
# Bachelor's Thesis

# Implementation, Optimization, and Evaluation of real-time Path Tracing in DirectX

for the award of the academic degree

## Bachelor of Science (B. Sc.)

in the study program of

## Informatik


| | |
|---|---|
| Submitted by | Malte Lanz |
| Examiner | |
| Submission Date | 17.04.2025 |

# Declaration under Oath

I hereby declare in lieu of an oath that I have written this bachelor's thesis independently and without the use of any sources other than those indicated. Any ideas taken directly or indirectly from external sources are identified as such.

The thesis has not yet been submitted in the same or a similar form to any other examination authority and has not yet been published.

Ulm, 17.04.2025

_____

Malte Lanz

# Abstract

Real-time path tracing has become increasingly feasible on modern consumer graphics hardware thanks to the synergy of hardware-accelerated ray tracing and optimized light-transport algorithms. This work investigates Spatiotemporal Reservoir Resampling (ReSTIR), an approach that integrates local sampling with spatiotemporal reuse to significantly increase the effective sample pool in real-time settings. By merging samples within and across frames via importance sampling and reservoir-based methods, ReSTIR reduces variance more efficiently than conventional path tracers that rely solely on multiple importance sampling. This thesis presents an implementation of ReSTIR in a DirectX 12 DirectX Ray Tracing (DXR) ray-tracing renderer with key algorithmic components, such as Bidirectional Scattering Distribution Function (BSDF) and area-light importance sampling, next-event estimation, multiple importance sampling, and reservoir sampling. It also researches various optimizations that can be applied to the implementation to mitigate cache pressure, reduce spatiotemporal correlation artifacts, and maintain warp occupancy. An empirical evaluation of two test scenes demonstrates that the implementation converges to low-noise images in fewer frames than a comparable MIS-based path tracer under the same time budget.

### Keywords

Path Tracing, Reservoir Sampling, Spatiotemporal Reuse

# Table of Contents

# List of Figures

# List of Tables

# List of Listings

# List of Abbreviation

| | |
|---|---|
| **CGI** | Computer Generated Imagery |
| **GPU** | Graphics Processing Unit |
| **ReSTIR** | Spatiotemporal Reservoir Resampling |
| **RIS** | Resampled Importance Sampling |
| **MIS** | Multiple Importance Sampling |
| **DXR** | DirectX Ray Tracing |
| **PBR** | Physically Based Rendering |
| **BSDF** | Bidirectional Scattering Distribution Function |
| **GGX** | Trowbridge-Reitz Distribution Function |
| **LUT** | Look-Up-Table |
| **NEE** | Next Event Estimation |
| **CDF** | Cumulative Distribution Function |
| **PDF** | Probability Distribution Function |
| **RRS** | Russian-Roulette Sampling |
| **GI** | Globial/Indirect Illumination |
| **DI** | Direct Illumination |
| **TLAS** | Top-Level Acceleration Structure |
| **BLAS** | Bottom-Level Acceleration Structure |
| **BVH** | Bounding Volume Hierarchy |
| **SM** | Shader Multiprocessor |
| **RGB** | Red-Green-Blue colorspace |
| **RMSE** | Root Mean Squared Error |

# 1 Introduction

## 1.1 Motivation: Real-time path tracing

Interactive real-time computer graphics and visualizations have been defined by the rasterization technique since its first appearance in the 1960s [5]. It allows the efficient display of triangle-based 3D objects in a 3D space projected onto the 2-dimensional image plane. While being effective for scenes with a limited number of triangles, rasterization alone struggles to reproduce the complex, photorealistic lighting effects that arise from global illumination, reflections, and refractions. Additionally, due to the linear scaling with triangle count, a basic rasterization renderer is unable to render highly complex environments typically found in movie productions in real-time and limits the triangle fidelity of interactive applications like video games and visualizations. In the 1980s, a different approach to shading was developed, tracing rays to map scene positions to pixel space [6].

In its most simple form, ray tracing is less efficient for rendering tringles than rasterization, as each pixel requires testing a ray - extending from the camera into the scene — against every triangle. To address this, acceleration structures that divide a scene into hierarchical volumetric segments are used, significantly reducing intersection tests and allowing the cost per pixel to scale logarithmically. This way, modern ray tracing implementations used in production renderers surpass rasterization performance in complex scenes to a point where memory becomes the limiting factor. Furthermore, recent advances in consumer-grade hardware have made real-time ray tracing increasingly feasible for video games and other interactive applications.

As defined by the rendering equation [7], ray tracing can also be used to simulate more complex true-to-life lighting effects typically referred to as path tracing. Path traced rendering can produce global illumination effects, such as indirect lighting, producing photorealistic images. This capability has made path tracing a cornerstone of modern production renderers in Computer Generated Imagery (CGI). However, due to its high computational demands, it has historically been impractical for widespread deployment on consumer-grade hardware, limiting its use in interactive applications with strict performance constraints.

This challenge can be addressed in two complementary ways: enhancing hardware with dedicated ray-tracing accelerators and optimizing software algorithms. As we will see, a combination of both approaches typically yields the best results. The focus of this work lies in improving path-tracing algorithms for real-time rendering, using dedicated hardware acceleration provided by the state-of-the-art graphics API DirectX 12.

## 1.2 Objective and Structure of the thesis

This thesis explores how targeted software optimizations, namely NVIDIA-developed ReSTIR, can enhance the quality and performance of real-time path tracing on modern graphics processing units (Graphics Processing Unit (GPU)s).

In chapter 2, a brief overview of the concept of using Monte Carlo integration to evaluate the rendering equation for each pixel as a path integral is given. As path tracing aims to simulate real lighting conditions as correctly as possible, a material model based on physical surface properties, that resemble those of true-to-life material models, is presented, to produce a realistic-looking result.

A key focus of this thesis is the implementation and evaluation of “ReSTIR,” an NVIDIA-developed technique for direct and indirect illumination in real-time path tracing [8, 9]. The thesis presents the mathematical foundations and core implementation concepts for ReSTIR in chapter 2, along with two crucial statistical methods on which it depends: RIS [10] and Multiple Importance Sampling (MIS)[11]. In chapter 4, implementation details are presented, along with optimizations to improve the performance of ReSTIR.

Since the implementation relies on the DirectX 12 application programming interface (API) and leverages hardware-accelerated ray tracing support, a concise overview of DirectX’s ray-tracing pipeline (DXR) [3] is provided in chapter 3, focusing on elements relevant to the techniques used.

Finally, in chapter 5 and 6 the complete system is benchmarked to evaluate its impact on image quality and performance in real-time rendering scenarios and to assess its performance gains critically.

# 2 Background and Related Work

This chapter gives a theoretical overview of a conventional path tracer and the foundation for improved sampling used in the ReSTIR algorithm. Typically, there is term ambiguity between ray tracing and path tracing. Essentially, ray tracing is defined as a term for every rendering technique that performs world-space ray-scene intersection evaluations.

## 2.1 Foundational Techniques in Path Tracing

Path tracing is based on the idea that lighting at each pixel on the screen can be approximated by taking light samples from the scene. Formaly, this is expressed as a path integral that solves the rendering equation [7] for a given surface point in the scene.

### 2.1.1 The Rendering Equation and Path Integral

The rendering equation provides a unifying framework for understanding how light is transferred between surfaces in a scene and is based on the physical law of conservation of energy [7]. This means that in a physically-based setting, there can never be more radiance reaching the observer than is emitted by light sources in the scene. The formula states that the radiance $L_o(\mathbf{x}, \omega_o)$ leaving a point $\mathbf{x}$ in direction $\omega_o$ is composed of any emission at $\mathbf{x}$ plus the reflected (or scattered) incident radiance coming from all directions $\omega_i$ in the hemisphere above $\mathbf{x}$. Formally, it is expressed as:

$$L_o(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + \int_\Omega f_r\big(\mathbf{x}, \omega_i, \omega_o\big)\, L_i\big(\mathbf{x}, \omega_i\big)\, \big(\omega_i \cdot \mathbf{n}\big)\, d\omega_i, \tag{1}$$

where:

- $L_o(\mathbf{x}, \omega_o)$ is the outgoing radiance at point $\mathbf{x}$ in the direction $\omega_o$.
- $L_e(\mathbf{x}, \omega_o)$ is the radiance emitted by the surface at $\mathbf{x}$.
- $f_r(\mathbf{x}, \omega_i, \omega_o)$ is the Bidirectional Scattering Distribution Function (BSDF), which models how incoming light from $\omega_i$ is scattered toward $\omega_o$.
- $L_i(\mathbf{x}, \omega_i)$ is the incident radiance at $\mathbf{x}$ from direction $\omega_i$.
- $-\omega_i \cdot \mathbf{n}$ is the cosine of the angle between $\omega_i$ and the surface normal $\mathbf{n}$.
- $\Omega$ represents the hemisphere of all incoming directions above the surface.

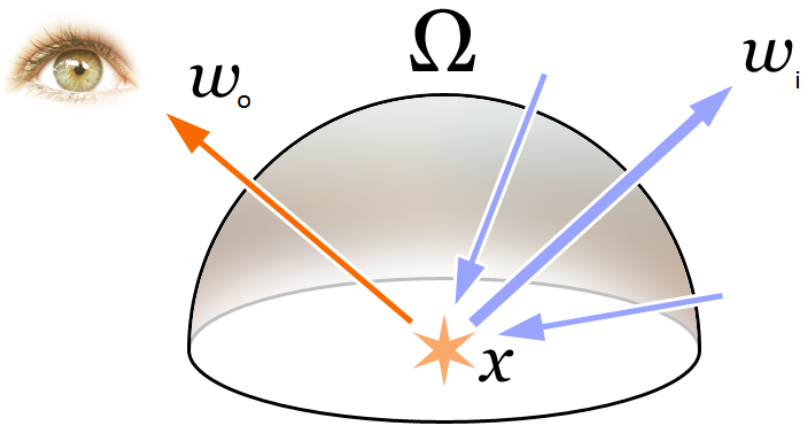


Figure 1: Rendering Equation sketch, from [1]

From here on, the vector directions are defined as: the outgoing direction $\omega_o$ points from the surface toward the observer (or camera), while the incoming direction $\omega_i$ points from the scene's light sources — or the environment — toward the surface point $\mathbf{x}$.

While it is possible to solve it analytically, for complex 3D scenes, the rendering equation can get exceedingly difficult to solve [12]. The approach applied in path tracing tackles the problem using numerical integration, namely the Monte Carlo Integration method [7].

### 2.1.2 Monte Carlo Integration

The Monte Carlo integration is a numerical integration method that approximates the value of an integral by summing up samples taken from the integration domain.

To approximate an integral of the form

$$I = \int_{\mathcal{D}} g(\mathbf{x})\, d\mathbf{x}, \tag{2}$$

Monte Carlo methods draw $N$ random samples $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ from the domain $\mathcal{D}$ according to a probability density function (PDF) $p(\mathbf{x})$. The integral is then approximated by

$$\hat{I} = \frac{1}{N} \sum_{k=1}^{N} \frac{g(\mathbf{x}_k)}{p(\mathbf{x}_k)}. \tag{3}$$

**Application in Path Tracing** In the context of path tracing, the technique transforms the high-dimensional integral underlying the rendering equation into a summation over randomly chosen samples from the domain. Each pixel's color is computed by estimating the solution to the rendering equation, which can be viewed as an integral over all possible light transport paths. Monte Carlo integration is used to sample directions from $\Omega$ and bounce rays through the

scene, accumulating the contributions from both direct and indirect lighting. As more samples are taken, the estimated radiance converges to the physically correct solution described by the rendering equation.

**Convergence** Monte Carlo methods provide an unbiased estimator for integrals by relying on random sampling and the notion of expected value. Considering the simplified integral form of the outgoing radiance for Red-Green-Blue colorspace (RGB) color spaces [13]:

$$L_o(\mathbf{x}, \omega_o) \;=\; L_e(\mathbf{x}, \omega_o) + \int_\Omega f_r(\mathbf{x}, \omega_i, \omega_o)\, L_i(\mathbf{x}, \omega_i)\, \big(-\omega_i \cdot \mathbf{n}\big)\, d\omega_i, \tag{4}$$

where the domain of integration $\Omega$ represents all relevant incoming directions. Define an integrand

$$g(\omega_i) \;=\; f_r(\mathbf{x}, \omega_i, \omega_o)\, L_i(\mathbf{x}, \omega_i)\, \big(-\omega_i \cdot \mathbf{n}\big) \quad \text{and} \quad I \;=\; \int_\Omega g(\omega_i)\, d\omega_i. \tag{5}$$

Evaluating $I$ numerically, directions $\{\omega_i^1, \omega_i^2, \ldots, \omega_i^N\}$ can be sampled from $\Omega$ according to some probability density function (PDF) $p(\omega_i)$. A Monte Carlo estimator for $I$ is then

$$\hat{I}_N \;=\; \frac{1}{N} \sum_{k=1}^{N} \frac{g(\omega_i^k)}{p(\omega_i^k)}. \tag{6}$$

Assuming the estimator is unbiased, its expected value should equal the true integral $I$.

Taking the expectation of $\hat{I}_N$,

$$\begin{aligned}
\mathbb{E}\big[\hat{I}_N\big] &= \mathbb{E}\Big[\frac{1}{N} \sum_{k=1}^{N} \frac{g(\omega_i^k)}{p(\omega_i^k)}\Big] \\
&= \frac{1}{N} \sum_{k=1}^{N} \mathbb{E}\Big[\frac{g(\omega_i^k)}{p(\omega_i^k)}\Big] \\
&= \frac{1}{N} \sum_{k=1}^{N} \int_\Omega \frac{g(\omega_i)}{p(\omega_i)}\, p(\omega_i)\, d\omega_i \\
&= \frac{1}{N} \sum_{k=1}^{N} \int_\Omega g(\omega_i)\, d\omega_i \\
&= \frac{1}{N} \sum_{k=1}^{N} I \\
&= I.
\end{aligned}$$

Since $\mathbb{E}[\hat{I}_N] = I$, the estimator $\hat{I}_N$ is *unbiased*. Furthermore, by the law of large numbers, $\hat{I}_N \to I$ almost surely as $N \to \infty$. Hence, the Monte Carlo method yields a correct estimate of the integral as the number of samples increases.

**Variance and Convergence Rate** In order to improve the quality of Monte Carlo integration, it is often beneficial to reduce the variance of the estimator. Since the samples in Monte Carlo integration are assumed to be independent, we can derive the variance of the estimator $\hat{I}_N$ as follows [12, 13]:

$$\sigma^2\big[\hat{I}_N\big] \;=\; \sigma^2\Big[\frac{1}{N}\sum_{k=1}^{N}\frac{g(\omega_i^k)}{p(\omega_i^k)}\Big]. \tag{7}$$

As each term in the sum has the same variance and the samples $\{\omega_i^k\}$ are i.i.d., this can be rewritten as

$$\sigma^2\big[\hat{I}_N\big] = \frac{1}{N^2}\sum_{k=1}^{N}\sigma^2\Big[\frac{g(\omega_i^k)}{p(\omega_i^k)}\Big] = \frac{1}{N^2}\times N\times\sigma^2\big[Y\big] = \frac{1}{N}\,\sigma^2\big[Y\big],$$

where $Y = \frac{g(\omega_i^k)}{p(\omega_i^k)}$. Taking the square root of both sides yields

$$\sigma\big[\hat{I}_N\big] \;=\; \frac{1}{\sqrt{N}}\,\sigma\big[Y\big]. \tag{8}$$

Hence, the standard deviation (or root-mean-square error, Root Mean Squared Error (RMSE)) of the Monte Carlo estimator decreases proportionally to $1/\sqrt{N}$. Although this convergence rate can be slow, especially in real-time path tracing applications, it can be improved through various variance reduction techniques, such as importance sampling, and Multiple Importance Sampling, which aim to lower $\sigma^2\big[Y\big]$.

**Bias in Path Tracing** Although Monte Carlo integration is theoretically unbiased, practical implementations of path tracers do not always preserve this property. Certain types of numerical or algorithmic errors can introduce unintended bias; however, in some cases, a controlled introduction of bias is deliberately employed to improve computational efficiency. This trade-off can lead to significantly faster render times at the cost of physical accuracy.

Bias in path tracing commonly manifests as spatial inconsistencies in image brightness, often indicating a violation of energy conservation and, by extension, the rendering equation [14, 11]. As a result, offline production renderers — which prioritize physical correctness — typically avoid biased techniques. A good development strategy is to first establish a fully unbiased

implementation of optimizations where possible, followed by targeted optimizations where acceptable. Given the complexity of optimized path tracer implementations, identifying bias is non-trivial. The most reliable approach involves comparing the output of a potentially biased renderer against a reference image produced by a simpler, slower, and probably unbiased implementation.

### 2.1.3 Materials in Physically Based Rendering

**Materials** Physically Based Rendering (PBR) materials aim to capture real-world light interaction with surfaces by adhering to physical laws such as energy conservation and reciprocity [12, 7]. This involves defining a Bidirectional Scattering Distribution Function (BSDF) that models how incident radiance is reflected and/or transmitted by a surface. In the rendering equation, the BSDF is written as $f_r(\mathbf{x}, \omega_i, \omega_o)$. The better a surface model approximates the physical properties of real-world materials [15], the more photorealism the output image can achieve.

Typically, a surface in PBR is constructed from several BSDF "lobes", evaluations of different material models that mimic the layered structure of real-world materials. An easy example is car paint, which has a clear coat layer that reflects mirror-like, and a diffuse layer that gives the car paint its color [12].

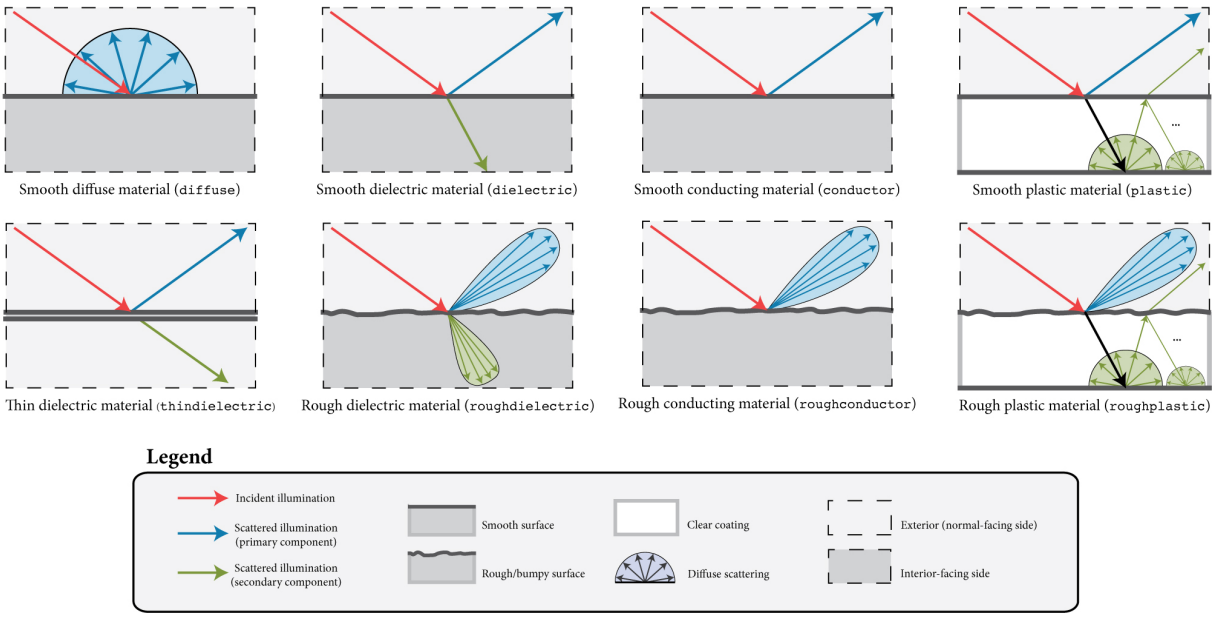


Figure 2: Example of the Mitsuba3 renderers material system [2]. This work's implementation supports diffuse, smooth, and rough conducting and dielectric materials but no transmission.

While material layers can get very complex when light interaction between layers is simulated, real-time rendering usually only uses a combination of a specular and diffuse lobe and sometimes also a dedicated lobe for refraction.

**Lambertian Reflection** A commonly used diffuse reflection model in PBR is the Lambertian model, which assumes that light is scattered uniformly in all directions [12]. This yields a simple BRDF:

$$f_r(\mathbf{x}, \omega_i, \omega_o) \;=\; \frac{\rho}{\pi}, \tag{9}$$

where $\rho$ is the diffuse reflectance. Despite being a strong simplification, the Lambertian model remains popular in both offline and real-time renderers for its very efficient evaluation. Alternatives, such as the Oren-Nayar model [16], provide more accurate results for rough surfaces at the expense of increased computational cost.

**Microfacet Models** Specular reflection in Physically Based Rendering (PBR) is often modeled using microfacet theory, which approximates a surface as a collection of tiny, perfectly reflective facets oriented according to a statistical Normal Distribution Function (NDF) $D(\mathbf{h})$ [17]. A generic microfacet BRDF can be expressed as:

$$f_r(\mathbf{x}, \omega_i, \omega_o) \;=\; \frac{F(\omega_i, \mathbf{h})\; G(\omega_i, \omega_o, \mathbf{h})\; D(\mathbf{h})}{4\,|\omega_i \cdot \mathbf{n}|\;|\omega_o \cdot \mathbf{n}|}, \tag{10}$$

where:

- $\mathbf{h}$ is the half-vector, $\mathbf{h} = \frac{\omega_i + \omega_o}{\|\omega_i + \omega_o\|}$.
- $F(\omega_i, \mathbf{h})$ is the Fresnel term, describing the portion of light reflected versus transmitted at a microfacet boundary.
- $G(\omega_i, \omega_o, \mathbf{h})$ is the geometric term, accounting for masking and shadowing among the microfacets.
- $D(\mathbf{h})$ is the normal distribution function that defines how microfacet orientations are statistically distributed.

This formulation provides a unified approach for modeling both mirror-like and rough reflections based on physically meaningful parameters such as surface roughness and metallicness [18].

**GGX Distribution** Among the various choices for $D(\mathbf{h})$, the Trowbridge-Reitz Distribution Function (GGX) (also referred to as Trowbridge-Reitz) distribution has become the de facto standard in modern PBR pipelines [19]. It is favored for its ability to better reproduce long reflection "tails" observed on rough, metallic surfaces. The GGX distribution is typically written as:

$$D_{\mathrm{GGX}}(\mathbf{h}) = \frac{\alpha^2}{\pi \left[(\mathbf{n} \cdot \mathbf{h})^2(\alpha^2 - 1) + 1\right]^2}, \tag{11}$$

where $\alpha$ is a roughness parameter controlling the slope distribution of the microfacets. GGX tends to provide visually plausible highlights.

Although microfacet models like GGX are physically based, they sometimes fail to conserve energy accurately when multiple internal reflections occur among the microfacets [20]. This leads to substantial darkening of the material, especially at grazing angles. As this is typically undesired, a reasonable approach for real-time rendering is to precompute a compensation term that provides the microfacet model with approximate energy conservation [20].

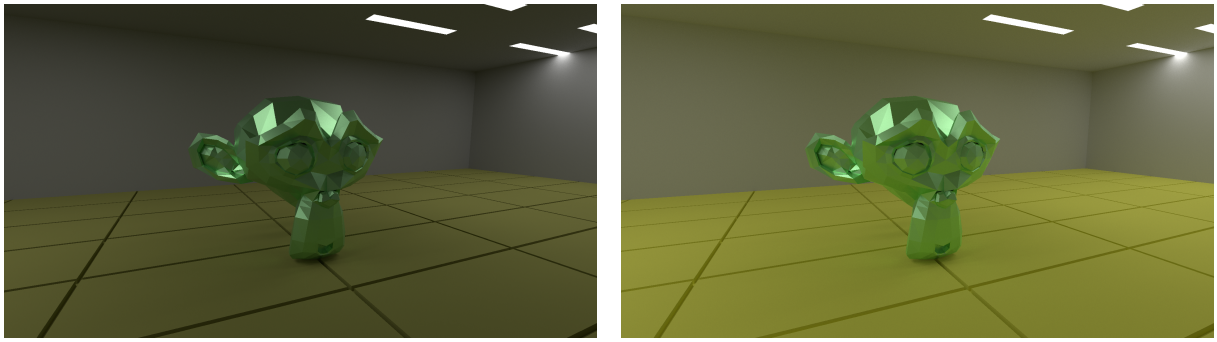

Figure 3: Left: native GGX without energy loss compensation term shows significant darkening. Right: the compensation term results in a more true-to-life image.

The approach described by Turquin in [20] can be split into two steps:

- A Look Up Table (Look-Up-Table (LUT)) is created that maps incoming light angles to a weight and precomputes the weights. This is done by performing a full incoming light integration for the material at a range of angle steps ranging from 0° to 90° of incoming light. For each step, the ratio of energy lost by evaluating the material is calculated and the weight is set to account for the energy loss.

- When evaluating a material in path tracing, adjust the calculated RGB value by multiplying it using the weight from the LUT:

$$E_{\text{ss}} = \text{LUT}(\text{mat}, \mathbf{n} \cdot \omega_o)$$

$$k_{\text{ms}}(\omega_o) = \frac{1 - E_{\text{ss}}}{E_{\text{ss}}}$$

$$f_{\text{ms}}(\omega_o) = f_r(\omega_o) \left[1 + K_0 \cdot k_{\text{ms}}(\omega_o)\right]$$

  where:

  - $E_{\text{ss}}$ is the single-scatter energy term (from `LUT`),
  - $k_{\text{ms}}(\omega_o)$ is the multi-scatter correction factor,
  - $f_{\text{ms}}(\omega_o)$ is the multi-scatter GGX specular term,
  - $f_r(\omega_o)$ is the original GGX BRDF specular term,
  - $K_0$ is a material-specific specular parameter,
  - $\mathbf{n} \cdot \omega_o$ represents the angle between the surface normal and the outgoing (view) direction and serves as the key when indexing into the LUT.

### 2.1.4 Implementation Concepts for Path Tracing

Basic path tracing can be performed in essentially two ways [11]:

**Forward Path Tracing** In forward path tracing, rays are emitted from the light sources and traced through the scene until they possibly reach the camera. This approach can directly handle caustics and complex light transport originating from intense or focused light sources; however, it can become inefficient when few of these light-emitted rays ultimately intersect the camera.

**Backward Path Tracing** In backward path tracing, rays are emitted from the camera (or eye) into the scene. This method often matches typical viewing applications better because every traced path directly contributes to the final image. It is relatively easy to implement and commonly used in production renderers but might have issues tracking complex light phenomena, like caustics caused by a lens.

For a real-time path tracer, the backward approach is usually the best choice as it is much more efficient for lowering lighting variance and therefore reducing noise. Using the following

optimization techniques, really good results can also be achieved for small and intense light sources, which a naive backward path tracer might struggle with.

## 2.2 Optimization techniques for path tracing

A typical core concept for tracking the path contribution in a backward path tracer is to track both the path throughput and path PDF as one bounces from surface to surface until finally hitting a light source or terminating the path. The throughput and PDF are updated within a loop by repeatedly sampling the BSDF at each intersection, applying geometry terms (such as the cosine factor and any visibility checks), and accumulating the corresponding PDF for the chosen directions.

Once a light-emitting surface is reached, the contribution of that path can be estimated using:

$$\text{contribution} = \frac{\text{throughput} \times \text{emission}}{\text{PDF}}.$$

where:

1. **Path Throughput:** The product of BSDF values and geometry terms (e.g., the cosine term for the projected area and any visibility factors) across each bounce.
2. **Path PDF:** The cumulative probability density of choosing the specific sequence of directions and intersection points.
3. **Emission:** The emitted radiance from light-emitting materials or light sources reached by the path.

As defined by the Monte Carlo estimator, averaging path contributions will yield an approximation of the rendering equation.

### 2.2.1 BSDF Importance Sampling

At each scene intersection position acquired by tracing a ray (also referred to as path vertex), a path tracer creates samples lying on a hemisphere around the sample position on a surface to sample the next path vertex by tracing a ray in the sampled direction. Generally, these samples are chosen randomly [11]. The simplest way to create a sample is to draw it from the uniform hemisphere sample distribution.

The Probability density function (PDF) is

$$p(\omega_i) = \frac{1}{2\pi} \tag{12}$$

for that distribution, where $\omega_i$ is the sampled direction on the hemisphere. A PDF is a weight that describes how likely a sample $\omega_i$ would have been drawn from the distribution. It is used to weight the sample contribution in the Monte Carlo estimator [12].

In practice, few materials reflect the incoming radiance in all directions equally. Especially specular materials often reflect mostly around a reflection vector based on the outgoing direction. Selecting a random direction in the hemisphere leads to many samples with negligible contributions and therefore high variance. Importance sampling of the material lobe improves that by only selecting samples proportionally to the BSDF lobe, reducing the number of samples with zero or very low throughput and thus lowering variance.

**Lambertian diffuse importance sampling** Due to the wide scattering lobe of a Lambertian diffuse material, importance sampling yields a less noticeable variance reduction compared to very sharp specular lobes. However, it still helps to sample in proportion to the cosine of the outgoing angle (often called cosine-weighted hemisphere sampling) [12].

A Lambertian BRDF in local coordinates (where $n = (0, 0, 1)$) is given by:

$$f(\omega_o, \omega_i) = \frac{\rho}{\pi},$$

where $\omega_o$ is the outgoing direction and $\omega_i$ is the incoming direction. A suitable PDF for importance sampling this BRDF is:

$$p(\omega_i) = \frac{\cos(\theta_i)}{\pi},$$

where $\theta_i$ is the angle between $\omega_i$ and the surface normal $n$.

**Specular materials (GGX) importance sampling** For specular reflection, the GGX microfacet BRDF can be written as:

$$f(\omega_o, \omega_i) = \frac{F(\omega_o \cdot h)\, D(h)\, G(\omega_o, \omega_i)}{4\,(\omega_o \cdot n)\,(\omega_i \cdot n)},$$

For isotropic GGX,

$$D_{\mathrm{GGX}}(h) = \frac{\alpha^2}{\pi \left[1 + (\alpha^2 - 1)\,(n \cdot h)^2\right]^2},$$

where $\alpha$ is the roughness parameter [12].

**Half-vector sampling**: First, the half-vector $h$ is sampled according to $D(h)\,(n \cdot h)$. Then the outgoing direction $\omega_o$ is reflected about $h$ to get $\omega_i$. The PDF for sampling $\omega_i$ in this approach becomes

$$p(\omega_i) = \frac{D(h)\,(n \cdot h)}{4\,|\omega_o \cdot h|}.$$

For GGX, sampling the half-vector $h$ proceeds as:

$$\theta_h = \arctan\Big(\alpha\sqrt{\tfrac{\xi_1}{1-\xi_1}}\Big), \quad \phi_h = 2\pi\,\xi_2,$$

where $\xi_1, \xi_2 \in [0, 1)$ are random samples. Convert $(\theta_h, \phi_h)$ to the 3D vector $h$, and finally compute $\omega_i = -\omega_o + 2(\omega_o \cdot h)\,h$ [12].

### 2.2.2 Next-Event Estimation

One major drawback of relying solely on BSDF importance sampling is the uncertainty of how many bounces are needed to hit a light source. This can lead to excessively long paths in scenes dominated by diffuse surfaces, as well as high variance. Next Event Estimation (Next Event Estimation (NEE)), sometimes referred to as Direct Light Sampling, addresses these issues by explicitly sampling directions toward light sources at each bounce. In essence, the path integral can be split into a sum of “local” integrals, each describing the incident radiance at a given path vertex, which makes it possible to reuse previously BSDF-sampled and traced path segments [21].

By tracing one path of $n$ bounces but also connecting each bounce directly to a light source, a path tree is constructed containing $n$ additional light-sample paths. Although each extra connection requires an additional shadow ray, the overall cost only increases by roughly a factor of two compared to a standard path tracer (since each bounce now issues two rays: one BSDF ray plus one shadow ray to the light). In return, the estimator gains multiple additional samples of light transport per path.

**Sampling of Light Sources** A straightforward way to apply NEE to triangular area lights is to randomly select a triangle and then pick a point on it, connecting the current path vertex to that point. A simple uniform sampling of light triangles is:

$$p(i) = \frac{1}{N}, \quad p(\mathbf{x} \mid i) = \frac{1}{\text{area}_i}, \tag{13}$$

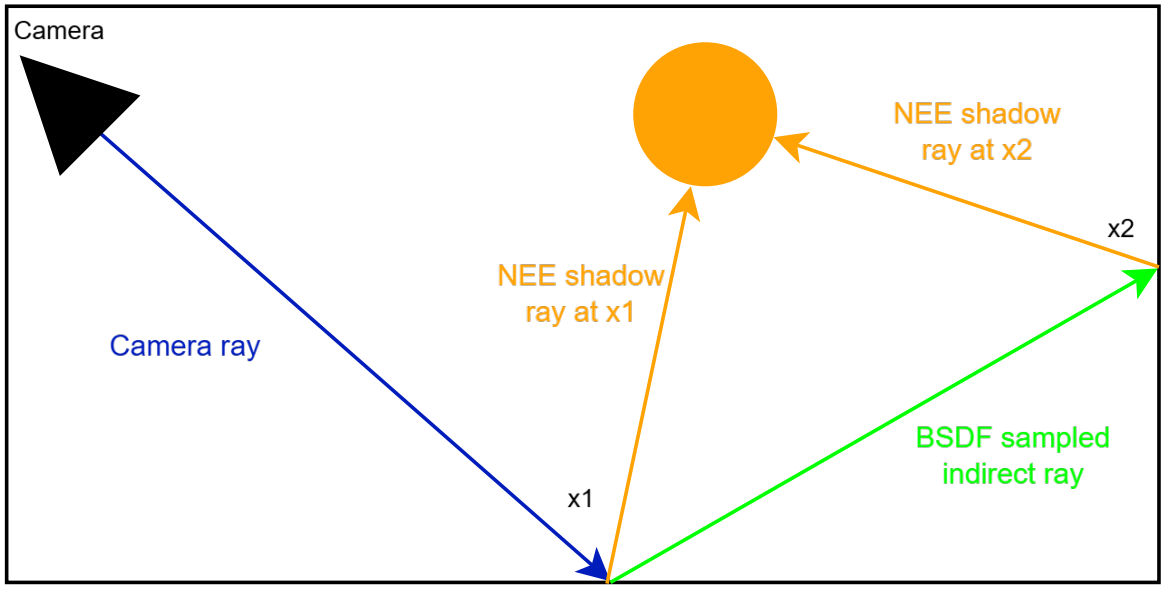


Figure 4: NEE is performed at any bounce. In the case of this sketch, one indirect path vertex is sampled using BSDF sampling which results in a total of two NEE samples.

but this may produce biased results when triangles differ in area or luminance. Weighting each triangle by $W_i = \text{area}_i \times \text{luminance}_i$ leads to importance sampling [21]:

$$p(i) = \frac{W_i}{\sum_j W_j}, \qquad p(\mathbf{x} \mid i) = \frac{1}{\text{area}_i}. \tag{14}$$

A cumulative distribution function (CDF) is preconstructed on the CPU from these weights. Conveniently, a binary search (in $O(\log n)$) can be used on the GPU side to pick a triangle. After selecting the triangle, a point on it is sampled uniformly.

```
function SampleLightTriangle(seed):
    randomValue = RandomFloat(seed)
    left        = 0
    right       = triangleList.length - 1
    selectedIndex = 0
    while (left <= right):
        mid     = left + (right - left)
        midCdf  = triangleList[mid].cdf
        if (randomValue < midCdf):
            selectedIndex = mid
            right         = mid - 1
        else:
            left = mid + 1
    sampleLight = triangleList[selectedIndex]
    return sampleLight
```

This approach effectively balances the sampling effort according to each triangle's contribution (area$_i$ $\times$ luminance$_i$). Consequently, large and/or bright lights are sampled more frequently, improving the rate of convergences.

### 2.2.3 Multiple Importance Sampling

While Next Event Estimation (NEE) can significantly improve convergence in predominantly diffuse scenes, it imposes a constraint for unbiasedness: one cannot simultaneously use BSDF sampling and NEE at the same bounce without careful weighting. Relying solely on NEE yields high variance in scenes with strong specular components (and vice versa). This motivates the use of *Multiple Importance Sampling* (MIS) [11].

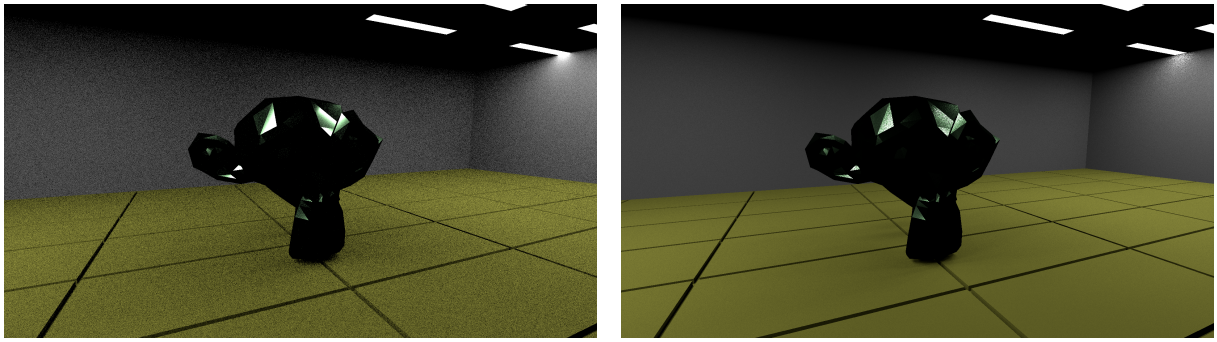

Figure 5: Left: direct lighting using only BSDF sampling; Right: direct lighting using only NEE. While specular materials have lower variance using BSDF sampling, diffuse materials experience lower variance using NEE. This also applies to path tracing at each bounce.

When generating multiple samples at each bounce, each sample must be appropriately weighted to avoid overestimating the integral. A naive approach is to assign equal weights, $w = 1/N$, where $N$ is the number of samples. Although this method is unbiased, it does not fully leverage the different sampling strategies to reduce variance.

Multiple Importance Sampling [11] introduces heuristics to combine samples from different proposal distributions in a statistically optimal way. Let $p_i(\mathbf{x})$ be the probability density function (PDF) for the $i$-th sampling strategy, and let $n_i$ be the number of samples drawn from that strategy. If $\mathbf{x}$ is a sample (e.g., a direction or a point) generated by strategy $i$, then MIS assigns a weight $\omega_i(\mathbf{x})$ such that the final estimator remains unbiased but reduces variance.

**Balance Heuristic** A commonly used heuristic is the *balance heuristic*, defined as:

$$\omega_i(\mathbf{x}) \;=\; \frac{n_i\, p_i(\mathbf{x})}{\sum_k n_k\, p_k(\mathbf{x})}. \tag{15}$$

In the special case where each strategy draws only one sample (i.e., $n_i = 1$), this simplifies to:

$$\omega_i(\mathbf{x}) \;=\; \frac{p_i(\mathbf{x})}{\sum_k p_k(\mathbf{x})}. \tag{16}$$

**Power Heuristic** The *power heuristic* takes this idea further by raising each term to an exponent $\beta > 0$ to favor the better-suited proposal distribution more strongly:

$$\omega_i(\mathbf{x}) \;=\; \frac{\big[n_i\, p_i(\mathbf{x})\big]^{\beta}}{\sum_k \big[n_k\, p_k(\mathbf{x})\big]^{\beta}}. \tag{17}$$

In practice, the balance heuristic often provides a good balance, though the optimal choice can be scene-dependent. Under either heuristic, computing the MIS weight $\omega_i(\mathbf{x})$ requires that each strategy's PDF be evaluated for *all* samples, even those drawn by other strategies. For example, when combining NEE with BSDF sampling, the path sampled by the BSDF must also be evaluated under the NEE PDF, and vice versa, to compute the proper MIS weight.

**Nested MIS for Multiple Lobes** MIS can be applied in a nested manner. When a material contains multiple lobes (e.g., diffuse and specular), MIS can be used to combine these lobes into a single sampled direction [11]. Subsequently, that direction may be combined with other strategies (such as NEE) through another level of MIS. For instance, if a material has $m$ lobes with PDFs $p_{l_1}(\mathbf{x}), \ldots, p_{l_m}(\mathbf{x})$, a direction from one of the lobes (say $l_i$) is sampled and then assigned a lobe-level MIS weight:

$$\omega_{l_i}(\mathbf{x}) \;=\; \frac{p_{l_i}(\mathbf{x})}{\sum_{j=1}^{m} p_{l_j}(\mathbf{x})}. \tag{18}$$

Then, having a chosen direction $\mathbf{x}$, we combine it with NEE by employing the broader MIS weight involving both BSDF-based sampling and NEE-based sampling:

$$\omega_{\text{combined}}(\mathbf{x}) \;=\; \frac{[n_i\, p_i(\mathbf{x})]^{\beta}}{\sum_k [n_k\, p_k(\mathbf{x})]^{\beta}}, \tag{19}$$

where each $p_k$ is now the total PDF for a given technique (e.g., the combined lobe PDF for BSDF sampling or the direct light sampling PDF for NEE).

When combining two BSDF lobes (indexed by $i = 0, 1$) using lobe probabilities $\alpha_0$ and $\alpha_1$, the resulting PDF can be written as:

$$p_{\text{combined}}(\omega_o, \omega_i) \;=\; \big[\alpha_0\, p_0(\omega_o, \omega_i) \;+\; \alpha_1\, p_1(\omega_o, \omega_i)\big] \tag{20}$$

Here, $p_i$ is the individual BSDF Probability Distribution Function (PDF) (for lobe $i$), $\omega_o$ is the outgoing direction, $\omega_i$ is the incoming direction.

In code, combining lobes looks like this:

```
// The sample the lobes should be evaluated for, storing position, normal,
    material data, etc.
Sample S;
// Compute probabilities for each lobe (e.g., diffuse vs. specular)
float2 probabilities = CalculateStrategyProbabilities(S);

// Evaluate each lobe's BRDF
float3 brdf_d = EvaluateBRDF(S);
float3 brdf_s = EvaluateBRDF(S);

// Compute the PDF for each lobe
float3 pdf_d  = BRDF_PDF(S);
float3 pdf_s  = BRDF_PDF(S);

// Combine final BRDF and PDF
float3 brdf_combined = probabilities.x * brdf_d + probabilities.y * brdf_s;
float3 pdf_combined          = probabilities.x * pdf_d + probabilities.y *
    pdf_s;
```

### 2.2.4 Resampled Importance Sampling (RIS)

Importance Resampling (IR) is a technique for generating samples from a target probability density function (PDF) $g(\mathbf{x})$ when direct sampling from $g$ is infeasible or too complex. One particular strategy is often referred to as Sampling Importance Resampling (SIR) [22].

**Sampling Importance Resampling (SIR)** Considering two distributions:

- A *source* distribution $p(\mathbf{x})$, from which samples can be readily drawn.
- A *target* distribution $g(\mathbf{x})$, which cannot be sampled directly.

A sample $\mathbf{Y}$ can be generated approximately distributed according to $g(\mathbf{x})$ by:

1. Drawing $M$ samples $\{\mathbf{X}_1, \ldots, \mathbf{X}_M\}$ from $p(\mathbf{x})$.
2. Computing weights
$$w_j = \frac{g(\mathbf{X}_j)}{p(\mathbf{X}_j)}$$
for $j = 1, \ldots, M$.
3. Picking a single sample $\mathbf{Y}$ from among $\{\mathbf{X}_j\}$ with probability proportional to $w_j$.

For large $M$, the distribution of $\mathbf{Y}$ converges to $g(\mathbf{x})$. Unfortunately, SIR is not feasible in path tracing as it requires normalization of the weights through the path integral, which is intractable as it requires solving $g(\mathbf{x})$. In the case of path tracing, this would be a flattened path integral.

**Resampled Importance Sampling (RIS)** Resampled Importance Sampling (RIS) combines SIR with Monte Carlo integration [10]. Let the integral to estimate be

$$I = \int_\Omega f(\mathbf{x})\, d\mu(\mathbf{x}),$$

and let $p(\mathbf{x})$ be a pdf that can be sampled easily, while $g(\mathbf{x})$ is an approximation to $f(\mathbf{x})$ but potentially difficult to sample directly. Draw $N$ "outer" samples via SIR, each using $M$ "inner" draws from $p(\mathbf{x})$.

**RIS Estimator** After generating $\{\mathbf{X}_{ij}\}$ for $i = 1, \ldots, N$ and $j = 1, \ldots, M$ (where each set of $M$ samples is used to pick a single $\mathbf{Y}_i$ via SIR), the RIS estimator is [10]:

$$\hat{I}_{\text{ris}} = \frac{1}{N} \sum_{i=1}^{N} \left[ \frac{f(\mathbf{Y}_i)}{g(\mathbf{Y}_i)} \frac{1}{M} \sum_{j=1}^{M} \frac{g(\mathbf{X}_{ij})}{p(\mathbf{X}_{ij})} \right]. \tag{21}$$

Here, $\mathbf{Y}_i$ is drawn from the $M$ candidates $\{\mathbf{X}_{i1}, \ldots, \mathbf{X}_{iM}\}$ using weights

$$w_{ij} = \frac{g(\mathbf{X}_{ij})}{p(\mathbf{X}_{ij})}.$$

### Process Description

Resampled importance sampling can be applied to path tracing with the following process, with the respective algorithm 1 [23]:

1. Take candidates $(X_1, \ldots, X_M)$ in a common domain $\Omega$.
2. Evaluate resampling MIS weights $m_i(X_i)$ for all $X_i$. MIS weights are required when resampling from different domains to avoid double-counting of samples. A simple example is using RIS when combining NEE and BSDF light samples.
3. Evaluate resampling weights

$$w_i = m_i(X_i)\,\hat{p}(X_i)\,W_{X_i}$$

   for each candidate $X_i$. $\hat{p}(X_i)$ is the target function that serves as a proxy for measuring how well a sample covers the targeted intractable integral. In the context of path tracing, a good target function for RIS is the contribution of a sample without its PDF weight. For RIS to converge, the target function has to be proportional to the integrated function.
4. Choose a candidate $X$ randomly from $\{X_1, \ldots, X_M\}$ proportionally to the weights $w_i$.
5. Evaluate the unbiased contribution weight (W)

$$W_X = \frac{1}{\hat{p}(X)}\left(\sum_{j=1}^{M} w_j\right). \tag{22}$$

   This replaces the reciprocal PDF in the Monte Carlo estimator.

This process yields a sample $X$ drawn from a pdf that is approximately proportional to the target function $\hat{p}$. Although $X$ is a single sample, it effectively represents the contribution of many candidate samples through its unbiased contribution weight $W_X$.

**Algorithm 1:** Resampled importance sampling (from [23])

**Input:** $M$: number of candidates to generate

**Output:** Sample $Y$ and its unbiased contribution weight $W_Y$

**Function** `randomIndex`$(w_1, \ldots, w_M)$**:**
  $r \leftarrow \texttt{rand}()$;
  **for** $s \leftarrow 1$ **to** $M$ **do**
    **if** $w_s > 0$ **then**
      $r \leftarrow r - \frac{w_s}{\sum_i w_i}$;
      **if** $r \leq 0$ **then**
        **return** $s$;
  **return** $\emptyset$;
**Function** `ResampledImportanceSampling`$(M)$**:**
  **for** $i \leftarrow 1$ **to** $M$ **do**
    generate $X_i$;
    $w_i \leftarrow m_i(X_i)\hat{p}(X_i)W_{X_i}$;
  $(Y, W_Y) \leftarrow (\emptyset, 0)$;
  $s \leftarrow \texttt{randomIndex}(w_1, \ldots, w_M)$;
  **if** $s \neq \emptyset$ **then**
    $Y \leftarrow X_s$;
    $W_Y \leftarrow \frac{1}{\hat{p}(Y)} \sum_i w_i$;
  **return** $Y, W_Y$;

Importantly, if all $w_i$ are zero (i.e., no valid sample is produced), the algorithm returns a null sample $\varnothing$ with $W_\varnothing = 0$.

### 2.2.5 Russian-Roulette Sampling

Russian-Roulette Sampling (Russian-Roulette Sampling (RRS)) is a path termination method that trades increased variance for efficiency and unbiasedness. The primary goal of RRS is to probabilistically terminate or continue simulation paths, balancing computational efficiency and accuracy by reducing unnecessary computation on paths with minimal contribution [12].

In practice, Russian-Roulette Sampling involves assigning a continuation probability $q$, typically dependent on path contribution or some heuristic. A path continues with probability $q$ and terminates with probability $1 - q$.

To ensure unbiasedness, the surviving paths must be reweighted by dividing by the continuation probability $q$:

$$X_{\text{RR}} = \begin{cases} \frac{X}{q}, & \text{with probability } q, \\ 0, & \text{with probability } 1-q. \end{cases} \tag{23}$$

Here, $X$ is the original estimate without Russian-Roulette applied, and $X_{\text{RR}}$ represents the Russian-Roulette estimator [12].

A typical strategy to select $q$ is to make it proportional to the importance or expected magnitude of $X$, such that paths with low contributions have higher termination probabilities. A commonly suggested heuristic for defining $q$ based on the path throughput $T$ is:

$$q = \min\left(1, \max\left(q_{\text{min}}, \frac{||T||}{T_{\text{threshold}}}\right)\right), \tag{24}$$

where $q_{\text{min}}$ is a lower bound to prevent excessively small continuation probabilities, and $T_{\text{threshold}}$ is a threshold parameter controlling path termination sensitivity [12].

For Russian Roulette Sampling to ensure the complete unbiasedness of a path traced render, paths can't be limited by the number of bounces. As this isn't feasible in real-time renderers, a small bias remains. Furthermore, while improving frame time it also increases variance, making it not optimal for real-time applications.

## 2.3 Theoretical Concept of ReSTIR

Spatiotemporal Reservoir Resampling (*ReSTIR*) [8, 9, 23] is a recently introduced method that greatly improves efficiency in real-time path tracing by reusing light or path samples temporally (successive frames) and spatially (among neighboring pixels). By merging these previously gathered samples in an unbiased manner, ReSTIR can effectively boost the number of relevant samples observed per pixel while keeping the overall per-frame cost affordable.

### 2.3.1 Overview and Motivation

Traditional one-shot importance sampling at each bounce does not always produce enough locally important samples for complex lighting or large numbers of emitters. ReSTIR reduces this problem by pooling or reusing samples from different pixels or frames and then resampling them so that each pixel ultimately retains one representative sample, similar to RIS. This resampling

step is driven by reservoir sampling (Sec. 2.3.2), a technique that enables the streaming of RIS [23]. ReSTIR works on a per-pixel basis, therefore the current pixel shall be defined as the pixel position defined by the shader dispatch grid.

ReSTIR initially targeted direct illumination, but subsequent work has shown how to extend it to global illumination via specialized shift mappings [9, 23], allowing reuse of multi-bounce paths in a similar resampling framework.

A ReSTIR pipeline can be built out of three main steps:

- Generates one candidate sample at each pixel using RIS. The selected sample is stored in a so-called reservoir. Typically, a higher quality input results in an improved ReSTIR output result.
- *Temporally* merges in samples from the previous frame, using motion vectors to decide how to transfer old samples to the current pixel.
- *Spatially* merges in samples from a small neighborhood of pixels to further increase the diversity of candidates. The samples are filtered based on how well they align with the target pixel domain.

**Definition: Canonical Sample** Spatial and temporal candidates in almost all cases cover a different integration domain, like a dissimilar normal and position from which the sample was taken. This means that when reusing such a sample, even tho it can be adjusted to fit the domain of the current pixel, it often does not cover the whole domain. For example, a sample that is occluded in a spatial candidate might not be occluded at the pixel location. An initial sample, created at the pixel position, does always cover the pixels domain and is therefore a safe integrator. This sample is called *canonical sample*. Spatial or temporal candidates are *neighbor* samples [23].

### 2.3.2 Reservoir Sampling

Reservoir sampling is a more memory-efficient approach for drawing a single sample out of a larger collection of candidates compared to naive RIS sampling. Rather than storing every candidate's weight and then sampling from a large cumulative distribution function (CDF)—as is typical in regular RIS— a single *reservoir* is maintained that gets updated as new samples arrive. This helps reduce memory usage and cache pressure in GPU implementations. It can be seen as a streaming algorithm for RIS samples [23].

The reservoir receives candidates one by one with a RIS weight $w_i$. For each candidate, it is randomly decided whether to replace the sample currently in the reservoir with the new candidate. The probability of replacement is proportional to the weight of the new candidate relative to the sum of all weights seen so far. After processing all candidates, the reservoir holds exactly one sample $\mathbf{y}$ with probability proportional to $\mathbf{y}$'s weight.

**Reservoir Data Structure.** The reservoir stores two pieces of information:

- chosenSample: the current sample.
- sumWeights: the sum of all candidate weights seen so far, $W = \sum_{i=1}^{M} w_i$.

Algorithm 2 illustrates this minimal structure.

**Algorithm 2:** Reservoir Data Structure [23]

**Data:** Reservoir $r$ with fields $(Y, W_Y, w_{\text{sum}})$

$(Y, W_Y) \leftarrow (\emptyset, 0)$; `// The output sample`

$w_{\text{sum}} \leftarrow 0$; `// The sum of weights`

**Updating the Reservoir** Whenever a new sample $\mathbf{s}$ with weight $w$ is sampled, its weight is added to sumWeights. The current sample stored in the reservoir is replaced with the probability $\frac{w}{\text{sumWeights}}$. This process is shown in Algorithm 3.

**Algorithm 3:** Reservoir Update [23]

**Input:** Reservoir $r$, candidate $X_i$, weight $w_i$

$r.w_{\text{sum}} \leftarrow r.w_{\text{sum}} + w_i$;

**if** $\texttt{rand}() < \frac{w_i}{r.w_{sum}}$ **then**

  $r.Y \leftarrow X_i$;

**Combining Two Reservoirs.** A key advantage of reservoir sampling is that multiple reservoirs can be merged into a single reservoir in an unbiased manner. With two reservoirs $\mathcal{R}_1$ and $\mathcal{R}_2$, a combined reservoir $\mathcal{R}_{12}$ can be produced by adding up the total weights and then performing a random draw based on the ratio of $\mathcal{R}_2$'s total weight to the new sum. Algorithm 4 shows the combination process.

**Algorithm 4:** Merge Reservoirs [23]

**Input:** Reservoirs $r_1, r_2$

**Output:** Merged Reservoir $r_{\text{merged}}$

$v_1 \leftarrow \text{MIS}(r_1) \cdot \hat{p}(r_1) \cdot r_1.W$;

$v_2 \leftarrow \text{MIS}(r_2) \cdot \hat{p}(r_2) \cdot r_2.W$;

$r_{\text{merged}}.w_{\text{sum}} \leftarrow v_1 + v_2$;

**if** $\texttt{rand}() < \frac{v_2}{r_{merged}.w_{sum}}$ **then**

  $r_{\text{merged}}.Y \leftarrow r_2.Y$;

**else**

  $r_{\text{merged}}.Y \leftarrow r_1.Y$;

$r_{\text{merged}}.W_Y \leftarrow \frac{r_{\text{merged}}.w_{\text{sum}}}{\hat{p}(r_{\text{merged}}.Y)}$;

**return** $r_{merged}$;

**Unbiased Final Estimator.** At the end of the sampling process, the reservoir contains a single sample $\mathbf{y}$ chosen with probability $\propto w_y$. To produce an unbiased estimate, the typical PDF for the sample is replaced with the Unbiased Contribution Weight (UCW):

$$\mathcal{W}_{\mathbf{y}} = \frac{W}{w_{\mathbf{y}}}$$

Often, such as in the implementation presented, it is directly stored in the reservoir to optimize computation, simplifying the calculation of the final estimator:

$$\text{final estimator} \;=\; f(\mathbf{y})\,\mathcal{W}_{\mathbf{y}}.$$

### 2.3.3 Temporal Reuse

One of ReSTIR's significant benefits is reusing samples from previous frames. Rather than regenerating all samples in each frame, the previous frame's reservoir (with total weight $W_{\text{prev}}$) is reprojected onto the current pixel using motion vectors.

Concretely:

1. Identify the previous pixel $p_{\text{prev}}$ whose reservoir is relevant for the new pixel $p$.
2. Evaluate whether the sample from $p_{\text{prev}}$ remains valid for $p$.
3. Merge that old reservoir's sample with the current pixel's newly generated candidate reservoir.

By chaining these updates frame to frame, each reservoir can represent an accumulation of many past samples. This reaches comparable convergence to simply accumulating all samples but providing unbiased motion reuse.

### 2.3.4 Spatial Reuse

After generating fresh samples and injecting any reprojected reservoir, the pixel may also combine its reservoir with that of a small neighborhood (e.g., $16 \times 16$ or $32 \times 32$ window). Each neighbor contributes exactly one sample stored in the pixel reservoir, which is merged into the local reservoir using the weighted update. Over many pixels, this can significantly boost the variety of samples, thereby lowering noise in screen-space areas with a similar geometric structure.

However, sharing samples across potentially very different pixels is not always beneficial, since their lighting distributions may not match. Neighbor reuse should therefore use heuristics on depth, normals, and material ID to avoid injecting irrelevant samples and potential bias.

**Shift Mappings for GI** In multi-bounce scenarios, a neighbor's path differs substantially if surfaces are reflective or refractive. Consequently, ReSTIR GI performs a reconnection-shift [9] to adjust a path accounting for the different geometry.

**Generalized MIS weights** As ReSTIR not only resamples from independent distributions, such as one distribution for light sampling and another for BSDF sampling, but also mixes samples from different pixels, each with its own effective distribution, MIS weights need to be used to unbiasedly merge reservoirs. Denote the pixel of interest as $i$, which has a *target function* $\hat{p}_i$, and one spatial neighbor as $j$, with its own function $\hat{p}_j$. Naively reusing neighbor $j$'s sample $\mathbf{x}_j$ for pixel $i$ would require evaluating $p_i(\mathbf{x}_j)$ to form a proper MIS weight. However, we may only know $\hat{p}_j(\mathbf{x}_j)$—the *target* function used when that sample was created in neighbor $j$. Hence, the "plain" balance heuristic

$$m_i(\mathbf{x}) \;=\; \frac{p_i(\mathbf{x})}{p_i(\mathbf{x}) + p_j(\mathbf{x})}$$

cannot be directly applied, since $p_j(\mathbf{x})$ is intractable.

Instead, ReSTIR adopts a Generalized balance heuristic [23] where each neighbor $j$ contributes $\hat{p}_j$ as an approximation to its true PDF. Concretely, assuming the union of pixels $\{i, j, k, \ldots\}$, each has a known target function $\hat{p}_i(\cdot), \hat{p}_j(\cdot), \hat{p}_k(\cdot), \ldots$. Then the generalized balance weights are

$$m_i(\mathbf{x}) \;=\; \frac{c_i\, \hat{p}_i(\mathbf{x})}{\sum_r \; c_r\, \hat{p}_r(\mathbf{x})}, \tag{25}$$

where $c_r$ is a per-distribution confidence weight, capturing how many samples a neighbor's reservoir represents, which gives pixels with a higher number of effective samples stored a higher weight.

This form ensures each pixel's domain is properly covered by some distribution in the neighborhood, without double-counting. Moreover, the denominator $\sum_r c_r \hat{p}_r(\mathbf{x})$ acts as an on-the-fly normalization to mix distributions. Crucially, the union of all pixels' supports must include the full set of samples that can contribute to pixel $i$. Thus, for reuse to remain unbiased:

$$\bigcup_r \text{supp}(\hat{p}_r) \supseteq \text{supp}(\hat{p}_i).$$

When combined with standard reservoir sampling (see Sec. 2.3.2), Eq. (25) guarantees an unbiased mixture of all relevant distributions, making spatial sharing of samples robust even for very different local shading contexts.

### 2.3.5 Pairwise MIS

While the Generalized MIS weights enable unbiased reuse between reservoirs, they have $O(n^2)$ complexity. To obtain a cheaper but still effective MIS scheme, *pairwise MIS* (originally proposed by Bitterli et al. [23]) can be used. The key observation is that among $M$ distinct sampling techniques (or "candidate distributions"), one can designate a *canonical* technique $c$, and then assign weights by comparing each other technique $i \neq c$ *pairwise* against $c$.

**Basic Formulation** Having $M$ sampling techniques, with PDFs $p_i(x)$, one of them (indexed $c$) is picked to be the "canonical" technique. The pairwise balance heuristic proposed in [24] for each sample drawn from technique $i$ yields weights:

$$m_i(x) = \frac{1}{M-1} \frac{p_i(x)}{p_i(x) + p_c(x)} \quad (i \neq c), \tag{26}$$

$$m_c(x) = \frac{1}{M-1} \sum_{j \neq c}^{M} \frac{p_c(x)}{p_j(x) + p_c(x)}. \tag{27}$$

These weights are valid MIS weights—they are nonnegative, sum to one, and preserve unbiasedness.

**Defensive Variant** A known concern is that $p_c(x)$, the canonical PDF, can appear in denominators for many pairs. If $p_c(x)$ is occasionally under-approximated or if $p_i(x)$ is over-approximated, then (26) can assign overly large weight to the $i \neq c$ technique. To mitigate this, a *defensive*

version assigns a higher weight to the canonical sample [23]:

$$m_i(x) = \frac{1}{M} \frac{p_i(x)}{p_i(x) + \frac{p_c(x)}{M-1}} \quad (i \neq c), \tag{28}$$

$$m_c(x) = \frac{1}{M}\Big(1 + \sum_{j \neq c} \frac{\frac{p_c(x)}{M-1}}{p_j(x) + \frac{p_c(x)}{M-1}}\Big). \tag{29}$$

**Generalization** In ReSTIR, the PDFs $p_i(x)$ may be target functions when used with ReSTIR. Denoting these approximations by $\hat{p}_i(x)$, substituting $\hat{p}_i$ for $p_i$ in the above yields *generalized pairwise MIS*. Similar to regular generalized MIS weights, the Pairwise MIS weights can allow each sampling technique $i$ to have an associated *confidence weight* $c_i \geq 0$, which acts like a multiplier on $\hat{p}_i(x)$ to reflect either how many samples are being drawn from technique $i$, or some estimated quality of that technique.

# 3 Methodology: Rendering backend

This chapter discusses the low-level rendering architecture used in the project, focusing on DirectX 12 ray tracing (DXR). It introduces how the ray tracing pipeline is structured and exemplifies key components such as acceleration structures and shaders. Finally, it explains how these elements come together in this implementation using an NVIDIA DXR sample [25] as a reference for the C++ backend.

## 3.1 Graphics API: DirectX 12 DXR

DirectX 12 is a low-level graphics API introduced by Microsoft to provide more direct control over graphics hardware compared to its predecessors [3]. One of its major improvements over previous versions is a closer-to-the-metal programming model, enabling fine-grained resource management and synchronization control. These changes help in optimizing performance-critical rendering workflows. Debugging and profiling as part of developing performant shaders are supported by several profiling tools such as NSight Graphics [26] by NVIDIA which will be used in this project as well. Programming shaders in DirectX is performed using the C-like "high-level shader language" (hlsl). DirectX 12 DXR is only supported on Windows and requires a GPU that enables hardware acceleration of ray tracing, starting with the NVIDIAs Turing and AMDs RDNA 2 architecture (NVIDIA RTX 20xx and RADEON RX6xxx respectively).

DirectX 12 was released with the aim of reducing driver overhead and allowing advanced techniques, such as explicit multi-threading and descriptor heap management. In parallel, GPU vendors started adding dedicated hardware support for ray tracing. Eventually, Microsoft introduced the *DirectX ray tracing* (DXR) extension, which leveraged existing DirectX 12 infrastructures to allow hardware-accelerated ray intersection [3].

ray tracing in DXR allows the project to take advantage of hardware-accelerated operations for bounding volume hierarchy (BVH) construction and traversal. For these reasons, DXR is an ideal choice for this work, particularly in conjunction with algorithms like ReSTIR and real-time focus, as DirectX is used in the majority of interactive applications.

While *Vulkan* also offers ray tracing extensions (namely, `VK_KHR_ray_tracing_pipeline`), the choice of DirectX 12 for this project was primarily driven by established ecosystem support, development tools, and existing sample frameworks that could be adapted with minimal overhead [25].

## 3.2 ray tracing Pipeline Structure in DXR

DXR introduces several new concepts to the DirectX 12 pipeline to facilitate ray tracing. At a high level, a typical DXR application must:

1. Construct one or more *acceleration structures*,
2. Define a *ray tracing pipeline state object* containing relevant ray tracing shaders,
3. Dispatch *ray generation* and handle shader invocation based on the scene geometry.

### 3.2.1 Acceleration structures

Acceleration structures lie at the core of DXR, enabling efficient ray intersection tests and, consequently, real-time or near-real-time ray tracing. Instead of testing each ray against all triangles in the scene, a well-constructed acceleration structure quickly eliminates large sets of geometry that a ray cannot intersect. This significantly reduces the total number of intersection tests and scales better as scene complexity grows [3].

DXR classifies acceleration structures into two main categories:

- **Bottom-Level Acceleration Structures (BLAS)**: These contain geometry data (e.g., triangles) for individual meshes. Typically, one BLAS is constructed per mesh or per set of static geometry.
- **Top-Level Acceleration Structures (TLAS)**: These reference one or more BLAS, along with per-instance metadata such as instance transforms and instance IDs. This allows multiple instances of the same geometry to appear in the scene at different locations or with different orientations.

#### BVH building in DirectX 12

To construct acceleration structures in DirectX 12 DXR, developers issue commands that describe the geometry and hierarchy to the GPU [3]. A typical build process involves:

1. **Bottom-Level Acceleration Structure (BLAS) Build**: The engine specifies vertex and index data for each mesh, then issues a *build* command (e.g., `Buildray tracingAccelerationStructure`) to the GPU. This command instructs the GPU to create a hierarchical tree of axis-aligned bounding boxes (AABBs) that encloses the mesh geometry.

2. **Top-Level Acceleration Structure (TLAS) Build**: The engine references previously built BLAS objects, supplying instance transforms and IDs. Another *build* command creates the top-level structure, which serves as the primary entry point for ray tracing queries.

### Bounding Volume Hierarchy (BVH) data structure

At its core, each BLAS is typically stored as a tree of bounding volumes, where each internal node bounds a subset of the scene geometry using an axis-aligned bounding box (AABB). Leaves of the tree directly reference geometry primitives. Figure 6 conceptually illustrates a small portion of a BVH.

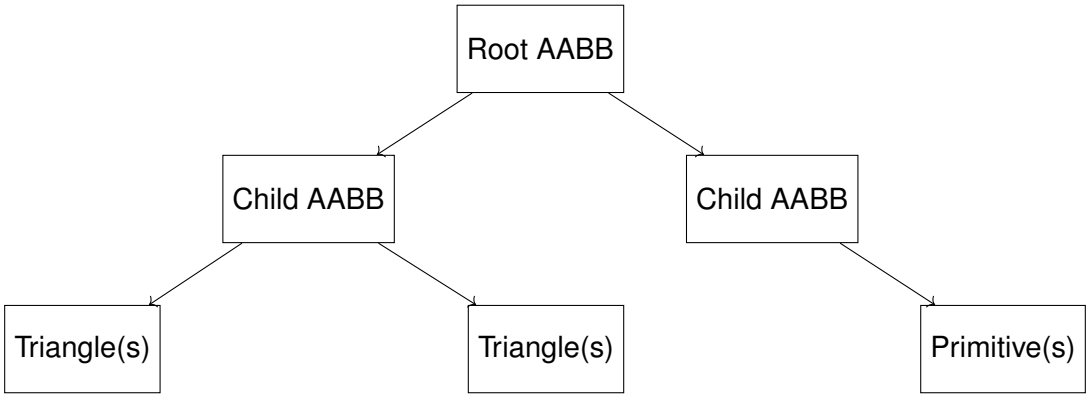


Figure 6: Conceptual structure of a BVH, where each node holds an AABB that encloses its children. Leaves reference actual geometry (e.g., triangles or other primitives like quads or spheres).

In practice, the BVH may store multiple primitives in one leaf node for efficiency; details depend on the specific build parameters and performance trade-offs chosen by the engine. In the implementation presented by this work, only triangles are valid primitives as their intersection can be fully accelerated by the hardware.

### Time complexity of BVH traversal

Without acceleration structures, a ray tracer would have to intersect each ray with every triangle in the scene, incurring a cost proportional to $\mathcal{O}(N)$, where $N$ is the number of triangles. With a BVH, the ray first tests against bounding volumes in a top-down fashion, quickly discarding large numbers of primitives. In an ideal balanced tree, this can reduce the intersection tests to approximately $\mathcal{O}(\log N)$, drastically improving performance [27].

### Hardware acceleration

Leveraging the specialized units of modern GPUs, both BVH traversal and ray-triangle intersection can be significantly sped up. By offloading these operations from general-purpose compute

units, the hardware can perform bounding box and triangle intersection tests more efficiently in parallel. This enables real-time ray tracing for scenes containing millions of triangles, provided the BVH is built and updated effectively.

#### Scaling compared to rasterization

While rasterization pipelines efficiently handle the projection and shading of large batches of triangles (with complexity often tied to screen resolution rather than scene size), ray tracing scales with the number of rays cast and the complexity of the scene's geometry. However, the hierarchical BVH approach mitigates the worst-case scaling, making ray tracing feasible for interactive rendering. While combining rasterization for primary visibility and DXR-based ray tracing for secondary effects (e.g., reflections, shadows) can yield good efficiency in scenes with low to medium triangle count (ca. 1 to 10 million triangles), with a growing number of triangles the primary visibility rasterization scales more aggressively than the ray tracing. As ray tracing primary visibility using ray tracing only requires one ray per pixel, it is relatively cheap compared to tracing possibly several full paths per pixel per frame. That is why this work's implementation uses a full ray tracing rendering approach without rasterized geometry which also leads to lower overhead in the rendering pipeline.

### 3.2.2 Ray Tracing Shaders

DXR introduces several shader types to DirectX 12 [3]:

- **Ray Generation (RayGen) Shader**: The entry point for ray tracing. It dispatches rays into the scene.
- **Intersection Shader**: Used for custom geometry intersection logic; is not required by the pipeline and can therefore not leverage the Hardware accelerated primitive intersection.
- **Any Hit Shader**: Invoked upon hitting geometry; can be used to evaluate whether the hit should be ignored or proceed.
- **Closest Hit Shader**: Invoked after all potential intersections along a ray have been processed, and determines the final shading at the closest intersection point.
- **Miss Shader**: Invoked if a ray does not intersect any geometry in the scene.

Figure 7 shows the typical flow of how a ray is traced through these stages.

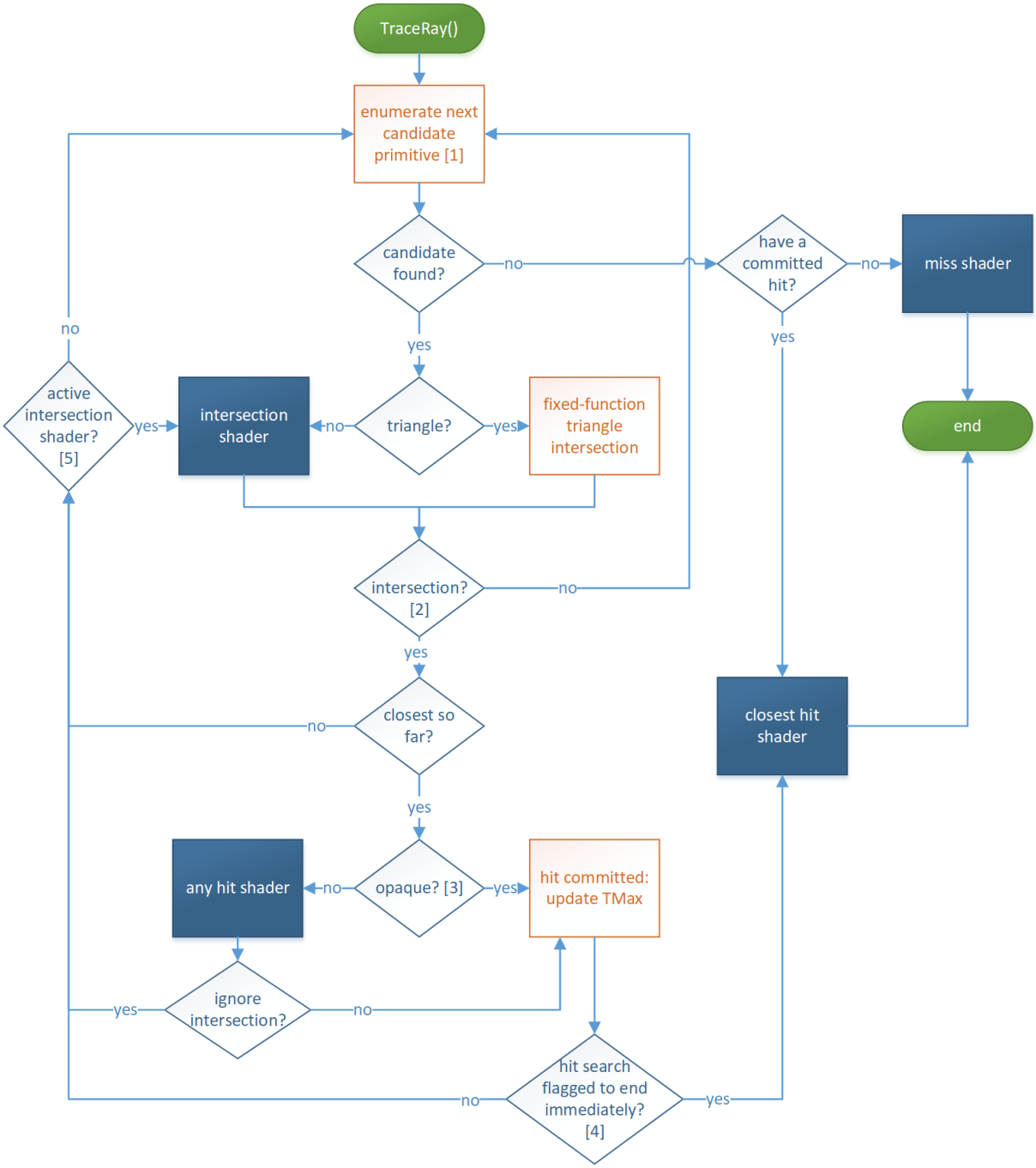


Figure 7: Trace ray control flow [3].

### 3.2.3 NVIDIA DXR Sample and project setup

For this implementation, an NVIDIA DXR sample [25] was used as the foundation of the C++ backend. This sample provides a working demonstration of:

- Setting up a DXR-compatible swap chain,
- Creating and managing acceleration structures (BLAS and TLAS),

- Defining ray tracing pipeline state objects and descriptors

**Changes for ReSTIR integration** Several adaptations were made to accommodate the ReSTIR-based frontend in this DXR framework:

1. **Material and 3D object loading with TinyOBJLoader** [28]: This allows the backend to import complex meshes and material attributes from .obj file formats. The .mat file format is used to describe the surface materials. As the material model of the renderer is physically based, PBR material extensions adhering to the OpenPBR standard are required for a realistic material representation.

   Object loading also involves precalculating the LUT for multiscatter-adjusted GGX (Section 2). This is performed whenever a material is loaded by integrating the BSDF for each LUT entry on the CPU.

Table 1: Example lookup table for $\cos\theta$ and corresponding $E_{ss}$ values. The material this table is used for is a high-roughness GGX material.

| $\cos\theta$ | 0.00 | 0.07 | 0.13 | 0.20 | 0.27 | 0.33 | 0.40 | 0.47 | 0.53 | 0.60 | 0.67 | 0.73 | 0.80 | 0.87 | 0.93 | 1.00 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $E_{ss}$ | 2.22 | 2.38 | 2.57 | 2.63 | 2.75 | 2.79 | 2.89 | 2.97 | 3.03 | 3.07 | 3.11 | 3.17 | 3.19 | 3.16 | 3.20 | 3.26 |

   As the evaluation of the integral is numerical in this case, good results require a significant amount of samples, resulting in calculation times of around 0.3s per material with 16000 samples and a LUT size of 16. In its current implementation, this is performed every time the scene is loaded together with the loading of the model. Optimizing this, each 3d object could be loaded only once and stored in a binary of its backend representation, especially as loading large .obj files can take minutes to parse.

2. **Instanced geometry**: The pipeline was extended to allow multiple instances of the same object, each with unique transforms. This enables tests with dynamic scenes where objects move in addition to the camera. As each Instance has a matrix representation of position, rotation, and scale, the inverse is stored as well for calculating per-object motion vectors.

3. **Creation of a CDF list with light triangles**: A cumulative distribution function (Cumulative Distribution Function (CDF)) is built over the set of light-emitting triangles for importance sampling. This is crucial for ReSTIR, which relies on efficient light sampling. As the CDF is computed only once when parsing all objects, changing geometry and disabling and enabling lights at runtime isn't supported.

# 4 Methodology: Implementing ReSTIR

As ReSTIR is a novel method, there are few openly accessible implementations and even fewer are well documented. Although "A Gentle Introduction to ReSTIR" [23] covers the core concepts and mathematical foundation, important details concerning an implementation are missing. This chapter aims to provide an in-depth look at how ReSTIR can be implemented in hlsl and what ideas and best practices can be used to improve performance and image quality. The optimizations section then gives an overview of how ReSTIR can be improved in practice in a DirectX environment.

## 4.1 Preliminaries

In implementing ReSTIR, a good practice is to fully split the integration of direct and all remaining indirect illumination. This work will adopt this principle. While it results in higher memory usage, variance reduction, and reduced artifacts (Discussed in chapter 6) outweighs the drawbacks.

### 4.1.1 Render Passes

A (render-) pass in the ray tracing pipeline is defined as a full dispatch of a raygen shader for all pixels. As ReSTIR requires spatial and temporal pixel-based data, a minimum of two render passes are required.

As shader data is mostly not persistent between passes, Unordered Access View Buffers (UAV) are used to store data between render passes and frames. As this implementation relies on ray tracing only, a rasterized GBuffer does not exist.

To store initial deterministic camera ray data, the SampleData structure is used:

```
struct SampleData
{
    float3 x1;
    uint16_t  mID;
    half3 L1;
    float3 n1;
    float3 o;
    uint  objID;
};
```

Listing 1: Example definition of SampleData struct in hlsl

with:

- `x1`: the camera ray scene intersection position in world space, similar to a depth buffer
- `n1`: the normal at x1
- `o`: the outgoing vector $\omega_o$, in this case, it is defined as the vector from the intersection position to the camera
- `L1`: stores the RGB emissive value of the material hit by the camera ray.
- `mID`: the material ID of the triangle hit.
- `objID`: the instance ID of the triangle hit used for calculating per-object motion vectors.

The reservoirs for ReSTIR are stored similarly. Due to the split of Direct and Indirect Illumination, a total of four reservoir buffers are required. This results in this buffer structure:


Figure 8: A total of six buffers is used to transfer reservoir and sample data between render passes and frames.

## 4.2 Direct Illumination (DI)

Direct Illumination (DI) in the context of ReSTIR involves estimating illumination at a point by directly sampling emissive surfaces or light sources in the scene.

### 4.2.1 Reservoir Structure

In this implementation, only area triangle lights are supported, resulting in this reservoir structure:

```
struct Reservoir_DI
{
    float3 x2;    float w_sum;
    float3 n2;    float W;
    half3 L2;     uint16_t M;
};
```

Listing 2: Example definition of SampleData struct in hlsl

where:

- `x2` is the sample point on a light triangle
- `n2` is the normal at x2
- `L2` is the light triangle emissive strength
- `w sum` is the sum of the weights acquired from RIS
- `W` is the unbiased contribution weight
- `M` is a confidence weight

The confidence weight M can be understood as an indicator of how many samples a reservoir contains. Higher M values typically indicate a higher-quality sample stored. M is increased when reservoirs are merged.

### 4.2.2 Initial Sampling

RIS is used to create an initial sample using Reservoir sampling. To provide a reasonable initial sample quality, several NEE and BSDF samples are resampled into one sample. To improve performance, the NEE samples are evaluated without calculating the occlusion $V$, requiring a shadow ray test for visibility. MIS is used to optimize variance between NEE and BSDF sampling.

```
function InitialSampling(M1, M2, reservoir):
    for i = 1 to M1:
        sample_NEE, pdf_light, pdf_bsdf = SampleLightNEE()

        mi = pdf_light / (M1 * pdf_light + M2 * pdf_bsdf)
        wi = (mi * p_hat) / pdf_light
        UpdateReservoir(reservoir, wi, sample_NEE)

    for j = 1 to M2:
        sample_BSDF, pdf_bsdf, pdf_light= SampleLightBSDF()

        mi = pdf_bsdf / (M1 * pdf_light + M2 * pdf_bsdf)
        wi = (mi * p_hat) / pdf_bsdf
        UpdateReservoir(reservoir, wi, sample_BSDF)

    reservoir.M = 1
    if(Visibility(reservoir) = false)
        InvalidateReservoir(reservoir)
end function
```

Listing 3: Pseudocode for Initial RIS Sampling

where:

- `M1` is the number of BSDF samples.
- `M2` is the number of NEE samples.
- `mi` is the MIS weight to account for BSDF/NEE sampling. It is calculated using the balance heuristic.
- `wi` is the reservoir resampling weight.

Importantly, a BSDF sample cannot be obtained without tracing a shadow ray. A good balance between performance and quality is to use M1 = 1 and M2 = 10. This requires tracing a total of two rays to fill the initial reservoir. A visibility check by tracing a shadow ray is performed to verify that the stored sample is valid. The reservoir is invalidated if that is not the case to remain unbiased, as otherwise, the NEE samples would result in light leakage.

### 4.2.3 Temporal Reuse

A reservoir from initial sampling can be combined with the reservoir from the last frame using algorithm 4. To retrieve the last frame's reservoir in a dynamic scene, typical for real-time

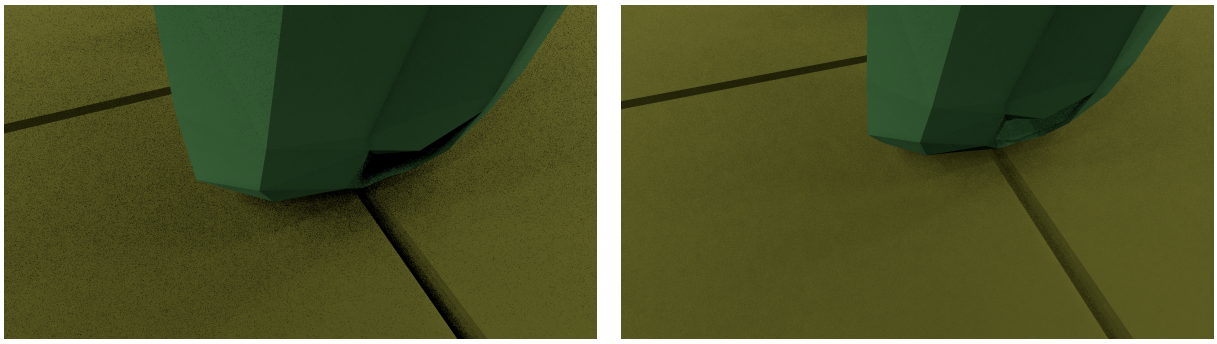

Figure 9: Left: postponing the visibility check leads to fully black pixels but no bias; Right: omitting the shadow ray completely leads to significant light bleeding and incorrect shadows.

applications, a motion vector has to be used to determine the pixel position of the last frame's reservoir [29]:

1. Transform current world-space position $P^t_{world}$ into local object-space:

$$P_{local} = M^t_{world \to object} \cdot P^t_{world}$$

2. Reproject $P_{local}$ to the previous world-space using the last frame's object transform:

$$P^{t-1}_{world} = M^{t-1}_{object \to world} \cdot P_{local}$$

3. Project $P^{t-1}_{world}$ to previous clip-space:

$$P^{t-1}_{clip} = P^{t-1} \cdot V^{t-1} \cdot P^{t-1}_{world}$$

4. Convert clip-space to normalized device coordinates (NDC):

$$P^{t-1}_{ndc} = P^{t-1}_{clip}.xy / P^{t-1}_{clip}.w$$

5. Map NDC to pixel coordinates:

$$P^{t-1}_{pixel} = (P^{t-1}_{ndc} \times 0.5 + 0.5) \times \text{resolution}$$

If a pixel lies outside the screen space, a reservoir merge can not be performed.

The second relevant factor is the MIS weights between the canonical sample and the temporal sample. As the PDF for the temporal reservoir is not known, a generalized MIS weight is used to

replace the PDF with the target function

$$m_i(x) = \frac{M_i \hat{p}_i(x)}{\sum_{j=1}^{M} M_j \hat{p}_j(x)}$$

where M is the confidence weight. The MIS weight is used to weigh the weight for each reservoir, which is computed as $\hat{p}$ * W. $\hat{p}$ can be calculated by reconnecting the camera ray path vertex textttx1 from the current pixel with textttx2 stored in each reservoir.

```
function TemporalReuse(c):
    t = GetReprojectedReservoir()

    if IsValid(t):
        c.M = min(cap, c.M)
        t.M = min(cap, t.M)
        mi_c = ComputeMIS(c)
        mi_t = ComputeMIS(t)

        w_c = mi_c * EvaluatePHat(c) * c.W
        w_t = mi_t * EvaluatePHat(t) * t.W

        c.w_sum = w_c

        UpdateReservoir(c, w_t, t.sample)

        c.W = c.w_sum / EvaluatePHat(c)

    end if

end function
```

Listing 4: Temporal Reservoir Reuse

Temporal reuse in combination with spatial reuse can introduce significant temporal correlation artifacts as visible in figure 10. To limit this, the confidence weight M should be capped to a reasonable value, typically 20 [23].

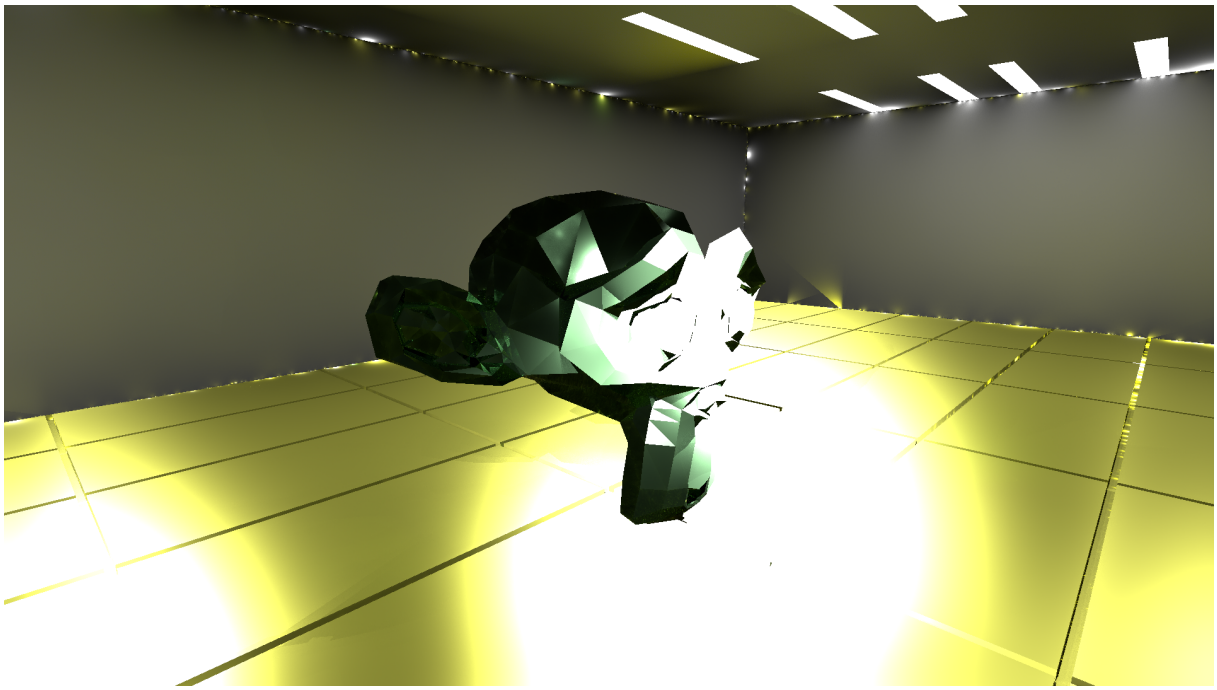

Figure 10: Uncapped confidence weights result in significant image artifacts.

### 4.2.4 Spatial Reuse

Spatial reuse of reservoirs works similarly to temporal reuse. Instead of merging the canonical sample reservoir with a reservoir from a previous frame, reservoirs are randomly taken from the adjacent pixels. Good results can be achieved by using several adjacent reservoirs. Not all possible candidates are fitting candidates, mostly because of geometric differences or differences in materials. It is good practice to reject pixels that store significantly different normals, different materials, and different depth in the sample buffer. In the optimizations section, an improved way of selecting good neighbor samples will be introduced.

The algorithm can be extended by a simple loop and rejection mechanism:

```
function SpatialReuse(c):
    for n pixels:
        s = GetRandomPixelInCircle(c)

        if IsValid(s) and not Reject(s):
            c.M = min(cap, c.M)
            s.M = min(cap, s.M)
            mi_c = ComputeMIS(c)
            mi_s = ComputeMIS(s)

            w_c = mi_c * EvaluatePHat(c) * c.W
            w_t = mi_s * EvaluatePHat(s) * s.W

            c.w_sum = w_c

            UpdateReservoir(c, w_s, s.sample)

            c.W = c.w_sum / EvaluatePHat(c)
        end if
    end for
end function
```

Listing 5: Spatial Reservoir Reuse

In this implementation, pairwise MIS is used to reduce the computational complexity of the MIS weights. This is especially important as each weight requires tracing a shadow ray.

## 4.3 Indirect Illumination (GI)

ReSTIR GI extends the concept of ReSTIR DI to indirect lighting. Instead of reconnecting to a light vertex, the sample buffer point is connected to a path vertex of a sampled path stored in a reservoir.

### 4.3.1 Reservoir Structure

To store a path sample, the reservoir structure for GI samples uses a similar concept compared to the DI variant:

```
struct Reservoir_GI
{
    float3 xn;      float w_sum;
    float3 nn;      float W;
    float3 Vn;      uint16_t mID2;  uint16_t M;
    half3 En;
};
```

Listing 6: Example definition of SampleData struct in hlsl

where:

- `xn` is the path vertex to reconnect to
- `nn` is the normal at xn
- `En` is the path contribution from the remaining path sample
- `Vn` is the incoming direction at xn
- `w sum` is the sum of the weights acquired from RIS
- `W` is the unbiased contribution weight
- `M` is a confidence weight

### 4.3.2 Initial Sampling

The initial sample stored in the GI reservoir is computed by tracing a path tree and using RIS to select one subpath [23]. This involves generating an NEE sample at each bounce, with the first bounce excepted as the direct lighting is covered by ReSTIR DI.

Subpaths, as well as their path throughput, need to be tracked independently and only carry one light contribution. For each NEE sample, a visibility check is performed. While this could be omitted and performed after path sampling, significant noise would be introduced, especially in areas that indirect light can only reach in several bounces.

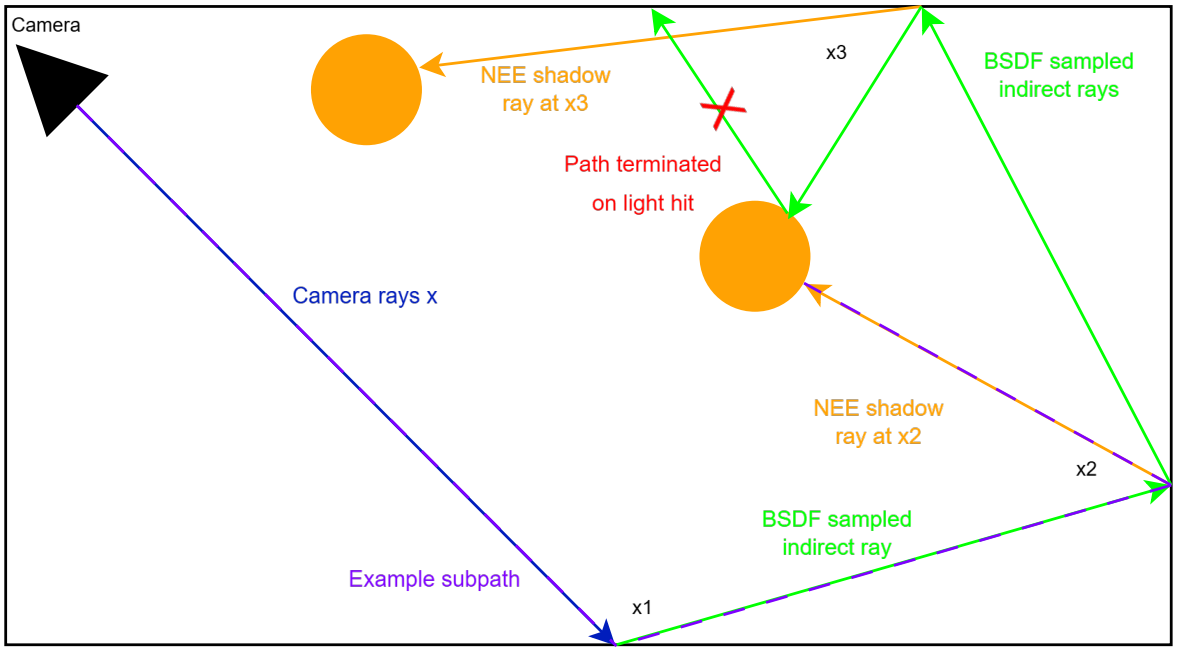


Figure 11: Initial sampling of paths. Every subpath is evaluated and added to the reservoir individually.

The resulting algorithm then is:

```
function SamplePaths(origin, normal):
    direction = SampleBSDF()
    hit       = TraceRay(origin, direction)
    if IsLight(hit):
        break

    for bounce in [1 .. maxBounces]:
        subpath_nee = SampleAndEvaluateNEE()
        UpdateReservoir(subpath_nee)

        subpath_bsdf = SampleAndEvaluateBSDF()
        if IsLight(L_bsdf.emission):
            UpdateReservoir(subpath_bsdf)
            break

        origin = hit.position
        normal = Normalize(hit.normal)
    end for

    return radiance
end function
```

Listing 7: Path sampling with reservoir updates

### 4.3.3 Temporal and Spatial Reuse

Temporal and spatial reuse work in a similar manner as in ReSTIR Direct Illumination (DI). A reconnection shift between the x1 path vertex (stored in sample data for a pixel) and xn from the Globial/Indirect Illumination (GI) reservoir is performed to adjust for the difference in locality for the path contribution. Figure 12 shows an exemplary path reconnection. Unlike in ReSTIR DI, a Jacobian determinant has to be included in every phat calculation to account for the changed path PDF as the GI paths are using a solid angle space PDF instead of the area space PDF ReSTIR DI uses. The Jacobian determinant for the reconnection between x1 and xn can be calculated from the division of the geometry terms [9]:

$$\left|\frac{\partial\omega_{k-1}^{y}}{\partial\omega_{k-1}^{x}}\right| = \frac{\left|\cos\theta_{k}^{y}\right|}{\left|\cos\theta_{k}^{x}\right|}\frac{\|x_{k}-x_{k-1}\|^{2}}{\|x_{k}-y_{k-1}\|^{2}},$$

In more recent ReSTIR variants [24], the reconnection shift mapping is extended to also include the second BSDF term. Not including this leads to a slight bias for specular materials. As this work is focused on real-time applications, this bias is not visible to the eye without significant image enhancements, while requiring significantly more compute time and can introduce temporal artifacts.

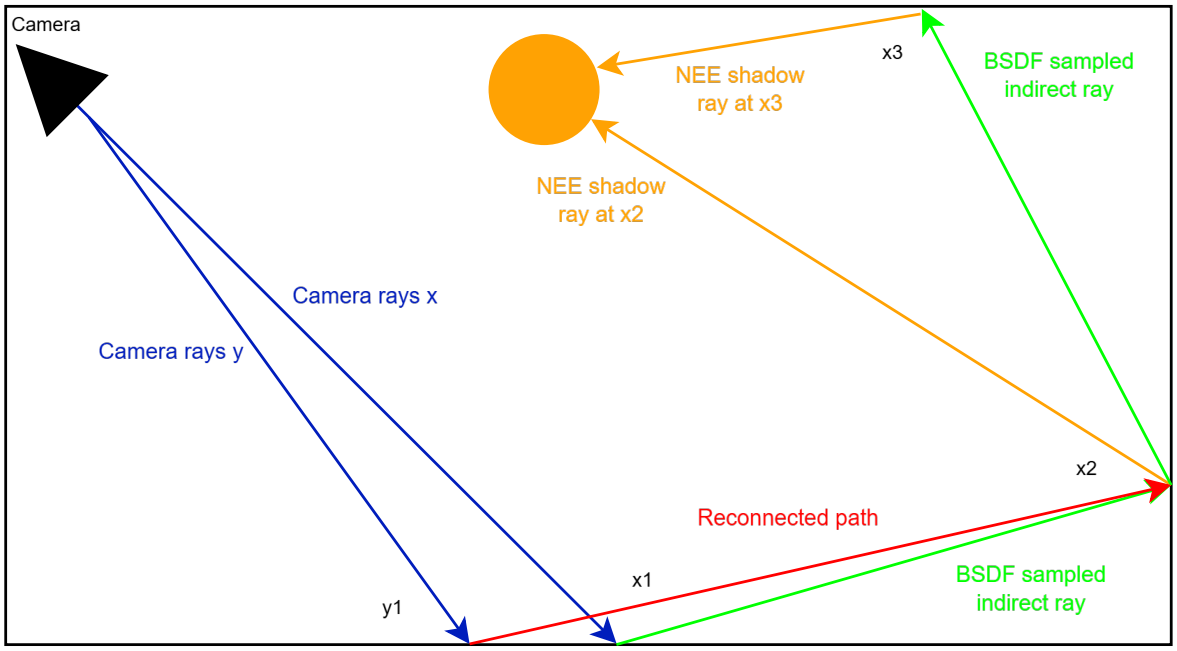


Figure 12: Reconnection of the path vertex y1 with the path x1.

## 4.4 Implementation Decisions

As ReSTIR itself only provides a framework for a spatiotemporal accelerated path tracer, several design decisions have to be made. As the presented implementation focuses on real-time rendering, the emphasis lies on optimal reuse and RIS chaining and less on unbiasedness.

### 4.4.1 Pipeline

A ReSTIR implementation typically follows a *pipeline* architecture. As stated in the Render Passes section, each stage of ReSTIR can be implemented as a self-contained pass. In this work, each render pass is set up as a raygen shader (since the introduction of inline ray tracing, a compute shader works too). The order of the passes except the initial sampling step can be freely rearranged. This results in different structures for different implementation goals. A good real-time design can be seen in Figure 13.

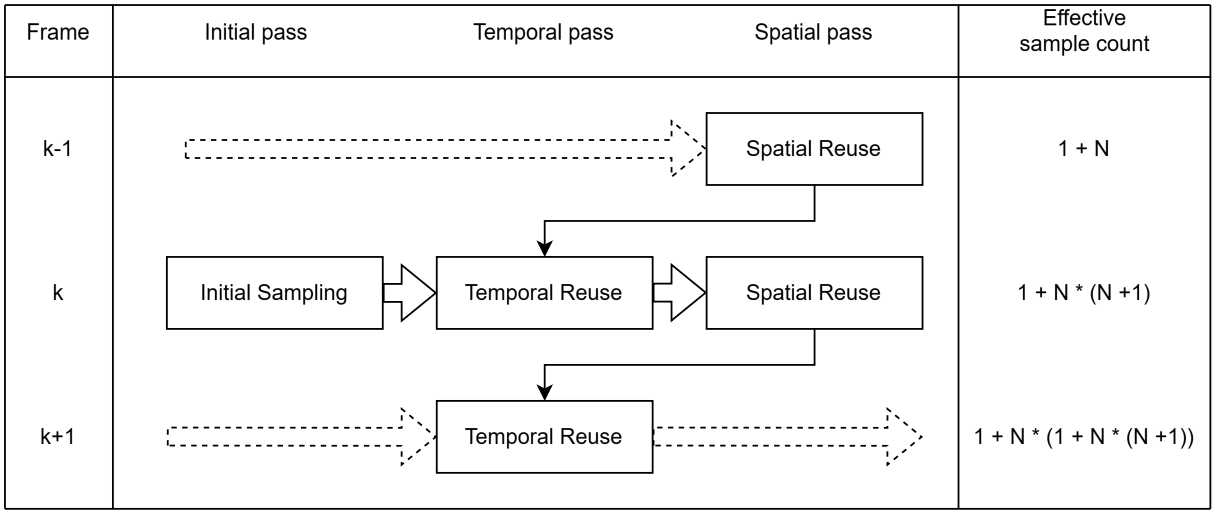


Figure 13: The chosen architecture creates a pipeline that creates a quadratic scaling of effective sample count. N is the number of neighbor samples for spatial reuse. A naive path would create 3 samples in 3 frames, while ReSTIR can achieve 40 combined samples in the best case.

In this case, reservoirs are first updated temporally. Usually, the temporal reservoir carries a more refined sample than the initial sampling can provide, thus improving the spatial reuse significantly. While more than one spatial reuse pass would be possible, the inherent computational cost and render pass overhead render a single spatial reuse pass the optimum.

### 4.4.2 Spatiotemporal Reuse

**ReSTIR GI variant** The implemented GI variant uses the simplified biased reconnection. Tests with the unbiased variant introduce significant temporal correlations, even at a low M-cap:

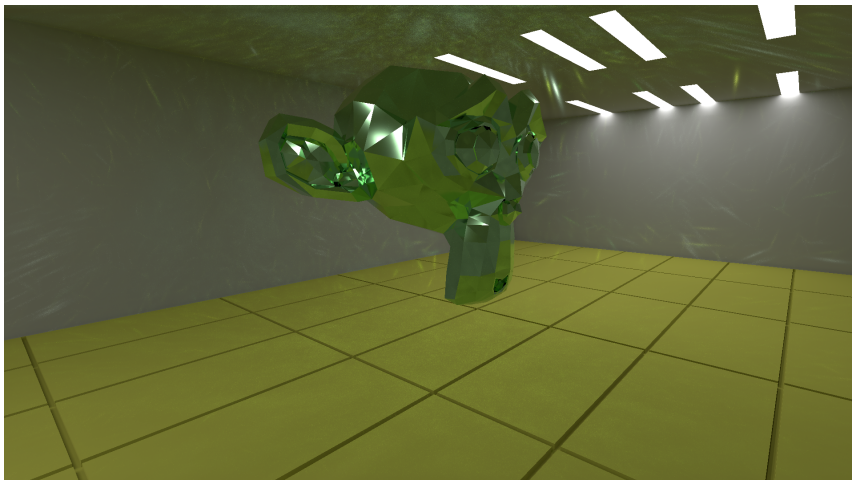

Figure 14: Temporal correlations resulting from an unbiased reconnection, specifically caused by a too high M-cap and specular material.

**Sample PDF Measure** Typically, the PDF of a sample can have two forms [23]:

- *Solid angle measure*: does not take into account distance and angle at the sampled point. This PDF often comes from a BSDF sampler (e.g., GGX).
- *Area measure*: extends the solid angle measure PDF by factoring in the distance and the local angle at the sampled point. This is typical for a sampler that performs Next Event Estimation (NEE).

Each PDF can be converted into the other via the following relationships. Let $p$ be the sampled point, $r$ the distance from the reference point to $p$, and $\theta$ the angle between the surface normal at $p$ and the direction to the reference point:

$$\text{pdf}_{\text{area}}(p) \;=\; \text{pdf}_{\text{solid}}(\omega) \;\times\; \frac{|\cos(\theta)|}{r^2}$$

$$\text{pdf}_{\text{solid}}(\omega) \;=\; \text{pdf}_{\text{area}}(p) \;\times\; \frac{r^2}{|\cos(\theta)|}.$$

This Resampled Importance Sampling implementation for Direct Illumination (DI) uses the *area measure*, which removes the Jacobian term in the $\hat{p}$ calculation and simplifies reuse. On the other hand, the Global Illumination (GI) implementation uses solid-angle space for the path extension, since it is sampled via chained BSDF sampling.

## 4.5 Optimizations

This section will present optimizations for the ReSTIR pipeline implemented in DXR using GPU profiling, as well as own optimizations that improve the ReSTIR output.

### 4.5.1 GPU architecture and profiling

Successful profiling for performance improvements requires general knowledge of GPU architectures. As this implementation targets NVIDIA GPUs, optimizations were done based on NVIDIA hardware. The most recent generation of GPUs (NVIDIA Blackwell) features the following architecture:

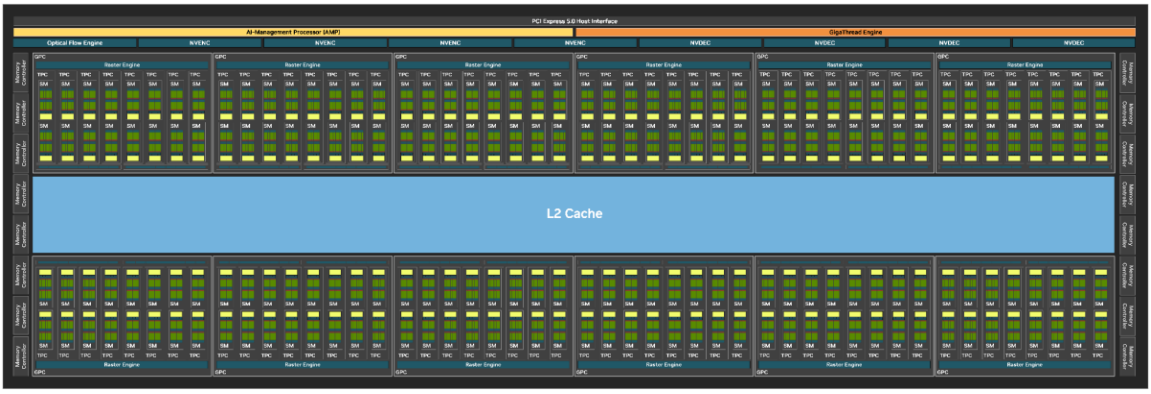


Figure 15: NVIDIA Blackwell GB202 GPU [4]

Importantly, it features an L2 cache that is shared between all shader multiprocessors (Shader Multiprocessor (SM)) that perform ray tracing computations. Each SM features a smaller L1 cache. A SM features several Warp units that each contain 32 threads (NVIDIA GPUs) but only one type of instruction can be executed at a time.

These concepts are important when optimizing path tracing GPU code [30, 31]:

- Latency hiding: Accessing data from global memory or cache can incur significant latency. GPUs hide this latency by switching to other warps that are ready to execute, allowing computation to continue while some threads wait. This is effective when there are many independent warps available, helping mask memory delays and maximize throughput.
- Memory coalescence: Refers to aligning memory access patterns so that threads in a warp access contiguous memory locations. This allows multiple memory requests to be combined into a single transaction, improving memory throughput and reducing bandwidth waste. Uncoalesced access leads to inefficient memory usage and more transactions.

- Active Threads per Warp: As each warp can only execute one type of instruction at a time, shaders should minimize branching through if conditions. Path tracers are known for significant branching. Branches that conclude the shader prematurely are especially inefficient. In extreme cases, it is possible that a warp uses only one thread, effectively limiting the GPU performance to 1/32 of its maximum.
- Register Pressure: SMs have a limited number of registers. If a shader gets too complex and too many variables need to be kept alive at a time (register pressure), variables can spill into the caches increasing the cache load. As path tracers are often cache-limited, this can directly degrade performance. A wavefront path tracing architecture can be a solution for this issue, which might be an interesting option for this implementation.

NVIDIA Nsight is used for GPU profiling [26]. It provides an overview of the GPU performance and possible bottlenecks. Furthermore, the shader code can be profiled directly. The Nsight graphics debugger output usually looks like this:

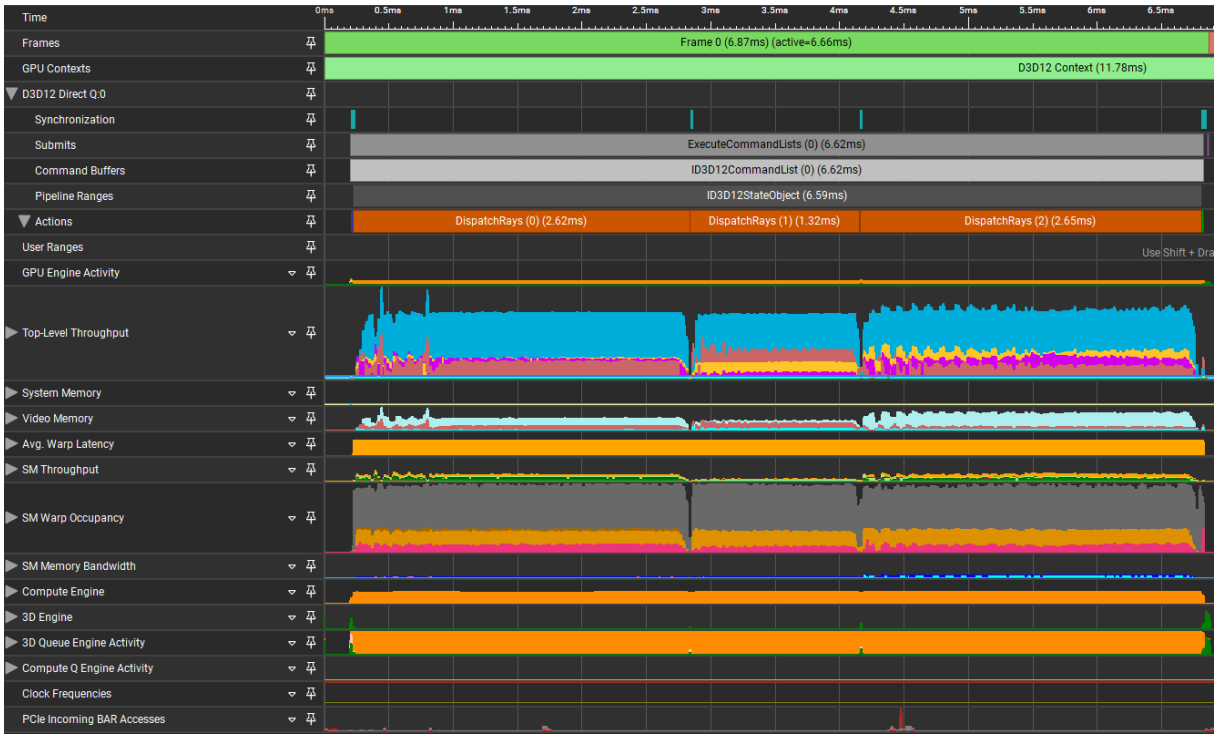


Figure 16: Nsight graphics profiler screen.

The most apparent indication of performance is the top-level throughput. The highest throughput category typically indicates a bottleneck. Unfortunately, ray tracing hardware core throughput is only available in this version of Nsight. Not much can be done to optimize ray hardware acceleration further without external libraries or reducing the number of traced rays, so this is the targeted bottleneck. In the example, the blue throughput graph indicates an L2 cache limitation. This is expected as ReSTIR requires a lot of memory for the reservoirs.

### 4.5.2 Minimizing reservoir size

Reducing the size of the reservoirs can result in significantly lower cache usage and reduced register pressure as reservoir structures are kept alive in the shader. A simple way to do this is to use half-precision variables in the storage for variables that do not require high precision. For the DI Reservoir, L2 and M can be replaced by half precision.

Another issue is that the smallest supported struct byte alignment in hlsl is 4 bytes, similar to C++. This means that the total number of bytes in the reservoir needs to be a multiple of 4. Lastly, GPUs load data packages at the minimum size of 16 bytes. Aligning variables to 16-byte packages can prove beneficial for performance, especially if the package contains complete variable representations. This is why the DI reservoir is ordered in a way that float3 variables are not split and align with the 16-byte rule.

```
1 struct Reservoir_DI
2 {
3     float3 x2;     float w_sum; // 16 bytes total
4     float3 n2;     float W;     // 16 bytes total
5     half3 L2;      uint16_t M;   // 8 bytes total
6 };
```

This reservoir structure results in a size reduction of 8 bytes and good memory alignment. As each pixel has 4 reservoirs and a sample data container, the size reduction is significant, especially as the L2 cache, sized 98mb on an RTX 5090 GPU, is unable to fully contain even two sets of reservoirs at 1080p resolution for the whole window as one set of reservoirs is 82mb.

### 4.5.3 L2 cache optimizations

Using raygen shaders for spatial reuse limits the controllability of warp dispatches of shaders. Typically, the dispatch is either performed tile-based or row/column major in lines of pixels. This poses a serious issue, as reservoir locations and order in the memory using UAVs are shader-defined.

To optimize memory coalescence, a number of different mapping methods were evaluated using spatial reuse with 3 neighbor candidates and equal rendering conditions:

- **Column Major Mapping:**

$$\text{pixelID} = x \times \text{resolution}.y + y$$

  This method places consecutive elements of a column contiguously in memory. It processes pixels down each column before moving to the next column.

- **Row Major Mapping:**

$$\text{pixelID} = y \times \text{resolution}.x + x$$

  This method places consecutive elements of a row contiguously in memory. It processes pixels across each row before moving to the next row.

- **Swizzling:** Uses small square tiles (e.g., $4 \times 4$ blocks) to preserve 2D locality in memory. The tile size determines how many pixels fit in each mini-block, grouping them to improve cache coherence when accessing neighboring pixels in both $x$ and $y$ directions depending on the GPU warp layout.

Here, x and y are the dispatch coordinates of the current pixel. The pixelID is a flattened representation of the 2d pixel position that is used as memory offset. The following table shows the frame time average of 10 frames per method:

Table 2: Frame time average in ms of 10 frames per method. Swizzling is tested with 1, 2, 4, 8 and 40 tilesizes

| Column maj. | Row maj. | SWx1 | SWx2 | SWx4 | SWx8 | SWx40 |
|---|---|---|---|---|---|---|
| 7.28 | 8.74 | 8.67 | 7.60 | 7.27 | 7.73 | 7.40 |

Swizzling with a tile size of 4 and column-major mapping provides the best performance. The tile size of 40 for SWx40 was chosen as the spatial reuse radius was set to 20 in this test.

### 4.5.4 Optimizing Active Threads per Warp

Typically, spatial reuse with a fixed number of neighbors and strict acceptance criteria produces uneven GPU warp loads resulting in few active threads per warp. A simple greedy algorithm was implemented to retry finding accepted neighbors for a limited number of tries.

This is significantly cheaper than fully evaluating the neighbors, especially with the formerly introduced L2 optimization.

```
foundCount = 0
for attempt in 0 .. max_tries-1 while (foundCount < candidate_count):
    pixel_n = GetRandomPixelCircleWeighted(...)
    if candidateAccepted_DI(pixel_r):
        spatial_candidates[foundCount] = pixel_n
        foundCountI++
```

Listing 8: Greedy neighbor search

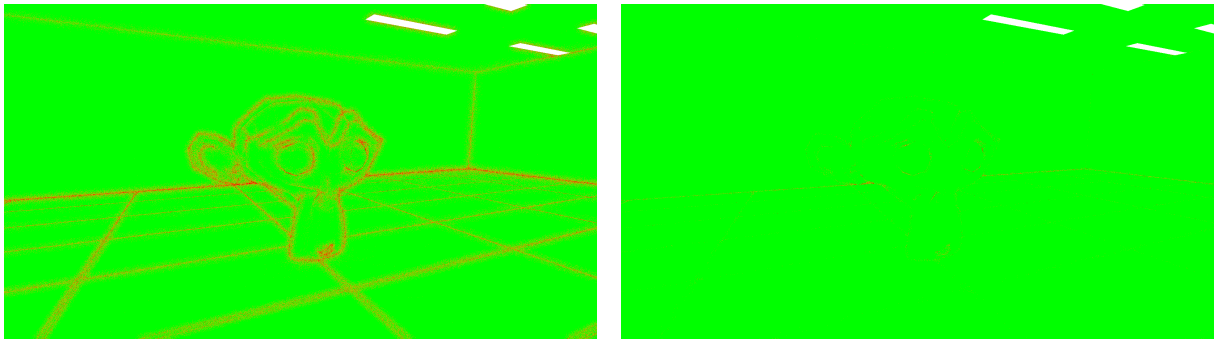

Figure 17: Left: No greedy neighbor search; Right: 3x greedy neighbor search, with negligible performance impact. Red indicates few matching neighbors, and green corresponds to the maximum number of neighbors matching.

Additionally, this reduces branching on the GPU as more threads per warp are active. Without greedy neighbor search, spatial reuse uses 60% of threads in a warp on average. Using the 3x variant by using a maximum of three times the spatial neighbor count as number of tries, 81% of threads in a warp are used on average. Additionally, the frame time decreased due to better GPU utilization from 7,55ms to 7,23ms on average despite a higher load.

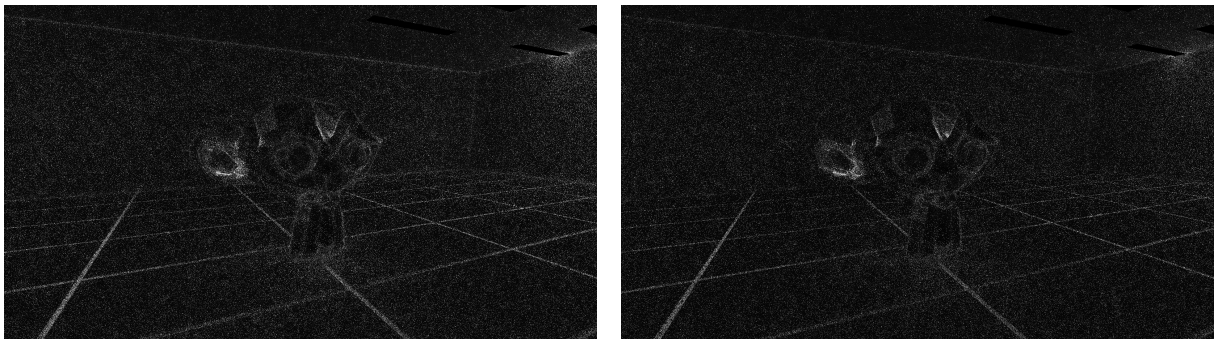

Figure 18: Display of the change in pixel error using the greedy neighbor search with only spatial reuse. The difference is multiplied by 5. Left: No greedy neighbor search; Right: 3x greedy neighbor search.

### 4.5.5 Reducing correlation artifacts

A significant disadvantage of spatiotemporal reuse is correlation artifacts and bright samples spreading spatially due to high variance. This is especially important in ReSTIR GI as the initial samples have a higher variance than in ReSTIR DI.

Generalized Resampled Importance Sampling [24] advises to only accept samples as valid spatial neighbors that satisfy a distance condition:

$$\min\big(\|x_2 - x_1\|, \|x_2 - y_1\|\big) \;\geq\; d_{\min} \tag{30}$$

where $x$ is the canonical sample and $y$ is the neighbor sample. This idea can be extended to the reconnection Jacobian, as it provides a similarity measurement between the shifted path and the base path. Paths with significantly different Jacobians tend to introduce stronger correlation artifacts. To mitigate this, this implementation uses the Jacobian of a reconnection to reject neighbor candidates. This can effectively reduce correlation artifacts at edges. As this is performed based on geometric information, it does not introduce bias.

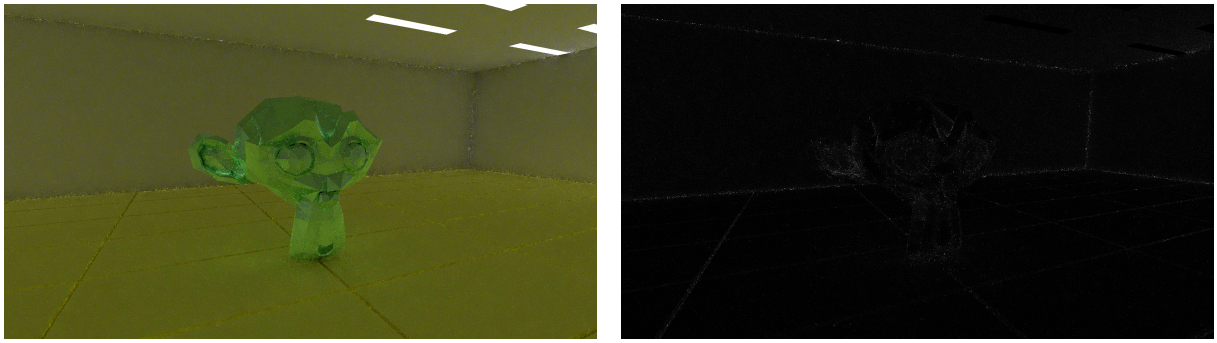

Figure 19: ReSTIR GI only, without using the Jacobian rejection. The difference is multiplied by 5.

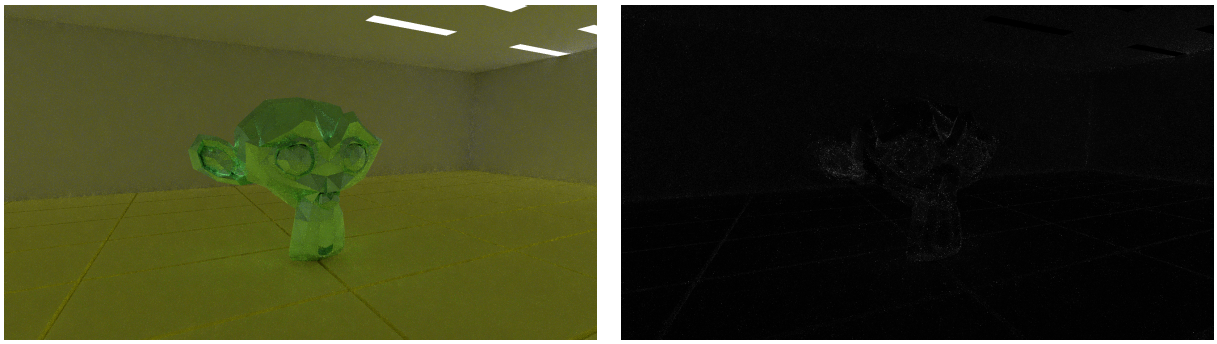

Figure 20: ReSTIR GI only, using the Jacobian rejection. The difference is multiplied by 5.

# 5 Experimental Setup

To assess the performance of the ReSTIR implementation, the following experimental setup was established.

## 5.1 GPU Setup

All experiments were conducted on an NVIDIA RTX 5090 GPU. Profiling and performance analyses were carried out using NVIDIA Nsight Systems, as previously discussed.

## 5.2 Scenes

To evaluate the effectiveness of the ReSTIR algorithm, two distinct test scenes were utilized. Each scene was selected to assess various aspects of rendering performance, including geometric complexity, lighting conditions, and material variations:

**Intel Sponza [32] and Stanford Dragon [33] in a difficult lighting scenario**

- **Complexity:** Approximately 18 million triangles, with the camera positioned inside the BVH, thereby increasing BVH traversal load.
- **Lighting Setup:** Several small spherical triangle light sources. This configuration results in suboptimal sampling of light triangles due to occlusions, creating a challenging lighting scenario.
- **Material Properties:** Diffuse Sponza environment combined with a specular material for the dragon.
- **Purpose:** Stress-testing the ReSTIR implementation under high geometric complexity with challenging lighting and material conditions.

**Garage and Blender Suzanne**

- **Complexity:** Approximately 3k triangles.
- **Lighting Setup:** A few area lights arranged optimally.
- **Material Properties:** Diffuse garage environment paired with a diffuse Suzanne material.
- **Purpose:** Reduced ray tracing workload and simpler lighting.

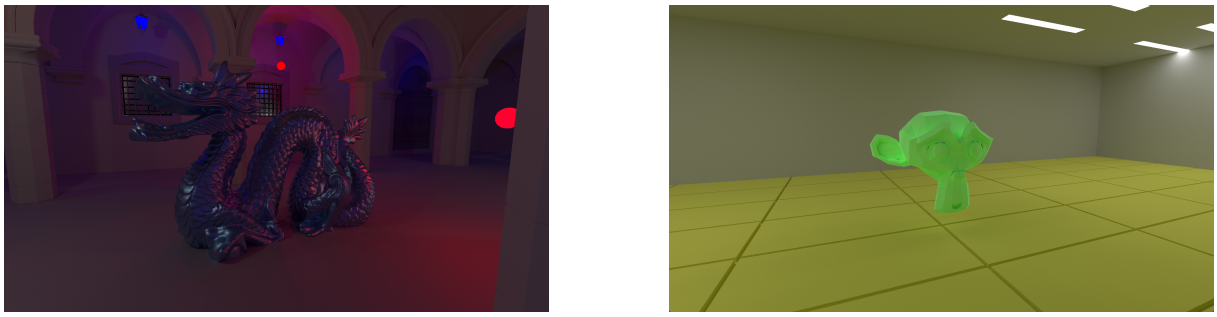

Figure 21: Ground truth render for the sponza and garage scene.

## 5.3 ReSTIR Configuration

For consistency and reproducibility, the ReSTIR algorithm was evaluated using standardized settings:

- **Temporal reuse parameters:** M-cap of 16.
- **Spatial reuse parameters:** M-cap of 128, 3 neighbor samples, and 9 total retries to populate the neighbor samples (3× greedy neighbor search).
- **Reservoir parameters:** Reservoirs optimized for size, with a total buffer size of 646.963 MB.
- **Sampling strategy:** Initial sampling uses the same features as the renderer employed for real-time comparison, except for additional RIS samples taken for direct lighting.

Any deviations are explicitly indicated in scenarios where individual aspects, such as neighbor count, are evaluated separately.

## 5.4 Error Metrics

Error evaluation for path tracers involves two primary components [11]:

- **Variance**: Measures the amount of noise present in a rendered image. Variance decreases as the number of samples increases.
- **Bias**: Represents the systematic deviation of a converged path tracer's output from the true ground-truth image.

Variance is commonly quantified using the Root Mean Squared Error (RMSE). The RMSE compares a noisy (unconverged) rendered image $I_{\text{noisy}}$ to the ground truth image $I_{\text{gt}}$. The RMSE is calculated as follows:

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(I_{\text{noisy}}(i) - I_{\text{gt}}(i)\right)^2}$$

where:

- $N$ is the total number of pixels in the image,
- $I_{\text{noisy}}(i)$ is the intensity value at pixel $i$ in the noisy image,
- $I_{\text{gt}}(i)$ is the intensity value at pixel $i$ in the ground truth image.

Determining bias is significantly more challenging. To evaluate bias, the converged render is compared against the converged ground truth. A practical method for identifying biased regions within an image is to visually compare it with the ground truth in combination with the RMSE. To simplify this process, the image can be enhanced to accentuate areas of bias. This enhancement is performed by setting each pixel of the resulting debug image to the local error multiplied by a scaling factor:

$$\text{error} = |\text{i}_{\text{noisy}} - \text{i}_{\text{gt}}| \times \text{scaling}$$

# 6 Evaluation and Discussion

This chapter presents a comparative analysis of the ReSTIR algorithm against MIS path tracing. For the test scenes introduced in Chapter 5, an evaluation of variance reduction, bias, and computational performance of ReSTIR will be conducted. Several different renderer configurations will be compared:

- **MIS PT**: regular path tracing with one NEE ray at each path vertex, weighted using MIS.
- **MIS PT + RIS**: included RIS at each path vertex to create multiple NEE samples cheaply. A singular shadow test is performed after sampling.
- **ReSTIR**: ReSTIR using MIS PT to create initial samples.
- **ReSTIR + GNSx3**: ReSTIR with additionally using greedy neighbor search and a retry factor of 3.
- **ReSTIR + RIS + GNSx3**: ReSTIR and GNS using the MIS PT + RIS samples as input.

The ground truth was rendered using MIS PT accumulated over several thousand frames.

## 6.1 Evaluation Methodology

The evaluation will focus on the following metrics:

- **Variance**: Variance is assessed through a comparative analysis between intermediate (unconverged) renders and the ground truth render. To illustrate convergence behavior effectively, the initial three rendered frames following scene loading will be examined. Additionally, Root Mean Square Error (RMSE) calculations for each render will quantitatively support the evaluation.

  Finally, a render produced using ReSTIR at its maximum sample capacity (M-cap) will be compared against the ground truth. This comparison targets the residual variance inherent to ReSTIR, which arises from deliberate trade-offs aimed at minimizing temporal correlation artifacts. The smaller the M-cap is set, the higher the minimum variance level is.

- **Bias**: The bias introduced by ReSTIR will be evaluated by directly comparing a fully converged ReSTIR-rendered image against the ground truth. While numerical metrics (RMSE) will provide quantitative insight, visual comparison remains essential to qualitatively identify any systematic deviations or artifacts indicative of bias.

- **Performance**: Given that ReSTIR exhibits higher per-frame computation times compared to traditional path tracing—primarily due to additional sampling and resampling steps—the variance reduction will be evaluated on a per-frame time basis to asses potential gains for real-time rendering.

## 6.2 Variance

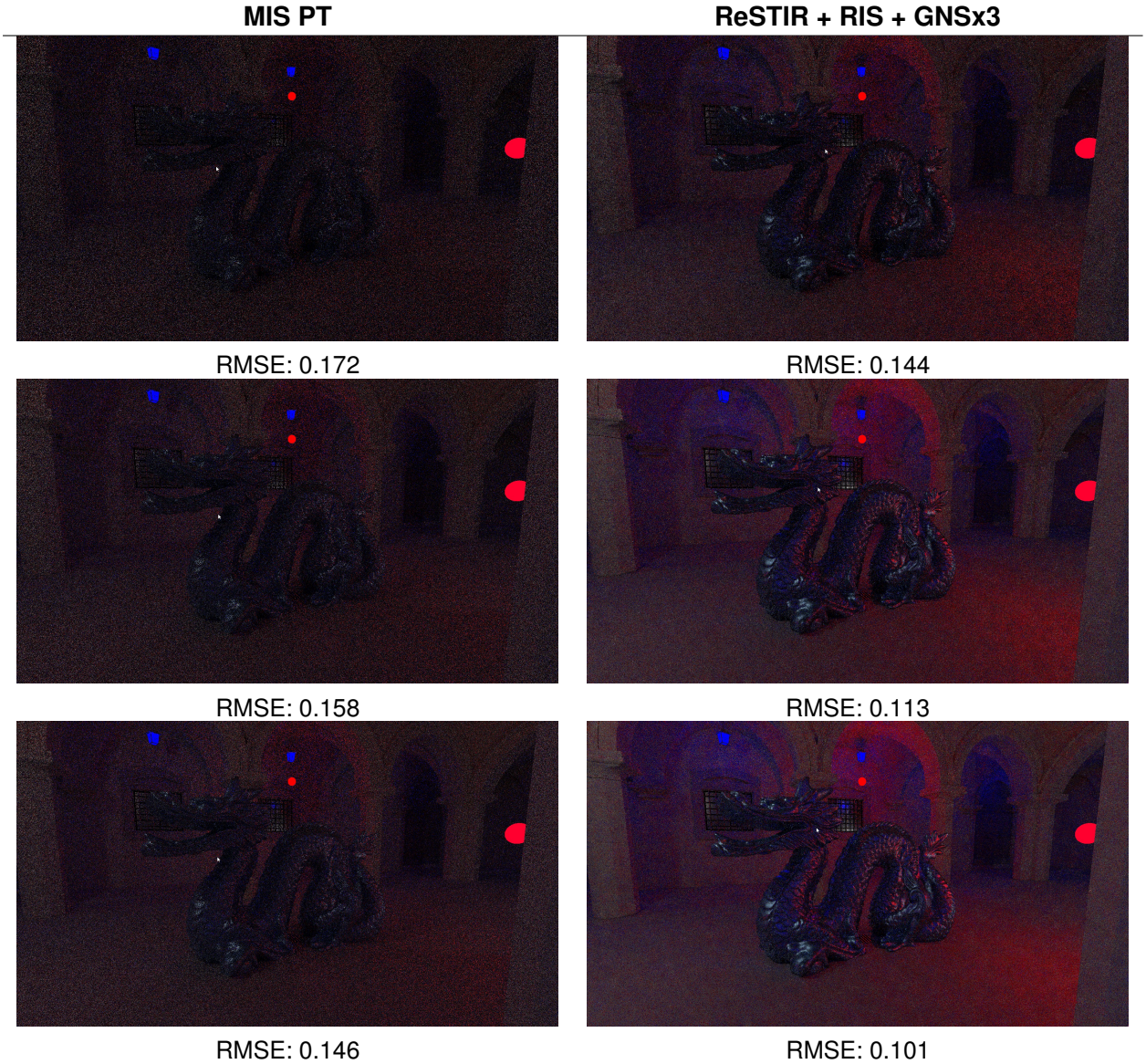


Figure 22: Side-by-side comparison of ReSTIR and a MIS path tracer. Due to pixels without light contribution, the MIS path tracer's output appears significantly darker.

Figures 22 and 23 show a detailed comparison of rendering convergence in the sponza scene. In the initial three frames after loading a scene (each column represents the first three frames after loading a scene, from top to bottom), the path tracer with MIS exhibits significantly higher visible variance, amplified by a large number of pixels missing a valid light sample altogether resulting in a black pixel. ReSTIR, on the other hand, produces brighter and more stable results

closer to the ground truth from the first frame due to spatiotemporal reuse. Numerically, these observations are supported by the RMSE values, which show a consistently faster decrease in variance using ReSTIR compared to the MIS approach.

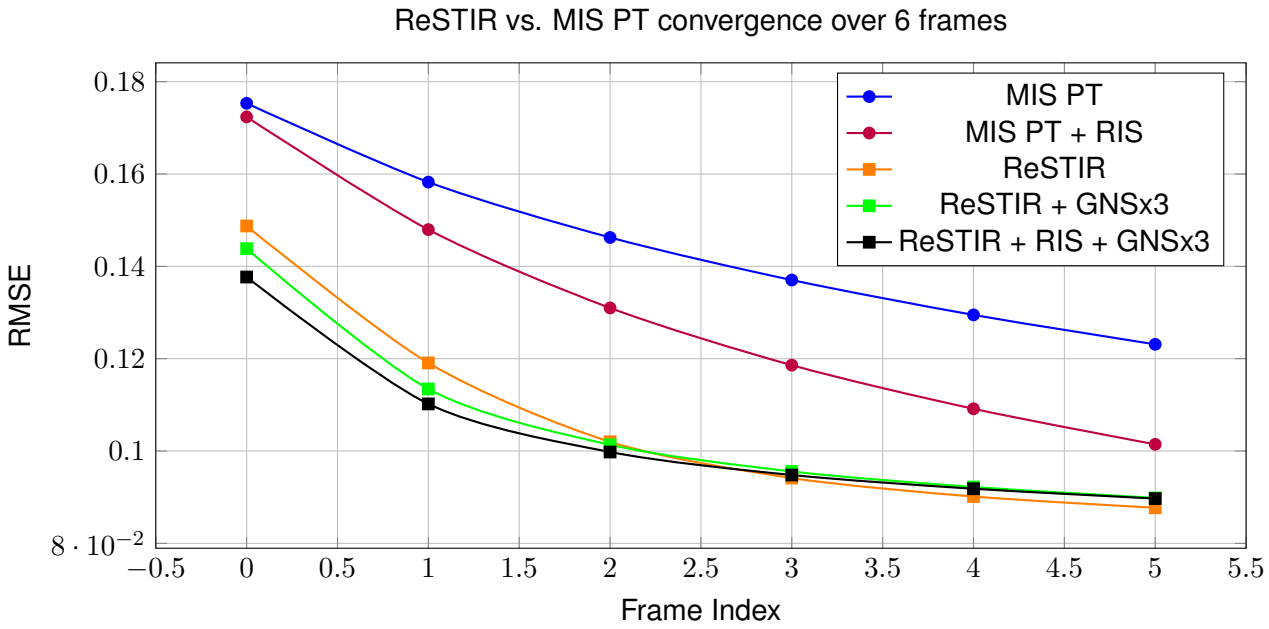


Figure 23: Convergence of the ReSTIR variants compared to MIS path tracing in the sponza scene.

In the convergence plot (Figure 23 top), it can be observed that both ReSTIR and ReSTIR with Greedy Neighbor Search (GNS) significantly reduce RMSE within the first few frames, compared to MIS path tracing and a more optimized approach that combines MIS with Resampled Importance Sampling (RIS). Notably, ReSTIR + GNSx3 slightly outperforms plain ReSTIR by converging to a lower RMSE more rapidly as early frames exhibit a higher number of invalid samples in surrounding pixels than later frames. Interestingly, later frames experience higher variance using the advanced ReSTIR variants. This is likely a result of color mismatch, as ReSTIR can only output one light sample color at a time. While the RMSE might be higher, the perceived noise is lower, as visible in figure 24

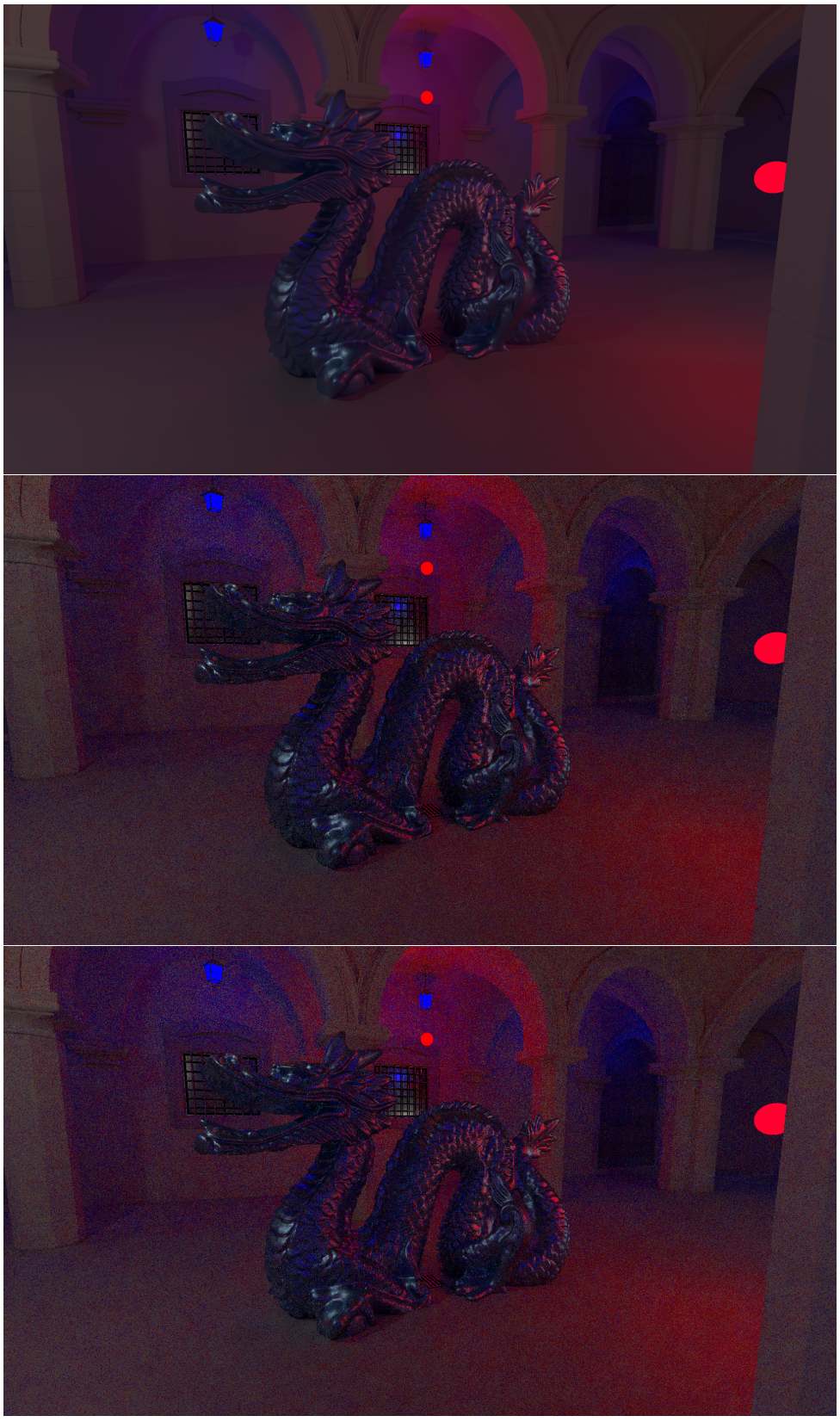

Figure 24: Top: ground truth; Middle: ReSTIR; Bottom: ReSTIR + RIS + GNSx3

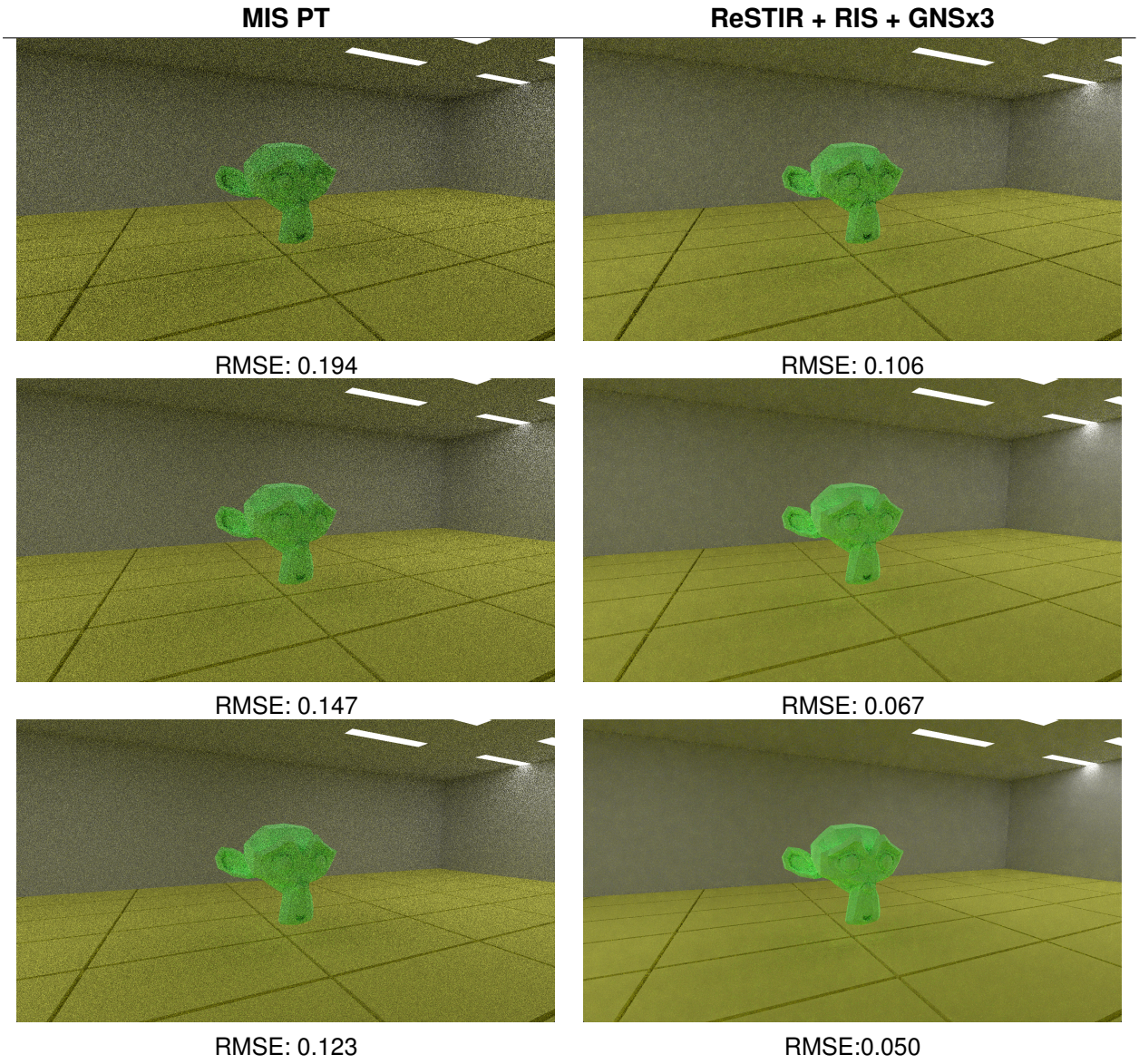


Figure 25: Side-by-side comparison of ReSTIR and the MIS path tracer used to create the initial samples for ReSTIR.

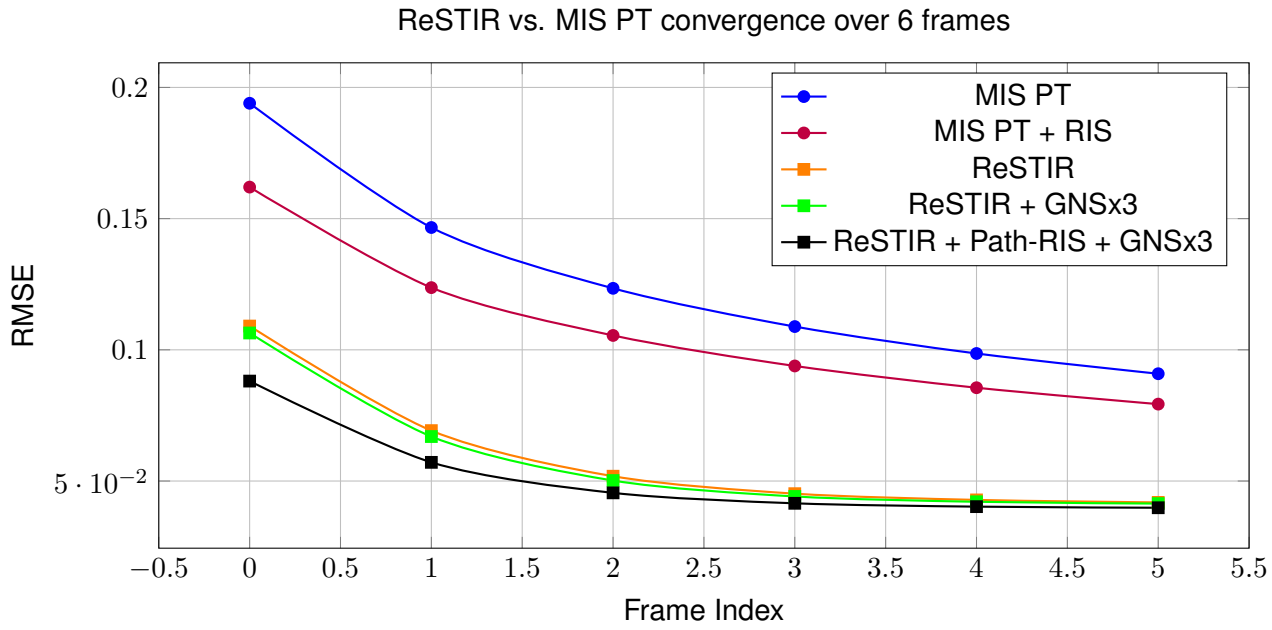


Figure 26: Convergence of the ReSTIR variants compared to MIS path tracing in the garage scene.

A similar trend is evident in the garage scene, illustrated in Figures 25 and 26. Again, MIS path tracing trails behind in early frames due to sparse and noisy sampling. Although the garage scene has different lighting conditions, the advantages of ReSTIR in reducing variance remain. Within the first three frames, ReSTIR achieves visibly smoother and more detailed renders; the corresponding RMSE plots confirm this, with ReSTIR and ReSTIR + GNSx3 tracking lower error values more quickly than the MIS baselines.

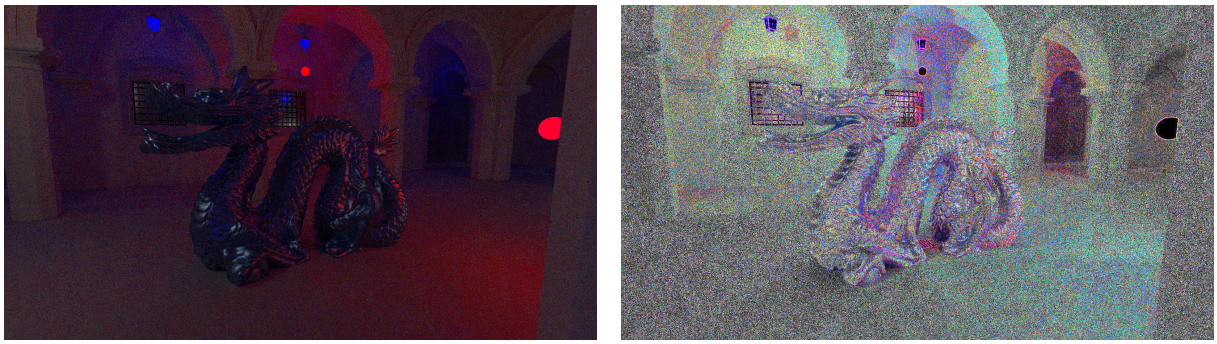

Figure 27: ReSTIR Maximum sample capacity vs. ground truth variance. Left: converged ReSTIR render; Right: visualization of the colored variance, scaled by a factor of 10. The minimal RMSE for ReSTIR with an M-cap of 16 is approximately 0.06.

Figure 27 further illustrates the near-converged ReSTIR result in the sponza scene by comparing a high-sample ReSTIR image to the ground-truth reference. The difference image on the right, scaled up by a factor of 10, shows that residual noise is mostly uniform, which indicates a stable noise level in all parts of the image. Numerically, the RMSE stabilizes to about 0.06 at maximum sample capacity (M-cap of 16). While a higher M-cap could provide reduced noise, correlation artifacts would be much more severe. As ReSTIR still requires a denoiser to be applied to the

final image, a higher variance is to be preferred as modern denoising solutions are not tuned to filter out strong correlation artifacts.

## 6.3 Bias

To assess bias, the fully converged ReSTIR image is compared against the ground truth in Figure 28. The difference image (scaled by a factor of 20 for visibility) shows minimal bias in diffuse regions while specular regions exhibit bias. This is expected as the biased variant of ReSTIR GI is used. In [9], a similar observation is made. In practice, this bias is not recognizable and is far outweighed by noise in real-time applications.

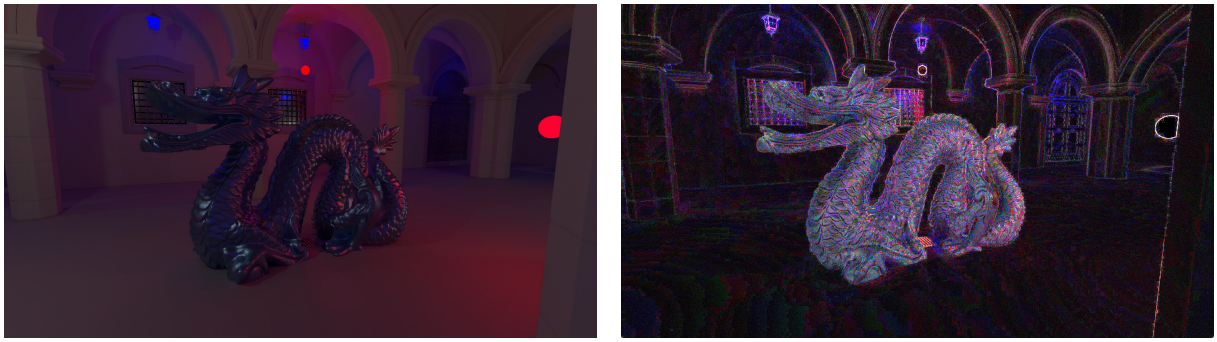

Figure 28: Left: Converged image of converged ReSTIR renders; Right: the difference between the ground truth and ReSTIR render, multiplied by 20.

## 6.4 Performance

In both the *sponza* and *garage* scenes, pairs of frames can be used to approximate equal frame times performances between ReSTIR and MIS PT. Table 3 and 4 show that ReSTIR consistently achieves a lower RMSE for the same overall render cost.

Table 3: RMSE comparison for matched compute times in the **sponza** scene. Each ReSTIR frame $i$ is paired with MIS frame $2i + 1$.

| Pair | ReSTIR RMSE | MIS RMSE |
|---|---|---|
| Frame 0 vs. Frame 1 | 0.1376 | 0.1582 |
| Frame 1 vs. Frame 3 | 0.1102 | 0.1370 |
| Frame 2 vs. Frame 5 | 0.0997 | 0.1231 |

Table 4: RMSE comparison for matched compute times in the **garage** scene. Each ReSTIR frame $i$ is paired with MIS frame $2i + 1$.

| Pair | ReSTIR RMSE | MIS RMSE |
|---|---|---|
| Frame 0 vs. Frame 1 | 0.0880 | 0.1466 |
| Frame 1 vs. Frame 3 | 0.0571 | 0.1088 |
| Frame 2 vs. Frame 5 | 0.0454 | 0.0908 |

Figure 29 visualize the difference in RMSE for ReSTIR and MIS given a similar compute budget. In the *sponza* scene, ReSTIR offers about a 20% reduction in RMSE on average compared to MIS under matched frame times; in the *garage* scene, the improvement is even more pronounced at around 40–50% lower RMSE.

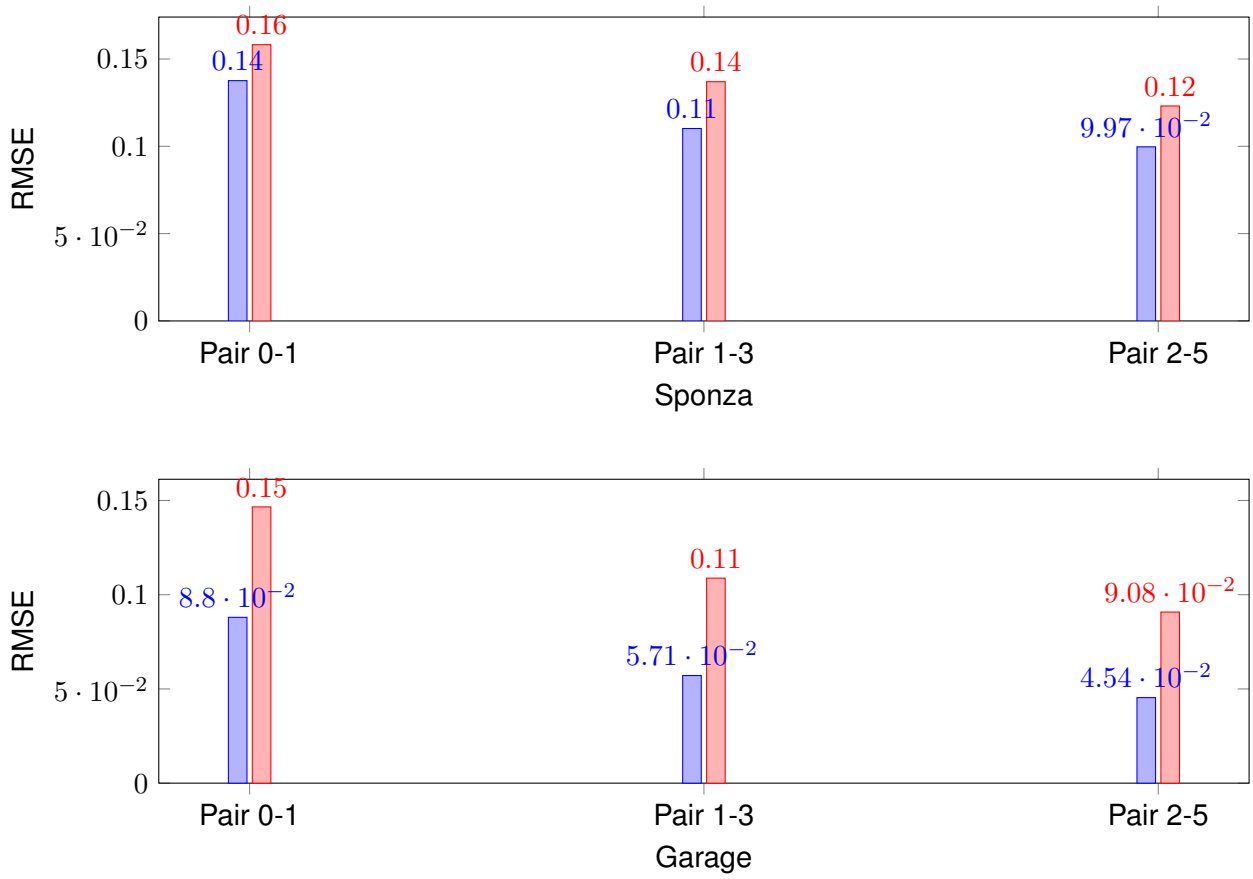


Figure 29: Comparison of ReSTIR and MIS renders for three matching-frame-time pairs in the *sponza* (top) and *garage* (bottom) scenes. Each pair is selected so that ReSTIR's frame $i$ has approximately the same compute cost as MIS's (red) frame $2i + 1$. ReSTIR (blue) yields significantly lower RMSE in all pairs.

# 7 Conclusion and Future Work

## 7.1 Summary

The goal of this thesis was to investigate how the real-time ray-tracing performance of a path tracer can be improved through software optimizations and hardware acceleration, specifically focusing on ReSTIR. The work presented:

- **Overview over Foundational Path-Tracing Techniques**
  A background on ray tracing and path tracing was provided, outlining how global illumination effects can be captured by simulating complex light transport paths in a scene. Methods such as BSDF importance sampling, next-event estimation, multiple importance sampling, and reservoir sampling were discussed to establish the groundwork for improved sampling strategies.
- **Introduced the theory behind ReSTIR**
  The thesis provided a step-by-step explanation of ReSTIR. ReSTIR takes advantage of resampled importance sampling and the concept of a reservoir that retains exactly one sample. By merging reservoirs spatially and temporally, ReSTIR effectively pools samples to lower variance.
- **Developed a DirectX 12 Ray-Tracing (DXR) Renderer**
  A prototype renderer was implemented in DirectX 12 DXR based on an NVIDIA DXR sample project. Key features, such as model loading and material preparation, were added.
- **Applied and Optimized ReSTIR.**
  The ReSTIR pipeline for direct and indirect lighting was integrated into the DXR renderer. In doing so, multiple optimizations were researched and employed:
  - Reservoir struct layout in GPU memory was optimized to reduce cache throughput.
  - Pairwise MIS was used in place of the typical balance heuristic to speed up spatial reuse.
  - Greedy neighbor searching (GNS) and geometry-based sample rejection were introduced to reduce correlation artifacts and improve active warp utilization.
- **Evaluated and Analyzed Performance and Quality**
  The system was benchmarked on two scenes of varying complexity (the Sponza scene with

a high triangle count and complex lighting, and a simpler garage scene) to test robustness. In both cases:

- ReSTIR significantly reduced variance compared to conventional path tracing, even when factoring in its higher per-frame cost.
- Greedy neighbor search (GNS) yielded slightly faster convergence by increasing the pool of valid neighbor samples.
- Some minor bias was observed in highly specular regions for the chosen ReSTIR GI variant, but was negligible in typical real-time viewing.

Overall, the results demonstrate that ReSTIR can provide substantially better visual fidelity in early frames of rendering while converging faster to low-noise images than a standard MIS-based path tracer, particularly in real-time contexts where each frame must be computed under time constraints. However, strong variance reductions as described in the ReSTIR papers [8, 9, 24] could not be measured, likely because the initial frame of reference in the papers is a naive path tracer, which also performs very poorly in comparison to NEE and MIS.

## 7.2 Key Findings

Although each ReSTIR pass requires additional memory transfers and reservoir resampling steps, the overhead is more than offset by the improved sampling efficiency. Empirically, the ReSTIR frames can be about 1.8–2.0 times more expensive than the baseline path tracer, but they often achieve lower noise levels within fewer frames of rendering. Real-time applications that maintain stable framerates can thus use ReSTIR to achieve higher visual quality under the same time budget, such as this implementation.

One known concern with spatiotemporal reuse is the introduction of correlation artifacts, particularly in scenes with sharp or glossy reflections. Strategies such as capping the reservoir's confidence weight (*M-cap*) and applying geometric acceptance criteria are shown to minimize these artifacts.

In a DXR environment, memory layout optimizations for reservoir data substantially impacted performance. Ensuring 16-byte alignment and minimizing reservoir size not only improved cache coherence but also helped avoid register pressure. Greedy sampling of neighbors further enhanced throughput. Profiling showed that the L2 cache limitations remain a bottleneck for large scenes, reinforcing the importance of reservoir compaction and further data-structure optimizations.

## 7.3 Limitations and Future Work

Although this thesis has shown promising results, multiple options remain for expanding the implementation. Future implementations could incorporate more sophisticated shift mappings for multi-bounce paths that respect material properties. These mappings can reduce bias but they must be carefully designed to avoid excessive temporal artifacts or performance overhead in real-time scenarios [24].

Modern real-time renderers often employ post-process denoisers, most notably the NVIDIA Realtime Denoiser (NRD) and NVIDIA DLSS ray reconstruction. Investigating how ReSTIR's distribution of noise interacts with these denoisers could reveal further improvements, possibly allowing for fewer denoiser iterations or lower spatiotemporal smoothing thresholds [34].

This work focused primarily on rigid objects with transformations. In highly dynamic scenes with changing topology (e.g., animated characters), one must rebuild or refit BLAS structures frequently. Recently, NVIDIA introduced "Mega Geometry" [35], which improves BVH performance for animated models. Studying ReSTIR's effectiveness in these scenarios is a possible future work.

In the current implementation, reservoirs are handled as an array of structures. As this can lead to memory divergence, a structure of arrays approach could be helpful for reducing the significant L2 cache throughput [31].

# Bibliography

[1] Wikipedia contributors, “Rendering equation — Wikipedia, The Free Encyclopedia,” https://en.wikipedia.org/wiki/Rendering_equation, accessed: 15 Apr 2025.

[2] M. Documentation, “Plugins: Bsdfs,” https://mitsuba.readthedocs.io/en/stable/src/generated/plugins_bsdfs.html, accessed: 15 Apr 2025.

[3] M. Corporation, “DirectX Raytracing (DXR) Functional Specification,” https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html, accessed: 15 April 2025.

[4] NVIDIA, “Nvidia RTX blackwell GPU architecture white paper,” https://images.nvidia.com/aem-dam/Solutions/geforce/blackwell/nvidia-rtx-blackwell-gpu-architecture.pdf, accessed: 15 Apr 2025.

[5] J. E. Bresenham, “Algorithm for computer control of a digital plotter,” *IBM Systems Journal*, vol. 4, no. 1, pp. 25–30, 1965.

[6] T. Whitted, “An improved illumination model for shaded display,” *Commun. ACM*, vol. 23, no. 6, p. 343–349, Jun. 1980. [Online]. Available: https://doi.org/10.1145/358876.358882

[7] J. T. Kajiya, “The rendering equation,” *SIGGRAPH Comput. Graph.*, vol. 20, no. 4, p. 143–150, Aug. 1986. [Online]. Available: https://doi.org/10.1145/15886.15902

[8] B. Bitterli, C. Wyman, M. Pharr, P. Shirley, A. Lefohn, and W. Jarosz, “Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting,” *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, vol. 39, no. 4, Jul. 2020.

[9] Y. Ouyang, S. Liu, M. Kettunen, M. Pharr, and J. Pantaleoni, “Restir gi: Path resampling for real-time path tracing,” *Computer Graphics Forum*, vol. 40, pp. 17–29, 11 2021.

[10] J. Talbot, D. Cline, and P. Egbert, “Importance Resampling for Global Illumination,” in *Eurographics Symposium on Rendering (2005)*, K. Bala and P. Dutre, Eds. The Eurographics Association, 2005.

[11] E. Veach, “Robust monte carlo methods for light transport simulation,” Ph.D. dissertation, Stanford, CA, USA, 1998, aAI9837162.

[12] M. Pharr, W. Jakob, and G. Humphreys, *Physically Based Rendering: From Theory to Implementation*, 4th ed. Cambridge, MA: The MIT Press, 2023.

[13] W. Jarosz, “Efficient monte carlo methods for light transport in scattering media,” Ph.D. dissertation, University of California, San Diego, 2008, appendix A available at https://cs.dartmouth.edu/~wjarosz/publications/dissertation/appendixA.pdf.

[14] R. Tamstorf and H. W. Jensen, “Adaptive sampling and bias estimation in path tracing,” in *Rendering Techniques ’97*, J. Dorsey and P. Slusallek, Eds. Vienna: Springer Vienna, 1997, pp. 285–295.

[15] J. Dupuy and W. Jakob, “An adaptive parameterization for efficient material acquisition and rendering,” *Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, vol. 37, no. 6, pp. 274:1–274:18, Nov. 2018.

[16] M. Oren and S. K. Nayar, “Generalization of lambert’s reflectance model,” in *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH ’94. New York, NY, USA: Association for Computing Machinery, 1994, p. 239–246. [Online]. Available: https://doi.org/10.1145/192161.192213

[17] R. Cook and K. Torrance, “A reflectance model for computer graphics,” *ACM Trans. Graph.*, vol. 1, pp. 7–24, 01 1982.

[18] “OpenPBR Standard,” https://academysoftwarefoundation.github.io/OpenPBR/, accessed: 15 Apr 2025.

[19] T. S. Trowbridge and K. P. Reitz, “Average irregularity representation of a rough surface for ray reflection,” *Journal of the Optical Society of America (1917-1983)*, vol. 65, no. 5, p. 531, May 1975.

[20] E. Turquin, “Practical multiple scattering compensation for microfacet models,” 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:221737278

[21] P. Shirley, C. Wang, and K. Zimmerman, “Monte carlo techniques for direct lighting calculations,” *ACM Trans. Graph.*, vol. 15, no. 1, p. 1–36, Jan. 1996. [Online]. Available: https://doi.org/10.1145/226150.226151

[22] M. A. Tanner and W. H. Wong, “The calculation of posterior distributions by data augmentation,” *Journal of the American Statistical Association*, vol. 82, no. 398, pp. 528–540, 1987. [Online]. Available: http://www.jstor.org/stable/2289457

[23] C. Wyman, M. Kettunen, D. Lin, B. Bitterli, C. Yuksel, W. Jarosz, and P. Kozlowski, “A gentle introduction to restir path reuse in real-time,” in *ACM SIGGRAPH 2023 Courses*, ser. SIGGRAPH ’23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3587423.3595511

[24] D. Lin, M. Kettunen, B. Bitterli, J. Pantaleoni, C. Yuksel, and C. Wyman, "Generalized resampled importance sampling: foundations of restir," *ACM Trans. Graph.*, vol. 41, no. 4, Jul. 2022. [Online]. Available: https://doi.org/10.1145/3528223.3530158

[25] NVIDIA, "Directx 12 raytracing tutorial, part 2," https://developer.nvidia.com/rtx/raytracing/dxr/dx12-raytracing-tutorial-part-2, accessed: 15 Apr 2025.

[26] NVIDIA, "NVIDIA Nsight Graphics," https://developer.nvidia.com/nsight-graphics, accessed: 15 Apr 2025.

[27] T. Aila and S. Laine, "Understanding the efficiency of ray traversal on gpus," in *Proceedings of the Conference on High Performance Graphics 2009*, ser. HPG '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 145–149. [Online]. Available: https://doi.org/10.1145/1572769.1572792

[28] "Tinyobjloader: A single file wavefront obj loader," https://github.com/tinyobjloader/tinyobjloader, accessed: 15 Apr 2025.

[29] L. Yang, S. Liu, and M. Salvi, "A survey of temporal antialiasing techniques," *Computer Graphics Forum*, vol. 39, no. 2, pp. 607–621, July 2020.

[30] NVIDIA Developer Blog, "Improve shader performance and in-game frame rates with shader execution reordering," https://developer.nvidia.com/blog/improve-shader-performance-and-in-game-frame-rates-with-shader-execution-reordering/, accessed: 15 Apr 2025.

[31] ——, "RTX Best Practices," https://developer.nvidia.com/blog/rtx-best-practices/, accessed: 15 Apr 2025.

[32] Intel Corporation, "Intel Graphics Samples," https://www.intel.com/content/www/us/en/developer/topic-technology/graphics-research/samples.html, accessed: 15 Apr 2025.

[33] Stanford University, "Stanford 3D Scanning Repository," https://graphics.stanford.edu/data/3Dscanrep/, accessed: 15 Apr 2025.

[34] NVIDIA Blog, "Ai decoded: Ray reconstruction," https://blogs.nvidia.com/blog/ai-decoded-ray-reconstruction/, accessed: 15 Apr 2025.

[35] NVIDIA, "NVIDIA RTX Kit," https://developer.nvidia.com/rtx-kit?sortBy=developer_learning_library%2Fsort%2Ftitle%3Aasc, accessed: 15 Apr 2025.