enable a high-performance GPU implementation, a stochastic and lightweight data structure in the form of a *reservoir* [Chao, 1982, Vitter, 1985] is employed. In particular, the reservoir allows RIS to omit storing all the generated candidate samples. Instead, the mechanism for storing these samples in the reservoir is stochastically driven by an unfair coin toss based on the RIS weights (given in Equation 2.5) of the generated samples. Finally, with little additional cost, the quality of the samples is further enhanced through *spatiotemporal resampling*, which increases the effective sample count for each pixel. Ultimately, Bitterli et al. presented a simple algorithm that requires no complex data structures, requires a fixed number of computations per frame, and can sample the full product of the rendering equation approximately.

Since this algorithm—coined *ReSTIR* by the authors—is core to this thesis, an in-depth explanation of the algorithm will be given in section 4.1.

CHAPTER 4

Algorithm

This chapter aims to present our proposed algorithm, which addresses some of the limitations of ReSTIR. In short, the algorithm proposed in this work uses the GPU hash tables data structure found in the path space filtering algorithm [Binder et al., 2021] to facilitate for *sampling lights in world space* using ReSTIR [Bitterli et al., 2020]. Accordingly, section 4.1 and section 4.2 will detail how the ReSTIR algorithm and the GPU hash table work, respectively. Finally, the technicalities of the proposed algorithm are presented in section 4.3.

4.1 ReSTIR

The ReSTIR algorithm [Bitterli et al., 2020] is based on two techniques: RIS [Talbot et al., 2005] and weighted reservoir sampling (WRS) [Chao, 1982, Vitter, 1985]. A general overview of RIS was presented earlier in subsection 2.2.2. We will, as such, in this section, instead present how RIS is used for many-light sampling (subsection 4.1.1) as well as how WRS works (subsection 4.1.2). Accordingly, details on how the aforementioned algorithms are combined to transform RIS into an efficient streaming algorithm, suitable for real-time GPU implementations, will be given in subsection 4.1.3.

4.1.1 Resampled Direct Lighting

Given the rendering equation (Equation 2.16), we can reparameterize it to compute the direct lighting due to an area light source a:

$$L_d(\mathbf{x}, \omega_o) = \int_{\mathbf{y} \in A} f_r(\mathbf{x}, \omega_{\mathbf{y}}, \omega_o) L_e(\mathbf{y}, \omega_{\mathbf{y}}) V(\mathbf{x}, \mathbf{y}) \frac{(\omega_{\mathbf{y}} \cdot \hat{\mathbf{n}}_{\mathbf{x}})(-\omega_{\mathbf{y}} \cdot \hat{\mathbf{n}}_{\mathbf{y}})}{||\mathbf{x} - \mathbf{y}||^2} dA \quad (4.1)$$

where A is the surface of the area light a, $\hat{\mathbf{n}}_{\mathbf{y}}$ is the normal of the light source surface at point $\mathbf{y}$, $\omega_{\mathbf{y}} = \frac{(\mathbf{y}-\mathbf{x})}{||\mathbf{y}-\mathbf{x}||}$ is the direction from $\mathbf{x}$ toward $\mathbf{y}$ on the light source, $L_e(\mathbf{y}, \omega_{\mathbf{y}})$ is the radiance emitted from $\mathbf{y}$ in direction $\omega_{\mathbf{y}}$, and $V(\mathbf{x}, \mathbf{y})$ is a visibility term that equals 1 if the point $\mathbf{x}$ is visible from $\mathbf{y}$ and 0 otherwise.

For the sake of brevity, we omit the exitant direction ω_o and shading point $\mathbf{x}$, define the *geometry term* as $G(\mathbf{x}, \mathbf{y}) = \dfrac{(\omega_{\mathbf{y}} \cdot \hat{\mathbf{n}}_{\mathbf{x}})(-\omega_{\mathbf{y}} \cdot \hat{\mathbf{n}}_{\mathbf{y}})}{||\mathbf{x}-\mathbf{y}||^2}$, and denote the differential area as dx. Thus, the rendering equation for computing direct lighting calculation can be rewritten as:

$$L = \int_A \underbrace{f_r(x) L_e(x) G(x) V(x)}_{F(x)} \, dx \tag{4.2}$$

When sampling direct lighting, we wish to sample light from a distribution p that is proportional to the full integrand F. However, due to the complexities of computing visibility $V(x)$, it becomes inherently difficult to compute such a distribution. Particularly, computing the visibility requires tracing additional *visibility rays* which can be prohibitive in real-time contexts.

Instead, Bitterli et al. [Bitterli et al., 2020] propose to use RIS: Sample M "cheaper" samples from from a candidate distribution q (which should be easy to sample from, and may be sub-optimal) and *resample* them for $N \ll M$ higher quality samples that are distributed according to a target PDF $\hat{p}$. In particular, instead of sampling the full product $F(x)$, we sample from only one of its factors which is cheaper to compute. Bitterli et al. choose $q(x) \propto L_e(x)$ as candidate distribution and $\hat{p}(x) = f_r(x)L_e(x)G(x)$ as target distribution (note the omission of visibility); as such, a light can be sampled by simply sampling it from a PDF based on the scene lights' power and area. Moreover, the visibility computations can be postponed to the end (when the final light sample is selected), making the visibility computations independent of M. Knowing that RIS can generate unbiased samples and may also reduce the estimator variance, we can safely generate a large number of cheaper candidate samples that can robustly be transformed into (fewer) high-quality samples that are approximately distributed according to our desired probability distribution and are, accordingly, used in the radiance calculations.

4.1.2 Weighted Reservoir Sampling

There are a few limitations with RIS. One of the most significant practical limitations is that no matter how we sample our candidates—be it with bisection or linear search—we have to generate and *store* all candidates upfront before selecting the output. This does not map well to GPUs because we want to resample many thousand times in parallel, and we, accordingly, do not want to store thousands of these samples per pixel.

To solve this issue, Bitterli et al. propose to use *weighted reservoir sampling* [Chao, 1982, Vitter, 1985] for stochastically selecting and storing the samples. In short, the algorithm can randomly select an item from a stream with preassigned weights in a single pass over the data—without needing to store the entire stream.

More formally, WRS is a stream-based sampling algorithm for choosing N random samples from a stream $\{x_1, x_2, \ldots, x_M\}$ of possibly unknown length M in a single pass over the data. Each element in the stream has a corresponding weight $w(x_i)$ so that x_i is selected with the probability $P(x_i) = w(x_i) / \sum_{j=1}^{M} w(x_j)$. In our contexts, the N samples are chosen *with replacement*, since we want independent selections x_i for the Monte Carlo integration.

Subsequently, an input stream is processed in order, and a *reservoir* of N samples is maintained. At any point in the stream, WRS maintains the invariant that the samples in the reservoir are drawn from the original stream's distribution (over all elements processed so far). When processing a new stream element, the reservoir is updated to maintain the invariant that sample x_i occurs in the reservoir with probability $w(x_i) / \sum_{j=1}^{M} w(x_j)$. When updating the reservoir with a fresh sample x_{m+1}, another sample in the reservoir x_i is stochastically replaced. The probability of x_{m+1} successfully being stored in the reservoir is,

$$p_{\text{select},m+1} = \frac{w(x_{m+1})}{\sum_{j=1}^{m+1} w(x_j)} \tag{4.3}$$

ensuring that x_{m+1} appears in the reservoir with a probability maintaining the mentioned invariant. Implementing WRS is surprisingly simple, and the pseudocode that backs up this claim is given in Algorithm 1. Note that only the samples of relevance and a running sum of weights are stored, making WRS very space-efficient.

Finally, WRS is combined with RIS into what Bitterli et al. call *Reservoir Resampling*, which selects an output sample from a stream of generated candidate samples without storing all of them: For each number of target samples, a reservoir is created and stochastically updated with the generated candidate samples x_i, using their corresponding RIS weights $w(x_i)$. Ultimately, RIS is transformed into a *streaming algorithm*.

4.1.3 Streaming RIS with Spatiotemporal Reuse

Nonetheless, another problem remains: Even with the use of WRS, we still have to *compute* all of the candidate samples. For complex scenes, a large

Algorithm 1 WRS for storing $N = 1$ samples

```
// $\mathbb{S}$ is the input stream containing pairs of samples and corresponding weights
function WeightedReservoirSampling($\mathbb{S}$)
    total_weight $\leftarrow 0$
    selected_sample $\leftarrow$ None
    for (item, weight) $\in \mathbb{S}$ do
        total_weight $\leftarrow$ total_weight + weight
        if random() < weight/total_weight then
            selected_sample $\leftarrow$ item
    return selected_sample
```

number of candidate samples might be required, and computing all these candidates may not be practical in real-time contexts.

With the knowledge that resampling provides a slightly improved distribution, Bitterli et al. suggest feeding the reservoir resampling algorithm with other samples also generated by a similar RIS procedure. This will increase the effective sample count and help the algorithm by feeding samples from similar distributions. In particular, by *resampling each sample with their spatially and temporally adjacent neighbors*, we can increase the effective sample count by orders of magnitude at an amortized cost. Moreover, the sample variance may get further reduced since the spatially and temporally neighboring RIS samples likely follow similar distributions. Feeding these correlated samples to RIS may result in more well-behaved estimates of the target function [Talbot et al., 2005].

Combining multiple reservoirs is surprisingly straightforward: You simply treat each reservoir's selected sample y as a fresh sample with weight w_{sum} (i.e., the reservoir's current sum of weights of all candidates seen so far) and feed these samples as input into a new reservoir (regarded as the "output or combined reservoir"). Mathematically, the result is equivalent to having performed reservoir sampling on the two reservoirs' combined input streams. However, with the proposed method, the reservoir merging is instead done in *constant time*. This is because it only requires access to the reservoir's current state—thus, avoiding the need to store and retrieving elements of either input stream.

An initial set of M candidates are first generated per pixel in order to produce a reasonably good initial set of samples. These candidates are streamed through their corresponding reservoirs, which are stored in image-sized buffers. *Spatial reuse* of neighboring reservoirs can then be performed, for each pixel, by selecting k neighbors and combining their reservoirs with the pixels' reservoir using the procedure outlined above. Notably, per pixel costs are $\mathcal{O}(k + M)$ but each pixel *effectively* sees $k \cdot M$ candidates. In addition, spatial reuse can be repeated multiple times using the outputs of the prior reuse pass as input, which can further increase the effective sample count.

Taking it a step further, *temporal reusage* can also be performed since the prior frame can provide additional candidates that are similar enough to be reused. After rendering a frame, the final reservoirs for each pixel are stored for reuse in the next frame by, for example, *reprojecting* the current reservoir to the previous frame using 2D motion vectors. This is done similarly to how it was done with temporal accumulation for the screen space filters. Accordingly, if frames are rendered sequentially, and the final reservoirs are passed to the next frame, a frame combines candidates, not just with those of the previous frame, but all previous frames in the sequence—which may significantly improve the image quality and temporal stability [Bitterli et al., 2020].

Computations including the visibility term have been omitted until now. Since the target distribution $\hat{p}$ is set to the unshadowed illumination, noise due to visibility starts to dominate as the number of candidate samples M grows large [Bitterli et al., 2020]. However, we cannot simply incorporate visibility to the target function because we evaluate it for every candidate to compute the RIS weight. We cannot afford to trace a visibility ray for each candidate sample. To solve this issue, *visibility reuse* is performed. Before performing spatiotemporal reuse, the visibility of the selected sample y for each pixel's reservoir is evaluated. If y is occluded, the reservoir is discarded by setting the sample's corresponding weight to 0. Consequently, if we follow up visibility reuse with spatiotemporal reuse, the occluded samples will not propagate to neighboring pixels after this initial visibility test, and all candidates going forward will incorporate visibility in their distribution. Thus, each pixel will leverage the information from many visibility rays from its neighboring pixels and its past to greatly improve its distribution and approximately sample all terms of the product in Equation 4.2.

The final algorithm, dubbed *ReSTIR*, is performed per pixel q in the following order:

1. Generate M initial candidates and resample them to N target candidates, maintained in the reservoirs.

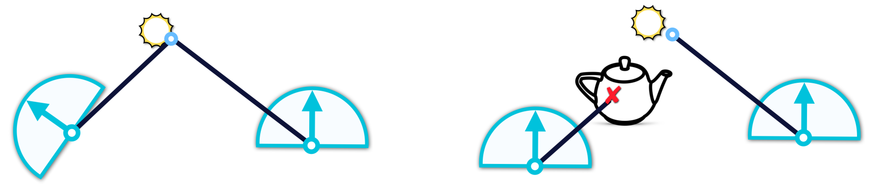
2. Evaluate visibility for the initial candidates and set their weight to 0 if they are in shadow.

3. Perform temporal reservoir reusage, for instance, using temporal reprojection with 2D motion vectors.

4. Perform spatial reservoir reusage n times for k neighbor, for example, by sampling spatial neighbors in a disk of user-defined radius R.

5. Compute the radiance estimates, given the current reservoir's state, using Equation 2.6.

Finally, it is worth noting that RIS is unbiased as long as $q(x) > 0$ where $\hat{p}(x) > 0$. Bias will be introduced as such if this requirement is not met. This could, for instance, happen when we use samples from neighboring pixels, and Figure 4.1 shows two examples of situations where this might be the case. Furthermore, Bitterli et al. mathematically demonstrate that bias occurs when the RIS weighting factor is no longer an estimator of the inverse PDF. They propose a method for correcting this bias which, however, is rather expensive in real-time rendering as it requires tracing additional visibility rays. Instead, a more practical solution is to attempt *minimizing bias*: Using a simple heuristic, the normals and depths of the neighboring pixels are compared, and if they are too different, the samples from those pixels can be rejected.



(a) Sample with wrong orientation (b) Occluded sample

Figure 4.1: Two examples of where bias can occur when performing spatiotemporal resampling with ReSTIR: (a) when the neighbor sample is below the normal-oriented hemisphere, or (b) when the neighbor is occluded. In both cases, $q_{neighbor}(x) > 0$ while $\hat{p}_{current}(x) \leq 0$, which results in bias if that neighboring sample is combined with the current sample.

For real-time, this bias-speed trade-off is beneficial, and Bitterli et al. show that the biased algorithm can result in considerably less variance, at the cost of some energy loss and image darkening, when compared to the unbiased variant executed and measured at equal times.

4.2 GPU Hash Tables

The hash table implementation in this thesis follows the work by Binder et al. [Binder et al., 2021] and Pascal Gautron [Gautron, 2020]. In short, these methods attempt to discretize the vertices, in world-space, into voxels of non-uniform size, which are stored in a hash table for efficient look-up and insertions. The following will present how the table is generated and used.

4.2.1 Spatial Hashing

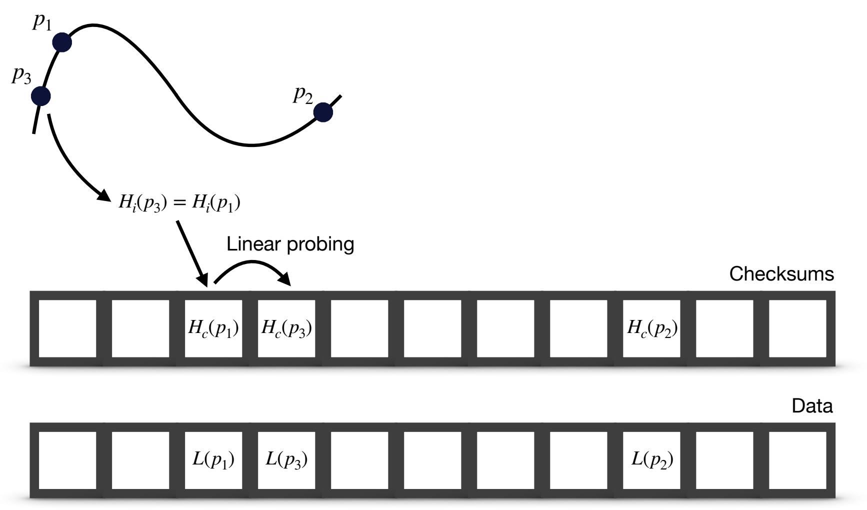


Figure 4.2: Simplified example demonstrating how spatial hashing is performed when inserting an entry into the hash table. Hash cell conflicts are solved through linear probing. The sample data stored here is some output of an arbitrary function $L(p)$.

The whole procedure for performing spatial hashing and inserting an entry into the hash table is summarized in Figure 4.2: Given a position p in world space, an n-dimensional hash function $h_i(p)$ is applied, which yields a hash key that acts as the entry *index* to the hash table (i.e., $h_i : \mathbb{R}^n \mapsto \mathbb{N}$). Moreover, in order to uniquely identify each entry, a *checksum* is computed and stored with each entry using a secondary hash function $h_c(p)$. As such, collisions between entries can efficiently be identified using these checksums. As mentioned by Binder et al., *linear probing* has shown to perform well in practice when implemented on GPUs [Binder et al., 2021]. Accordingly, we also apply linear probing in this work to resolve conflicts among hash table entries.

4.2.2 Construction of hash keys through discretization

To discretize, say, a 3D point p, *nesting* is performed [Perlin, 1985]: Each component of the point (e.g., p_x, p_y and p_z in 3D) are individually hashed using some hash function h:

$$H_{3D}(p) = h(p_z + h(p_y + h(p_x))) \tag{4.4}$$

However, naively hashing the bits to the floating-point coordinates of a point may result in that two spatially nearby entries not appearing close to each other in the hash table. In other words, that would mean that the output of $H_i(p)$ would be different for two nearby points, and it would not be possible to process the two points together. To avoid this, Gautron and Binder et al. suggest resorting to *discretization*, which would mean that the points instead are discretized into *voxels* of size s:

$$H_{3D}(p) = h(\lfloor p_z/s \rfloor + h(\lfloor p_y/s \rfloor + h(\lfloor p_x/s \rfloor))) \tag{4.5}$$

As such, two points within the same voxel (i.e., they are within distance s of each other) will end up with the same hash index.

For the discretization to work in dynamic contexts, Binder et al. suggest using an adaptive voxel size $s_{adaptive}$ by parameterizing it on the projected size of a defined screen area using the projection theorem. Consequently, the voxels will have a non-uniform resolution that adaptively changes based on the size of the projected screen area of a voxel when using the following revised hash function:

$$H_{3D}(p) = h(\lfloor p_z/s_{adaptive} \rfloor + h(\lfloor p_y/s_{adaptive} \rfloor + h(\lfloor p_x/s_{adaptive} \rfloor))) \tag{4.6}$$

To make the hash key dependent on the voxel size resolution (which can also be seen as a level of detail identifier), we can further nest $H_{3D}(p)$ with the adaptive voxel size $s_{adpaptive}$, making the function 4D:

$$H_{4D}(p) = h(s_{adaptive} + H_{3D}(p)) \tag{4.7}$$

Finally, for the hash function to take into consideration that two nearby points still may be different due to different surface orientations, the surface normal n is also nested:

$$H_{7D}(p) = h(\lfloor n_z s_{nd} \rfloor + h(\lfloor n_y s_{nd} \rfloor + h(\lfloor n_x s_{nd} \rfloor + H_{4D}(p)))) \tag{4.8}$$

where s_{nd} is a discretization factor (e.g., $s_{nd} = 10$). Ultimately, $H_{7D}(p)$ becomes the spatial hashing function for the hash table, and it is applied to any input point p in world space.

4.2.3 Reducing quantization artifacts with jittering

Due to the discretization applied, quantization artifacts may occur. In a similar fashion to Binder et al., and as explained in subsection 3.1.2, this is addressed by resolving the quantization artifacts in noise. Specifically, when inserting and looking up entries in the hash table, the world space positions are jittered in the tangent plane of the surface.

4.3 Path Space Importance Resampling

The proposed algorithm, dubbed *Path Space Importance Resampling*—or, PSIR for short—is an extension to the original ReSTIR algorithm for it to work in path space (interchangeably referred to as world space in this thesis).

The key idea of PSIR is to address one of the main limitations of the original ReSTIR algorithm: It is only able to operate on the first vertex of the camera path (i.e., the primary hit point), which makes it difficult for ReSTIR to be extended to direct lighting and global illumination beyond the first hit. This limitation stems from the fact that the original ReSTIR algorithm operates on image buffers. While this makes the algorithm fast, simple and, memory-efficient, it will result in the aforementioned limitation. Moreover, ReSTIR is also limited because the initially selected light samples are selected through importance sampling based on a PDF computed from the light sources' power and area. Consequently, the initial candidates do not account for the actual distances to the light samples. This can be of practical relevance, as we will waste computational resources on distant light sources with high power that likely will not contribute to the final lighting surfaces due to lighting attenuation. PSIR attempts to mitigate these limitations by extending ReSTIR beyond screen space.

In particular, the proposed algorithm augments ReSTIR as following: A *hash table* is, firstly, filled with light *samples* using the procedure presented in section 4.2—that is, light samples are inserted into the hash table by constructing a spatial hash key based on their positions and surface normals. Moreover, once the hash cell—containing a fixed number of lights—is full, the *light samples are stochastically inserted* to the hash table by applying RIS based on the lights' expected contribution to the hash cell volume (i.e., the voxel's volume). Concretely, we apply RIS by setting the target PDF $\hat{p}(x)$ to the expected unshadowed illumination from the light to any surface covered by the hash cell (i.e., $\hat{p}(x) = f_r(x)L_e(x)G(x)$) and the candidate distribution $q(x)$ proportional to the probability of selecting a light (i.e., $q(x) \propto L_e(x)$).

For each light sample, we store the light's properties, such as its orientation, position, and emissive color, as well as its selection probability. Moreover, our hash table "build step" is performed on a per-frame basis and is performed by iterating through each light in the scene, in parallel, using the thousands of threads available in GPUs. There are n_{cell} hash cells, and for each of those, $n_{entries}$ of light samples are stored. Consequently, at least $n_{cell} \cdot n_{entries}$ bytes of data needs to be allocated for storing the values of the hash table if each light sample stores a byte each. A visualization displaying how the hash cells are assigned when hashing the visible surfaces, using the procedure outlined in subsection 4.2.2, is shown Figure 4.3.

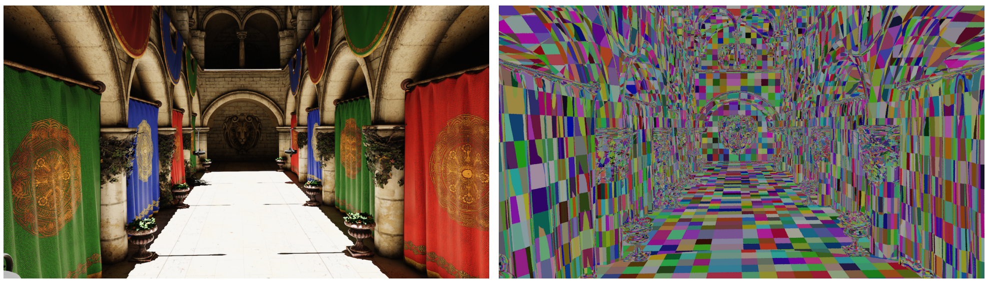


Figure 4.3: Visualization of how the hash cells are assigned (b), with the reference scene shown in (a). In (b), each hash key is given a color based on their cell index, which is then directly visualized. Notice how the granularity of the cells depends on both the position, distance from the camera, and the surface orientation at that point.

At this point, the hash table is filled with relevant light samples. The data structure can now, in theory, be used in any step of the ReSTIR algorithm: Simply provide the necessary data of a surface (e.g., its world position and surface normal) to construct a spatial hash key in order to look up if light sample data is available in that corresponding cell. If not, perform the resampling as you normally would do with ReSTIR; otherwise, use the light sample data retrieved from the hash table. If there are multiple light candidates in the hash cell, resampling may be applied to those lights to select the final light sample stochastically. Since any world space position and surface normal may be used to construct the spatial hash for performing hash table lookups, this data structure may be beneficial in shading secondary surfaces where a primary hit G-buffer is unavailable. For instance, this structure could be instrumental in performing light sampling for secondary rays during path tracing.

Intuitively, the described hash table construction step could also be interpreted as a process for generating *unordered groups of the light samples*: Lights with similar characteristics—that is, lights that are nearby or on similar oriented surfaces—are grouped in the same voxels. Subsequently, when sampling lights from this structure, only ”relevant” lights are considered, and lights that are likely too far away to contribute are inherently filtered out during the lookup of the light samples.

Our algorithm directly extends ReSTIR and, accordingly, follows the same pipeline on a per-frame basis. Subsequently, the concrete steps of our algorithm are, orderly, given as follows:

1. Build hash table by iterating through each light and fill the light samples in the hash cell. If a hash cell is full, use RIS to maintain the hash cell stochastically.
2. Generate M initial candidates as in ReSTIR but sample them from the hash table instead. If the corresponding hash cell for the surface position and normal is empty or invalid, perform ReSTIR as you otherwise would.
3. Perform visibility reuse.
4. Perform spatiotemporal resampling as standard ReSTIR.
5. Compute radiance estimates.

Finally, note that we also jitter (as explained in section 4.2) the sample positions to minimize quantization artifacts when inserting and looking up data from the hash table.

CHAPTER 5

Experiment

This chapter begins by presenting the experiment conducted for evaluating the performances of PSIR. The purpose of this experiment is to evaluate how well PSIR extends ReSTIR. With this, we can investigate when or if the former should be preferred over the latter. Subsequently, the results discovered during this thesis are presented and are accompanied by an analysis for it.

In this chapter, we highly recommend the reader to zoom in when inspecting the figures, assuming this document is read digitally, to get a better and more clear view of ReSTIR's and PSIR's noise characteristics.

5.1 Setup

5.1.1 Our Test Scenes

We test our implementation on two different scenes: Bistro and Zero-Day. The first and most important reason for this selection of scenes is that the lighting in these scenes fully depends on the direct illumination from the many area light sources—assuming we neglect the environment map contribution. Consequently, this allows us to demonstrate the results of performing *efficient* many-light sampling; specifically, by resampling many lower-quality samples into fewer but higher quality samples. The second reason is that both scenes contain challenging, professionally created, high-quality assets, and—even better—they are publicly available and are free for anyone to use. Finally, do note that scenes also are employed to stress our implementations under varying conditions.

Bistro. Amazon Lumberyard Bistro (Fig. 5.1a) is the first scene we use. It was revealed at the 2017 Game Developer Conference to showcase new anti-aliasing and transparency features in the Lumberyard game engine. It

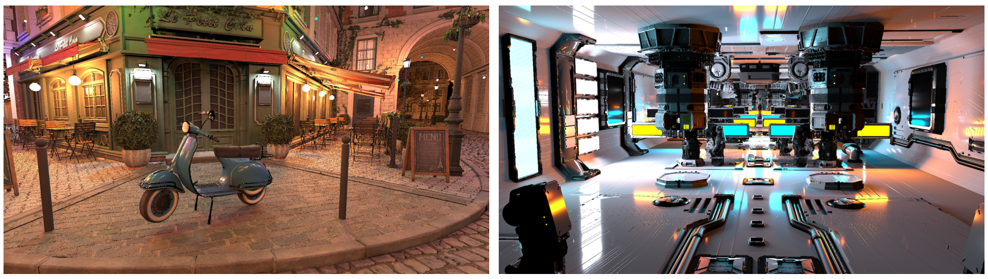


Figure 5.1: Sample renderings of our test scenes, (a) Bistro and (b) Zero-Day. Sample images from [Bitterli et al., 2020].

was later donated to NVIDIA's Open Research Content Archive (ORCA) [Lumberyard, 2017]. The scene consists of two sub-scenes, an interior, and an exterior scene, containing $1,046,609$ and $1,293,691$ triangles, respectively. Bistro contains in total 20,638 emissive triangles, which makes it a great environment for testing and comparing our resampling algorithms.

Zero-Day. BEEPLE Zero-Day (Fig. 5.1b) is the other scene we employ. It was created by Mike Winkelmann for a short film called Zero-Day in 2015 [Winkelmann, 2019] and was also donated to ORCA. It consists of two sub-scenes, Measure One and Measure Seven, which contain $1,372,670$ and $1,294,866$ triangles, respectively. In addition, the two sub-scenes also contain $10,103$ and $10,989$ dynamic emissive triangles, respectively. Subsequently, this gives us another scene with many emissive triangles to benchmark; however, this time in another context, with different materials and lighting conditions.

5.1.2 Implementation Details

We build our implementations of ReSTIR and PSIR on top of our custom-made, fully ray-traced, rendering engine. It uses the NVIDIA CUDA [NVIDIA et al., 2020] parallel computing platform for general GPU computations and the NVIDIA OptiX [Parker et al., 2010] ray tracing engine to take advantage of NVIDIA's ray tracing hardware.

Generating A Frame

Our engine renders a frame as follows: First, a geometry pass is executed by tracing *primary rays* for each pixel and computing the corresponding geometry and material properties for each visible surface. The output from this pass is a *G-buffer* (see subsection 3.1.1). Secondly, we compute the direct lighting for each pixel by sampling a light and computing the illumination by performing a single evaluation of the rendering equation. For this, we need to evaluate the BRDF of the visible surfaces by using the recently generated G-buffer. Subsequently, we need to evaluate the visibility by casting shadow rays. It is in this stage of the pipeline that we would apply ReSTIR or PSIR for sampling the lights for the direct lighting calculations. In the third stage, indirect rays may be traced by sampling new directions based on the current surface's BRDF. In our engine, we use a unified material model consisting of a GGX microfacet layer [Walter et al., 2007] atop a Lambertian substrate, enabling us to render physically based materials efficiently. The indirect lighting stage is optional, and the maximum number of indirect rays per surface hit is limited to 1 to stay within the target render time of 16.67ms (i.e., 60 frames per second). Finally, we perform a composition pass to combine the indirect and direct illumination.

Resampling Settings

For initial candidate generation and spatiotemporal resampling, in both PSIR and ReSTIR, we found the implementation choices made by Bitterli et al. [Bitterli et al., 2020] to work quite well in practice. Our resampling implementation follows most of their suggestions, albeit with some disparities. In particular, we generate $M = 32$ and resample with candidate PDF $q(x) \propto L_e(x)$ and target PDF $\hat{p}(x) = \rho \cdot L_e \cdot G$. In more practical terms, this means that we generate our initial samples by importance sampling emissive surfaces based on their power and solid angle. We then evaluate the target PDF by computing a single sample of the rendering equation without considering visibility.

However, in contrast to the implementation proposed by Bitterli et al., we do not consider the contributions made by the environment map since we assume they are not present in our scenes. Nonetheless, it should be relatively straightforward to add support for environment maps. For instance, a proportionality, or probability factor, may be defined, which yields the portion of initial samples that must come from the environment map. Moreover, we only implemented the biased version of the resampling algorithm since we quickly observed that the improvement in visual quality was not worth it, compared to the performance decline for the unbiased implementation. Accordingly, we evaluate $N = 1$ samples for each frame (i.e., one shadow ray is traced per pixel each frame), resample $k = 5$ spatial neighbors in $n = 1$ spatial reuse pass, followed by a temporal resampling reuse pass. We also discard different samples using the suggested heuristic, which compares the surface and material properties of the samples.

Hash Table Settings

Our GPU hash table implementation mainly follows the work by Binder et al. [Binder et al., 2021] and is presented in section 4.2. Although the parameters for the hash table are initially assumed to be scene-dependent, we found it sufficient to use the same parameters for both our test scenes. Namely, we generate the hash keys using the procedure outlined in subsection 4.2.2. However, instead of computing the adaptive discretization factor $s_{adaptive}$ from the projected pixel size of a voxel, we instead found it effective to compute $s_{adaptive}$ based on the distance between the camera and the target world position:

$$s_{adaptive} = 2^{\left\lfloor \log_2\left(\frac{||p_{camera}-p||}{s_{scale}}\right)\right\rfloor} s_{scale} \tag{5.1}$$

Where p_{camera} is the world space camera position, p is the world space position of the point we want to hash, and s_{scale} is a scene distance scale constant that works as a discretization constant for quantizing our computed distances. In our experiments, we found $s_{scale} = 0.01$ to work well as it resulted in smaller-sized hash cells for positions far away and larger-sized cells for nearby positions. Finally, to nest our normals into our hash key, we use $s_{nd} = 10.0$ as discretization constants to quantize our normals.

We use `pcg` [O'Neill, 2014] as our index hash function h_i and `xxhash32` [Collet, 2016] as our checksum hash function h_c, since those have been shown to be fast enough to be useful for real-time, while also being high-quality enough (e.g., these lies along the Pareto frontier) for almost any graphics use-case [Jarzynski and Olano, 2020]. Prior to insertion or retrieval of data from the hash table, we jitter the world-space position p on the corresponding tangent plane of its surface before performing the discretization for generating the position-component h_{3D} of the hash key. In case two hash keys map to the same hash cell, we perform linear probing to resolve the conflict, as outlined in subsection 4.2.1. If the hash table lookup fails, even after linear probing—for instance, due to a hash cell being empty—we fall back to the original ReSTIR algorithm. Finally, for our two test scenes, we allocated a total of $n_{cells} = 1,000,000$ hash cells and $n_{entries} = 32$ entries per hash cell.

5.2 Results

We test our implementations at standard HD resolution (1280×720 pixels) on a system with an NVIDIA GeForce RTX 2070 SUPER GPU. We initially attempted to implement the algorithms at full HD (1920x1080 pixels), however, due to the memory available on the GPU, we had to render at the first-mentioned resolution. The rendering times we report only include the total *lighting computation timings*; as such, the measured timings include the initial sample generation, spatiotemporal resampling, direct lighting computation, optionally the indirect lighting computation, and the hash-table build step if PSIR is applied. Moreover, we report the image errors in *Relative Mean Squared Error* (RMSE):

$$\text{RMSE} = \frac{1}{N}\sum_{i=1}^{N}\frac{(y_i - \hat{y}_i)^2}{y_i^2} \tag{5.2}$$

Where N is the total number of pixels in our rendered output, while $\hat{y}_i$ and y_i are our estimated and reference value of pixel i, respectively. We use this measure since previous work has reported that computing the relative error can be less sensitive to outliers than, for instance, computing the mean squared error (MSE) [Bitterli et al., 2020]. This observation was also consistent with our findings of the various error metrics. Finally, the data is generated by rendering a frame using the setup presented in subsection 5.1.2; that is, a 1 spp ray tracer is used to render the scene, where one of PSIR or ReSTIR is employed to sample lights when evaluating the rendering equation. In general, our figures show the first frame rendered. The exception is if temporal reusage in ReSTIR or PSIR is used—in that case, our figures show the final frame of a 20 frame render in order to "warm up" our history buffer with temporal data [Karis, 2014, Bitterli et al., 2020, Koskela et al., 2019].

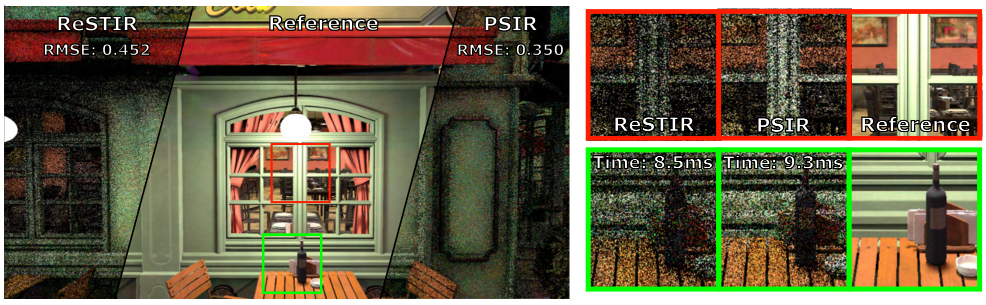


Figure 5.2: Single-frame snapshot of Bistro rendered using ReSTIR and PSIR, accompanied with a 2048 spp reference render. There is less energy loss in the images rendered with PSIR since it is better able to filter distant lights that are likely noncontributing.

Figure 5.2 and Figure 5.3 show equal-time comparisons of PSIR against ReSTIR—which works as the baseline for our comparisons in our two test scenes. Quantitatively, in both the test scenes, our technique exhibits lower or equal error than Bitterli et al.'s ReSTIR. Moreover, inspecting the rendered frames of both algorithms, we also see that PSIR yields frames of higher quality. That is, frames that, qualitatively, look more similar to the reference render. Specifically, we see that PSIR results in images with less energy loss

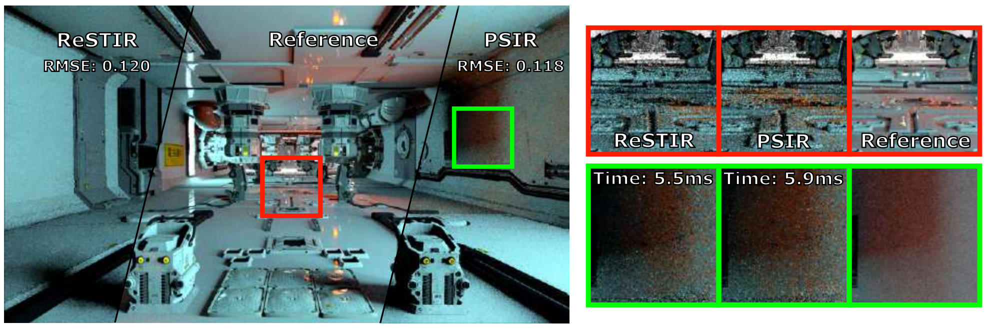


Figure 5.3: Single-frame snapshot of Zero-Day rendered using ReSTIR and PSIR, accompanied with a 2048 spp reference render. Notice that PSIR can better sample the non-visible lights (e.g., the images rendered with PSIR have more orange lighting in them) in the scene.

in certain areas in Figure 5.2. Furthermore, we also see from Figure 5.3 that PSIR is able to sample the light sources outside the primary view, resulting in more orange-colored highlights in certain areas, which corresponds better with the ground truth. In general, we see that the quality between the two algorithms is near-identical. Ultimately, it is worth noting that the extra steps PSIR incur to ReSTIR are not free, and, in average, comes at the cost of from 0.7ms to at most 2.0ms for our two test scenes.

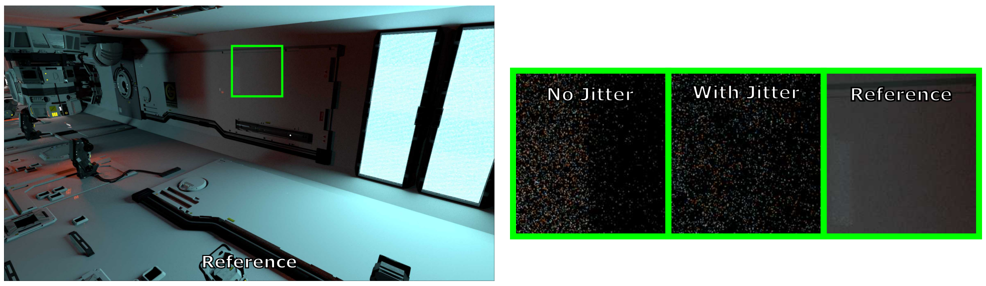


Figure 5.4: Single-frame snapshot of Zero-Day rendered with PSIR, with and without jittering applied. The boundaries between adjacent hash cells are visible when jittering is omitted. In contrast, when jittering is applied, these quantization artifacts are traded for noise which is easier for the eyes to ignore.

We also tested the importance of jittering our samples when looking up or inserting samples into the data structure. From the leftmost zoomed-in crop of Figure 5.4, we see that quantization artifacts can occur when jittering is neglected. In fact, we see the discretization boundaries between neighboring hash cells when jittering is omitted. However, by performing jittering, we see that these artifacts effectively are removed—at the cost of little to no additional performance penalties, assuming that the amount of jitter is not excessive (we would otherwise be trashing the GPU cache lines). In particular, from the middle crop of the mentioned figure, we see that there is a bit more additional unstructured noise, but, at the same time, the color boundaries that previously stood out are now eliminated.

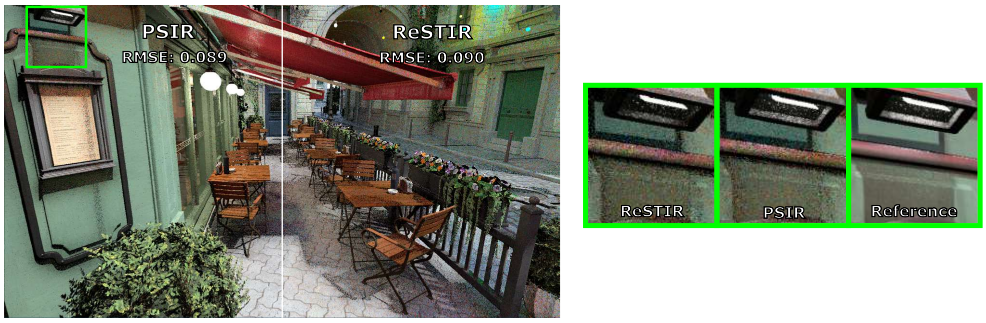


Figure 5.5: Snapshot of Bistro (at day time) rendered with PSIR and ReSTIR, *only using temporal resampling*. There are only minor differences between ReSTIR and PSIR, both qualitatively and quantitatively.

Bitterli et al. showed that temporal reusage could dramatically increase the effective sample count and, consequently, drastically improve the final image quality. To further test the effectiveness of PSIR, we decided to evaluate the algorithm with and without temporal reusage. Figure 5.5 displays a snapshot of Bistro rendered with ReSTIR and PSIR, with only temporal reusage enabled (in addition to initial sample generation). The results between PSIR and ReSTIR are qualitatively and quantitatively close to virtually identical—with almost the same reported RMSE and image quality. However, from closer inspections (e.g., the selected crop in green), we see that PSIR, in some areas, is still better than ReSTIR at outputting images that better resemble the reference image. In particular, we see additional white lighting on the border of the rectangular spotlight on the image rendered using PSIR. There is otherwise very little that separates the two images.

Our algorithm was, furthermore, also tested without temporal reusage (i.e., we only executed spatial resampling and initial candidate generation). Figure 5.6 shows a render from the same view as Figure 5.5. There is a great reduction in image quality, and the reported error is almost a magnitude larger than previously—demonstrating the impact of reusing temporally adjacent samples. However, looking at the errors of the two algorithms relative to each other, we see that the disparity between them is still not that big. Further inspection of the image (e.g., the selected crop in green) shows that PSIR results in less noise and can result in more evident structures in the image.

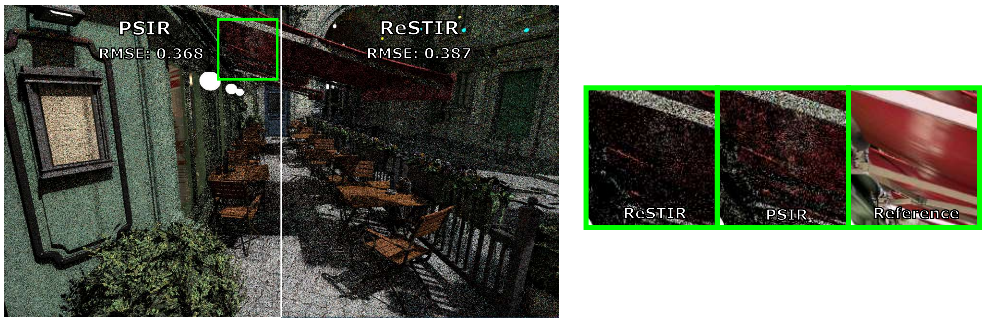


Figure 5.6: Snapshot of Bistro (at day time) rendered with PSIR and ReSTIR, *only using spatial resampling*, from the same view as Figure 5.5. There are visibly larger differences between PSIR and ReSTIR when temporal sample reusage is omitted.

Specifically, we see that PSIR is able to show more of the red color of the awning and its white specular highlights.

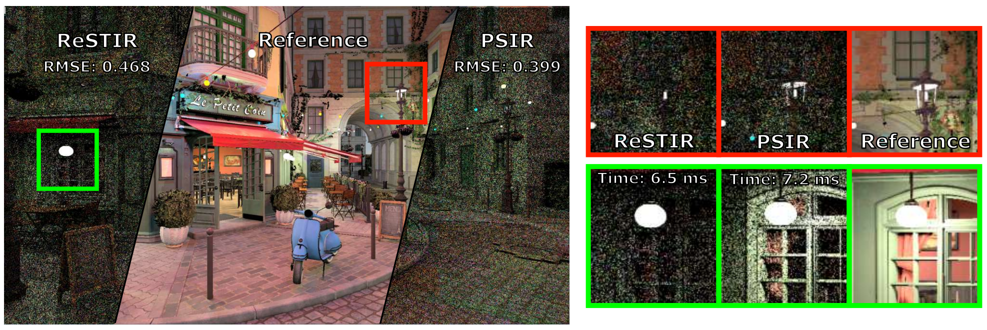


Figure 5.7: Snapshot of Bistro rendered with PSIR and ReSTIR, *only using the initially generated samples*, accompanied with a 2048 spp reference render. Notice that, without any kind of sample reusage, the effectiveness of PSIR is visibly improved over ReSTIR.

Accordingly, we also tested our algorithm without any kind of sample reusage. That is, we only executed the initial sample generation stage where we generated M candidate samples. For PSIR, we also performed the hash table build step before the initial sample generation and used the generated table as an ”initial sample pool” for generating the candidate samples. Figure 5.7 shows that the difference in error between the two algorithms is now even more prominent. Looking at the differences between the images (e.g., see selected crops in red and green), we see evident disparities between the outputs produced by the two algorithms. On surfaces close to the light sources, we see that ReSTIR results in noisy outputs with little to no structure. In contrast, the selected crops indicate that PSIR has better capabilities at sampling lights which are more likely to contribute to the final illumination.

Finally, we remark that the additional performance cost of using PSIR on top of ReSTIR was roughly between 0.7ms to around 2.0ms on our two test scenes. We found the additional performance cost of performing lookups of light sample data from the hash table negligible when we performed resampling. The main cost came from constructing the hash table at each frame. For our two test scenes, we found the build step performance to be approximately the same, with an average execution time of 0.5ms-1.2ms, depending on the total number of lights in the scene and the number of lights in the proximity of the primary view. Another aspect of PSIR is memory usage. Extending ReSTIR with a hash table required an additional $\sim$ 500MB of memory—which was used for storing hash keys, the hash cells' age, and the corresponding values—so that the cells in the data structure would sufficiently "cover" the test scenes. This additional memory usage could be critical depending on the target platform. Allocating too few cells would result in fuller hash tables, which prompted linear probing to occur more often (due to the higher likelihood of a hash cell collision) and would, consequently, lead to slower hash table build times. However, on the contrary, allocating too many hash cells would result in excessive use of the limited amount of GPU memory we have. Nonetheless, compared to the other screen-sized buffers, such as the reservoir buffer, the diffuse and specular illumination buffer, and the G-buffer, the overall memory impact of the hash table size was found to be relatively minimal for our test scenes.

5.3 Discussion

From the reported results, we saw that PSIR generally had the edge over ReSTIR—both quantitatively and qualitatively. However, the differences between the outputs of the two algorithms were mostly rather minor. Through further visual inspections, we noticed there were only certain areas in the scenes where PSIR drastically improved the image quality over ReSTIR; in particular, the biggest differences between PSIR and ReSTIR only occurred when we omitted temporal resampling. We believe this is due to the nature of how temporal resampling between consecutive frames is performed: A frame combines candidates not just with those of the previous frame but also with all previous frames in the sequence. Consequently, reusing data in this manner improves the effective sample count dramatically and, accordingly, the final image quality. Furthermore, this also meant that the effect of applying PSIR would be hidden or masked by the temporal sample reusage since the majority of the sample contributions would come from reusing the samples temporally. This can be deduced through visual assessments of Figure 5.2, Figure 5.3 and Figure 5.5 where the output images from both PSIR and ReSTIR look to be nearly converged. Though, even while the images look to be almost converged, PSIR still shows some improvements over ReSTIR. For instance, we saw in Figure 5.5 that PSIR was better able to capture the correct lighting around the border of the spotlight, which gave some—perhaps weak—indications that the algorithm was properly able to sample the relevant (i.e., the most likely contributing) lights.

Note that we can also draw similar conclusions, regarding the improved effective sample counts, from Figure 5.6, where we only perform spatial resampling—albeit, the effect is not to the same degree as when applying temporal resampling. When we perform spatial resampling, samples from similar surfaces and in close proximity are reused and combined. Accordingly, by only performing spatial resampling, we will not achieve the same dramatic increase in sample counts as we did with temporal resampling. The combined samples are not propagated to their spatially adjacent neighbors efficiently, which means that the sample counts may not increase drastically. The consequences of this become evident when comparing Figure 5.5 and Figure 5.6, where the reported RMSE increases by almost an order of magnitude and the visual quality drastically deteriorate when going from temporal-only resampling to spatial-only resampling. The differences between PSIR and ReSTIR become more apparent as such, in the form of better visible specular highlights and improved lighting in certain areas. This discrepancy is likely because PSIR can now contribute more due to the lower effective sample counts when performing spatial resampling only.

PSIR's Role

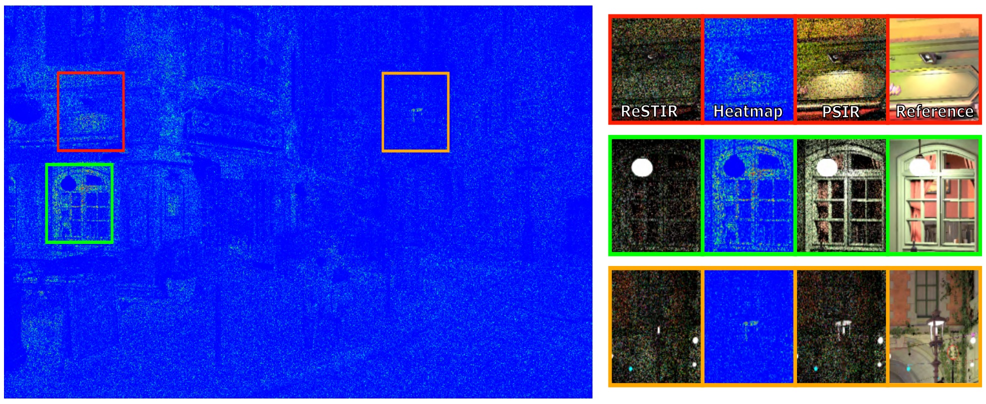


Figure 5.8: Error heat map of Figure 5.7 between PSIR and ReSTIR.

To better understand where, why and how PSIR performs better than ReSTIR, we refer to Figure 5.8, which shows the error heat map of Figure 5.7 between ReSTIR and PSIR. Specifically, it shows the error between ReSTIR and

PSIR without any kind of sample reusage since we found sample reusage (albeit mostly the temporal sample reusage) to ”hide” the contributions of PSIR. From the heat map, we see at least three areas where the RMSE is particularly dense and has the highest errors. We highlight these three areas in the red, green, and orange crops. Interestingly, the selected crops show that the differences between PSIR and ReSTIR are the highest on surfaces closest to the light sources. When we further inspect the renders for ReSTIR and the corresponding one for PSIR and compare it against the reference, we see that PSIR indeed is better able to account for the contribution from the closest light source—which is often the one contributing the most to the final lighting for that surface.

The fact that PSIR shows better tendencies at sampling light from the closest light source corresponds well with our intuition of the algorithm, which we explained at the end of section 4.3. In short, what our path space data structure *effectively* does is to create an unordered set of light samples, which are stored in the same or close-by voxels and, accordingly, in the same or nearby hash cells. To understand why this helps, we will look at how ReSTIR performs the initial sample generation. As mentioned in section 5.1, with ReSTIR, we generate our M initial samples by importance sampling emissive surfaces based on their expected contribution. The expected contribution is computed based on the lights’ power and area; that is, the distances to each light from the surface points are not considered. Thus, light sources with high emissive power are still sampled a lot, even if they are positioned far away from the surface points. In reality, computing the illumination from a distant light source will result in an attenuated contribution due to the inverse-square attenuation law [Pharr et al., 2016]. Fortunately, the probabilities of selecting these lights are stochastically filtered when using RIS, and the samples are completely removed when the surface is in shadow and visibility reuse is performed [Bitterli et al., 2020]. Nevertheless, even if our sampling probabilities are effectively filtered, we still waste our computational resources on generating and filtering these samples that might not even contribute to the final result.

To demonstrate how such distant lights affect ReSTIR and PSIR, we examine an extreme case where an ”anomalous light source” with a large and disproportionate amount of emissive power is inserted in Bistro, at a position far away from the scene. Figure 5.9 shows how PSIR and ReSTIR handle such a case where we have a highly skewed initial light distribution; that is, a light distribution where the probability mass is skewed towards a particular light source. Remember, we pre-process (specifically, we generate an alias table [Walker, 1977]) our lights before we start rendering so that we can efficiently sample lights using this initial light distribution. Since ReSTIR samples its candidate sample only based on that initial distribution, it will likely sample the anomalous light without considering the distance towards

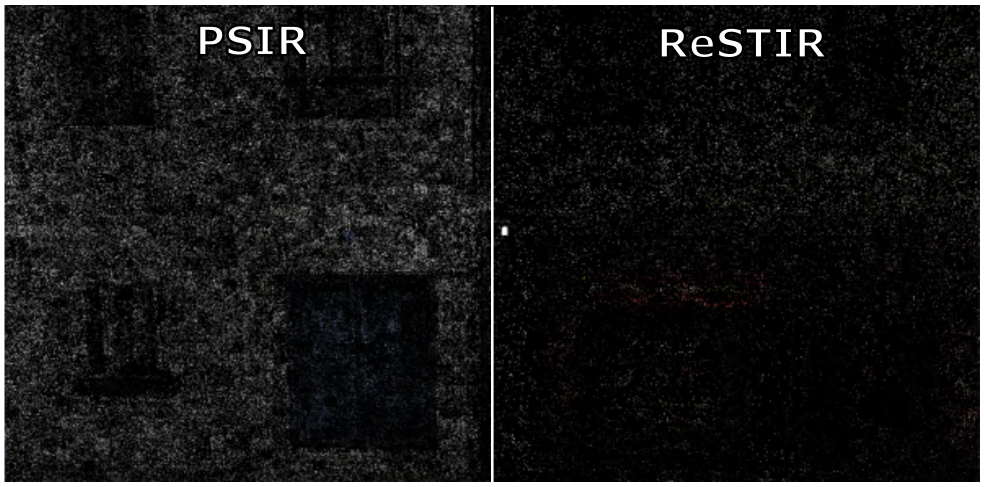


Figure 5.9: PSIR vs. ReSTIR when an anomalous light source, with an excessive amount of emissive power, is placed far away from the scene.

it. Consequently, these selected anomalous samples will be wasted since RIS or the visibility tests will "trash" them. In fact, this is what we see happens with ReSTIR in Figure 5.9, where the final image produced hardly contains any visible structure at all.

On the contrary, PSIR generates its initial samples by looking up available sample data from the path space hash table, based on the current surface's position and normal. Since we build the hash table effectively through an unordered grouping of the lights into discrete voxels, we will essentially have filtered away many of the distant noncontributing lights in the hash table construction step. Moreover, since we select the lights stochastically with RIS, the generated light pool should be robust, containing mostly relevant samples—assuming that our selected candidate and target distribution is well-behaved and that our hash cells are full. As such, we believe that the main contribution of PSIR to ReSTIR is that it *boosts the quality of the samples*—particularly by boosting the quality of the candidate samples. Indeed, from Figure 5.9 we see this in practice. Here we see that since PSIR additionally samples its candidate samples from the hash table, it becomes more robust due to it being capable of disregarding the anomalous light. Subsequently, this results in a significantly more visible structure to the final image with a lot more low-frequency detail.

To summarize our discussion up to now, we have seen that PSIR extends and improves ReSTIR mainly by boosting the quality of the generated samples. By pre-processing the lights and grouping them into discrete voxels in a hash table construction step, we would effectively pre-filter noncontributing and distant light sources. Consequently, we saw the most significant differences in sample quality between ReSTIR and PSIR on surfaces close to the light sources since ReSTIR would not be able to ignore the distant and noncontributing lights directly. In contrast, with PSIR, we would effectively ignore these distant and noncontributing lights when sampling from the hash table due to how we construct these light pools. On the final image, around the surfaces close to the light sources, we would have more noise with ReSTIR, whereas PSIR would have less noise and exhibit more low-frequency detail with more evident structure. Although there were improvements in image quality with PSIR when using spatiotemporal resampling, we saw the biggest quality improvements when we rendered images by only using the initial candidate samples. These improvements over ReSTIR made sense, as ReSTIR only generated the initial samples by importance sampling emissive surfaces based on PDFs generated from the light sources' power and area. By also considering the samples in our path space data structure, the selected candidate samples would be "pre-filtered" and take the distance to the light sources into account—which would effectively cull distant and weak lights that likely would not contribute much to the final result.

PSIR's Performances

The focus of the discussion has so far been about what PSIR brings over ReSTIR to the final image quality. However, as we saw from the results presented earlier, PSIR does in fact incur extra performance costs. The main drawback of PSIR's performance is that it depends on the number of lights in the scene. Moreover, for the hash cells to sufficiently "cover" the scene, it becomes crucial to allocate a sufficient amount of memory for the hash table. Ideally, we would like to allocate enough memory so that a hash cell at least covers a certain area around its designated light source (i.e., the light source closest to the hash cell's unprojected world space position). For our two test scenes, which contained around $10,000$ and $20,000$ emissive triangles, the additional cost of looking up and inserting samples to the hash table was between 0.7ms and 2.0ms, depending on the view direction and the scene complexity.

The performance numbers in the figures presented earlier in this section were dominated mainly by the hash table construction step at each frame. As a matter of fact, the cost of looking up light samples from the hash tables seemed to be "hidden" by the performance costs of performing resampling. Curiously, this corresponds well with how the GPU is designed: Since many thousands of threads run in parallel, the performance of each individual thread will be hidden by the total performance. Consequently, the focus is on the overall throughput rather than the latency of each individual thread. We tested the scalability of the hash table build step by inserting up to one million artificial light sources placed randomly in the

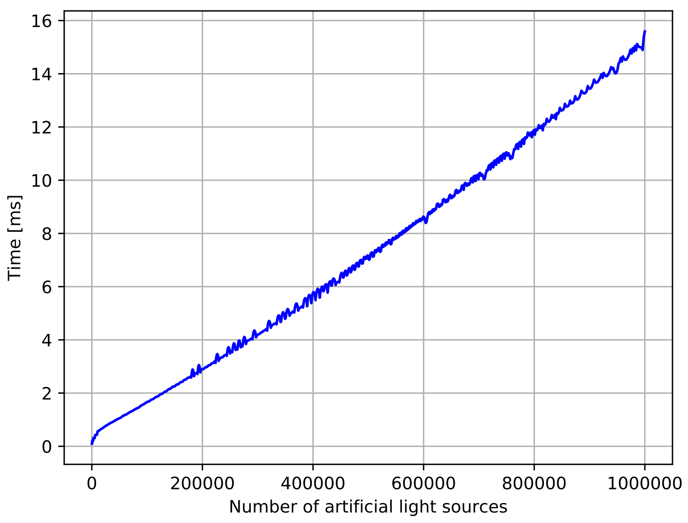


Figure 5.10: Smoothed graph of how the hash table construction step performs (in milliseconds) as a function of the number of lights to be inserted.

scene. Figure 5.10 shows the smoothed results of this simulation using the hash table settings specified in section 5.1. Notice how there is more variance to the measurements after 200, 000 lights are placed. This variability in execution timings is likely because of the linear probing being performed more frequently since the probability of hash collisions increases with the higher light counts (given a fixed number of hash cells and entries). Moreover, we find that when the number of light sources increases, the execution time varies almost linearly. This trend indicates that the hash table construction step, initially proposed by Binder et al. [Binder et al., 2021], scales well with an increasing scene complexity.

Even so, for use cases requiring more than 200, 000 light sources, we see that PSIR might not be viable for real-time contexts, including the fact that the hash table requires more than 2ms to be constructed per frame. As such, the question of whether it is worth implementing PSIR, given its performance costs, is a question that depends on a multitude of factors, such as the target platform, the use case (e.g., whether the rendered images are used in a game or a movie) and the desired scene complexity.

To conclude this discussion, we remark that while the differences between PSIR and ReSTIR are not drastic, PSIR's role in boosting the quality of the initial candidates and its intrinsic abilities to filter distant lights should not be understated. As we have seen from the results in this section, PSIR provides more low-frequency details to the final renders due to its abilities to inherently filter distant and likely noncontributing light sources. These considerations in details are paramount because it makes the job of an accompanying denoiser a lot simpler. For instance, in the selected crops of Figure 5.8, a denoiser would have it difficult to denoise the output from ReSTIR since there is little to no low-frequency detail in the output. On the other hand, PSIR visibly has more detail and structure, making the job of the denoiser more straightforward because the output better conforms with the G-buffer guide. Subsequently, for lower sampling rates, PSIR makes it simpler for the denoiser to output more temporally stable image sequences, as well outputs with less spatial ”boiling”. In use cases where we neither can perform spatial nor temporal reusage—for instance, due to various memory or performance constraints—PSIR should especially help to provide less noisy images, with better and more low-frequency structure, than ReSTIR.

CHAPTER 6
Conclusion

In this thesis, we have taken a closer look at the problem of many-light sampling. We began by investigating the related work in this field and looked at the different methods' limitations. Subsequently, we proposed an extension to the algorithm currently considered state-of-the-art for many-light sampling—namely, ReSTIR.

Our proposed algorithm, PSIR, attempts to extend ReSTIR by facilitating resampling in path space through the use of a hash table. This data structure was constructed by creating a discretized representation of the virtual scene. The discretization was performed by generating hash keys that would quantize the 3D space into dynamically sized voxels. In the hash table build step, the light sample data of the scene lights would be stochastically inserted into the hash table by: First, generating hash keys based on the lights' orientations and world-space positions; and, then, using RIS to, probabilistically, decide if the light sample was to be stored. Subsequently, the filled hash table could be used for lookups in any step of the original ReSTIR algorithm, given that the surface normal and world-space position was available—so that a hash key could be constructed and used for lookup or insertion into the data structure.

The results from our experiments showed that PSIR indeed was able to provide outputs with generally lower error and higher image quality than ReSTIR. However, while PSIR provided improved results, the improvements were generally rather minor over ReSTIR. In particular, we saw that PSIR generally outperformed ReSTIR on surfaces close to the light sources and when the effective sample counts were low. From the results, we discovered this was because the hash table in PSIR was able to pool the lights into local unordered groups, which could then be used as "initial sample sets" to boost the quality of the candidate generation or spatiotemporal resampling stage. As such, distant and weak lights would be pre-filtered during the hash table build stage. We concluded that PSIR mainly extends ReSTIR by boosting the quality of the generated samples and, especially, the initial candidate samples.

Ultimately, the work carried out in this thesis has shown that while we are not quite yet able to simulate artifact-free global illumination at real-time performances, we are getting close. With stochastic algorithms such as ReSTIR, PSIR and, stochastic lightcuts facilitating for sampling thousands to even millions of light sources, at least at interactive framerates, the previous restrictions limiting ourselves to only a couple of hand-picked light sources become lifted. This alleviation should make it easier for content creators to create detailed scenes with realistic and physically-based lighting. Indeed, with the steady improvements we see with rendering hardware, and on research on real-time rendering, it seems that that the dream of movie quality graphics in a fully dynamic virtual scene will more and more likely become closer to reality.

6.1 Limitations and Future Work

One of the main limitations of PSIR was that the spatiotemporal sample reusage still was performed among spatially and temporally adjacent samples *in screen space*. This is because what PSIR does is to provide ReSTIR with localized sample data from a spatial data structure which, in practice, improved the quality of the initial samples. Extending PSIR to perform spatiotemporal sampling in path space may be an exciting avenue for future work, and Bitterli et al. suggest rethinking how samples are combined [Bitterli et al., 2020].

Another limitation of our proposed algorithm was that the time complexity of constructing the hash table was linearly dependent on the number of lights in the scene. Thus, an avenue for future research could be to inspect and revisit the hash construction step and, perhaps, even rethink how the hash table stores its data. To give an example, the hash table could instead store the light sample data based on the current view and stream in, or rebuild itself, whenever the view changes drastically. For instance, instead of iterating through each light in the build step, we could instead iterate through each hash cell and "unproject" the cells back to their original input keys (i.e., the world space positions and surface normals) so that we can find and store the closest and most similar lights in proximity. This approach could, however, require designing a bijective hash key as we would need to go back and forth between hash cell and input keys. Furthermore, since the primary focus in this work was the image quality impact of PSIR, it would also be interesting to see a more extensive analysis on how much of a performance cost PSIR is to ReSTIR and how the algorithm could be optimized further.

We have in our work constructed the hash keys and discretized the virtual scene world based on the world positions and surface normals of objects. This design choice was mainly due to the suggestions provided by Binder et al. on designing the hash keys to be fast to construct but to also be of high quality [Binder et al., 2021]. It would, as such, be interesting to see how other kinds of hash functions would affect the performances of PSIR. For instance, one could also include the incident and exitant directions of a surface point, as well as the surface's material parameters, in order to introduce an even higher level of granularity for the hash cells. However, further investigations may need to be done when employing PSIR with high-dimensional keys due to the curse of dimensionality.

Finally, in this work, we have mainly focused on the impact PSIR has on the direct lighting quality. While direct lighting often is the kind of lighting that impacts the overall illumination the most, PSIR could also, in theory, be applied to indirect light sampling due to the sample data in the hash table being stored in world space. Consequently, it will be interesting to see how PSIR affects the convergence tails of Monte Carlo renderers, such as path tracers, when we apply it to higher-order bounces.

Bibliography

[Appel, 1968] Appel, A. (1968). Some techniques for shading machine renderings of solids. AFIPS '68 (Spring), page 37–45, New York, NY, USA. Association for Computing Machinery.

[Arvo and Kirk, 1990] Arvo, J. and Kirk, D. (1990). Particle transport and image synthesis. *SIGGRAPH Comput. Graph.*, 24(4):63–66.

[Bako et al., 2017] Bako, S., Vogels, T., Mcwilliams, B., Meyer, M., NováK, J., Harvill, A., Sen, P., Derose, T., and Rousselle, F. (2017). Kernel-predicting convolutional networks for denoising monte carlo renderings. *ACM Trans. Graph.*, 36(4).

[Binder et al., 2021] Binder, N., Fricke, S., and Keller, A. (2021). Massively parallel path space filtering.

[Bitterli et al., 2016] Bitterli, B., Rousselle, F., Moon, B., Iglesias-Guitián, J. A., Adler, D., Mitchell, K., Jarosz, W., and Novák, J. (2016). Nonlinearly weighted first-order regression for denoising Monte Carlo renderings. *Computer Graphics Forum (Proceedings of EGSR)*, 35(4):107–117.

[Bitterli et al., 2020] Bitterli, B., Wyman, C., Pharr, M., Shirley, P., Lefohn, A., and Jarosz, W. (2020). Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 39(4).

[Buades et al., 2005] Buades, A., Coll, B., and Morel, J. . (2005). A non-local algorithm for image denoising. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 2, pages 60–65 vol. 2.

[Burley, 2012] Burley, B. (2012). Physically-based shading at disney.

[Chaitanya et al., 2017] Chaitanya, C. R. A., Kaplanyan, A. S., Schied, C., Salvi, M., Lefohn, A., Nowrouzezahrai, D., and Aila, T. (2017). Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder. *ACM Trans. Graph.*, 36(4).

[Chao, 1982] Chao, M. T. (1982). A general purpose unequal probability sampling plan. *Biometrika*, 69(3):653–656.

[Collet, 2016] Collet, Y. (2016). xxhash: Extremely fast hash algorithm. https://github.com/Cyan4973/xxHash.

[Cook et al., 1984] Cook, R. L., Porter, T., and Carpenter, L. (1984). Distributed ray tracing. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, page 137–145, New York, NY, USA. Association for Computing Machinery.

[Cook and Torrance, 1982] Cook, R. L. and Torrance, K. E. (1982). A reflectance model for computer graphics. *ACM Trans. Graph.*, 1(1):7–24.

[Dahlberg et al., 2019] Dahlberg, H., Adler, D., and Newlin, J. (2019). Machine-learning denoising in feature film production. In *ACM SIGGRAPH 2019 Talks*, SIGGRAPH '19, New York, NY, USA. Association for Computing Machinery.

[Gautron, 2020] Gautron, P. (2020). Real-time ray-traced ambient occlusion of complex scenes using spatial hashing. In *ACM SIGGRAPH 2020 Talks*, SIGGRAPH '20, New York, NY, USA. Association for Computing Machinery.

[Hachisuka et al., 2008] Hachisuka, T., Jarosz, W., Weistroffer, R. P., Dale, K., Humphreys, G., Zwicker, M., and Jensen, H. W. (2008). Multidimensional adaptive sampling and reconstruction for ray tracing. In *ACM SIGGRAPH 2008 Papers*, SIGGRAPH '08, New York, NY, USA. Association for Computing Machinery.

[He et al., 2013] He, K., Sun, J., and Tang, X. (2013). Guided image filtering. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(6):1397–1409.

[Heath and Munson, 1996] Heath, M. T. and Munson, E. M. (1996). *Scientific Computing: An Introductory Survey*. McGraw-Hill Higher Education, 2nd edition.

[Jarzynski and Olano, 2020] Jarzynski, M. and Olano, M. (2020). Hash functions for gpu rendering. *Journal of Computer Graphics Techniques (JCGT)*, 9(3):20–38.

[Kajiya, 1986] Kajiya, J. T. (1986). The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, page 143–150, New York, NY, USA. Association for Computing Machinery.

[Kalantari et al., 2015] Kalantari, N. K., Bako, S., and Sen, P. (2015). A machine learning approach for filtering monte carlo noise. *ACM Trans. Graph.*, 34(4).

[Karis, 2014] Karis, B. (2014). High quality temporal anti-aliasing. *Advances in Real-Time Rendering for Games, SIGGRAPH Courses*.

[Keller et al., 2019] Keller, A., Viitanen, T., Barré-Brisebois, C., Schied, C., and McGuire, M. (2019). Are we done with ray tracing? In *ACM SIGGRAPH 2019 Courses*, SIGGRAPH '19, New York, NY, USA. Association for Computing Machinery.

[Koskela et al., 2019] Koskela, M., Immonen, K., Mäkitalo, M., Foi, A., Viitanen, T., Jääskeläinen, P., Kultala, H., and Takala, J. (2019). Blockwise multi-order feature regression for real-time path tracing reconstruction. *ACM Transactions on Graphics (TOG)*, 38(5).

[Kroese et al., 2014] Kroese, D. P., Brereton, T., Taimre, T., and Botev, Z. (2014). Why the monte carlo method is so important today. *Wiley Interdisciplinary Reviews: Computational Statistics*, 6:386–392.

[Li et al., 2012] Li, T.-M., Wu, Y.-T., and Chuang, Y.-Y. (2012). Sure-based optimization for adaptive sampling and reconstruction. *ACM Trans. Graph.*, 31(6).

[Lin and Yuksel, 2020] Lin, D. and Yuksel, C. (2020). Real-time stochastic lightcuts. *Proc. ACM Comput. Graph. Interact. Tech. (Proceedings of I3D 2020)*, 3(1).

[Lumberyard, 2017] Lumberyard, A. (2017). Amazon lumberyard bistro, open research content archive (orca). `http://developer.nvidia.com/orca/amazon-lumberyard-bistro`.

[Mara et al., 2017] Mara, M., McGuire, M., Bitterli, B., and Jarosz, W. (2017). An efficient denoising algorithm for global illumination. In *Proceedings of High Performance Graphics*, New York, NY, USA. ACM.

[Mara et al., 2014] Mara, M., McGuire, M., Nowrouzezahrai, D., and Luebke, D. (2014). Fast global illumination approximations on deep g-buffers. Technical Report NVR-2014-001. NVIDIA Corporation.

[Meister et al., 2021] Meister, D., Ogaki, S., Benthin, C., Doyle, M. J., Guthe, M., and Bittner, J. (2021). A Survey on Bounding Volume Hierarchies for Ray Tracing. *Computer Graphics Forum*.

[Moon et al., 2014] Moon, B., Carr, N., and Yoon, S.-E. (2014). Adaptive rendering based on weighted local regression. *ACM Trans. Graph.*, 33(5).

[Moreau et al., 2019] Moreau, P., Pharr, M., and Clarberg, P. (2019). Dynamic many-light sampling for real-time ray tracing. In *High Performance Graphics*.

[NVIDIA, 2018] NVIDIA (2018). Nvidia turing gpu architecture: Graphics reinvented. https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[NVIDIA et al., 2020] NVIDIA, Vingelmann, P., and Fitzek, F. H. (2020). Cuda, release: 10.2.89. https://developer.nvidia.com/cuda-toolkit.

[Olsson and Assarsson, 2011] Olsson, O. and Assarsson, U. (2011). Tiled shading. *Journal of Graphics, GPU, and Game Tools*, 15(4):235–251.

[Olsson et al., 2012] Olsson, O., Billeter, M., and Assarsson, U. (2012). Clustered deferred and forward shading. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*, EGGH-HPG'12, page 87–96, Goslar, DEU. Eurographics Association.

[O'Neill, 2014] O'Neill, M. E. (2014). Pcg: A family of simple fast space-efficient statistically good algorithms for random number generation. Technical Report HMC-CS-2014-0905, Harvey Mudd College, Claremont, CA.

[Pantaleoni, 2020] Pantaleoni, J. (2020). Online path sampling control with progressive spatio-temporal filtering.

[Parker et al., 2010] Parker, S. G., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., McAllister, D., McGuire, M., Morley, K., Robison, A., and Stich, M. (2010). Optix: A general purpose ray tracing engine. *ACM Trans. Graph.*, 29(4).

[Perlin, 1985] Perlin, K. (1985). An image synthesizer. In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '85, page 287–296, New York, NY, USA. Association for Computing Machinery.

[Pharr et al., 2016] Pharr, M., Jakob, W., and Humphreys, G. (2016). *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition.

[Saito and Takahashi, 1990] Saito, T. and Takahashi, T. (1990). Comprehensible rendering of 3-d shapes. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '90, page 197–206, New York, NY, USA. Association for Computing Machinery.

[Schied et al., 2017] Schied, C., Kaplanyan, A., Wyman, C., Patney, A., Chaitanya, C. R. A., Burgess, J., Liu, S., Dachsbacher, C., Lefohn, A., and Salvi, M. (2017). Spatiotemporal variance-guided filtering: Real-time reconstruction for path-traced global illumination. In *Proceedings of High Performance Graphics*, HPG '17, New York, NY, USA. Association for Computing Machinery.

[Talbot et al., 2005] Talbot, J. F., Cline, D., and Egbert, P. (2005). Importance resampling for global illumination. In *Proceedings of the Sixteenth Eurographics Conference on Rendering Techniques*, EGSR '05, page 139–146, Goslar, DEU. Eurographics Association.

[Tomasi and Manduchi, 1998] Tomasi, C. and Manduchi, R. (1998). Bilateral filtering for gray and color images. In *Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271)*, pages 839–846.

[Veach, 1998] Veach, E. (1998). *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, Stanford, CA, USA. AAI9837162.

[Veach and Guibas, 1995] Veach, E. and Guibas, L. J. (1995). Optimally combining sampling techniques for monte carlo rendering. In *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '95, page 419–428, New York, NY, USA. Association for Computing Machinery.

[Vitter, 1985] Vitter, J. S. (1985). Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57.

[Vogels et al., 2018] Vogels, T., Rousselle, F., Mcwilliams, B., Röthlin, G., Harvill, A., Adler, D., Meyer, M., and Novák, J. (2018). Denoising with kernel prediction and asymmetric loss functions. *ACM Trans. Graph.*, 37(4).

[Walker, 1977] Walker, A. J. (1977). An efficient method for generating discrete random variables with general distributions. *ACM Trans. Math. Softw.*, 3(3):253–256.

[Walter et al., 2005] Walter, B., Fernandez, S., Arbree, A., Bala, K., Donikian, M., and Greenberg, D. P. (2005). Lightcuts: A scalable approach to illumination. *ACM Trans. Graph.*, 24(3):1098–1107.

[Walter et al., 2007] Walter, B., Marschner, S. R., Li, H., and Torrance, K. E. (2007). Microfacet models for refraction through rough surfaces. In *Proceedings of the 18th Eurographics Conference on Rendering Techniques*, EGSR'07, page 195–206, Goslar, DEU. Eurographics Association.

[Whitted, 1979] Whitted, T. (1979). An improved illumination model for shaded display. page 14.

[Winkelmann, 2019] Winkelmann, M. (2019). Zero-day, open research content archive (orca). https://developer.nvidia.com/orca/beeple-zero-day.

[Yuksel, 2019] Yuksel, C. (2019). Stochastic lightcuts. In *High-Performance Graphics (HPG 2019)*, pages 27–32. The Eurographics Association.

[Zwicker et al., 2015] Zwicker, M., Jarosz, W., Lehtinen, J., Moon, B., Ramamoorthi, R., Rousselle, F., Sen, P., Soler, C., and Yoon, S.-E. (2015). Recent advances in adaptive sampling and reconstruction for Monte Carlo rendering. *Computer Graphics Forum (Proceedings of Eurographics - State of the Art Reports)*, 34(2):667–681.