Mathematisch-
Naturwissenschaftliche
Fakultät

Computergrafik

Masterarbeit

Advanced Path Tracing with NVIDIA RTX

Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik
Computergrafik
Felix Scholl, felix.scholl@student.uni-tuebingen.de, 2020

Bearbeitungszeitraum: 22.6.2020-22.12.2020

Betreuer/Gutachter: Prof. Dr. Hendrik Lensch, Universität Tübingen
Zweitgutachter: Prof. Dr. Andreas Schilling, Universität Tübingen

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Masterarbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Felix Scholl (Matrikelnummer 3927630), April 18, 2021

Abstract

Hardware accelerated ray tracing is now possible with the NVIDIA RTX and the newly released AMD RDNA 2 GPUs.
This master thesis implements common and advanced path tracing techniques while utilizing the new ray tracing hardware acceleration, and evaluating their performance and quality. The evaluated techniques are Next Event Estimation, Multiple Importance Sampling, Irradiance Caching, Adjoint-drive Russian Roulette and Splitting, and Parallax-aware Path Guiding.
Also included is a wide range of visualization modes that can be enabled interactively to better understand the inner workings of the implemented techniques. This, in combination with the way higher performance than CPU based path tracer and resulting fast convergence rate, allows for interactive comparisons between the different methods.

Acknowledgments

I would like to thank Lukas Ruppert for his technical insights, continuous support during the development, and proofreading.
I would also like to thank Prof. Dr.-Ing. Lensch for the opportunity to create this master thesis as well as for providing the NVIDIA RTX 2080Ti.

Contents

1 Introduction

Generating images from complex scenes that accurately depict global illumination is the driving force behind many developments in computer graphics. To precisely model global illumination, one has to evaluate the rendering equation [Kaj86] accurately, which is most often achieved with Monte Carlo methods like path tracing.

One obstacle of path tracing is its low performance when compared to typical rasterization approaches, that in contrast with ray tracing, are in large parts implemented in specialized hardware on all GPUs.
In 2018, NVIDIA announced their new Turing architecture [NVI18] starting with the new RTX graphics cards. They contain new ray tracing cores (RT cores), that are supposed to accelerate ray tracing to levels that allow for real-time ray tracing applications. AMD, not being far behind, also just released their newest RDNA 2 GPUs [AMD20] containing their version of hardware ray tracing acceleration.

These RT cores take a specialized scene representation in the form of an acceleration structure and perform bounding box hierarchy traversals and triangle intersections in hardware. This, in combination with new programmable ray tracing shader constructs, allows for fast implementations of path tracers and existing advanced path tracing methods.

This master thesis aims to implement common and advanced methods, that can be used to enhance path tracers, on the GPU utilizing the Vulkan ray tracing extension for RTX accelerated ray tracing.
The implemented techniques are Next Event Estimation, Irradiance Caching [WRC88], Adjoint-driven Russian Roulette and Splitting [VK16], and Parallax-aware Path Guiding [RHL20].

These methods will then be evaluated to analyse whether they work well on the RTX enabled GPU implementation.

2 Related Work

The idea of hardware based ray tracing is not new and was first designed as custom hardware in the form of SaarCor [SWS02]. Their hardware ray tracing unit consisted out of three components: Ray generation and shading, ray tracing cores (RTC), and RTC-memory managers.
The RTCs performed the acceleration structure traversal and returned the result to the shading unit for color calculations. Their system was built in such a way, that the number of RTCs could be scaled to handle more rays at once.
SaarCor was only a proof of concept and only existed as a simulation, yet it portrays many ideas also used in the modern RTX architecture.

With the rise of the RTX hardware, NVIDIA also presented some fully ray traced games to highlight the importance of their newest development.

Minecraft RTX [NVI20c] was created in collaboration between NVIDIA and Microsoft and aimed to make Minecraft a fully path traced game.
Their implementation used Irradiance Caching for global illumination, by keeping Irradiance Caches for each vertex and face, and updating them asynchronously to achieve light transfer between different parts of the scene. Of course, just like all fully path traced games, the final image would still look very noisy, as there is simply not enough time to generate enough samples. To work around this issue, a denoising step was added, that produced high quality images from the very noisy and low samples per pixel input images. This denoising step is computationally expensive and takes 45% of the frame time as shown in their in-depth explanation [NVI20b].
In the end, they achieved a very impressive tech demo of all the things enabled through real-time ray tracing, like perfect reflections or accurate global illumination.

Another project the released early on was Quake II RTX, a path traced implementation of the 1993 game Quake II. It started as a personal project by Christoph Schied as Q2VKPT [Sch19], with the goal of creating a fully path traced version of the 1993 game. This was then further expanded by NVIDIA [NVI19] and contains fully path traced real-time graphics, where the only part utilizing the rasterization pipeline is the UI. Their focus was on impressive lighting effects and true indirect illumination created by the sun or projectiles and explosions caused by the player.
Just as Minecraft RTX, a big part of the development effort was spend on the denoising step that creates great images out of the unusable and noisy path traced results.

For real time ray tracing applications, denoising of the results after just a few samples

is an important topic to create noise-free and visually pleasing results.

Spatiotemporal Variance-Guided Filtering [SSK+17] aims to reconstruct a temporally stable image from just one sample per pixel. Their implementation takes direct and indirect light contributions and then removes the albedo of the surfaces to filter the light contribution and not some high frequency texture components. The direct and indirect light images are then separately filtered using data from previous frames in combination with motion data to perform temporal accumulation, which then get filtered using variance controlled wavelet filters. After recombining direct and indirect light, and adding the material albedos again, a final temporal antialiasing step is done.
This resulted in a temporally stable filter, that only took around 10 ms for 1920×1080, not including the ray tracing time. As their approach was created prior to the RTX hardware, they had to make some concessions like reducing their ray depth to only one bounce to achieve reasonable performance.

To reduce variance in the image, finding paths to the light sources is important, which is where path guiding comes into play. Path Guiding aims to guide rays along those paths, where they can actually receive light contributions. This is especially useful for scenes with complex light placements, e.g. a small light source behind glass, as otherwise most rays will never contribute any light.
To guide the rays along paths that can contribute light, representations of the local radiance need to be constructed. This can be done by collecting samples, i.e. what direction light was received from, in a region and fitting a distribution around the samples. These distributions can then be efficiently sampled to guide rays in the right directions.

Parallax-aware path guiding [RHL20] goes one step further and also collects the targets of the samples and uses them to adjust the distribution based on the position in the region as to always point at the location the light was received from and not just the direction, which leads to sharper distributions that stay valid for larger regions.
Parallax-aware path guiding is further explored in Sections 3.2.5 and 3.2.8.

A similar topic is explored in Temporal Sample Reuse for Next Event Estimation and Path Guiding [DHD20]. It uses information from previous frames to make the few samples that can be collected for real-time path tracing more efficient. They present two techniques that improve the importance sampling for these GPU based real-time path tracers.
They create Visibility Light Caches, information about which lights can be sampled with Next Event Estimation (NEE) from this location, and store them in a scene-spanning octree. These VLC can then be used to mainly sample those light sources, that are actually visible at the current location, which is especially important for scenes with many light sources. Their implementation also contains techniques to make the update process of this visibility data as fast as possible.
The other technique introduced is a path guiding approach to direct the rays into

directions of large indirect light contributions. This guiding information is stored in Compressed Directional Quadtrees that store approximations of the indirect radiance received in a compact structure, that can be easily enhanced over multiple frames.

The paper Spatiotemporal Reservoir Resampling [BWP+20] proposes a method to increase the direct light sampling performance in scenes with thousands to millions of light emitting triangles.
There, each pixel collects some random light samples and stores their computed expected contributions in a so called reservoir. To improve the number of samples used, a form of spatial accumulating is used to select k other reservoirs in the neighborhood, as well as reusing the reservoirs of previous frames, as they should generally portray a similar behaviour.
These reservoirs can then be sampled efficiently to prefer those light sources that are expected to have high contributions, and only perform the costly shadow ray generation for a small number of samples. The resulting images, with only direct light contributions, are of comparably high quality when compared to other methods and perform especially well in combination with an OptiX denoiser [PBD+10].

Other projects that utilize the new RTX hardware are for example Mitsuba 2 [NDVZJ19] or real-time ambient occlusion [Gau20].

Mitsuba 2 is a research oriented path tracer and rendering system that contains many possible rendering modes in addition to standard path tracing, like differential rendering, inverse rendering, or renders that take spectral properties or polarization into account.
Recently, support for NVIDIA OptiX [PBD+10] based ray tracing, which supports the new RTX hardware acceleration, was added to Mitsuba 2. This enables interesting new research opportunities due to the integration in this research based renderer.

Ambient occlusion (AO) is a widely used technique in games or CAD applications to simulate some global illumination properties that result from local occlusions. The paper [Gau20] proposes a real-time ray-traced ambient occlusion that works even for complex scenes. For this, they utilize the RTX ray tracing technology to generate exact ambient occlusion rays and store the resulting distance in a simple array at a position defined by spatial hashing.
This is done by rounding the position to a certain resolution and hashing the x/y/z components in combination with this resolution factor. This results in all primary rays in a certain small region adding their distance samples to the same array location, resulting in a better AO estimate, without having to manage a complex data structure. This estimate can then easily be accessed using the spatial hashing to retrieve the AO values for the query location.
The paper also introduces multiple resolution levels that all get stored in the same array as well as filtering options for better initial results, resulting in fast and accurate ambient occlusion, while drastically reducing the amount of data that needs to be stored.

The optimal structure of path tracers that use the new hardware accelerated ray tracing is still a topic worth some research. Researchers at NVIDIA presented their recommendation to use a wavefront-based approach to path tracing in the paper [LKA13].
This is an alternative to the classical *Mega kernel* approach that loops over the ray depth and handles material interactions as well as new ray creation in one central kernel. Their reasoning is, that this approach leads to frequent warp divergence triggered by control flow divergence, e.g. different materials or rays leaving the scene, which results in warps needing to serialize work.

In contrast, the wavefront approach [LKA13] splits these ray paths into different segments to achieve higher coherence. The kernel only ever traces one step of the ray path and stores the complete ray state in memory afterwards, as well as putting the material interaction as a separate task into a material specific queue. These different queues can then be processed independently and, due to being very similar materials, with minimal warp divergence. This is especially useful for complex multi-layer materials that would otherwise result in wait times for other rays in the warp.
The wavefront approach also aims to always processes a constant workload, as the number of rays in flight is always held constant by immediately creating new rays for all those that were terminated.
The goal of wavefront path tracing is to create a coherent workload with fully utilized warps as well as reducing the amount of registers needed.

3 Background

Whitted [FW80] introduced recursive ray tracing in 1980, where he proposed generating rays for each pixel and creating new reflection, refraction, or shadow rays from the first surface along the initial ray. This enabled him to simulate specular reflection, refraction, and accurate shadows, but was limited in the ray depth and had no way to handle scattering between diffuse surfaces or the effects of area lights. Further research [CPC84, Kaj86] lead to path tracing, which is a technique that can accurately generate these effects.

3.1 Path Tracing

The goal of path tracing is to create an accurate representation of global illumination. For this goal, an equation is needed that describes all these global illumination effects. This is called the rendering equation.

3.1.1 Rendering Equation

The outgoing radiance L_o at a point x in direction ω_o is a combination of the emission $L_e(x, \omega_o)$ and the reflected radiance $L_r(x, \omega_o)$:

$$L_o(x, \omega_o) = L_e(x, \omega_o) + L_r(x, \omega_o) \tag{3.1}$$

The reflected radiance from an incoming direction ω_i is dependent on the surface reflection properties (BRDF/BSDF) $f(\omega_i, x, \omega_o)$, the angle of ω_i to the surface, and the actual incoming light $L_i(x, \omega_i)$ [Kaj86]

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega f(\omega_i, x, \omega_o) L_i(x, \omega_i) \cos\theta_i \, d\omega_i \tag{3.2}$$

Of course, the incoming radiance $L_i(x, \omega_i)$ has to be an outgoing radiance at another location. In this case the location of the first object hit from x in direction ω_i

$$L_i(x, \omega_i) = L_o(y, -\omega_i) \tag{3.3}$$

The goal of any physically-based render is to solve the rendering equation for each pixel of the output to compute the final image. This is not analytically solvable as

the function is self-referential. This means, the radiance value for one location is dependent on all other radiance values that can contribute light to this location, which in itself are again dependent on all other radiance values.
Therefore, a numerical approximation is the only option for arbitrary scenes and materials [PJHa].

3.1.2 Monte Carlo

The goal of Monte Carlo simulations is to approximate a problem numerically, which is difficult or impossible to solve analytically, by taking a random set of discrete samples of the function and creating a (weighted) average of them. The *law of large numbers* stipulates, that the average result of many such samples should be close to the expected value of the original function.

In this case, we want to evaluate the complex and self referential rendering equation with a Monte Carlo simulation.

A multidimensional integral

$$I = \int_{\Omega} f(x)\,dx \tag{3.4}$$

can be approximated by evaluating N uniform samples $x_i \in \Omega$ with $p(x_i) = p(x_j) = p$ [NB99]

$$I \approx \frac{1}{Np} \sum_{i}^{N} f(x_i) \tag{3.5}$$

Importance Sampling

An extension of MC is importance sampling, a technique to reduce variance. Here, the x_i are not uniformly distributed, but distributed according to some probability distribution $p(x)$ that matches the shape of $f(x)$ better than a uniform distribution.

$$I \approx \frac{1}{N} \sum_{i}^{N} \frac{f(x_i)}{p(x_i)} \tag{3.6}$$

This only holds, if $p(x) \neq 0\ \forall x : f(x) \neq 0$, so if we actually fully sample the function's non-zero value parameter range.

The variance of $\frac{f}{p}$

$$\mathbf{V}\left(\frac{f}{p}\right) = \mathbf{E}\left[\left(\frac{f}{p}\right)^2\right] - \mathbf{E}\left[\left(\frac{f}{p}\right)\right]^2 \tag{3.7}$$

would be lowest, if $p \sim f$. This is of course often not easily achievable, due to the complex nature of f, but the closer the shape matches, the lower the variance.

For path tracers, this is most often used to sample according to the properties of the local material BRDF, e.g. on a very glossy surface, it is more efficient to create new rays close to the reflection direction, as all other directions have very low BRDF values that wouldn't contribute much if at all. This of course does not represent the shape of the incoming radiance at all, which might result in increased variance for some situations.

Monte Carlo approximation of the rendering equation

A Monte Carlo approximation of the rendering equation [Kaj86] can be described as:

$$L_o(x,\omega_o) \approx L_e(x,\omega_o) + \frac{1}{M}\sum_k^M \frac{f(\omega_{i,k},x,\omega_o)\,L_i(x,\omega_{i,k})\cos\theta_i}{p(\omega_{i,k})} \tag{3.8}$$

$$= L_e(x,\omega_o) + \frac{1}{M}\sum_k^M \frac{f(\omega_{i,k},x,\omega_o)\,L_o(y_k,-\omega_{i,k})\cos\theta_i}{p(\omega_{i,k})} \tag{3.9}$$

To actually calculate the approximation of the radiance L_o towards the camera, we need to send out rays from the camera and check for the closest intersection x with the scene. There, the local emission has to be queried and another, possibly set of, ray(s) created to again evaluate the reflective term of the radiance L_r, which again depends on L_o itself. So, M new rays in random directions have to be created and again intersected with the scene, which results in the location y.
Now, a recursive evaluation of $L_o(y,-\omega_i)$ is necessary.

When this method is used to approximate global illumination, and therefore approximating the rendering equation, it is commonly called *path tracing* [Kaj86] and can be described in pseudo-code as:

```
color pathTrace(point x, vector wo) {
    color Li = 0;

    for (int m = 0; m < M; m++) {
        vector wi = randomDirection(x, wo);
        float p = p(wi, x, wo);

        vector y = rayIntersection(x, wi);

        Li += brdf(wi, x, wo) * pathTrace(y, -wi)
              * cosTheta(x, wi) / p;
    }

    return Le(x, wo) + Li / M;
}
```

Of course, a real path tracer would need some extra cases for ray misses or materials with special properties like dielectrics or specular surfaces.

3.2 Advanced Path Tracing Methods

3.2.1 Next Event Estimation

With ordinary path tracing, the emission term L_e is mostly zero, and only in the rare occasion, when a light is hit, an actual contribution for that path is added. This creates a lot of variance, as whether a path actually contributes light is solely dependent on randomly hitting a light somewhere in this path.

To decrease the variance, Next Event Estimation can be used. Instead of querying the local emission, which is mostly zero, a random location on a random light is selected and checked for occlusion. If it is not occluded, this contribution is added to L_o. This basically skips the emission check at the current location and instead queries the emission one bounce further.

We can reconstruct, why this results in the same radiance values, by restructuring the rendering equation (3.2)

$$\begin{aligned}
L_o(x,\omega_o) &= L_e(x,\omega_o) + L_r(x,\omega_o) \\
&= L_e(x,\omega_o) + \int_\Omega f(\omega_i,x,\omega_o)\, L_i(x,\omega_i) \cos\theta_i \, d\omega_i \\
&= L_e(x,\omega_o) + \int_\Omega f(\omega_i,x,\omega_o)\, L_o(y,-\omega_i) \cos\theta_i \, d\omega_i \\
&= L_e(x,\omega_o) + \int_\Omega f(\omega_i,x,\omega_o)\, (L_e(y,-\omega_i) + L_r(y,-\omega_i)) \cos\theta_i \, d\omega_i \\
&= L_e(x,\omega_o) + \int_\Omega f(\omega_i,x,\omega_o)\, L_e(y,-\omega_i) \cos\theta_i \, d\omega_i \\
&\qquad + \int_\Omega f(\omega_i,x,\omega_o)\, L_r(y,-\omega_i) \cos\theta_i \, d\omega_i
\end{aligned} \tag{3.10}$$

Note: The first L_e term is only there for the first surface interaction, as there is no previous step that could sample it beforehand.

With this separation, we can approximate L_e and L_r separately and therefore use importance sampling to only sample directions, where a non-zero value for L_e is possible, i.e. lights.

In pseudo-code, this can be represented as:

```
color pathTraceNEE(point x, vector wo, bool primaryRay) {

    // Next Event Estimation
    color L_NEE = 0;
    for (int n = 0; n < N; n++) {
        Light light = randomLight();
        point yLight = light.randomPoint();
        vector wi = direction(x, yLight);
```

```
        float p = light.p(yLight, x);

        if (!occluded(yLight, x)) {
            L_NEE += brdf(wi, x, wo) * Le(yLight, -wi)
                    * cosTheta(x, wi) / p;
        }
    }

    // Reflected radiance
    color Li = 0;
    for (int m = 0; m < M; m++) {
        vector wi = randomDirection(x, wo);
        float p = p(wi, x, wo);

        vector y = rayIntersection(x, wi);

        Li += brdf(wi, x, wo) * pathTraceNEE(y, -wi, false)
              * cosTheta(x, wi) / p;
    }

    if (primaryRay) {
        return Le(x, wo) + Li / M + L_NEE / N;
    } else {
        return Li / M + L_NEE / N;
    }

}
```

Of course, this is not a complete algorithm as it is missing some special cases for materials with discrete directions, like dielectrics or specular materials, as NEE is not possible there.

3.2.2 Multiple Importance Sampling

The goal of multiple importance sampling is to combine different sampling methods to reduce variance.

Multiple Importance Sampling (MIS) was first proposed by [VG95] to address problems of NEE on very glossy surfaces. The problem that was observed, was that the NEE term depends on the distribution of lights in the scene and the BSDF, and sampling by the light distribution alone could completely miss the non-zero part of the BSDF. This was especially apparent on very glossy to almost mirror-like surfaces that reflect a light source, as NEE would almost never sample the exact location of the light that is in the actual reflection direction. See Figure 4.5 for an example.

MIS approximates an integral by combining T different sampling methods that generate $x_{t,j}$ based on probability distributions $p_t(x)$ and combines their contributions

using a weighting function $w_t(x)$.

$$\int_\Omega f(x)dx \approx \sum_{t=1}^{T} \frac{1}{n_t} \sum_{j=1}^{n_t} w_t(x_{t,j}) \frac{f(x_{t,j})}{p_t(x_{t,j})} \tag{3.11}$$

This approximation can be derived as shown in [WGGH20] by splitting the integral I into T weighted components, that again sum up to the original I.

$$\begin{aligned} I &= \int_\Omega f(x)dx \\ &= \int_\Omega \sum_t^T w_t(x) f(x)dx \quad \text{with } \sum_t^T w_t(x) = 1 \;\; \forall x : f(x) \neq 0 \text{ and } w_t(x) = 0 \;\; \forall x : p_t(x) = 0 \\ &= \sum_t^T \int_\Omega w_t(x) f(x)dx \end{aligned} \tag{3.12}$$

Each of the T integrals can then be approximated using importance sampling by a different probability distribution $p_t(x)$

$$\begin{aligned} I &= \sum_t^T \int_\Omega w_t(x) f(x)dx \\ &\approx \sum_t^T \frac{1}{n_t} \sum_j^{n_t} w_t(x_{t,j}) \frac{f(x_{t,j})}{p_t(x_{t,j})} \end{aligned} \tag{3.13}$$

This holds for all weighting functions $w_t(x)$ that satisfy $\sum_t^T w_t(x) = 1 \;\; \forall x : f(x) \neq 0$ and $w_t(x) = 0 \;\; \forall x : p_t(x) = 0$. The paper [VG95] specifically defines a few weighting functions:

- The balance heuristic, which they prove to be a good choice

$$w_t(x) = \frac{n_t p_t(x)}{\sum_{t'}^T n_{t'} p_{t'}(x)} \tag{3.14}$$

- The power heuristic, which should reduce the variance in situations where one of the $p_t(x)$ is a significantly better match for $f(x)$, thereby reducing fireflies or dark spots

$$w_t(x) = \frac{n_t p_t(x)^\beta}{\sum_{t'}^T (n_{t'} p_{t'}(x))^\beta} \tag{3.15}$$

Applying this method to NEE can be done by evaluating the NEE term (3.10) using one ray based on the light distribution $p_1(\omega_i, x, \omega_o)$, and one based on the shape of

the BSDF $p_2(\omega_i, x, \omega_o)$.

$$\int_\Omega f(\omega_i, x, \omega_o) L_e(y, -\omega_i) \cos\theta_i \, d\omega_i \approx \sum_{k=1}^{2} w_k(\omega_{i,k}, x, \omega_o) \frac{f(\omega_{i,k}, x, \omega_o) L_e(y_k, -\omega_{i,k}) \cos\theta_k}{p_k(\omega_{i,k}, x, \omega_o)} \tag{3.16}$$

This prevents the problem on glossy surfaces, as the BSDF-based ray is likely to hit the correct location on the light, while also keeping the benefit of reducing variance on more diffuse surfaces.

3.2.3 Irradiance Caching

Evaluating the indirect illumination on diffuse surfaces is very costly, as it depends on illumination from all directions and needs many samples to create a noise free result. The paper [WRC88] also observed, that in many cases the indirect illumination varies slowly over a surface, and that drastic changes only occur in the direct light component.

This, in combination with the fact that an ideal diffuse component is independent of the view-direction, lead them to create a method called Irradiance Caching (IC) [WRC88], where this indirect illumination is cached on diffuse surfaces, or surfaces with diffuse components, and can be used to replace the expensive calculation of the indirect illumination in a region around this cache location.

Their proposed workflow for a surface with diffuse components is:

1. Evaluate direct light contribution (NEE)
2. Generate rays to compute specular/non-diffuse components
3. If there is at least one valid IC-value in range, then use these IC values. Otherwise, generate a new cache value by generating a number of rays to evaluate the indirect illumination. This cache value is then valid in a sphere of influence based on the distance to occlusions.

By assuming their hypothesis, indirect illumination varying slowly over diffuse surfaces, to be correct, then we can again construct the theoretical background by converting (3.10)

$$\begin{aligned} L_o(x, \omega_o) &= \ldots + \int_\Omega f(\omega_i, x, \omega_o) L_r(y, -\omega_i) \cos\theta_i \, d\omega_i \\ &= \ldots + \int_\Omega f(x) L_r(y, -\omega_i) \cos\theta_i \, d\omega_i \\ &= \ldots + f(x) \int_\Omega L_r(y, -\omega_i) \cos\theta_i \, d\omega_i \\ &\approx \ldots + f(x) E(x) \end{aligned} \tag{3.17}$$

As $f(\omega_i, x, \omega_o)$ is constant in regards to changes in ω_i, ω_o for diffuse materials, it can be replaced with $f(x)$.

Here, $E(x)$ is a weighted average of irradiance cache values IC_i that can affect this location ($i \in S$).
The weight values $w_i(x)$ are a measure of similarity between the two surface points and dependent on the normal orientations of the two samples, the distance, as well as the size this IC entry is valid for [WRC88]. This results in IC entries only contributing to the result, if they were created on a similar surface section as well as most likely not containing wildly different occlusions due to the limited sphere of influence of the cache values.

$$E(x) = \frac{\sum_{i \in S} w_i(x) IC_i}{\sum_{i \in S} w_i(x)} \tag{3.18}$$

The IC entries are generated by approximating the indirect light integral

$$IC_i(x) \approx \int_\Omega L_r(y, -\omega_i) \cos\theta_i \, d\omega_i \tag{3.19}$$

Explicitly, they are generated by sampling the hemisphere of the sample location in a stratified way, so segmenting the hemisphere into regions and generating one ray per region.

$$IC_i(x) = \frac{\pi}{2n^2} \sum_{j=1}^{n} \sum_{k=1}^{2n} L_r^{x, \omega_{j,k}} \tag{3.20}$$

Note: In this case, the cos term was folded into the sampling strategy of $\omega_{j,k}$.
The quality of this approximation depends on the number of samples as well as the quality of the $L_r^{x,\omega_{j,k}}$ approximation. A simple approximation is to simply perform a NEE step at all intersected surfaces and disregard the indirect-indirect illumination. This of course also prevents true global illumination effects from occurring that need two or more bounces.

A further improvement to the IC method was the Irradiance Cache Gradients [WH97]. These gradients are created during the sampling of the hemisphere and take the distance of occluding objects into account to approximate how the IC value would change when subjected to displacement or rotation of the surface.

3.2.4 Adjoint-driven Russian Roulette and Splitting

Adjoint-driven Russian Roulette and Splitting (ADRRS) [VK16] aims to improve the common techniques Russian Roulette (RR) and the splitting of paths [Vea97].

Common RR implementations base their decisions about terminating a path solely on the path weight containing the effect of all encountered materials, called throughput.

The difference in light situations or the actual effect the path could have on the final result are disregarded and can lead to sub-optimal decisions.

At the heart of ADRRS lies the idea of the zero-variance scheme [WH98], where a single sample with the perfect weight could evaluate a complex function exactly. For path tracing, this would mean, that the path weight/throughput $\hat{v}(y,\omega_i)$ and the reflected radiance from that direction $L_r(y,\omega_i)$ would result in exactly the correct pixel value I

$$I = \hat{v}(y,\omega_i)\, L_r(y,\omega_i) \tag{3.21}$$

To achieve this, the incoming path weight $v_i(y,\omega_i)$ has to be scaled to fullfill this invariant

$$\hat{v}(y,\omega_i) = \frac{v_i(y,\omega_i)}{q(y,\omega_i)} \tag{3.22}$$

This scaling factor q then has to be calculated from some estimate of the final pixel value $\tilde{I}$ and an approximation of the irradiance $IC(y)$, as the real values are not known

$$q(y,\omega_i) = \frac{v_i(y,\omega_i)\, L_r(y,\omega_i)}{I} \approx \frac{v_i(y,\omega_i)\, f_{BSDF}(x)\, IC(y)}{\tilde{I}} \tag{3.23}$$

Of course, simply scaling the weight of a path is not allowed, as it would change the result. Therefore, the path itself has to be modified. This is done by applying RR or splitting.

If RR is applied, the path gets terminated with the probability $1-q$. To keep the MC estimate unbiased, any surviving paths get their weight scaled by $\frac{1}{q}$ to account for the missing contribution of the terminated rays. This raises the path weight and can be used to counteract too low path weights.

Splitting does the opposite, as it splits one incoming path into q outgoing paths that each only contribute $\frac{1}{q}$ of the total. This way, the path weight can be decreased and any interesting regions explored more thoroughly.

As the approximations used create some deviations, instead of there being a fixed desired weight, a range of acceptable weights called weight window is used and the path weight $\hat{v}(y,\omega_i)$ is only scaled to be somewhere in this range. See Figure 3.1.

The center of this weight window C_{WW} is calculated from the desired weight, see Equation 3.21, but has to be approximated as the real values are not known.

$$C_{WW} = \frac{\tilde{I}}{f_{BSDF}(y)\, IC(y)} \tag{3.24}$$

The weight window bounds δ^-, δ^+ are then

$$\delta^- = \frac{2C_{WW}}{1+s} \tag{3.25}$$

$$\delta^+ = s\,\delta^- \tag{3.26}$$

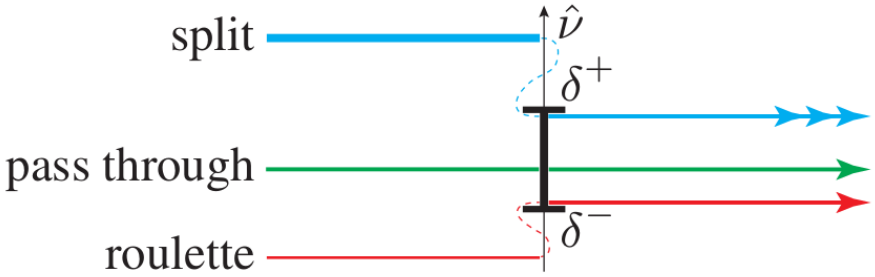

Figure 3.1: The acceptable weight window for ADRRS, where any paths with a weight higher than the desired range get split so that all resulting rays fall in the desired weight range. Paths with too low weights only survive RR with a certain probability and any surviving rays will therefore be scaled up to the desired range.
Image taken from [VK16]

where s is a window width factor and is suggested to be set to $s = 5$.

This all combined should result in decreased variance overall and a better exploration of more important regions, while paths in uninteresting regions can be terminated early to save on calculations.

3.2.5 Path Guiding

Evaluating the local BSDF and sampling according to it is often very easy and is therefore the prevalent method used to generate new rays. But of course, the rendering equation heavily depends on the incoming light which is not easy to approximate.

Path guiding describes the process of guiding rays along those paths or directions that will result in light contributions. There exist two different main categories of path guiding, local and global. This section is in many parts based on [RHL20].

Global path guiding aims to construct the whole ray path based on previous paths by e.g. influencing the primary sample space, and therefore the decisions made at each intersection, using a trained guiding model with neural networks [ZZ19, MMR+19] or other methods [GBBE18, RHJD18].

Local path guiding on the other hand aims to approximate the incoming local radiance and use it to guide new ray directions at each intersection. For this, the local radiance, a value dependent on the position as well as the direction, needs to be represented in a compact form that can be easily sampled and evaluated.

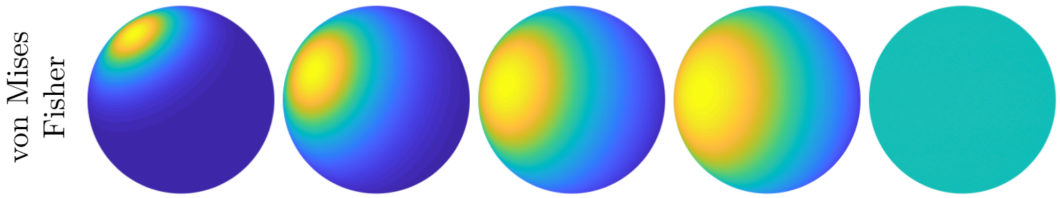

Figure 3.2: An example of what kind of distributions a von Mises-Fischer distribution can create based on the supplied μ and κ.
Image taken from [Hau18]

3.2.6 Von Mises-Fischer Mixture Model (VMM)

The Von Mises-Fischer Mixture Model (VMM) is based on the von Mises-Fischer distribution function (vMF) [FLE87] which is a direction-dependent probability distribution which for $\mathbb{R}^3$ is defined on a sphere. A numerically stable version was developed by [Jak12] and takes the form:

$$v(\omega|\mu,\kappa) = \frac{\kappa}{2\pi(1-\exp(-2\kappa))}\exp\left(\kappa\left(\mu^T\omega - 1\right)\right) \tag{3.27}$$

With this, directional distributions with varying sharpness can be created, see Figure 3.2.

By combining multiple vMF, i.e. mixing them, a Parametric Mixture Model (PMM) can be created. This is called the Von Mises-Fischer Mixture Model (VMM) and is defined as

$$V(\omega|\Theta) = \sum_{k=1}^{K} \pi_k v(\omega|\Theta_k) \tag{3.28}$$

where the parameter set $\Theta = \{\pi_1, \Theta_1, \ldots, \pi_K, \Theta_K\}$ contains the distribution parameters $\Theta_k = \{\mu, \kappa\}$ and the π_k are positive weights that sum up to 1.

This VMM can easily be sampled or evaluated for a direction [Jak12], which makes it a good choice for the local radiance approximation.

3.2.7 Fitting

Of course, the VMM actually needs to be fitted to the local radiance to be of any use. For this, first there need to be samples available that specify, what radiance they received and from what direction. These samples then get grouped into regions that will get represented by a common VMM, because the precision cant be increased indefinitely as that would result in too few samples or too much overhead for region selection.

These samples are then used to fit the VMM to the samples and therefore the local radiance distribution. This is done in a two step iterative process called EM-Algorithm

[DLR77, MK07], which stands for the two steps *Expectation* and *Maximization* and is a common tool in statistics and machine learning. Its purpose is to adjust the parameters of a statistical model, in this case the VMM, to increase the maximum likelihood, i.e. make the probability distribution best predict the actual samples encountered.

After fitting, the VMM should then represent the actual directions and probabilities for light contributions, which can then be used to sample new directions.

3.2.8 Parallax-Aware Path Guiding

Parallax-Aware path guiding by [RHL20] was developed to combat a main problem of ordinary local path guiding, which is that samples that originate from the same source of light differ wildly in the directions they are actually facing as the origin of the sample can lay anywhere in the region. This means that the vMF cone needs to be pretty wide to contain all of these directions, thereby making the whole VMM less accurate.

Parallax-aware path guiding works against this issue by saving the distance to the encountered light effect. With this information, the target of each sample can be calculated and then used to reproject all samples to a central location, thereby making all samples that show the same light effect actually point in the same direction, see Figure 3.3. This way, the resulting vMF cone can be pretty narrow and is a more accurate fit for the actual light effect. Information about the target of the distribution can then be used to again reproject the distribution to the current location so that the vMF stays valid for the whole region. Now, it always points at the correct target location and not just the direction it was encountered from, as that could be from a totally different point in the region.

Splitting and Merging of components

The VMM contains a number of vMF distributions that are jointly fit to the collected samples. During this fitting process, some vMF distributions might be fitted to the same effect which results in there being two almost identical distributions. This hinders the quality of the fit as now there is one less component that can actually represent different effects. These distributions can also never diverge again as the samples will get attributed to both of them equally.
Another possible problem is that one distribution contains samples from two to distinct effects that result in the distribution having a very wide cone to account for both effects.

Splitting and merging aims to to locate these problems and fixes them by either merging two components that represent the same effect into a single component or alternatively splitting one component that contains two effects into two distinct

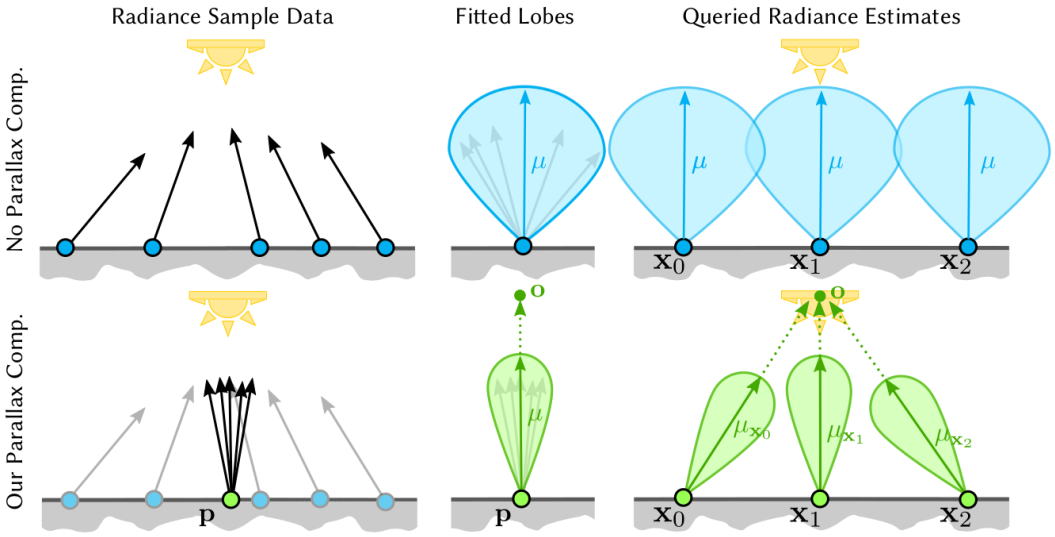

Figure 3.3: A comparison of how the received local radiance would need to be fitted, when using ordinary path guiding or parallax compensation, as well as how the resulting guiding information would look like. Using parallax compensation makes the resulting lobe significantly sharper, as effects from the same source also point at the same target and don't vary over the region.
Image taken from [RHL20]

distributions that each fit one of them. This can repair degenerate cases or increase accuracy where it is needed.

Of course, statistical analysis is necessary to detect such conditions, but as we use the finished implementation of [RHL20], no further statistical details will be provided, but can be found in [RHL20].

3.3 Vulkan Ray Tracing

To use the ray tracing features of the new NVIDIA RTX cards in Vulkan [Gro], NVIDIA, with input from AMD and Intel, developed the VK_NV_ray_tracing Vulkan extension [Wea18] that enabled the creation of acceleration structures and added support for new ray tracing shaders. This was then later transformed into a vendor-agnostic extension VK_KHR_ray_tracing [Wea19] that, with the help of other hardware manufacturers as well as game developers, was refined to create a standard that all manufacturers could later implement.

3.3.1 Acceleration Structure

An acceleration structure (AS) is some kind of organisation structure of the scene data, mostly in the form of axis-aligned bounding boxes (AABB), that allows fast traversal for intersection checks while keeping the number of objects to intersect against as small as possible. Any object of the scene will first have to be inserted into an acceleration structure.

Examples for such acceleration structures that are often used for CPU ray tracers are kd-trees [PJHc], a tree structure where each node represents a plane splitting one dimension with children containing further such splitting planes or all the objects in the current region, or octrees, where the bounding box encompassing the scene is split into 8 smaller bounding boxes around one central point, which can then be recursively split again or otherwise contain all the objects in that region.

In the context of Vulkan ray tracing, the actual structure of the AS is opaque and not controllable by the developer, except for some configuration options like *PREFER_FAST_TRACE*, *PREFER_FAST_BUILD*, or *OPAQUE*. The supplied triangle meshes or AABBs get constructed into bottom-level AS which can then be instanced into the top-level AS. The building/optimisation step is completely dependent on the hardware and drivers implementing this extension. NVIDIA states that their AS are bounding volume hierarchies [PJHb], meaning the primitives in the scene get divided into different sets, which then get recursively combined into larger bounding boxes encompassing all children, see Figure 3.4. Bounding Volume Hierarchies also have the desirable property of having a known and reasonable maximum memory footprint, as the number of internal nodes and leafs can be bounded based on the primitive count, as well as being faster to build than e.g. kd-trees [PJHb].

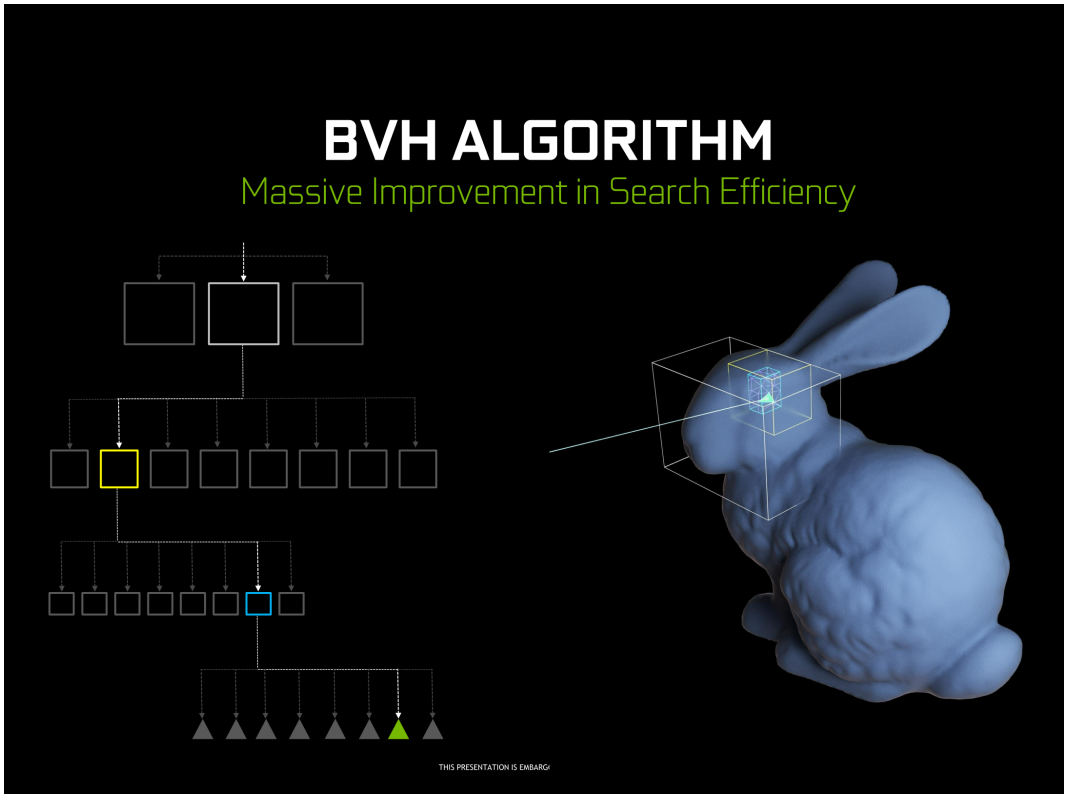

Figure 3.4: The Bounding Volume Hierarchy as used for the Acceleration Structures. Image taken from NVIDIAs Turing Editor's day presentation [NVI]

These AS can then be used in the ray tracing shaders for efficient hardware-accelerated traversal of the bounding box hierarchy to compute all ray-primitive intersections.

3.3.2 Shaders

To interact with the ray tracing hardware, five new shader types were introduced. The new shaders are all closely related to compute shaders and are therefore capable of all general purpose calculations and common operations like buffer or texture access.

RayGen

The ray generation shader is the entry point for the ray tracing pipeline where a configured number of ray generation instances are started, e.g. one per pixel. The RayGen shader can then start ray traces and store a final result in the appropriate location, i.e. an image pixel.

When calling the *traceRay* function, one can specify

- the AS to trace against
- the payload data structure passed to the other shaders
- min and max ray length
- a mask to exclude some objects
- flags like *TerminateOnFirstHit* or *SkipClosestHitShader*

Closest Hit

The closest hit shader gets called on the intersection with the smallest distance t. Here, primitive specific information like the exact location, normals, and materials can be accessed and used to fill in the payload with all relevant hit information.

It is also possible to start new *traceRay* calls from this shader, but the device specific maximum recursion depth for nested *traceRay* calls has to be taken into account, e.g. 32.

Any Hit

The any hit shader gets called for all hits along the ray, in undefined order, and can be used to reject or approve intersections. This can be used to for example use alpha textures to let rays pass through, or to gather data from all intersected objects.

Miss

If no intersections were reported for a ray, then the miss shader gets executed. This can be used to e.g. query a sky box. The miss shader is also able to trace new rays.

Intersection

Intersection shaders are only necessary if other primitives besides triangles should be used, as they use a fast hardware implementation for the intersections. These other primitives, called procedural, are only stored as AABB in the AS and any intersections with this AABB results in a call to the intersection shader. Here, the ray direction and origin can be used to perform any kind of intersection calculations that can result in reporting an intersection or not.

4 Method/Implementation

The path tracer is mostly implemented in the RayGen shader where one compute thread per pixel computes the corresponding pixel value.

For this, a configurable *raytrace* function was created that contains the loop-based path tracing implementation. The *raytrace* function takes an origin, a direction, a maximal depth, as well as a range of options to configure the behaviour of the path tracing, e.g. if NEE is enabled or if the local emission should be queried.

In this function, a while loop repeats the steps

1. Trace ray in the desired direction to get the next surface interaction
2. Optionally evaluate the local emission or NEE on the surface and store its contribution
3. Generate a new direction based on BSDF sampling or, when enabled, path guiding
4. Evaluate the BSDF and update the throughput value, which is the coefficient for all future light contributions

until either the maximum depth is reached or no intersection is found. The ray tracing is implemented with the *traceRay* function that traces a ray against an acceleration structure (AS) and a ClosestHit shader then evaluates the intersection and returns the location, normal, material, texture uv-coordiantes, and ray length. See Figure 4.1 for a more detailed description.

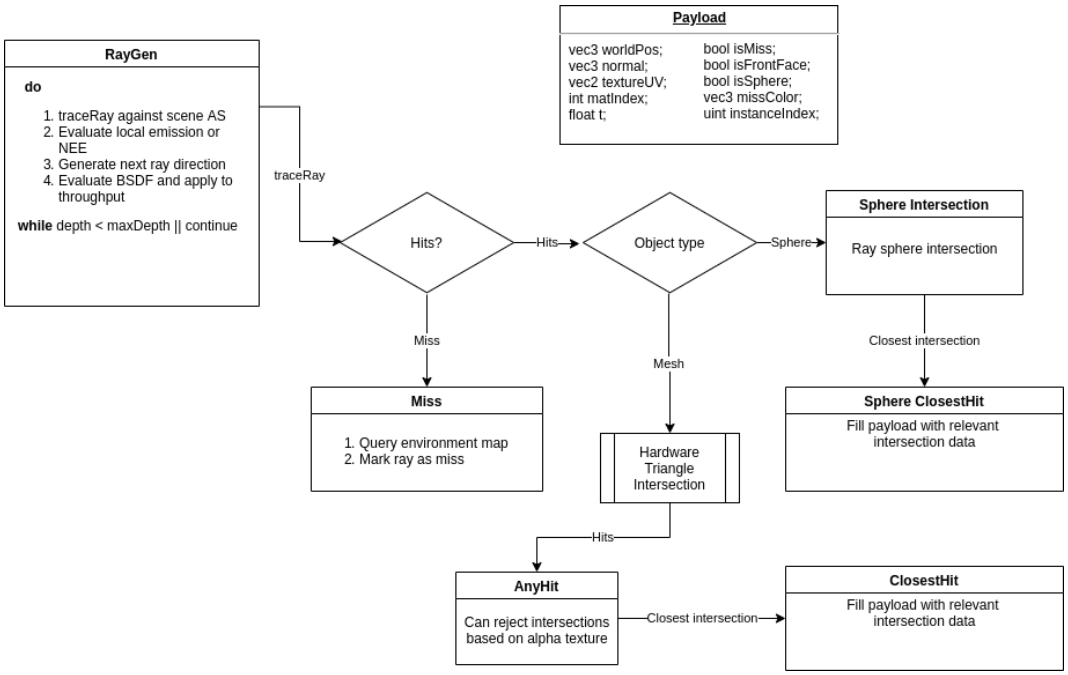

Figure 4.1: The shader structure of the main path tracing loop.

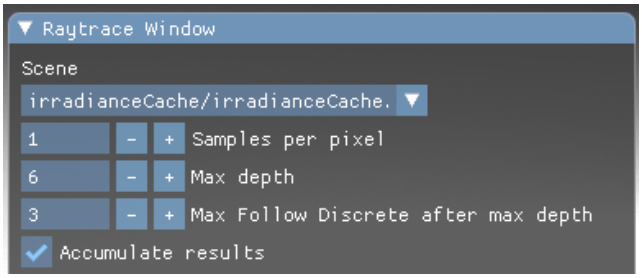

Figure 4.2: The scene selector as well as some general path tracing settings. These settings are: the number of samples per pixel per frame, the maximum ray depth, as well as the leeway for this depth, when materials with discrete directions are encountered last.
Accumulate results switches between showing only the samples collected in the current frame or accumulating the results of multiple frames.

4.1 Random numbers

A Monte Carlo path tracer depends heavily on random number generation as all decisions about sampling directions or light selections are based on random numbers. This means, the choice of pseudo random number generator (PRNG) is important for the final quality of the image.

For this, first a noise texture based approach was implemented where each thread traversed the noise texture in some direction to select its random values.
This was later replaced with an implementation by NVIDIA [NVI20a], which generates a seed based on the Tiny Encryption Algorithm [ZOC10] and uses a linear congruential generator as the PRNG. The linear congruential generator always takes the previous random number and generates a new one.

The seed gets calculated per pixel thread and is based on two numbers, one being the pixel index and the other per frame random number generated on the CPU. This makes sure, that the different pixel threads generate independent random numbers that differ from one frame to the next.

4.2 Next Event Estimation

The goal of NEE is to sample the lights in the scene explicitly at each intersection, for which we need to select random points on area lights.

To sample area lights uniformly and therefore each location of the light with probability $\frac{1}{area}$, a random triangle needs to be selected based on its area, as these triangles can have strong variations in size. To avoid having to iterate over many triangles of a light to generate a weighted random sample, these triangle indices get precomputed at the time of scene loading and stay constant. This way, a uniform random triangle location can be selected by simply querying a random index of the triangle indices array and picking a random point on that triangle.

To check if this light is visible from the current origin o, a shadow ray is shot in the direction of the sample point p with a maximum distance $t_{max} = ||p - o|| - \epsilon$ to check for occlusions. If no hit is reported and the miss shader gets executed, then it sets the *isShadowed* payload value to false and the light's contribution gets added to the received light. The *traceRay* flags *TerminateOnFirstHit* and *SkipClosestHitShader* are set, as we only care if the miss shader was executed and any hit at all results in no light contribution.

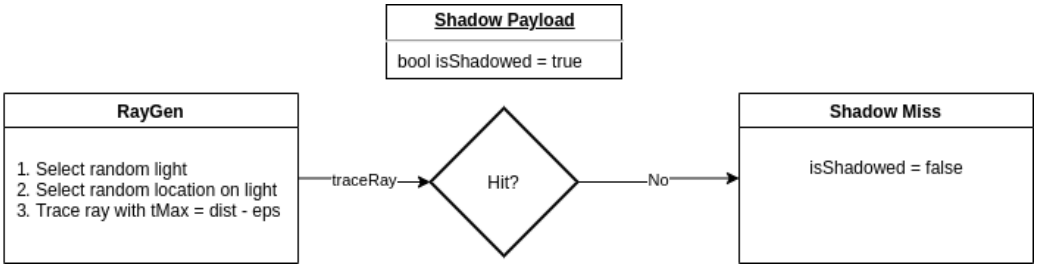

Figure 4.3: The shader structure and payload used for NEE.

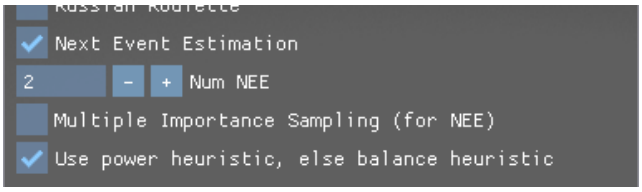

Figure 4.4: The configuration options for NEE and MIS.
The amount of NEE iterations at each intersection can be configured, where higher numbers result in less direct light noise but have some performance impact, especially for longer paths.
MIS can be configured to use the power heuristic or the balance heuristic to combine the two sampling methods.

4.3 Multiple Importance Sampling

To implement MIS for NEE, a second ray based on BSDF sampling needs to be traced. If it intersects a light or misses the scene and queries the environment map, then this contribution gets added to the result, but weighted using the balance or power heuristic based on the pdfs of sampling that light location and BSDF direction directly.

Figure 4.5 shows the effects of MIS in comparison to path tracing and NEE in the Veach MIS inspired scene.

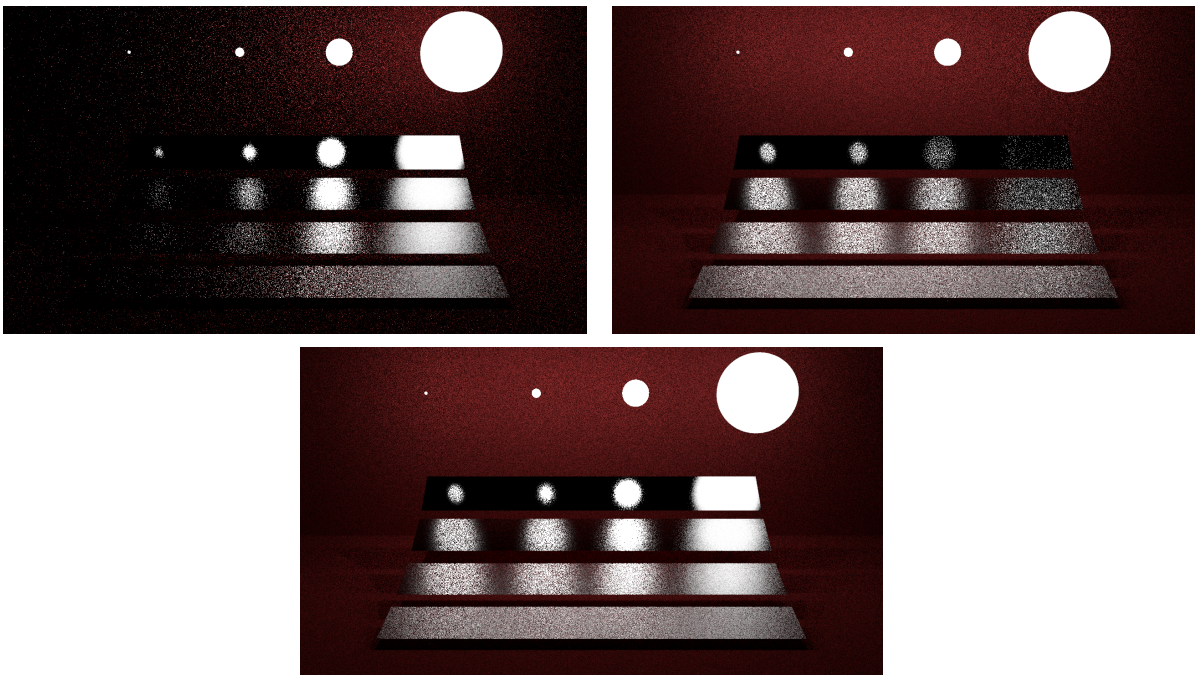

Figure 4.5: Comparison of 10spp images of a Veach MIS inspired scene [VG95] using a pure path tracer (left), NEE (right), and NEE with MIS (bottom)
The 4 plates are rough conductors with a transition from almost diffuse at the bottom, to almost specular at the top.
The pure path tracer has difficulties on diffuse surfaces as the chance to hit lights is relatively low, but reflections work well due to the BSDF based sampling. NEE helps on all diffuse surfaces, due to the light based sampling, but fails for the almost specular case, as it has difficulties to sample the exact reflected light spot. MIS combines the best of both.

4.4 Irradiance Caching

The IC entries get stored in two separate buffers. One describes the spheres representing the maximum sphere of influence and their AABB, the other contains the cache values, position, and normals. The buffer containing the sphere data is used to generate a separate AS consisting of AABBs, one for each cache entry.

To evaluate the IC for a diffuse location, all IC entries with a sphere of influence that overlaps the query location need to be evaluated and the similarity weights calculated. For this, a custom intersection shader is necessary that reports intersections, if the origin of the ray is within *radius* distance of the IC entry center, and therefore in its sphere of influence. *traceRay* is called with $t_{max} = \epsilon$ as it should only intersect those IC AABBs that contain the origin.

Each resulting hit is one IC entry that needs to be evaluated. For this, an *AnyHit*-shader is used. It calculates the weight w_i based on the position, normals, and distance of the query location and stores the resulting contribution, if any, to a shared ray payload, as well as adding the weight to a normalization term. See Figure 4.6 for a schematic representation.

This approach of using the hardware ray tracing functionality to query the IC entries replaces the normal approach of traversing a data structure to search for entries in range.

If any IC entries were valid, then the returned IC value is normalized and combined with NEE to create the final contribution to the pixel value, as the IC value was used in place of casting more rays for the indirect contribution.

If no IC entries were found, then the current location is put into a queue, so that after the ray tracing process, an IC entry can be created. This queue structure was mainly created because recursion is not possible in shaders, so calling the *raytrace* function to calculate the cache value from inside itself is not possible, but it might also improve coherence between rays.

The IC value calculation is implemented by sampling the hemisphere over the point in a stratified way and calling the *raytrace* function with $maxDepth = 1$ and $addDirectLights = false$, as the direct light contribution will be added by NEE. Of course, this limitation to only generate paths of depth one leads to this only being an approximation of the real value. The radius of influence gets calculated by taking the harmonic mean of the distance to all occluding objects and multiplying with a user defined value specifying the desired accuracy of the system. The result is then written into an IC entry at an index returned by an *atomicAdd* operation.

The calculation of the IC values can of course also use ICs to calculate its own value, thereby creating more accurate results for diffuse surfaces. To improve the quality of the Irradiance Caches, they can also be updated later by again sampling the hemisphere and combining this result with the previous value. This results in more

global illumination effects being accounted for in the ICs and is exactly the strategy used in Minecraft RTX to generate global illumination [NVI20b].

To include these new IC entries in the AS, we need to update it, which is a faster process than creating a completely new one. To make this easier, the AS contains all possible IC entries from the beginning, but with a radius of 0. This AS update is done after each frame, where IC creation or updating was enabled. Every 100 frames, the IC acceleration structure is rebuild to again create an optimal AS, as updating results in suboptimal structures that slow down ray traversal.

One limitation of this GPU implementation is, that IC entries can only be found in the next frame after creation, as the AS does not contain them until the AS update. The original IC algorithm [WRC88] always used the created cache value immediately for the current ray, as well as all being query-able by all following rays.
In a simple implementation, this would result in all rays creating IC entries on the first diffuse surface hit. To counteract this, new IC entries only get created with some creation probability, and continue on as normal path tracing rays otherwise. This way, only a fraction of the rays create IC entries and on the next frame, many rays can use the resulting IC. This probability as well as the update probability for irradiance caches can be configured, see Figure 4.8.

4.4.1 Gradients

During the hemisphere sampling step of the creation of the IC entries, the values and first-hit distances of the samples get saved and can then be compared to calculate approximate rotational and translational gradients, i.e. how much the irradiance would change when the location of the sample is moved slightly or its surface orientation changes. These gradients then get stored alongside the cache values and can be used to modify the cache value based on the position sampling the IC. See Figure 4.9 for a comparison of using IC gradients.

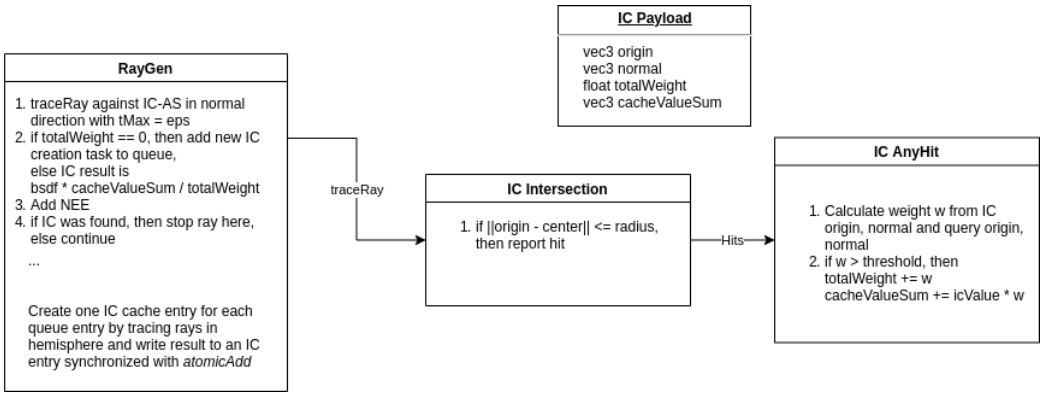

Figure 4.6: The shader structure and payload used for Irradiance Caching. The IC-Acceleration Structure contains the AABBs of the IC entries, while another buffer contains the actual cache values.

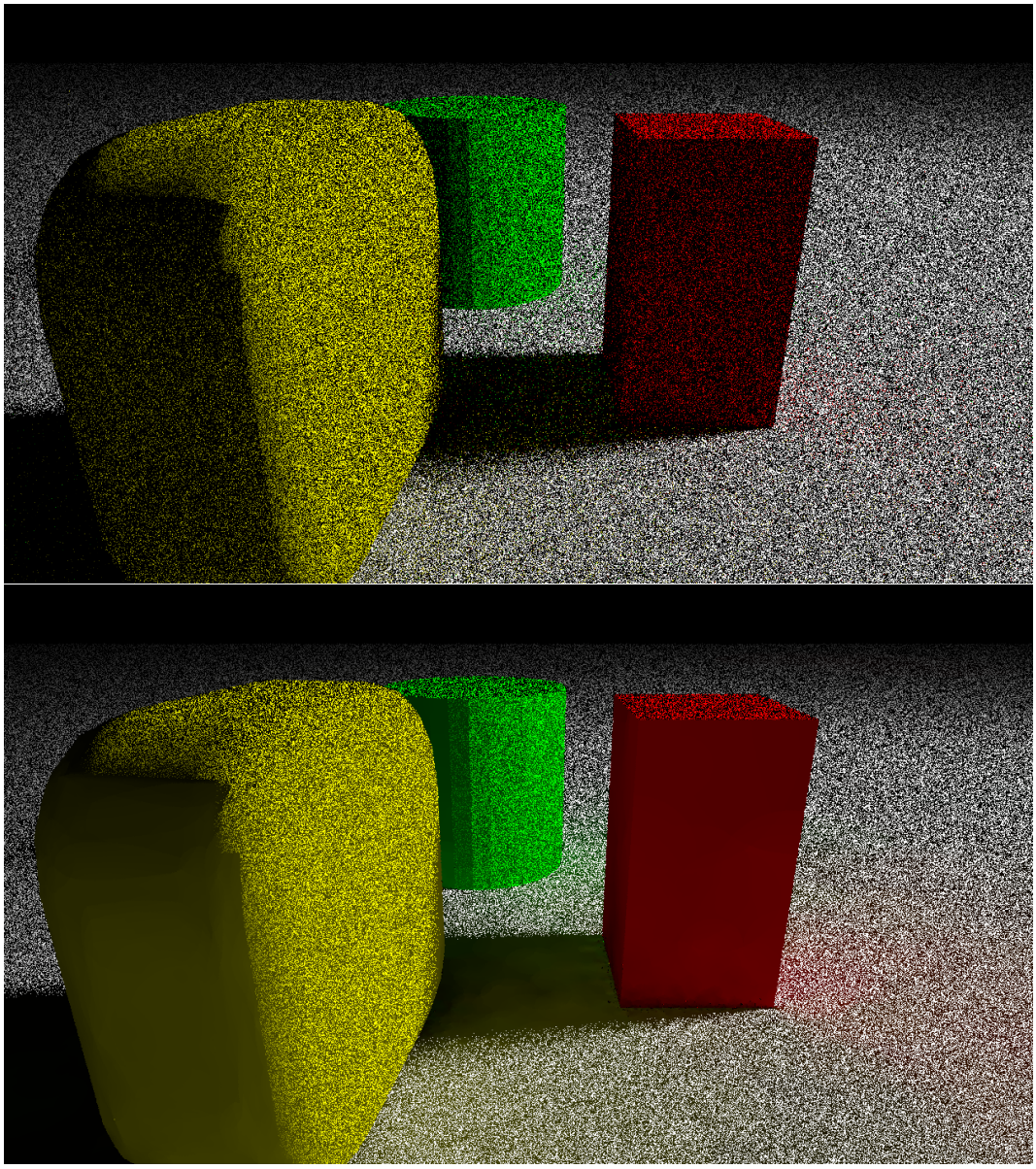

Figure 4.7: Comparison of a completely diffuse scene without (top) and with irradiance caching (bottom) while using only a single sample per pixel.
One can see that the indirect illumination effects, of e.g. color being visible on the floor in front of the object, are already visible when using IC and especially the shadow area in the center already has a way smoother appearance.
The effects are more pronounced on regions without direct light contributions, as there no noise from the NEE step exists.
Also visible are clear boundaries between regions with slightly different cache values, which are of course undesirable.

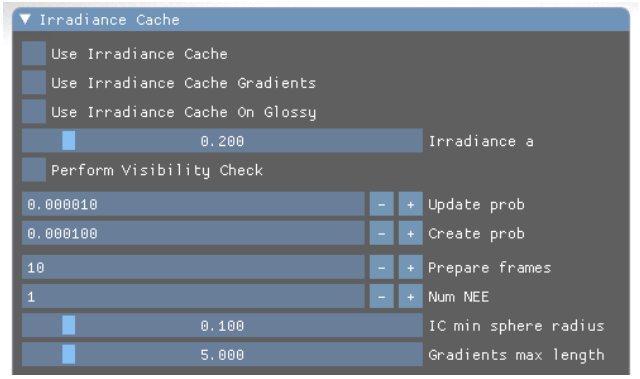

Figure 4.8: The configuration options for irradiance caching. The most important setting is the accuracy parameter a that controls the size of the IC entries and how closely they have to match the query location to be taken into account. Also interesting are the amount of NEE steps taken for each hemisphere sampling ray to calculate the IC value, or the creation and update probabilities controlling the rate of rays creating IC entries or updating them.
The visibility check is a additional criterion proposed in [WRC88] that rejects IC entries during querying, if they could be occluded by the surface itself.
The minimum sphere size is used to limit the amount of IC entries created in corners, as they would be very small otherwise due to the very close obstructing surface.
Prepare frames are used to pre-populate the IC on first scene load. They disregard pixel value calculations and only trigger IC entry creation for the first few surfaces intersected.
Gradients can also be enabled and their max length configured to prevent too high gradients increasing the variance.
The IC on glossy surfaces option is only used for the estimate image for ADRRS.

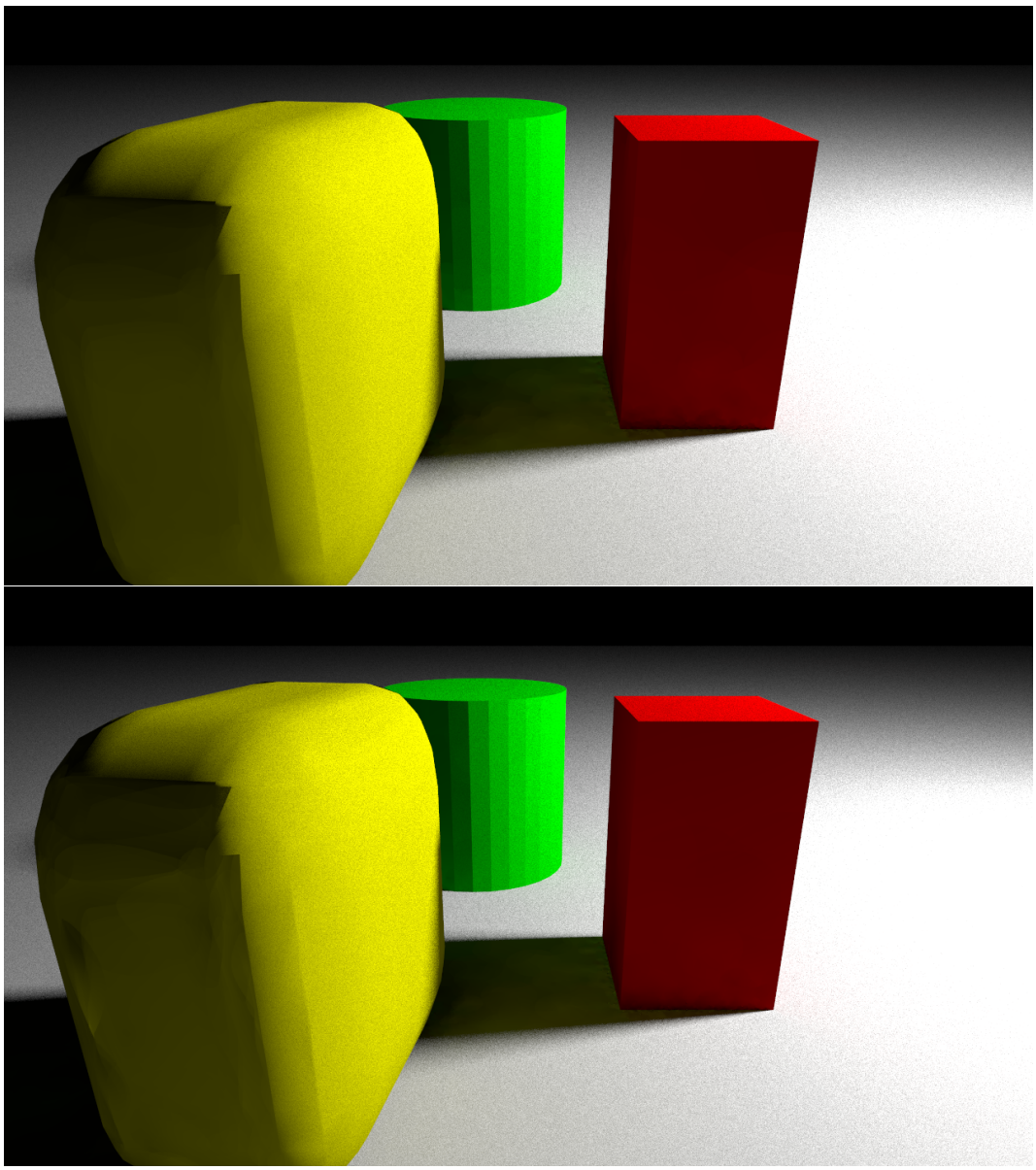

Figure 4.9: Comparison of using irradiance caches (IC) without (top) or with (bottom) IC gradients.
On the red surface and the shadow in the center, the gradients helped reduce the banding effect of caches with different values creating circular regions in the final image. On the other hand on the shadow side of the yellow object, the gradients over corrected and created visible artefacts.

4.5 Adjoint Driven Russian Roulette and Splitting

For adjoint driven Russian roulette and splitting (ADRRS), an estimate of the final image needs to be available to drive the decisions about whether a path will contribute a significant amount or if its contribution will be minimal.

This is currently implemented as described in their paper [VK16], where IC entries get created on all non-specular surfaces. Four samples per pixel then get taken, where the ray continues through specular interactions, until the first non-specular surface is found and the IC is queried there. The estimated image then gets stored in a separate image resource that can be queried by the shaders, see Figure 4.15 for an example.
An alternative approach would be to use the current accumulated image in denoised/blurred as the estimate, but this is currently not implemented.

To decide if a ray should be split, to explore the region further, or terminated early with Russian roulette, the desired weight window needs to be calculated. For this, the IC is queried $IC(x)$ and in combination with the estimate for the current pixel $\tilde{I}$, the desired particle weight C_{WW} and weight window bounds δ^+, δ^- get calculated

$$C_{WW} = \frac{\tilde{I}}{f_{BSDF}(x) IC(x)} \tag{4.1}$$

$$\delta^- = \frac{2C_{WW}}{1+s} \tag{4.2}$$

$$\delta^+ = s\delta^- \tag{4.3}$$

Here, the BSDF value $f_{BSDF}(x)$ is a directionally uniform approximation of the directional BSDF, e.g. the total reflectivity divided by π, as the IC contains no directional information. $s = 5$ was chosen, as this was the recommended value [VK16].

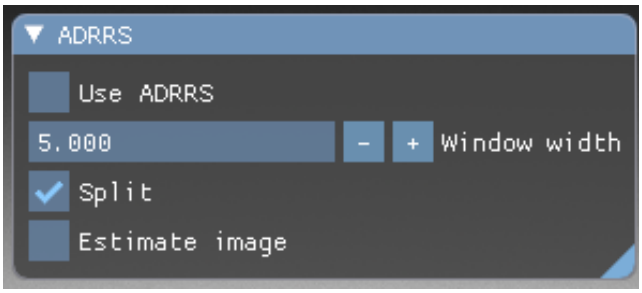

Figure 4.10: The configuration options for ADRRS. Here, the window width parameter s can be adjusted as well as controlling whether splits should occur. Also the creation of the estimate image $\tilde{I}$ can be triggered here.

With these desired weight bounds δ^+, δ^-, the incoming throughput of the ray v_i can then be compared against them:

- if $\delta^- \leq v_i \leq \delta^+$, then the ray is in the desired weight window and continues on unchanged
- if $v_i < \delta^-$, then its weight is too low. To increase its weight, Russian roulette is performed with survival probability $q = \max(\frac{v_i}{\delta^-}, 0.1)$, bounded to at least 10% as it could create fireflies otherwise.
  If the ray survives, then its outgoing throughput $\hat{v}$ gets scaled with $\frac{1}{q}$ which puts it back in the desired variance-reducing range.
- if $\delta^+ < v$, then its weight is too high. To scale it back to the desired range, the ray gets split into multiple rays, each having a throughput scaled with $\frac{1}{q}$ where $q = \frac{v_i}{\delta^+}$. This results in the outgoing throughput $\hat{v}$ of each ray being back in the weight window.
  The number of split rays should ideally be q, but this is not a whole number. Therefore an "expected-value split" [VK16] is used, where the ray is split in $n = \lfloor q \rfloor$ rays with probability $n + 1 - q$ or $n + 1$ otherwise.
  To prevent a huge number of splits, q and therefore n get bounded to be at most 10 total splits per pixel.

The information about where to split, as well as the current material, incoming direction, throughput, and all other relevant ray information get stored in a queue, while the original ray continues. This queue then gets evaluated afterwards, and its contribution is added to the pixel value. These split rays are also able to create new splits by adding them to the queue.

4.6 Path Guiding

To use path guiding, four main components need to be implemented:

- A method to query which guiding region is applicable for the current location
- The collection of samples, so data about what locations received light from which direction and at what strength and probability
- The optimization step that fits the Von Mises-Fischer Mixture Models (VMM) to the collected samples in a way that optimally represents the actual incoming light in that region
- The evaluation and sampling of the VMMs to generate new ray directions or evaluate the guiding-based pdf

The optimization step was not implemented on the GPU, but instead uses the reliable and tested implementation of [RHL20] which runs on the CPU in a highly vectorized form.

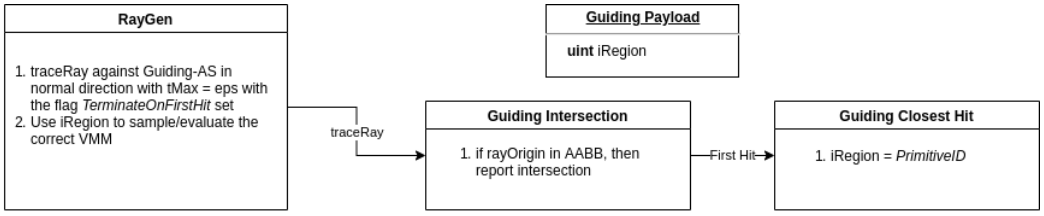

Figure 4.11: The shader structure and payload used for path guiding. The Guiding-Acceleration Structure contains AABBs that partition the scene bounding box into separate guiding regions.
A buffer contains the Von Mises-Fischer Mixture Models (VMM) that were optimized on the CPU, which can then be easily sampled to generate a guided direction or evaluated for the pdf.

4.6.1 The Guiding Acceleration Structure

To use path guiding, the scene needs to be partitioned into regions that each have different guiding distributions (VMMs) associated with them, as well as a method to query which guiding region is the one that is applicable for the current location.

To implement this, first the total scene AABB gets calculated and then recursively split along its largest axis a certain number of times which results in the initial guiding regions.

These regions, represented by AABBs, then get built into an Acceleration Structure (AS). Now, instead of having to implement some data structure traversal to find the matching guiding region, a *traceRay* call is used to implement this positional query with the RTX ray tracing hardware.

A simple intersection shader is used to intersect against the region AABB that overlaps the query location. This *traceRay* call is started with the *TerminateOnFirstHit* flag, as there should be (almost) no overlap of regions. There exists some small overlap between the region AABBs, as they get scaled up slightly to prevent numerical errors that result in no matching region being found, but for those locations either region is fine.

See Figure 4.11 for a representation of this method.

4.6.2 Collecting Samples

The samples need to store:

- the position of the sample
- the direction the light is coming from, which is the direction in which the path continues on

- the pdf of choosing the direction
- the weight of the sample, i.e. the light accumulated in successive bounces divided by the pdf of the sample direction
- the index of the region the sample is in
- optionally for parallax-aware path guiding also the the distance to the next surface along the path with a non-discrete BSDF (e.g. not dielectric or specular surfaces)

These samples can only be collected on surfaces with non-discrete BSDFs, as the combination of samples from surfaces where the direction is based on a probability density function (pdf), or for discrete BSDFs on a probability, are not combine-able.

To collect this data, for each bounce all previous samples of the ray need to be updated based on the light encountered at this bounce, as well as updating the distance for the previous sample.

If NEE is enabled, then direct light contributions at the sample location are ignored, as that contribution can easily be sampled with NEE and does not need to be evaluated by directing the path towards it. NEE contributions of the next bounces are of course again taken into account, as the path needed to actually reach that location.

To store this data, a large array of sample data is created and each pixel gets assigned a certain region of this array to store the samples in, but limited to a defined maximum number of samples per pixel. All these sample data are first marked as invalid at the beginning of each frame, so that only sample data from the current frame gets used.

This buffer is then copied to CPU memory and has to be sorted by region, so that each region's VMM can be separately fitted to only the data collected in its region. This is done in a CPU-based parallel sorting step with the end result being array ranges containing sample data of the same region.

4.6.3 Fitting VMMs

The sorted sample data is then used to fit the VMMs to the samples collected in their region using the implementation of [RHL20]. This implementation is written in a vectorized form that allows for fast processing and is also very robust as well as thoroughly tested.

When enabled, splitting and merging of components of the VMMs can be triggered. As described in Section 3.2.8, this splits a vMF into two separate vMFs if they are found to represent two or more different light effects, or merges two vMFs if they approximate the same one. More information about the merging and splitting criteria and implementation can be found in [RHL20].
The number of vMF components in a VMM are limited to a constant maximum and a field in the VMM keeps the number of active components.

For parallax-aware path guiding, the samples need to be re-projected to a central location before fitting, as well as an extra step that updates the distance value accompanying each Von Mises-Fischer distribution (vMF). With this distance value per vMF, the target location each vMF is pointing at can be calculated and also added to the vMF GPU struct.

This is also the step where decisions about refining the spatial partitioning of the guiding regions can be taken. If the number of samples collected for one region rises above a certain threshold, then the corresponding region gets split and the two new regions initialized based on the split regions guiding VMM. In the next iteration, these new regions then collect samples and get optimized independently, thereby increasing the accuracy of the guiding information by focusing on local effects.
As this changed the region AABBs, the AS needs to be updated.

These newly optimized VMMs then need to be synchronized to the GPU.

4.6.4 Using Path Guiding

When path guiding is enabled, a certain percentage, e.g. 50%, of new direction sampling is done by sampling the VMMs. The other part is still based on BSDF sampling and is used to keep exploring for new paths as well as preventing bad fits from only producing useless rays.

The VMM is selected by querying the current guiding region as described in 4.6.1. To sample the mixture model (VMM), one of its vMF distributions is randomly selected based on the associated weight π_k. This vMF can then easily be sampled using two random numbers [Jak12].

The pdf p for sampling that direction ω_i then needs to take the VMM into account as well as the BSDF.

$$p(\omega_i, x, \omega_o) = a\, p_{VMM}(\omega_i, x) + (1-a)\, p_{BSDF}(\omega_i, x, \omega_o) \tag{4.4}$$

Where a is the probability of using the VMM sample method.

4.6.5 Parallax-aware Path Guiding

When Parallax-aware Path Guiding is enabled, each vMF, except environment map contributions, has a target location t associated with it. With this information about at what surface-point the light contribution originated from, the direction μ of the vMF can be adjusted to point towards this location, independent from the location o in the region

$$\mu = \frac{t-o}{||t-o||} \tag{4.5}$$

This step is always done prior to sampling or evaluating the VMM.

4.7 Visualizations

This section contains the many visualization options, that are available to further explore the effects of the implemented methods or to get a better look at behind the scenes details.

All these visualizations can be enabled and configured during the runtime of the path tracer using the expansive UI (Figure 4.12).

The possible visualizations are:

- The values of the Irradiance Cache directly without surface interactions (Figure 4.13)
- The IC translational and rotational gradients (Figure 4.14)
- The pixel estimate image for ADRRS (Figure 4.15)
- The number of ray splits that were triggered by ADRRS (Figure 4.16)
- The depth of the rays until they escaped the scene or were terminated by ADRRS (Figure 4.17)
- The location of the path guiding regions in the scene (Figure 4.18)
- Spherical representations of the path guiding VMMs in the scene, as well as in 2D (Figure 4.19)
- A mode in which the spherical VMMs are animated to move inside their region to highlight the effects of parallax-aware path guiding (Figure 4.20)
- The number of active components in the path guiding VMMs to show the effects of splitting and merging (Figure 4.21)

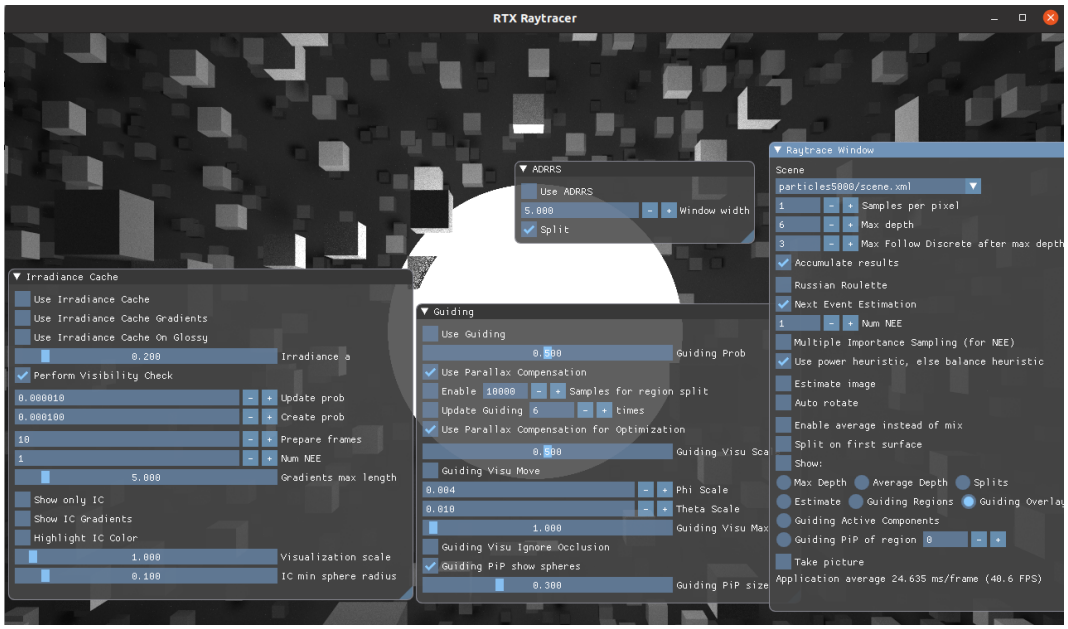

Figure 4.12: The UI of the RTX path tracer with all collapsible panels visible. The most important panel is the one on the right, containing basic settings like NEE, samples per pixel, depth, or different visualization modes (bottom).
All other methods also have varying customization options and can be enabled at any time for immediate comparisons between different modes. The lower third of Irradiance Caching and the lower half of Guiding contain only visualization options to get a look behind the curtains, by e.g. directly showing the path guiding VMMs in the scene.

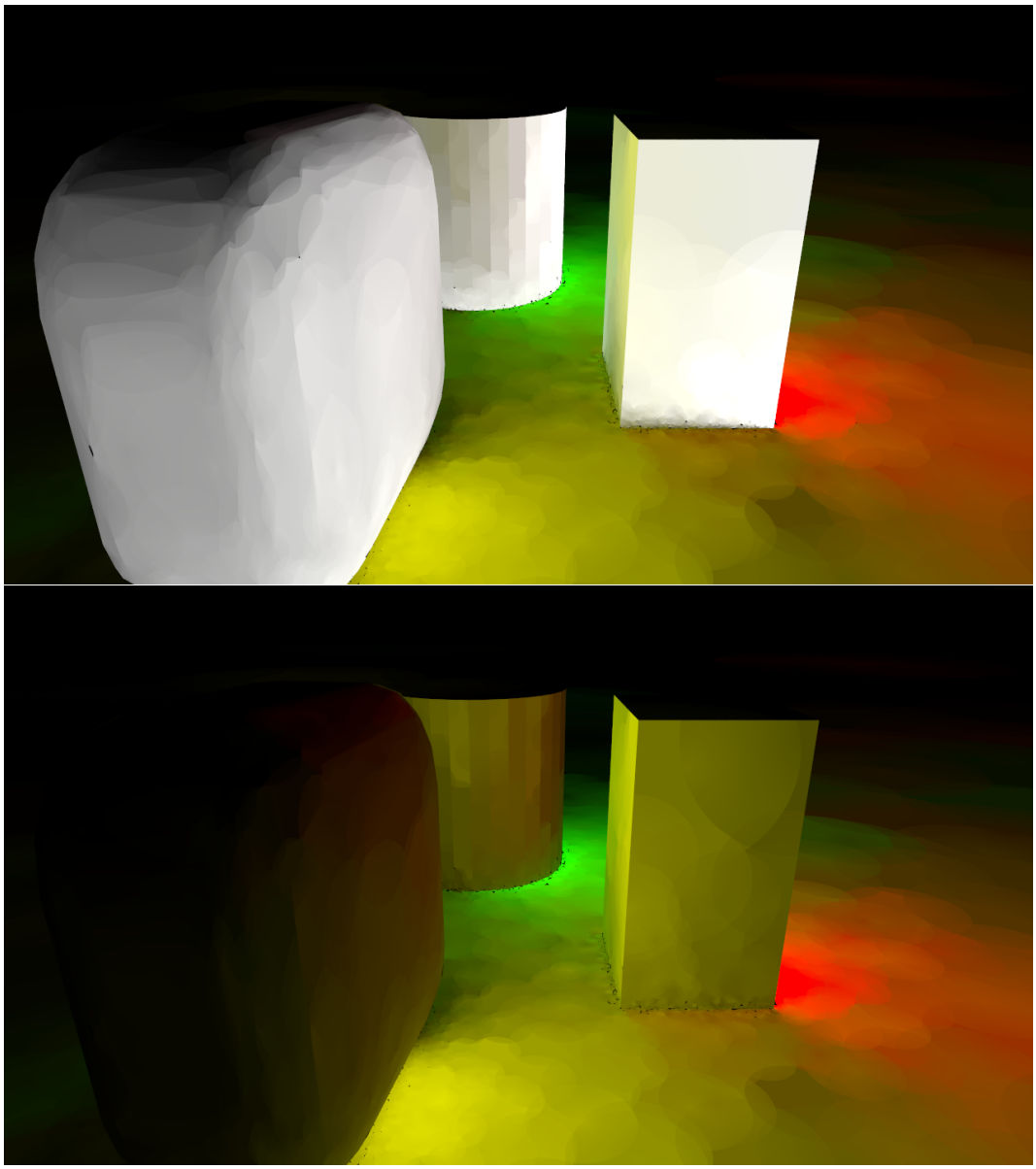

Figure 4.13: Visualization mode showing the IC values directly without any surface properties. The black spots along the bottom of the objects are holes between the irradiance caches, that occur especially in these corners, as the ICs naturally get very small and no caches overlap these locations. This visualization shows the important effects of indirect illumination and can be a good tool to follow the creation of the IC entries.
The bottom image uses the color highlighting mode that removes the white component of the irradiance cache and only shows the remaining color. In practice, this helps visualize the effects of updating caches and the resulting propagation of global illumination effects between caches.

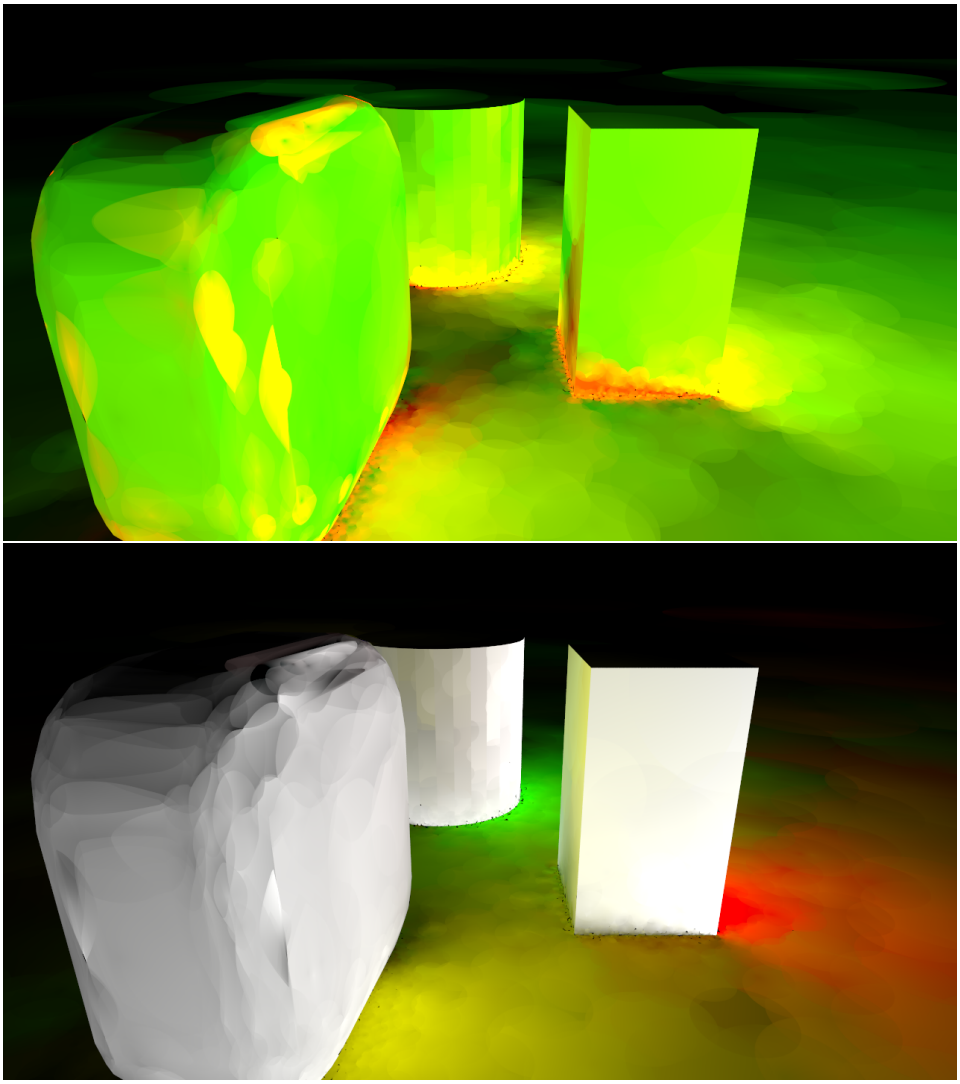

Figure 4.14: The top image shows the visualization mode that shows the strength of the IC gradients. The red component scales with the length of the translational gradient and the green component with the rotational gradient.
This visualization is a great tool to show what kind of situations result in larger gradients. Especially near corners, small translations can have big effects on the resulting irradiance as more of the hemisphere is obstructed, as can be seen by the red regions around the bottom of the objects. Many other regions are more dependent on the angle of the surface, as e.g. more of the bright floor would be visible if the surface tilts towards it.
The bottom image contains the IC values when gradients are used. Using gradients results in smoother irradiance distributions, as the areas between different IC entries are naturally smoothed as all IC entries are being adjusted for that position.
Also visible are artefacts that occur when the gradient estimated is too large, as can be seen on the left object.

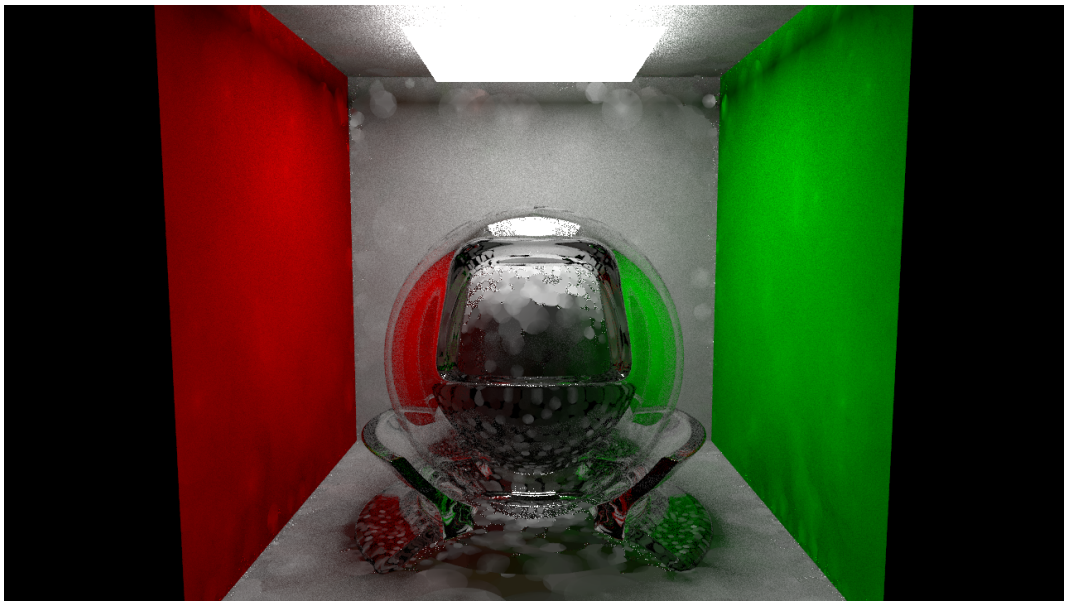

Figure 4.15: Visualizations of the estimate image used for ADRRS. This estimate is created by querying the IC on the first non-discrete surface and applying NEE a few times.

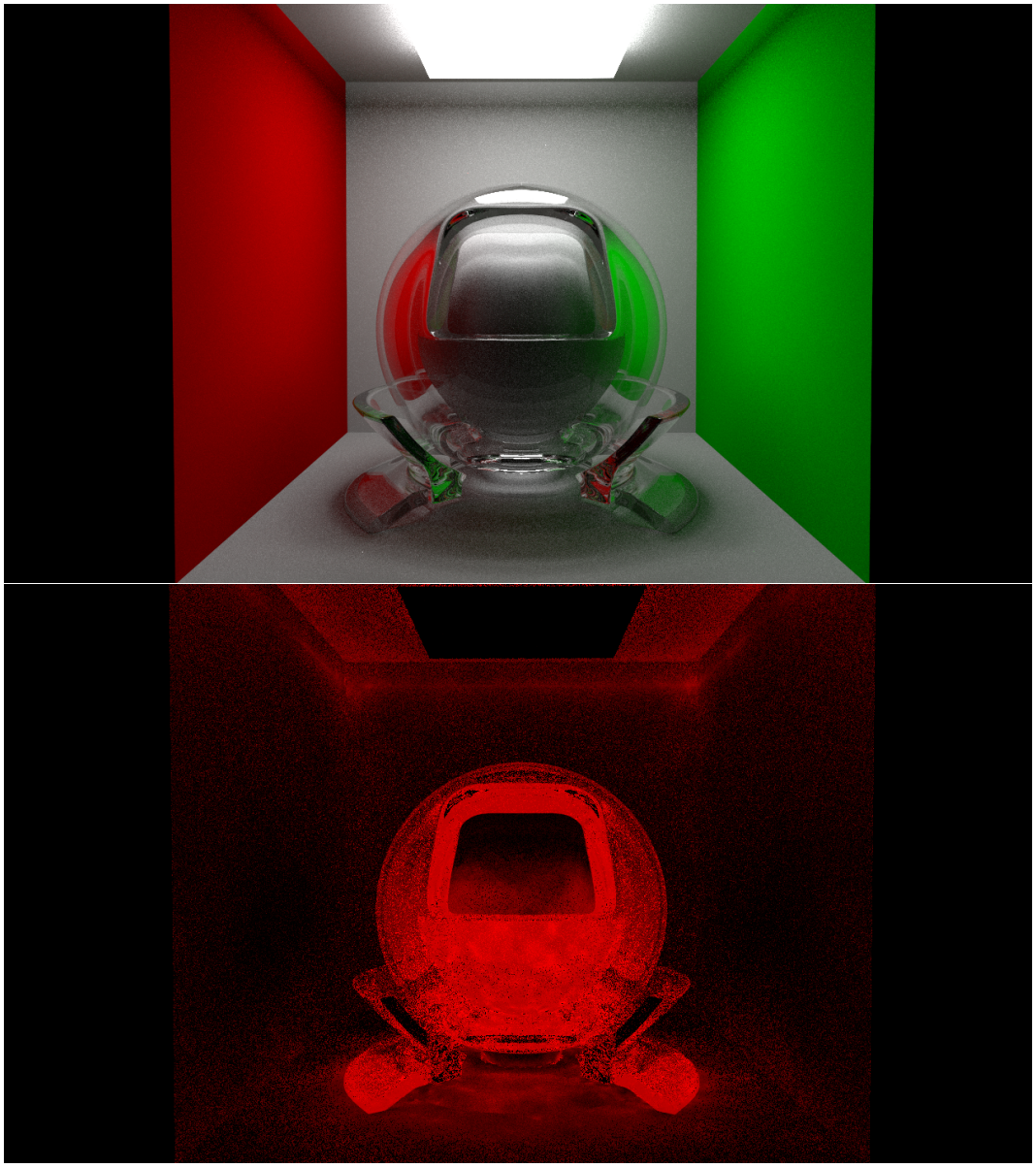

Figure 4.16: Visualizations of the ray splits per pixel, where stronger red values signify more splits. Interesting to see here is the dark region on a plane with the light source, that was explored with more splits which was probably due to the reduced direct light contribution and therefore resulting higher importance of indirect illumination, as well as the IC not being accurate enough and making the shadow estimate brighter than it actually is, see Figure 4.15. The difference between different regions of the glass body are also interesting, as well as the increased splits close to the glass object.
On the other hand, the rays hitting the light source directly did not get any splits at all, as further exploration of the indirect illumination would not change anything in the pixel value.

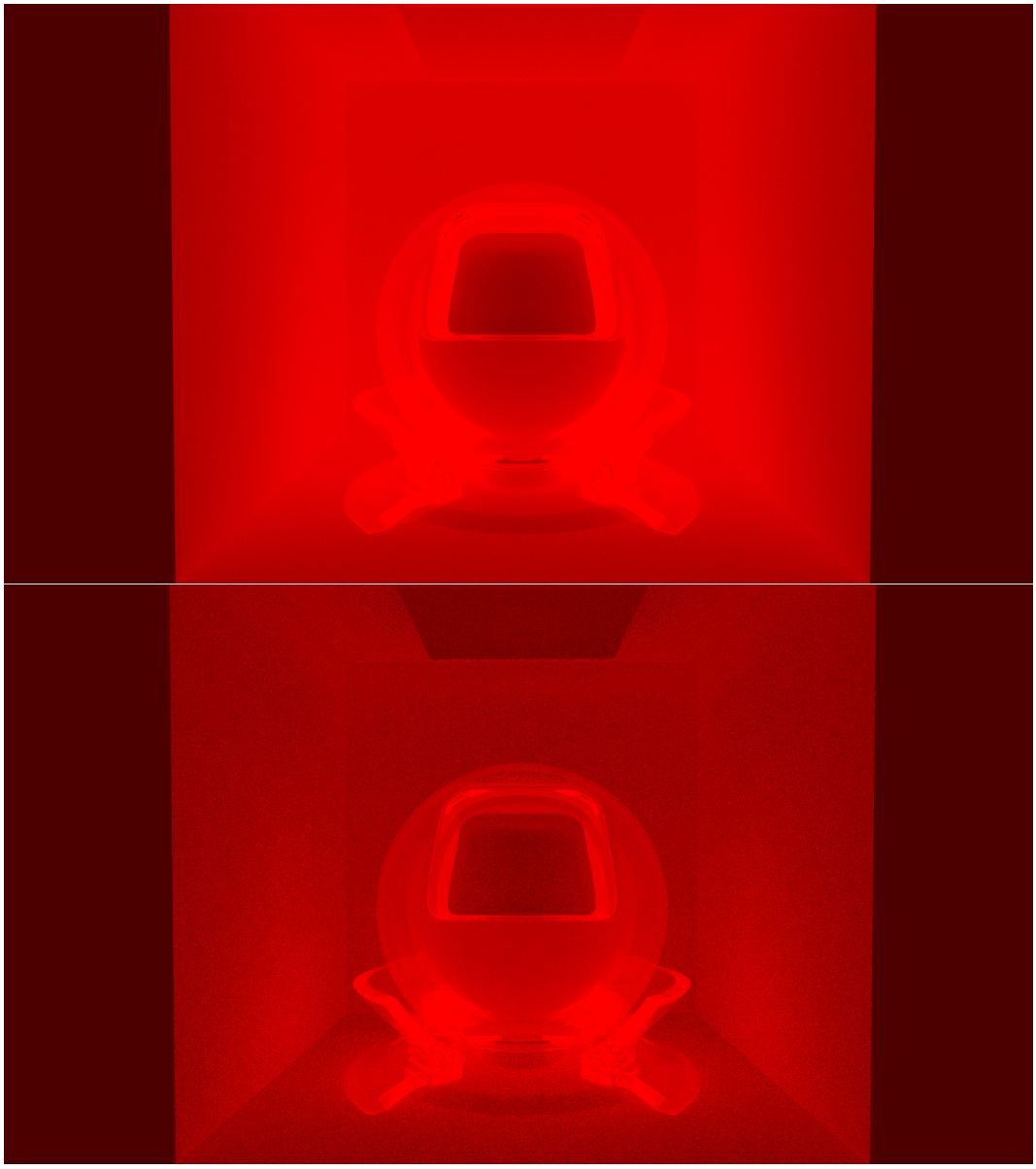

Figure 4.17: Visualizations of the average depth of the rays per pixel, where stronger red values signify higher depth values.
The upper image is from the normal path tracer, while the bottom one is with ADRRS active. In the standard path tracer image, one can see, that the average depth of the rays solely depend on the depth in the scene and how long it takes the ray to leave the scene. In contrast, ADRRS reduces the depth of rays in simple regions like the flat wall dramatically where few bounces are already enough to evaluate the most important lighting effects. Also regions with bright reflections, like the top of the glass object, or the light itself were evaluated with significantly reduced ray depth, as any following bounces and their expected contribution are not likely to change the final pixel value.

Figure 4.18: This is an image of the visualization mode, where each surface is colored by the color of the corresponding path guiding region.

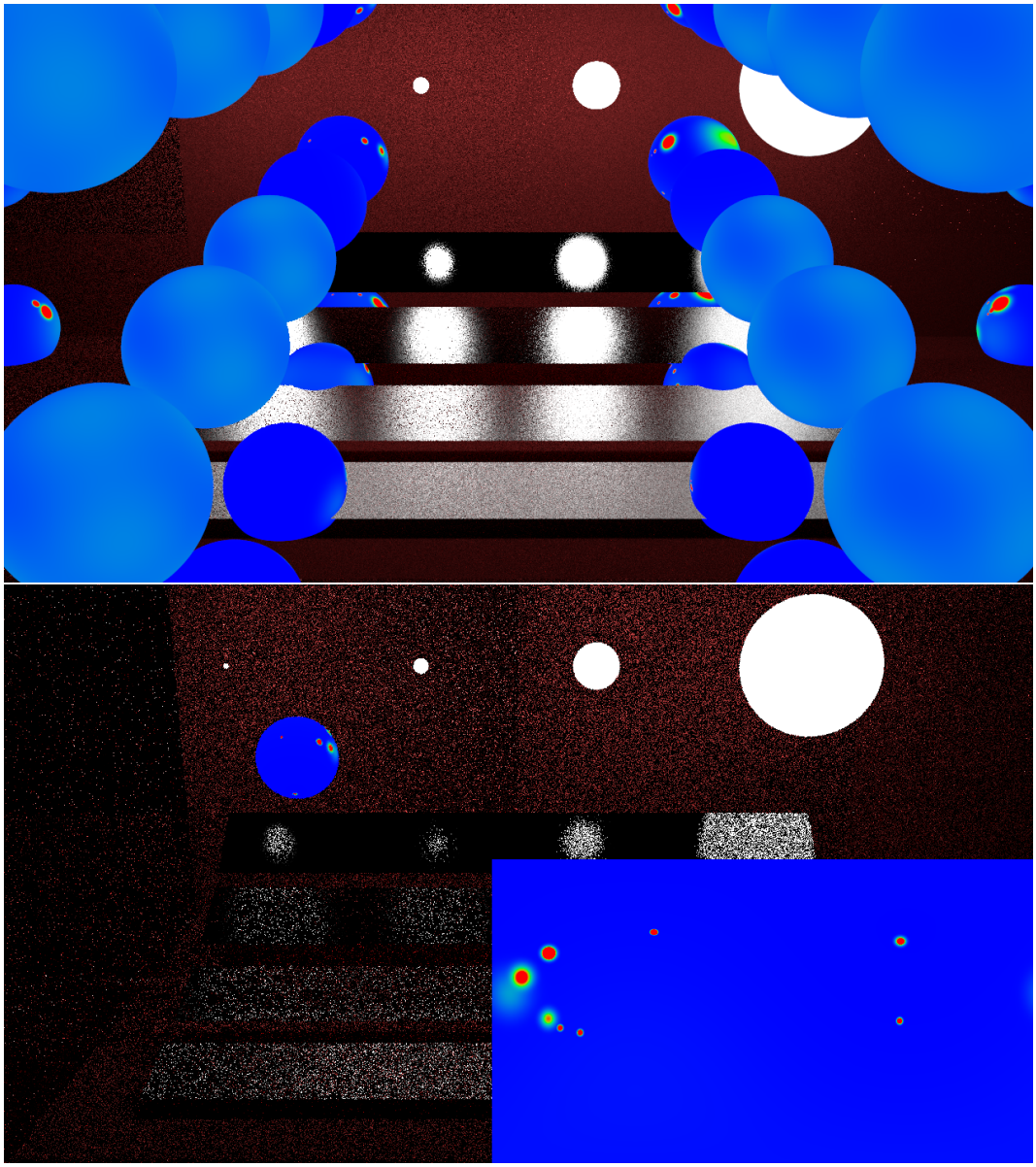

Figure 4.19: For path guiding, visualizations were created that show the actual VMM distribution as spherical representations in the scene (top) or a single one as a picture-in-picture 2D representation.

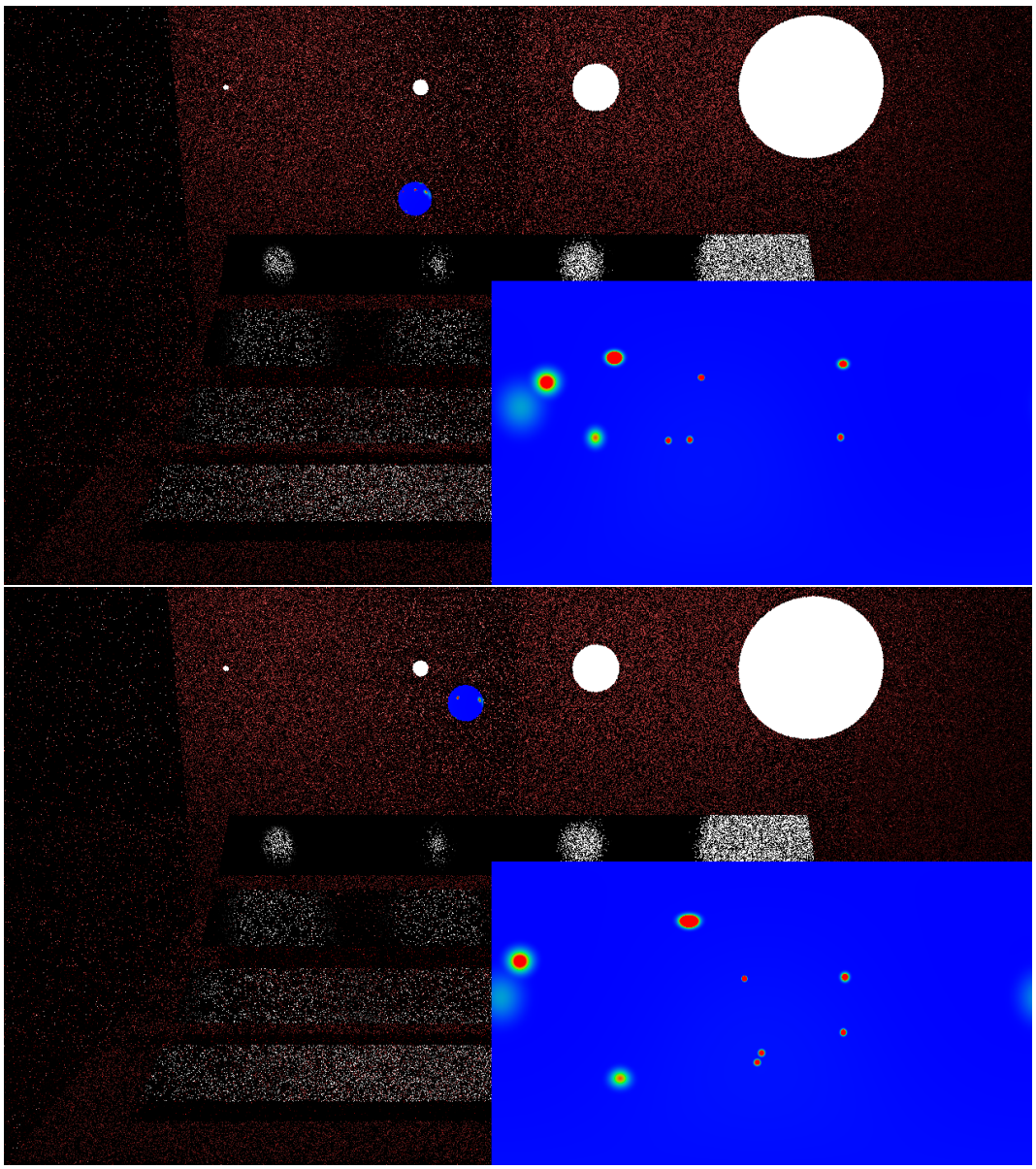

Figure 4.20: To highlight the effects of parallax-aware path guiding, the visualization of the VMM in the scene can also be animated, so that the query location of the VMM varies over time. This way, the parallax compensation can easily be seen and understood, as the distributions react to their position in real time.

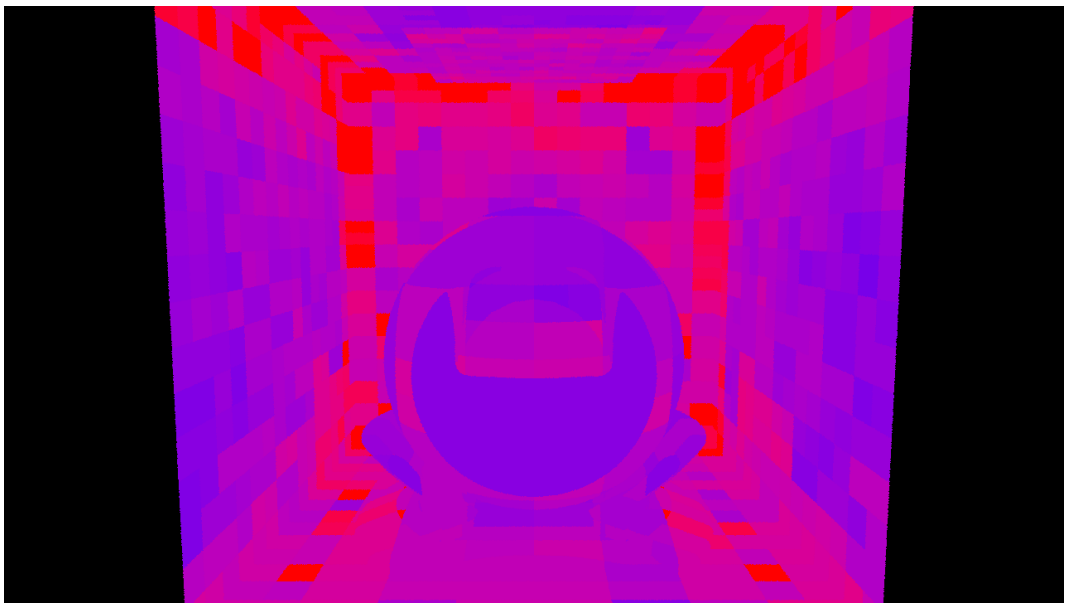

Figure 4.21: This visualization shows the number of active components of the guiding VMM, so how many vMF distributions were needed to accurately represent the local radiance. This highlights the effects splitting and merging have on the final guiding VMM, as they split components that contain samples from multiple distinct effects and merge components that actually show the same light path.
This image was taken with a maximum of 32 components, red representing more active components.

5 Evaluation

This chapter contains different evaluations that aim to show how the different implemented methods perform on different scenes, how scene size impacts performance for these GPU based ray tracing operations, the effects of the samples per pixel per frame setting, as well as some profiling and GPU utilization analysis.

All images used or shown in this chapter were rendered with a 1280x720 resolution on a NVIDIA RTX 2080Ti graphics card with an Intel i5 6600k CPU (4 cores, 4 threads).

5.1 Impact of scene size

To evaluate the impact of the scene size on the performance, multiple scenes were created that display approximately the same geometry, while containing very different amounts of triangles.

For this, the Crytek Sponza scene was loaded into Blender, where the mesh could be simplified with the *decimate* command. This way, a certain percentage of faces is removed by merging them with neighboring faces. To increase the number of faces, the mesh was *subdivided* so that each face got split into multiple faces.

With these modifications, 10 variants of the scene were created, that all contain wildly different amounts of triangles. 26k triangles at the lowest, and 4.2 million at the highest.

To remove all other variables, the scene was also configured to only contain pure white diffuse materials.

The resulting frames per second can be found in Figure 5.1. The 4 different data sets were created with different path tracer settings, like with or without NEE and to ray depth 6 or up to 20 bounces. Also, one of the two graphs shows the data point in a logarithmic scale. NEE seems to be less impacted by the scene size, as can be seen by the shallower angle of the graph.

With the logarithmic scale for the triangles axis, the correlation to the logarithm of the scene size becomes apparent, as independently of the path tracer configuration used, the resulting FPS always approximately lay on a line. This is the expected behaviour for classic acceleration structures like BVHs, as the intersection complexity scales with $\log N$.

This means, that NVIDIA's implementation of the acceleration structure does not seem to have any additional scene size dependent overhead and therefore scales well with the number of faces in the scene.

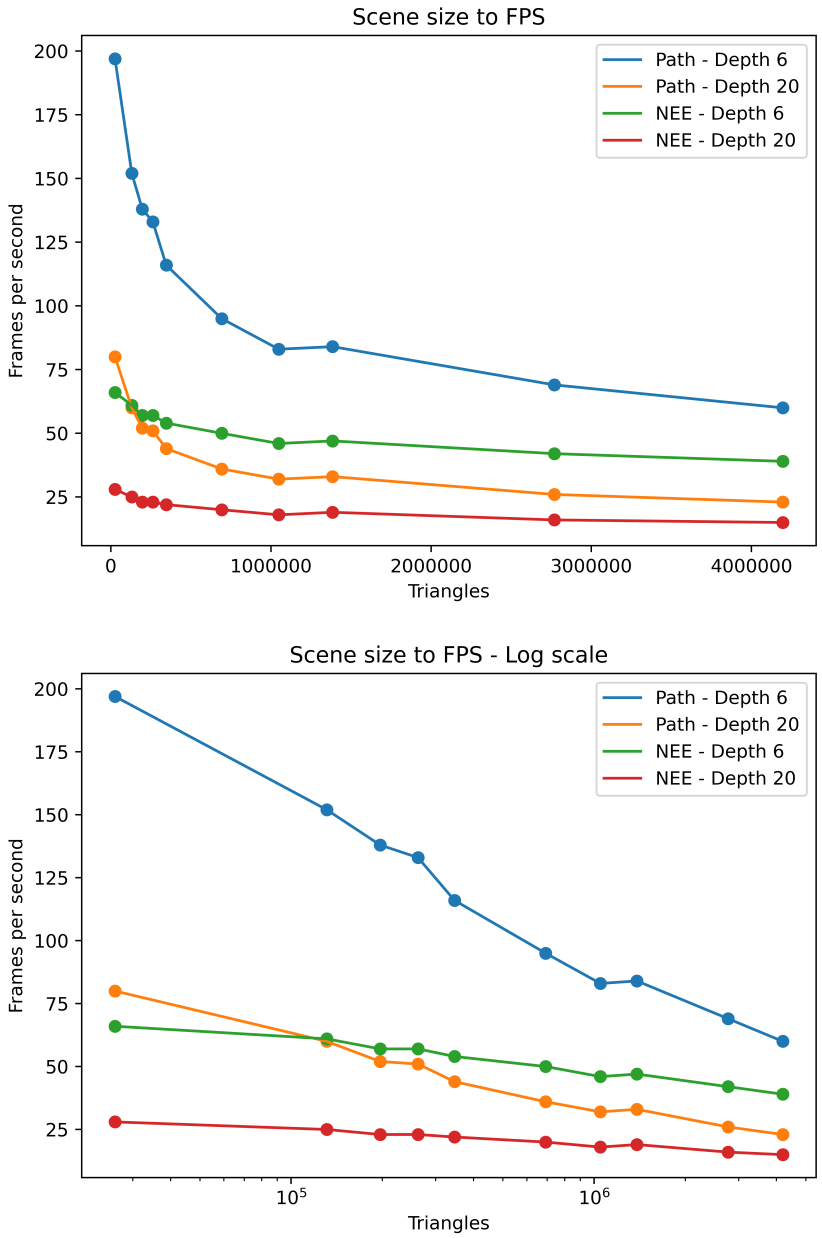

Figure 5.1: The frames per second for 4 different path tracing settings and different scene sizes. The different scene sizes were created by subdividing and decimating the mesh of the sponza scene.
One of the graphs has the triangle axis displayed in log scale. There, one can see, that the time per frame seems to scale with the logarithm of the scene size, which is the expected behaviour for acceleration structures like BVHs.

5.2 Sample per pixel per frame

One important setting for the path tracer is the amount of samples per pixel that should be calculated for each frame. It can of course be used to show how the quality of the image changes, when the number of samples per frame is varied and only the results from one frame are shown. But also an important consideration is, how the performance of the accumulated result over multiple frames is impacted by this samples per frame configuration. When the goal is to create a good result in the shortest possible time, then how is this dependent on the samples per frame?

To evaluate this, 4 different scenes and two path tracer configurations, with or without NEE, were used.
The 4 scenes vary drastically, especially in regards to scene size and the amount of regions where the ray can escape the scene. The irradiance cache scene (see Figure 4.7) is relatively simple and has an unobstructed sky. The rough conductor is just a floating material test object combined with an environment map. Crytek Sponza is relatively complex scene and has the central section that leads to the sky. The fireplace scene is a completely enclosed room and is also fairly complex.

Figure 5.2 contains the results, where the resulting samples per second are drawn against different samples per frame settings.

In general, the total samples per second seems to increase in most cases, when the samples per frame are increased, in some cases even significantly so. The overhead of displaying the frames has of course a smaller performance impact if they happen less often, as well as any other GPU to CPU communications that might be necessary, which is why a performance increase was expected with higher samples per frame.

Especially on scenes where the rays can escape easily, the performance increase was drastic. This performance increase seems to be more than just the frame display overhead being reduced. A possible explanation might be the better compute shader warp coherence, as with a lower samples per frame settings, rays escaping might lead to other rays in the same warp, that continue on for multiple bounces, to not have any parallel workload anymore.

A disadvantage of the increased samples per frame is the reduction of the frames per second to sometimes non-interactive fps. E.g. the Crytek Sponza scene with NEE and 32 samples per frame dropped to 1.1 fps which makes any interactions with the camera or rendering settings a very tedious process. Other methods with higher complexity like Path Guiding or ADRRS reach this non-interactive threshold even sooner.

A general recommendation would be to use low samples per frame (e.g. 1 to 16) for situations where the interactive component is important, such as live switching between different methods to show differences or when moving around the scene. High samples per frame can then be used for fixed camera renderings that should reach high quality quickly.

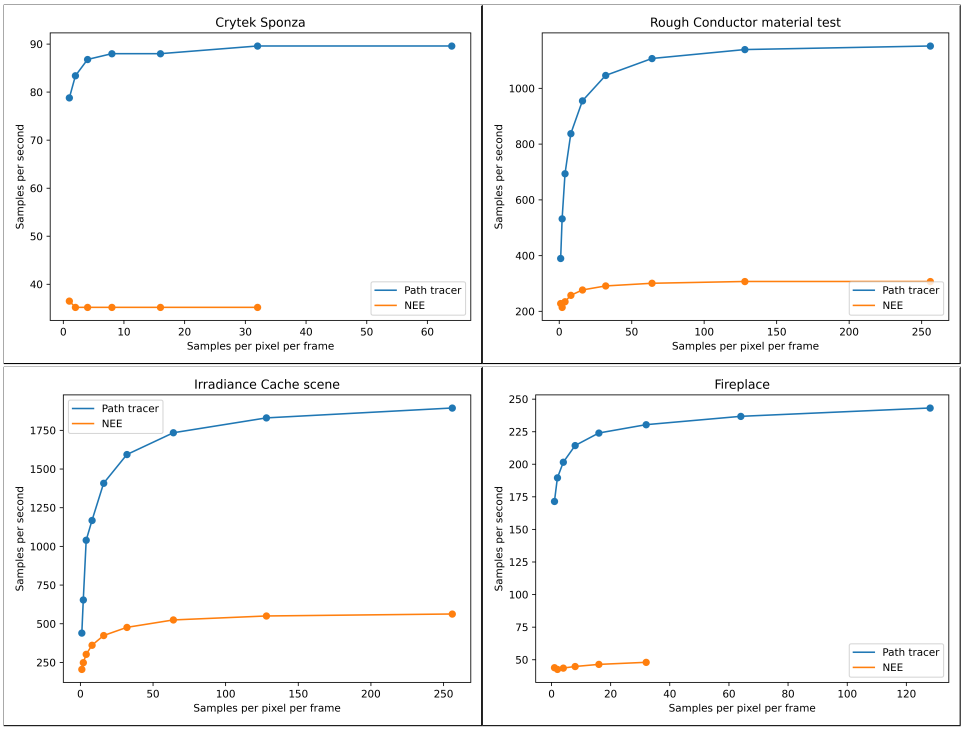

Figure 5.2: The path tracer can be configured to compute a specified amount of samples per pixel for each frame. These graphs show the resulting samples per second when using different amounts of samples per frame for different scenes.
The total amount of samples calculated increases in most cases, when the number of samples per frame is increased. This is especially apparent on scenes where the rays can escape the scene more easily, like the irradiance cache scene (open sky), or the rough conductor material test scene (free floating material test object).

5.3 Convergence rate for different methods

This section aims to analyze the quality and speed of the different methods that were implemented for the path tracer, and compare it against a standard CPU based path tracer in the form of Mitsuba [Jak10]. With Mitsuba, the path tracer integrator will be used, which is the equivalent of NEE with MIS. As the CPU used is comparably slow for parallel workloads with only 4 cores, most of the time, instead of equal time comparisons, also 5 times more time for Mitsuba will be provided.
For the RTX path tracer, all methods will be compared with an equal time comparison that includes the startup time of around 22 seconds.

To evaluate the quality of the results, the relative mean squared error (relMSE) metric is used, with some slight adjustments. The strongest outliers (0.1%) will be excluded to prevent e.g. fireflies from influencing the quality metric disproportionate. Also a small epsilon (0.001) is added to prevent division by zero. The lower this relMSE value is, the better the result. This is the evaluation setup as used in [RHL20].

For this, 4 very different scenes were used, and the reference image created with the RTX path tracer, sometimes with NEE or MIS, using many samples.

5.3.1 Sponza

The Sponza scene consists of many diffuse surfaces with textures, with a small light source slightly above the building. Results of the path tracer and NEE can be seen in 5.3, quality metrics in 5.1.

The path tracer created many more samples than any other method, but the resulting image is still entirely grainy and unusable. This is due to the very low probability of actually sampling the direction of the small light source, which results in most rays not having any light contribution at all.

NEE explicitly samples the light source and can therefore easily deal with this situation. This, in combination with the low complexity and resulting high sample count, results in an almost perfect image.

Mitsuba also creates a very high quality result after 10 minutes, and only the lower sample count still leaves some residual noise in the image.

Irradiance Cache was evaluated two times, once with IC updates over time and once without. Using updates resulted in slightly lower total samples, but better quality, as the IC values were refined over time and could use the values of the other caches for more accurate results.

ADRRS has too much overhead, which only resulted in 62 samples in two minutes. This low sample count is due to having to first fill the IC and then also querying the IC at each intersection instead of once per ray query for the IC method. Also, a possible explanation of the reduced performance is the strongly diverging behaviour

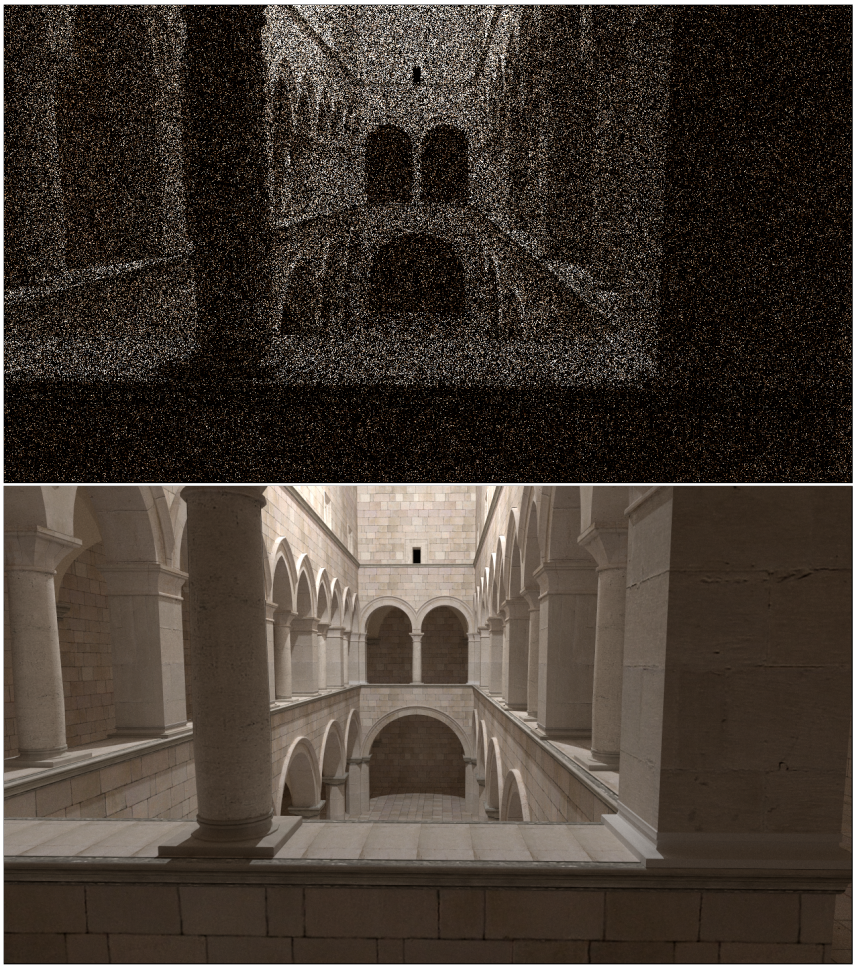

Figure 5.3: Pure path tracing in comparison with NEE on the sponza scene after 2 minutes

that can be found in neighbouring rays, as some get terminated early with RR and others even get split. This most likely reduces the warp coherence and leads to under utilized warps, that need to serialize work.

Path Guiding without NEE produced way better results in the upper area of the scene, as the rays could get guided to actually hit the light. But the low samples per pixel resulted in all other regions not being explored enough and the image still looking very unfinished.

Path Guiding with NEE just created computational overhead and therefore lower quality when compared to pure NEE with its significantly higher sample count.

	Mitsuba	Mitsuba	Path Tracer	NEE	NEE + MIS
relMSE	0.217	0.048	31.131	**0.017**	0.028
SPP	72	360	**3712**	1408	800
time	2m9	10m40	2m	2m	2m

	IC no updates	IC	ADRRS	Guiding Parallax	Guiding Parallax + NEE
relMSE	0.37	0.222	0.216	78.807	0.046
SPP	864	800	62	976	604
time	2m	2m	2m	2m	2m

Table 5.1: Quality metrics and total samples per pixel for the Sponza scene.

5.3.2 Veach MIS

The scene is inspired by Veach's scene in [VG95] and consists of rough surfaces with varying roughness values. The bottom most plate is close to diffuse and each plate further up gets closer to mirror properties. The four light sources are of very different sizes and always reflect off of all the plates towards the camera.

Figure 5.4 shows the results after 2 minutes for the path tracer and NEE with MIS. Table 5.2 shows the quality metrics for the different methods.

As this is a scene that was created to show the need for MIS, NEE with MIS resulted in the best result by far. The path tracer still could not remove the noise created on the diffuse surfaces trying to randomly hit the smallest light source, as can be seen on the bottom plate and the back wall around the light source. And this was even with around 10 times more samples per pixel than NEE with MIS.

NEE created good results in most parts of the image, but just as shown in Figure 4.5, still struggles with the combination of the large light source and the mirror-like plate, as even 12000 samples per pixel were not enough to sample the correct mirror direction spot of the large light source for some locations.

The other methods IC, ADRRS, and Path Guiding all perform okay, but are worse than Mitsuba's two minute result, as they were not configured to use MIS and therefore had the same problems as NEE while producing significantly less samples.

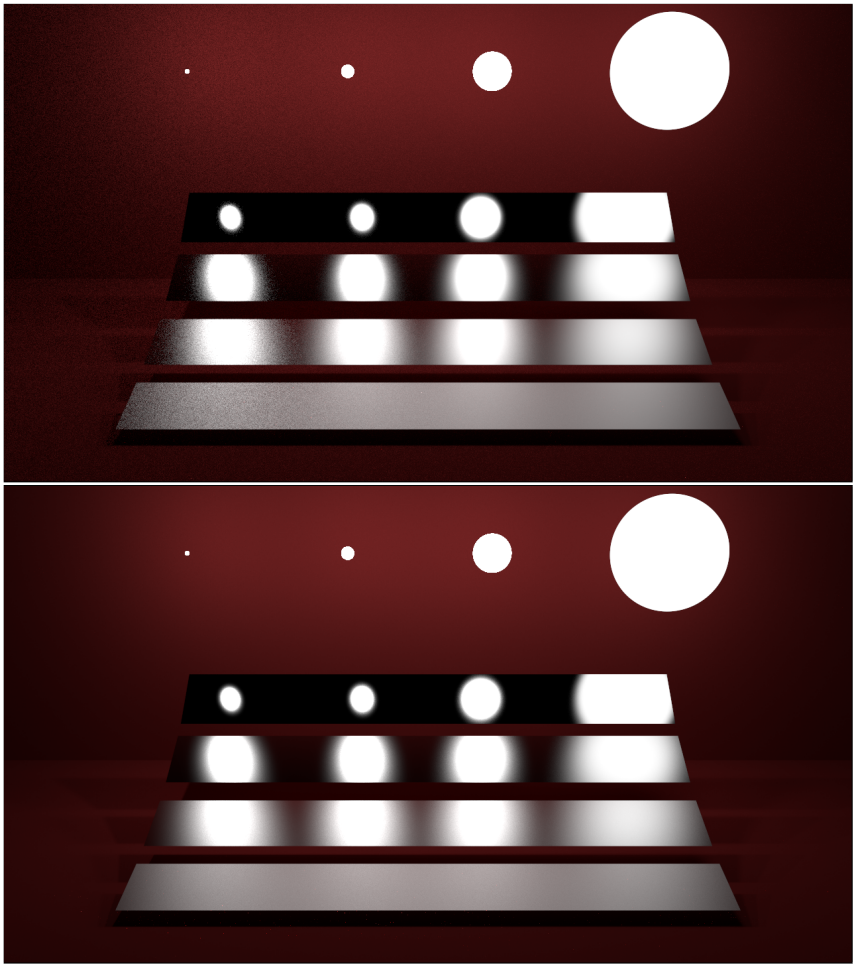

Figure 5.4: Pure path tracing (top) vs NEE with MIS on a Veach MIS [VG95] inspired scene

	Mitsuba	Mitsuba	Path tracer	NEE	NEE + MIS
relMSE	0.0182	0.0126	0.1206	0.0058	**0.0017**
SPP	500	2500	**50944**	12128	5888
time	2m3	10m17	2m	2m	2m

	Irradiance Cache	ADRRS	Guiding parallax + NEE
relMSE	0.0225	0.0349	0.035
SPP	4416	1376	1344
time	2m	2m	2m

Table 5.2: Quality metrics and total samples per pixel for the Veach MIS scene

5.3.3 Cornell Box Dielectric

The scene is based on a Cornell box, i.e. a box with colored walls on the left and right, with a dielectric material test object with a diffuse ball in the center. It contains light paths of widely different lenghts as seen in Figure 4.17. Some glass regions capture the ray for a long time, while other regions have rays escaping the scene easily.

Three resulting images of the path tracer, Mitsuba, and IC can be found in Figure 5.5, the quality metrics in Table 5.3.

NEE and path tracing both created great images, but NEE contains a few fireflies, which are completely absent from the path traced image.

MIS did not have any glossy regions where its strengths could be utilized, and so its overhead and resulting fewer samples just reduced the overall quality. But even then, it still significantly outperforms the equivalent Mitsuba integrator and even collects three times more samples in two minutes, than Mitsuba generated in 12 minutes.

The Path Guiding results are good, but again are outperformed by the significantly higher number of rays collected in the standard methods. But importantly, it outperformed the Mitsuba path tracers (2min and 12min) as it handled the region below the glass object much better, even with less samples.

The Irradiance Cache result beats Mitsuba in the relMSE metric, but the resulting image contains clearly visible circular patterns that make the image way less appealing and subjectively worse. The good relMSE value can probably be attributed to the general color and brightness similarity to the real value, while containing no high frequency high intensity noise.

	Mitsuba	Mitsuba	Path Tracer	NEE	NEE + MIS
relMSE	0.449	0.1076	**0.0078**	**0.0078**	0.0123
SPP	200	1100	**11872**	5152	3136
time	2m13	12m1	2m	2m	2m

	Irradiance Cache	ADRRS	Guiding Parallax	Guiding Parallax + NEE
relMSE	0.0551	0.1549	0.0466	0.0479
SPP	992	304	976	860
time	2m	2m	2m	2m

Table 5.3: Quality metrics and total samples per pixel for the Cornell Box Dielectric scene

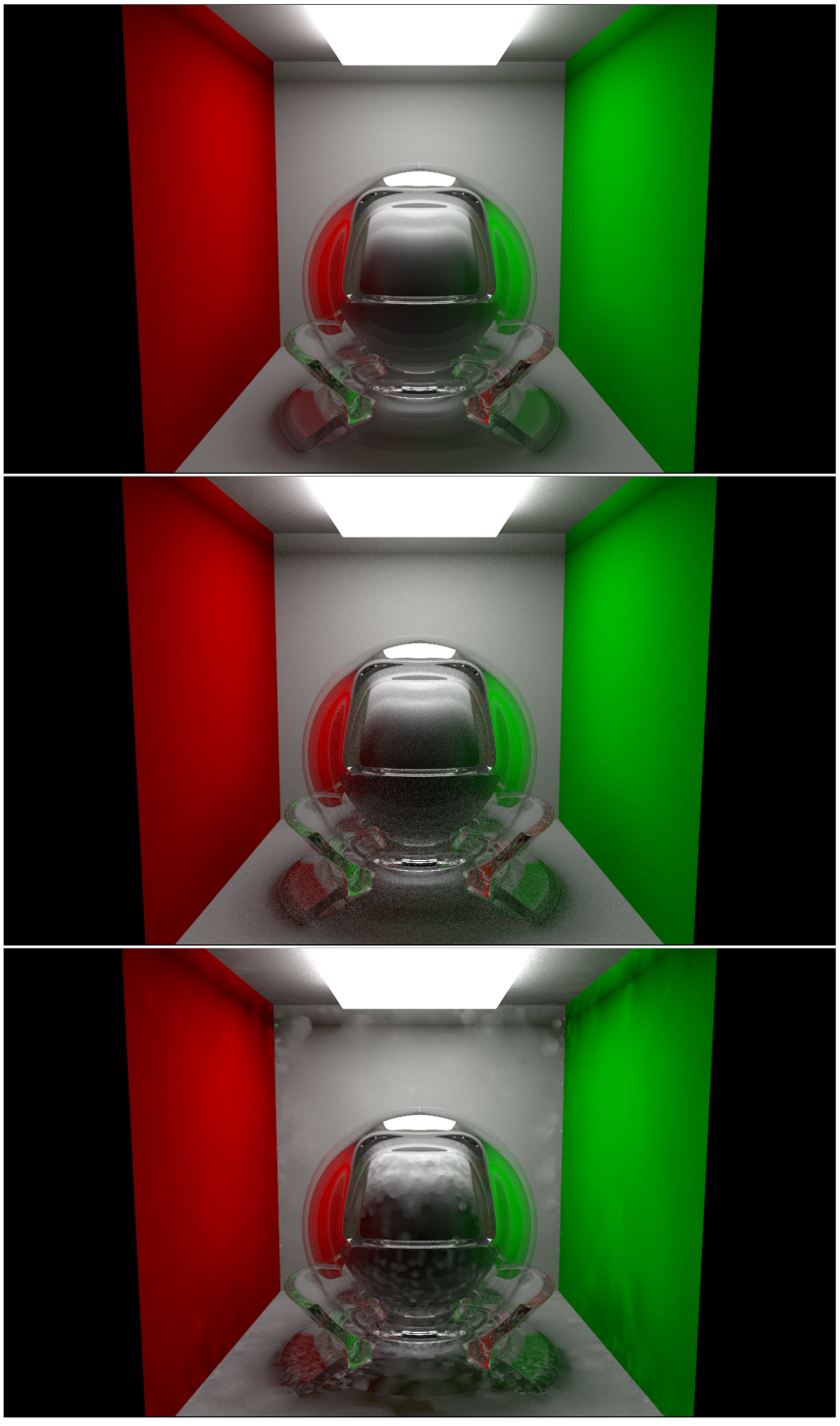

Figure 5.5: The resulting images for the path tracer in 2 minutes (top), and the result created with Mitsuba after 12 minutes (middle), and the Irradiance Cache result (bottom).

5.3.4 Clocks

The clocks scene is an extremely difficult path guiding scene where the only light source is behind a glass pane and most surfaces are diffuse. It was originally created for Stochastic Progressive Photon Mapping [HJ09]. This makes querying the light source directly impossible for all locations outside of the lamp shade, which makes the many diffuse surfaces especially hard to evaluate cleanly. To give this difficult scene more time to be sampled, 10 minutes was chosen as the render time. Also, the relMSE metric was adjusted slightly to exclude less outliers (0.001%), as the resulting images are still very noisy and this should be accounted for in the metric. Figure 5.6 and Table 5.4 show the results.

The first thing that becomes apparent is that 10 minutes is not even close to enough to create a clean image. This especially apparent for Mitsuba, as it only got a small portion of pixels that actually have any light contribution at all. 2550 samples per pixel was evidently not enough to navigate the many diffuse surfaces and the difficult light placement.

The RTX path tracer on the other hand was able to create 490k samples per pixel, which at least resulted in an image where an approximation of the final result can be seen, although still being very noisy. This huge number of samples is 190 times those created by Mitsuba in the same time, but of course Mitsuba also used NEE, which doesn't help much in this scene, and runs on a comparably weak CPU.

NEE created a worse result than the path tracer, as the overhead of sending extra rays to the light source, that in most cases could never be reached, reduced the sample count significantly enough to introduce noticeably more noise.

For parallax-aware path guiding, two renders were started, each training on the samples of 64 or 32 frames with 20 samples per frame, which took 190 seconds for the 64 samples. In the resulting image, two different regions can be seen, one where the training created viable guiding distributions and those where it didn't.
The regions that were trained well, produce generally good results, even with siginificantly less samples. Especially the floor is smoother in many areas than the path traced result. The area around one of the clocks and the left floor region could not be trained correctly and therefore only created rays that behaved like path tracer rays, but of course with significantly reduced sample count and therefore worse quality.

These regions could most likely not be trained, because the VMM fitting step needs a certain number of non-zero samples at once to start the fitting process. And as the fitting step is only ever based on the rays of the current frame, this means, that one frame of the render process had to have enough rays that intersected a point in the region and actually found a light source. For this configuration this would be 16 such samples that describe a path to a light source.
For the problematic regions, the probability of creating a path to the light source was

	Mitsuba	Path tracer	NEE	Guiding Parallax	Guiding Parallax
relMSE	28713	**347**	636	1351	2953
SPP	2550	**493312**	196736	16640	20848
time	10m12	10m	10m	10m	10m

Table 5.4: Clocks

so small, that these 16 samples in one frame never happened during the 64 training frames. The implementation of [RHL20] of course also contains methods to work around this issue, by e.g. collecting the directional samples in the beginning until enough samples for the initial training step were created. These methods were not yet implemented in the RTX path tracer.

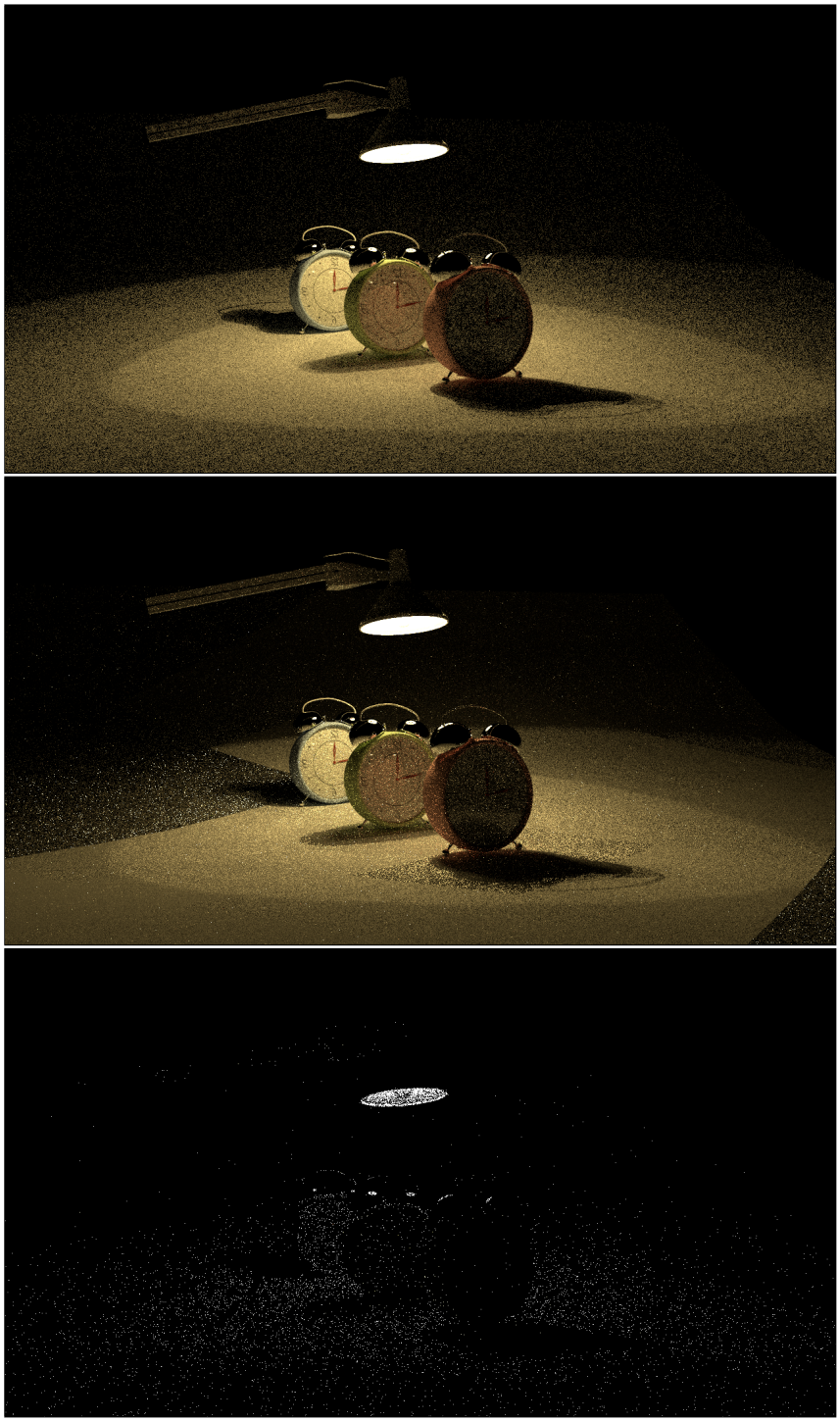

Figure 5.6: The images created in 10 minutes by path tracing (top), Parallax-aware Path Guiding (middle), and Mitsuba path tracing (bottom).

5.4 GPU Utilization and Profiling

To better understand the performance of the path tracer as well as possible bottlenecks, a profiling and GPU utilization trace were captured.

The GPU utilization over one frame as shown in Figure 5.7 highlights some important limitations of this implementation. It was created with NVIDIA Nsight Graphics.

During the Vulkan TraceRays command, the compute shader Streaming Multiprocessors (SM) are only utilized to approximately 50% of their capacity (unused warp slots in dark grey, active ones in orange). These compute warps, i.e. groups of parallel executions of the shaders, could not be started, because the shaders needed too many registers, and therefore had to use resources, that could otherwise have been used to run more warps in parallel.
This is possibly a result of implementing all these different methods in one combined shader to allow for easy method switching and method combination. A more specialised implementation might achieve higher Streaming Multiprocessor utilization and therefore higher parallel execution.

Also of note are the relatively low ALU (arithmetic and logical operations) and FMA (floating point operations) pipeline throughputs of 11% or 7% respectively. This makes it likely that the ray intersection unit is the limiting factor, but explicit ray tracing statistics are not available in this tool.

The profiling capture of the startup process, as seen in Figure 5.8, shows the Vulkan commands executed during startup as well as the first few frames and was captured with NVIDIA Nsight Systems.

The highlighted row contains the Vulkan API calls currently in progress. By far the most time consuming step in the startup phase is the call to create the ray tracing pipeline from the defined shaders and connected buffers, as it takes 25 seconds. In comparison, the acceleration structure creation was directly before the pipeline creation and took 4 milliseconds, which is not even visible in this graph.

This resulted in a total startup time of 28 seconds before the first frame could be created. In practice, this time varied between 20 and 40 seconds and was at the lower bounds, if the same shaders had been uploaded previously. So after developing new features or general changes, the startup time would be around 40 seconds, while any following starts would then only take 22 seconds. This is most likely a graphics card configuration caching feature, that increases performance for subsequent iterations.

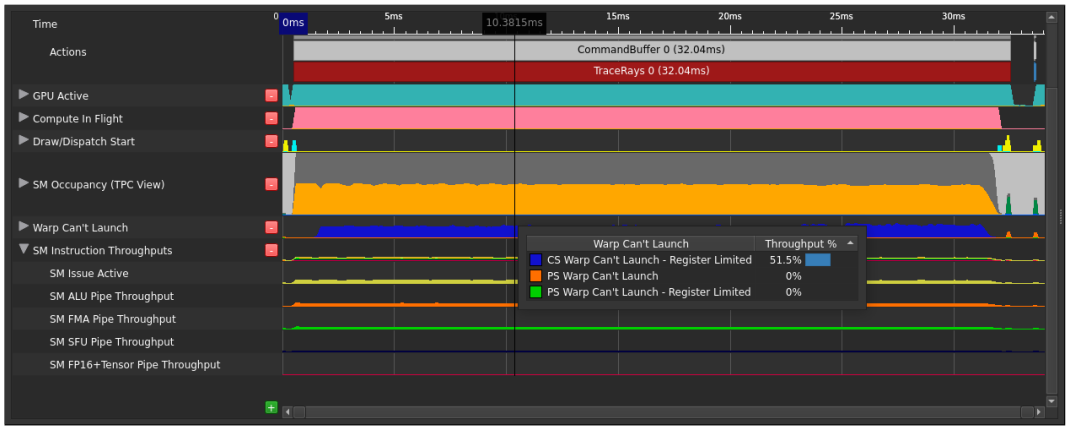

Figure 5.7: A GPU trace collected with NVIDIA Nsight Graphics of one frame.

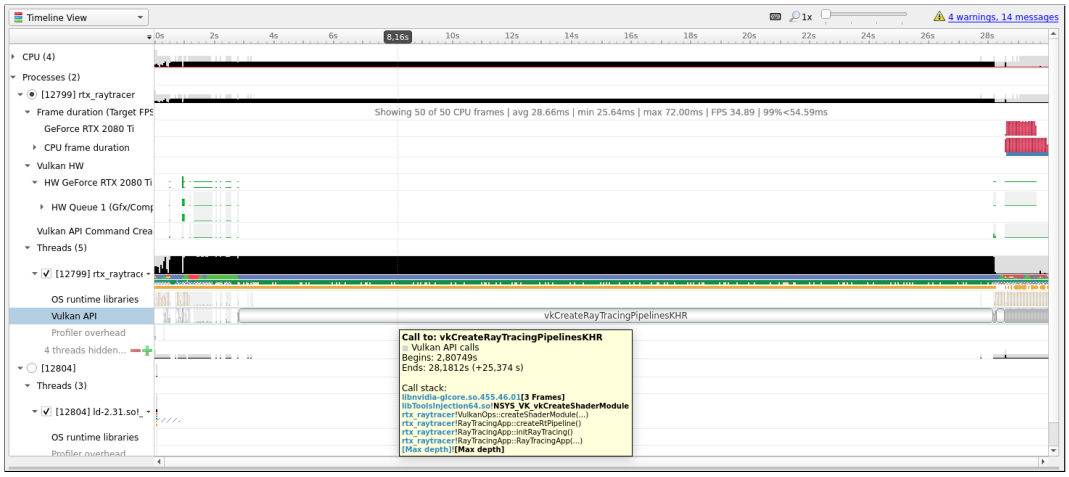

Figure 5.8: This is a profiling trace of the path tracer created with NVIDIA Nsight Systems. The red boxes in the top-right are the first ≈ 40 frames of the path tracer, which start after the startup phase of 28 seconds.

6 Conclusion

In conclusion, this master thesis resulted in an interactive path tracer that implements a wide range of advanced path tracing methods, that were adjusted to utilize the new RTX hardware.
This path tracer is very fast when compared to ordinary CPU path tracers, as the hardware accelerated ray tracing greatly increases the ray throughput and therefore the resulting sample count.

The implemented methods had very different performance impacts on the path tracing. NEE or MIS are important assets that in many cases increase the performance drastically, while being easy to implement. Other methods like ADRRS are too slow with the current implementation. ADRRS had no advantage over the vastly higher sample count achieved with simple path tracing or NEE. This was most likely related to the loss of warp coherence, as some rays get terminated early while others continue on or get split, which leads to work needing to be serialized.

Path guiding with training on the CPU works well with difficult path tracing scenes, while performing significantly worse for simpler scenes where the vastly higher performance of simpler methods outperforms it.
It would greatly benefit from more features of other implementations, like better initial training for complex regions, or in the best case even training on the GPU to remove the need for large data transfers between GPU and CPU after each training frame.

The combined implementation of all these methods in parallel, to allow for easy switching between methods and comparisons, resulted in too many and too large shaders, that increased the startup time substantially and resulted in reduced parallel execution due to resource conflicts in the form of registers. Specialized implementations are likely to experience significant performance gains in comparison to this broad implementation.

A positive of the combined implementation is that it allows for easy and fast switching between different methods to highlight the effects and differences interactively.
This, combined with the wide range of visualization modes, helps to understand the inner workings of the methods and the information used by them in the background.

All in all, this RTX accelerated path tracer is a great tool to interactively compare the quality and behaviour of the different methods, as results can be generated in a short amount of time and switching between different methods is easy and fast.

Future work could explore a wavefront based approach to reach higher utilization and better warp coherence.
For path guiding, the EM training step could be implemented on the GPU to optimize many guiding mixture models at once, while reducing the amount of data transferred between GPU and CPU.

Bibliography

[AMD20] AMD. Rdna 2 architecture, 2020. URL: https://www.amd.com/de/technologies/rdna-2.

[BWP+20] Benedikt Bitterli, Chris Wyman, Matt Pharr, Peter Shirley, Aaron Lefohn, and Wojciech Jarosz. Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting. *ACM Transactions on Graphics*, 39, 07 2020. doi:10.1145/3386569.3392481.

[CPC84] Robert Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. volume 18, pages 137–145, 07 1984. doi:10.1145/800031.808590.

[DHD20] Addis Dittebrandt, Johannes Hanika, and Carsten Dachsbacher. Temporal Sample Reuse for Next Event Estimation and Path Guiding for Real-Time Path Tracing. In Carsten Dachsbacher and Matt Pharr, editors, *Eurographics Symposium on Rendering - DL-only Track*. The Eurographics Association, 2020. doi:10.2312/sr.20201135.

[DLR77] Arthur Dempster, Natalie Laird, and Donald Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, 39:1–38, 01 1977.

[FLE87] N. I. Fisher, T. Lewis, and B. J. J. Embleton. *Statistical Analysis of Spherical Data*. Cambridge University Press, 1987. doi:10.1017/CBO9780511623059.

[FW80] J. D. Foley and Turner Whitted. An improved illumination model for shaded display, 1980.

[Gau20] Pascal Gautron. Real-time ray-traced ambient occlusion of complex scenes using spatial hashing. pages 1–2, 08 2020. doi:10.1145/3388767.3407375.

[GBBE18] Jerry Jinfeng Guo, Pablo Bauszat, Jacco Bikker, and Elmar Eisemann. Primary sample space path guiding. In *Proceedings of the Eurographics Symposium on Rendering: Experimental Ideas and Implementations*, SR '18, page 73–82, Goslar, DEU, 2018. Eurographics Association. doi:10.2312/sre.20181174.

[Gro] Khronos Group. Vulkan. URL: https://www.khronos.org/vulkan/.

[Hau18] Soren Hauberg. Directional statistics with the spherical normal distribution. pages 704–711, 07 2018. doi:10.23919/ICIF.2018.8455242.

[HJ09] Toshiya Hachisuka and Henrik Jensen. Stochastic progressive photon mapping. *ACM Trans. Graph.*, 28, 12 2009. doi:10.1145/1618452.1618487.

[Jak10] Wenzel Jakob. Mitsuba renderer, 2010. URL: http://www.mitsuba-renderer.org.

[Jak12] Wenzel Jakob. Numerically stable sampling of the von mises-fisher distribution on s 2 (and other tricks). Interactive Geometry Lab, ETH Zürich, Tech. Rep, 2012.

[Kaj86] James T. Kajiya. The rendering equation. In *Computer Graphics*, pages 143–150, 1986.

[LKA13] Samuli Laine, Tero Karras, and Timo Aila. Megakernels considered harmful: Wavefront path tracing on gpus. Technical report, NVIDIA, 2013.

[MK07] G. Mclachlan and Thriyambakam Krishnan. *Extensions of the EM Algorithm*, pages 159–218. 04 2007. doi:10.1002/9780470191613.ch5.

[MMR+19] Thomas Müller, Brian Mcwilliams, Fabrice Rousselle, Markus Gross, and Jan Novák. Neural importance sampling. *ACM Transactions on Graphics*, 38:1–19, 10 2019. doi:10.1145/3341156.

[NB99] M. Newman and Gerard Barkema. Monte carlo methods in statistical physics. 01 1999.

[NDVZJ19] Merlin Nimier-David, Delio Vicini, Tizian Zeltner, and Wenzel Jakob. Mitsuba 2: A retargetable forward and inverse renderer. *Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 38(6), December 2019. doi:10.1145/3355089.3356498.

[NVI] NVIDIA. 2020 editor's tech day - bvh. URL: https://images.anandtech.com/doci/13282/NV_Turing_Editors_Day_029.png.

[NVI18] NVIDIA. Nvidia turing gpu architecture, sep 2018. URL: https://www.nvidia.com/en-us/geforce/news/geforce-rtx-20-series-turing-architecture-whitepaper/.

[NVI19] NVIDIA. Quake ii rtx, mar 2019. URL: https://www.nvidia.com/en-us/geforce/news/quake-ii-rtx-ray-tracing-vulkan-vkray-geforce-rtx/.

[NVI20a] NVIDIA. Any hit shaders - tutorial - rng implementation, 2020. URL: https://github.com/nvpro-samples/vk_raytracing_tutorial_KHR/blob/master/ray_tracing_anyhit/shaders/random.glsl.

[NVI20b] Microsoft NVIDIA. Minecraft with rtx: Crafting a real-time path-tracer for gaming, mar 2020. URL: https://youtu.be/mDlmQYHApBU.

[NVI20c] Microsoft NVIDIA, Mojang, 2020. URL: https://www.nvidia.com/de-de/geforce/campaigns/minecraft-with-rtx/.

[PBD+10] Steven Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan Mcguire, R. Morley, Austin Robison, and Martin Stich. Optix: A general purpose ray tracing engine. *ACM Trans. Graph.*, 29, 07 2010. doi:10.1145/1778765.1778803.

[PJHa] Matt Pharr, Wenzel Jakob, and Greg Humphreys. Physically based rendering: From theory to implementation - analytic solutions to the lte. URL: http://www.pbr-book.org/3ed-2018/Light_Transport_I_Surface_Reflection/The_Light_Transport_Equation.html#AnalyticSolutionstotheLTE.

[PJHb] Matt Pharr, Wenzel Jakob, and Greg Humphreys. Physically based rendering: From theory to implementation - bounding volume hierarchies. URL: http://www.pbr-book.org/3ed-2018/Primitives_and_Intersection_Acceleration/Bounding_Volume_Hierarchies.html.

[PJHc] Matt Pharr, Wenzel Jakob, and Greg Humphreys. Physically based rendering: From theory to implementation - kd-tree accelerator. URL: http://www.pbr-book.org/3ed-2018/Primitives_and_Intersection_Acceleration/Kd-Tree_Accelerator.html.

[RHJD18] Florian Reibold, Johannes Hanika, Alisa Jung, and Carsten Dachsbacher. Selective guided sampling with complete light transport paths. volume 37, pages 1–14, 12 2018. doi:10.1145/3272127.3275030.

[RHL20] Lukas Ruppert, Sebastian Herholz, and Hendrik P. A. Lensch. Robust fitting of parallax-aware mixtures for path guiding. *ACM Trans. Graph.*, 39(4), July 2020. doi:10.1145/3386569.3392421.

[Sch19] Christoph Schied. Q2vkpt. http://brechpunkt.de/q2vkpt/, 2019.

[SSK+17] Christoph Schied, Marco Salvi, Anton Kaplanyan, Chris Wyman, Anjul Patney, Chakravarty Chaitanya, John Burgess, Shiqiu Liu, Carsten Dachsbacher, and Aaron Lefohn. Spatiotemporal variance-guided filtering: real-time reconstruction for path-traced global illumination. pages 1–12, 07 2017. doi:10.1145/3105762.3105770.

[SWS02] Jörg Schmittler, Ingo Wald, and Philipp Slusallek. Saarcor: a hardware architecture for ray tracing. pages 27–36, 01 2002.

[Vea97] Eric Veach. Robust monte carlo methods for light transport simulation. 01 1997.

[VG95] Eric Veach and Leonidas Guibas. Optimally combining sampling techniques for monte carlo rendering. volume 95, pages 419–428, 01 1995. doi:10.1145/218380.218498.

[VK16] Jiri Vorba and Jaroslav Krivanek. Adjoint-driven russian roulette and splitting in light transport simulation. *ACM Transactions on Graphics*, 35:1–11, 07 2016. doi:10.1145/2897824.2925912.

[Wea18] Eric Werness and Ashwin Lele et al. Vk_nv_ray_tracing, 2018. URL: https://www.khronos.org/registry/vulkan/specs/1.2-extensions/man/html/VK_NV_ray_tracing.html.

[Wea19] Eric Werness and Ashwin Lele et al. Vk_khr_ray_tracing, 2019. URL: https://www.khronos.org/registry/vulkan/specs/1.2-extensions/man/html/VK_KHR_ray_tracing.html.

[WGGH20] Rex West, Iliyan Georgiev, Adrien Gruson, and Toshiya Hachisuka. Continuous multiple importance sampling. *ACM Transactions on Graphics*, 39, 07 2020. doi:10.1145/3386569.3392436.

[WH97] Gregory Ward and Paul Heckbert. Irradiance gradients. *Proc. 3rd Eurographics Workshop on Rendering*, 8598, 11 1997. doi:10.1145/1401132.1401225.

[WH98] John Wagner and Alireza Haghighat. Automated variance reduction of monte carlo shielding calculations using the discrete ordinates adjoint function. *Nuclear Science and Engineering - NUCL SCI ENG*, 128, 11 1998. doi:10.13182/NSE98-2.

[WRC88] Gregory J. Ward, Francis M. Rubinstein, and Robert D. Clear. A ray tracing solution for diffuse interreflection. *SIGGRAPH Comput. Graph.*, 22(4):85–92, June 1988. doi:10.1145/378456.378490.

[ZOC10] Fahad Zafar, Marc Olano, and Aaron Curtis. Gpu random numbers via the tiny encryption algorithm. 01 2010.

[ZZ19] Quan Zheng and Matthias Zwicker. Learning to importance sample in primary sample space. *Computer Graphics Forum*, 38:169–179, 05 2019. doi:10.1111/cgf.13628.