Hasselt University

A thesis submitted in partial fulfillment of the requirements for the Master's degree in computer science

# A Scalable and Coherent Approach to Monte Carlo Path Tracing for Multi-User Environments

*Author*:

Jente Vandersanden

*Promotor*:

dr. Jeroen Put

Academic Year 2022 - 2023

# Acknowledgements

First of all, I would like to express my special gratitude to my promotor, dr. Jeroen Put, for the countless hours of engaging discussions we have had that formed the foundation for this work, and his great sense of humor that added a touch of levity to these sessions. His enthusiasm, invaluable guidance and support have played a crucial role in shaping the outcome of this thesis.

Secondly, I would like to thank all other members of the Visual Computing department: Prof. Michiels, Prof. Jorissen, Bram, Steven, Chen and Kristof for allowing me to write my thesis under their department. The opportunity to work in their lab throughout the year and the invaluable feedback received during our monthly meetings have been very inspiring and motivational.

I would also like to thank my fellow student and friend, Brent, in particular for his company throughout the past few years, our numerous idea exchanges and brainstorming sessions. His encouragement during the time we were working towards shared goals have been a great driving force during challenging moments.

In addition I am also grateful towards all other professors, teachers and fellow students who have crossed paths with me during my educational journey. Thank you for sharing your knowledge and wisdom, inspiring me and providing a challenging but comfortable learning environment. It is through these encounters that I have grown both academically and personally.

Another thank you goes out to my dear friends whose companionship helped me maintaining a balance between work and other aspects of life. The refreshing walks, gym sessions, and delightful banter we shared have not only provided much-needed respite, but also served as fertile ground for new ideas to flourish.

I cannot forget to dedicate a special thank you to my family. Thank you for all your support, for believing in me reaching my goals, and for showing interest in the topics I was working on. I am especially grateful for all the opportunities you have given me in life.

And last but not least, my wonderful girlfriend. Thank you for your unwavering patience, for encouraging me in difficult moments and most of all, for always believing in me. Without you, I would not be standing where I am today.

– *Jente*

# Summary


## Introduction

Over the years, we have seen the majority of entertainment applications make the transition to the cloud. One interesting question to ask is why this has not happened yet for graphics applications, such as video games. In *cloud gaming*, a centralized cloud provider provides the infrastructure for gaming. Cloud gaming removes all the workload of rendering and managing a video game away from the client. Instead, it is assigned to a server, which then streams the results to the client. This allows clients to play games on-demand on practically any device that has a display, input and internet connection. Although solutions for cloud gaming exist, they have not been widely adopted yet. The most plausible explanation for this is that the current solutions do not scale well to many-user scenarios, especially when we consider multiplayer applications.

With the advent of hardware-accelerated ray tracing, it has become possible to simulate light transport in real-time on consumer hardware. This allows for very realistic lighting effects in video games. These light transport simulations, nevertheless, need a lot of processing power. They compute, with physical accuracy, how an image would look taken from a certain viewpoint. Because existing light transport algorithms render individual viewpoints, they are not scalable when it comes to multi-user applications. This is because the underlying algorithms essentially require a separate processing unit per user due to their computational intensity. Therefore, it currently makes more sense for clients to purchase their own hardware rather than having distinct hardware provided by a cloud provider for each individual user.

Another problem natural to light transport simulation is that of ray incoherence. When light rays are traced throughout a scene, they interact with different materials causing them to change trajectory. This implies that a group of rays possibly has widely varying directions and/or origins. We call these rays incoherent. To make real-time light transport simulation possible, many rays are processed in parallel on a massively parallel processor. If incoherent rays are scheduled to be processed in parallel, they possibly access many different memory locations. This wastes the potential of the GPU's fast cache memory, because each time a cache miss is encountered, data needs to be retrieved from main memory which is several orders of magnitude slower. It is therefore that memory incoherence increasingly forms a large bottleneck in contemporary path-tracing algorithms [4].

## Objectives

This thesis focuses on two objectives. A first objective is to define an approach to Monte Carlo path tracing that benefits from a higher ray coherence than traditional Monte Carlo path tracers. The ideas behind our algorithm will be evaluated and demonstrated via a proof-of-concept implementation that should be representative for medium to large scenes in video games. A second topic of focus will be to confirm the hypothesis that an algorithm that calculates the full scene's irradiance scales better for many-user scenarios. We attempt to make this tangible by making an explicit comparison between the workloads performed by our

algorithm and a reference GPU Monte Carlo path tracer. It should furthermore be mentioned that we will focus on the computation of diffuse global illumination effects, for two reasons. First of all, diffuse lighting effects are the most relevant candidate for caching, due to their viewpoint-independent appearance. Secondly, they are the worst case when it comes to ray incoherence, because diffuse surfaces scatter rays uniformly in their normal-oriented hemisphere. Therefore, we expect to make the most impact there.

# Implementation

We propose an algorithm that computes and stores irradiance for the whole scene. The computed data can then be used by any amount of users to compose their viewpoint. This works for diffuse global illumination because the visual appearance of a point on a diffuse surface can be described by the product of the Lambertian BRDF and the irradiance at that point. We first briefly describe the components of our system. We make a distinction between direct and indirect lighting. We handle direct lighting in a separate render pass. For indirect lighting, the scene is subdivided in so-called *radiance cells*. These radiance cells operate independently on their local geometry. In other words, indirect lighting is computed cell by cell, each cell handling its local surfaces. We adopt a hybrid radiance propagation scheme for indirect lighting, meaning that nearby indirect ligthing is ray traced and distant indirect lighting is approximated. This approximation is done by making use of a nearby radiance probe. In our implementation, we place a radiance probe in the center of each radiance cell. Whether ray tracing or an approximation is used, is decided by a threshold parameter called the *tracing range*. We work per lighting bounce. An overview of our algorithm is presented in Figure 0.0.1.

## Unbiased approach

The algorithm presented in Figure 0.0.1 is biased. Besides this, we have also provided an unbiased approach in which we set the tracing range threshold parameter to infinity. This essentially means that we no longer make use of probe approximations for distant lighting, but instead ray trace everything. Because we no longer make use of the radiance probes, we can also skip the second render pass. The unbiased approach therefore only executes two render passes: the direct ligthing pass and radiance cell scattering pass.

## Data structure to store irradiance

Several data structures in which to store the computed irradiance and later sample from were considered. In particular, *GPU octree textures* and *per-object lightmaps* were implemented and experimented with. Because of their robustness and the lack of a necessity to transform 3D surface points into 2D space, GPU octree textures were appealing in the first place. However, we later decided to step away from them, mainly due to the lack of support for hardware texture filtering. This would make texture look-ups significantly slower, while our aim was to introduce minimal overhead. In contrast, per-object lightmaps do support hardware acceleration, although they suffer from other limitations, such as a high memory usage and *texture seam* artifacts. It is important to note that although the underlying data structure is interchangeable, there are certain minor nuances that must be taken into account in the rendering kernels' implementation because of the various representations.

## Limitations and optimizations

Some limitations that the finished implementation suffers from are artifacts, high memory usage and no support for specular effects. The first two of these are mainly caused by the underlying data structure for irradiance that was chosen. Texture seam artifacts are inevitable in most practical scenarios, because a mapping from 3D to 2D space needs to be made. The algorithm that performs this mapping is required to make 'cuts' in some places to unwrap the 3D surface. These cuts manifest themselves as seams when texture filtering is applied.

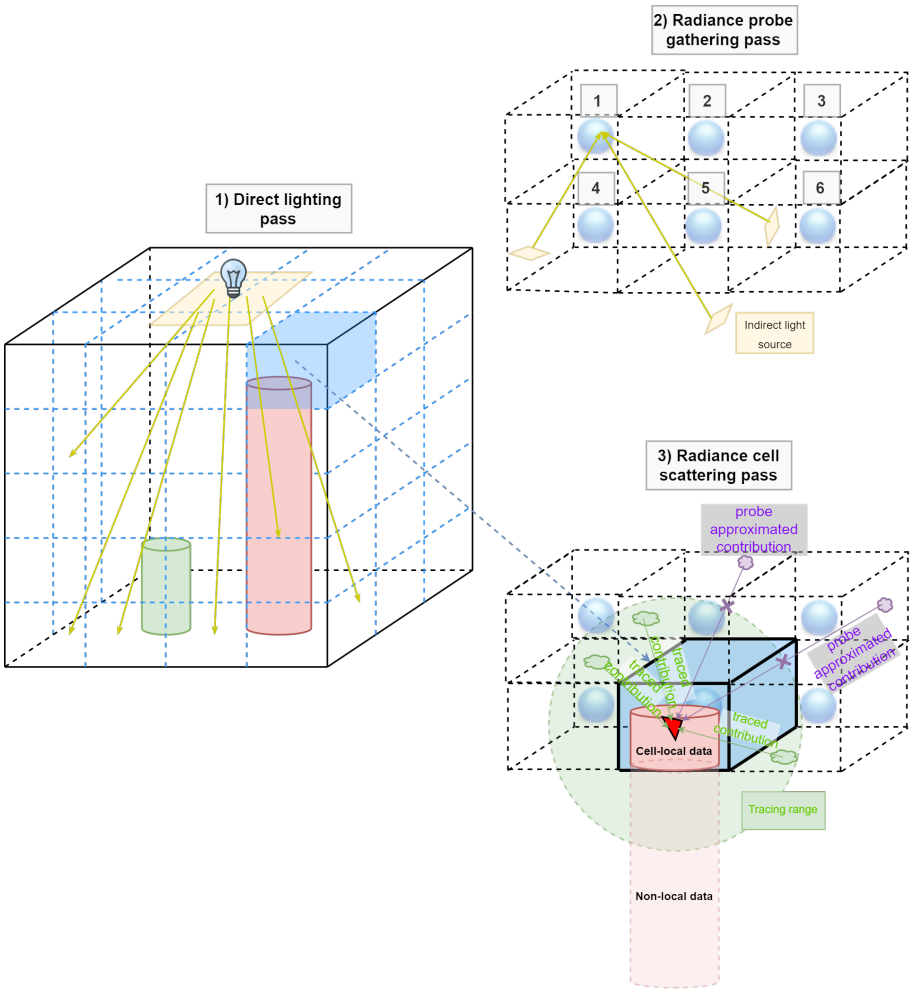



**Figure 0.0.1:** High-level overview of the thesis' proposed rendering algorithm and used data structures. The scene is subdivided into a regular grid of radiance cells. The algorithm starts with the direct lighting pass, in which direct radiance starting from emitters is propagated throughout the scene. Secondly, during the radiance probe gathering pass, indirect radiance from the previous lighting bounce is gathered into radiance probes. After this pass each radiance probe contains a 'omnidirectional view' of the incoming indirect radiance at their position. Finally, in the radiance cell scattering pass, indirect radiance is transferred onto the local geometry of each radiance cell. In other words, the scene's surfaces are illuminated with indirect radiance for the current lighting bounce. We make a distinction between nearby and distant indirect lighting based on a tracing trange threshold value. We ray trace nearby incoming indirect radiance, and approximate distant indirect radiance by making use of the radiance probes.

Although these limitations are currently present in our implementation, we believe they can be overcome, motivated by the fact that solutions already exist for these limitations. Unfortunately, these solutions were out of scope for this thesis and therefore we did not include them.

In addition we proposed several optimizations that can be considered as future work. One concern is that the algorithm currently computes irradiance for the entire scene. It should be intuitive that this is unnecessary for the goal that we want to achieve. Indeed, the goal is to centralize all computations required for all users in the scene, and reuse those computations between users. Therefore, one of the proposed optimizations explains that it would be sufficient to compute irradiance for the union of visible points to all users. Another optimization proposes a form of *asynchronous rendering*, i.e. updating irradiance first where it matters most and later in areas that are less visually impacted by the dynamic changes in the scene.

## Evaluation and analysis of the implementation

First of all, we have experimented with the configurable parameters of our implementation. More specifically, our hypothesis was that by decreasing the tracing range threshold, we would increase coherence. The reasoning behind this is that by limiting the space that a ray can traverse, we also limit the memory space that the ray can access. To verify this, we have run our algorithm for different tracing range values, keeping all other parameters constant. We measured both algorithm run time and cache hit ratios for the L1 and L2 caches. Our experiments indeed confirmed that decreasing the tracing range has a positive impact on both overall algorithm performance and L2 cache hit ratio (coherence). One thing to mention however is that setting the tracing range too low can introduce artifacts such as light leaks, because the assumption of distant lighting is violated.

Similarly, the impact of the radiance cell size on the algorithm's performance was measured. The radiance cell size's impact was much less significant. When the size is set too small, the algorithm's performance starts to suffer from the additional overhead that is introduced. This overhead is caused because a separate kernel needs to be launched per radiance cell. If the radiance cells become too small and numerous, the workload of each radiance cell also becomes too little to fully occupy the GPU.

The second experiment that was conducted compares the total work done by our algorithm to the workload of a traditional Monte Carlo path tracer on GPU. It might be interesting for cloud providers to have an idea starting from which amount of users it would be beneficial to use an algorithm like ours over a traditional approach, where each client's viewpoint is path traced independently. To make this possible, we implemented a reference path tracer that computes diffuse global illumination for 3 lighting bounces (like our algorithm does). We made our comparison in terms of algorithm run time. Our algorithm only needs to be run once per frame to serve all users, if we disregard the time it takes to compose a viewpoint from our cached irradiance data. This in contrast to the reference path tracer, that needs to be rerun entirely for each viewpoint that is to be rendered. We uniformly generated a set of viewpoints, and summed up the reference path tracer's render times for each of these. We compared this accumulated render time with the run time of our algorithm and identified the cross-over point.

This experiment provided us with two insights: First of all, it shows the cross-over point where our algorithm starts performing less work than the total work performed by all independent reference path tracers combined. This cross-over point also gives us the factor with which our algorithm should be sped up to achieve the same frame rate as the reference path tracer. Note that this speedup-factor represents the worst case: for 1 sample per pixel, the reference tracer is achieving frame rates of over 250 FPS, which is excessive for the majority of entertainment applications. For most games it is already acceptable to have frame rates between 30 and 60, especially when the game is fully ray traced. Secondly, this experiment also showed us that our

algorithm's workload is dominated by the amount of surface points that it needs to compute irradiance for, in contrast to the reference path tracer where the workload is mainly determined by the amount of samples that are taken per pixel. Therefore, if we increase the scene's size, the run time of algorithm becomes longer, and we need to apply more resources to achieve an interactive frame rate. But this is what we desired in the first place: from a cloud provider's point of view, it is much more interesting to scale resources based on the scene that needs to be rendered, instead of the amount of users that want to render the scene.

## Conclusion and future work

The objective of this thesis consisted of two subobjectives. First of all, we wanted to modify the Monte Carlo path-tracing algorithm to make it more scalable for multi-user applications. The second subobjective seeked to improve the coherence of Monte Carlo path tracing on GPU. An algorithm that keeps in mind both coherence and scalability was proposed. The scalability of the algorithm was experimentally verified by making a comparison in terms of total executed workload between the proposed algorithm and a reference GPU path tracer. The key finding from this experiment is that, in contrast to the traditional approach, the proposed algorithm scales computing resources depending on the size of the scene to be rendered instead of the number of users that want to render the scene.

Secondly, experimental results supported our hypothesis that reducing the tracing range parameter would benefit memory coherence. Briefly said, for the biased approach, we do observe an improved memory coherence. In contrast, for the unbiased approach, the tracing range is set to infinity. This means that the ray's traversal is no longer limited, with the consequence that the previously improved memory coherence is lost. Nevertheless, our experiments showed that the unbiased approach is still faster than the biased one. The reason for this is that the biased approach has to execute an additional render pass to update the probes each time the scene's lighting distribution changes, which is simply too expensive. This observation gave us the idea that *visibility propagation* might be more interesting than radiance propagation when it comes to improving coherence. In visibility propagation, we build up a data structure that contains discrete visibility information. This data structure could then potentially be used in several applications, one of which is importance sampling for indirect lighting. An implementation for visibility propagation is regarded as future work.

# Contents

# Chapter 1

# Introduction

During the last few decades our consumption of entertainment media has drastically shifted from predefined, static content to real-time, dynamic content that we can interact with. This effect can be observed, for instance, in the enormous popularity growth of live streaming, video games, and other interactive media such as virtual or augmented reality applications. Dedicated rendering hardware such as graphical processing units (GPUs) have allowed for rendering dynamic virtual scenes at interactive frame rates, which makes these applications possible in the first place. However, obtaining a realistic and immersive experience is desirable, and shading of the scene geometry plays a significant part in this. This shading can be done arbitrarily, but unsurprisingly, that will result in a fake look. The reason behind this is that we are used to interact with objects in the real world, which are shaded according to their material properties and the physical transport of light. Lighting is a very important contributor to the realism of a scene. It can be used to give us a sense of time, depth, atmosphere and mood. After all, the image of the real world we constantly see through our eyes is also governed by the physics of light transport. Therefore, it helps determine how we experience the world, whether it is virtual or not. Since its inception in the early 1960s [29], the field of computer graphics has come a long way toward achieving photorealism, but this process truly took off after the development of the ray-tracing algorithm [38]. This algorithm attempts to simulate the behavior of all light rays in space, according to the laws of physics. Since there are an infinite amount of light rays in any scene that contains a form of light, it should not be surprising that this is a very complex and computationally intensive problem. Throughout the years, researchers have attempted to find more optimal solutions to this problem, to decrease render times. In real-time applications such as video games this often resulted in simplifications and approximations, because it was simply unfeasible to provide the physically-accurate solution while maintaining interactive frame rates. Real-time *global illumination* has been a prominent topic of research during the past few decades. In fact, it is a problem that is still not entirely solved for some scenarios as of today.

Due to recent developments in specialized consumer hardware (NVIDIA RTX [35]), it became possible to reach interactive and physically-accurate global illumination in video games through hardware-accelerated ray tracing. NVIDIA RTX cards possess *RT cores* that are specialized to accelerate mathematical operations that are frequently used in ray tracing, such as ray-triangle intersections and *bounding volume hierarchy traversal*. Despite these fundamental advancements, a great bottleneck that we often see recurring in contemporary ray-tracing algorithms is the incoherence between light rays. This is particularly the case for the rays that account for indirect lighting. As a consequence, memory accesses that are made to swap scene asset data in and out of the GPU's memory are not only redundant but also scattered throughout memory space. As a result, these algorithms are not able to profit from the use of exceptionally performant cache memory. This effect is amplified for massively parallel processors like GPUs. They operate optimally on large tasks that require many similar operations on similar data, because they are largely constrained by memory bandwidth. As is the case with CPUs, the process-

ing cores of a GPU are many times faster than the available memory throughput. If memory bandwidth becomes insufficient to provide the processors with enough data and instructions to continue computation, the processors will be stalled waiting for this data. This phenomenon is also called the *memory wall*. GPU implementations of various kinds of algorithms are often adapted versions of the traditional CPU implementation to ensure maximal data parallelism. Sometimes it is even necessary to completely rethink and redesign the algorithm to benefit from the parallelism that the GPU offers. For Monte Carlo path tracing this has not yet happened as of today. Although many advancements have led to the availability of real-time path tracing on consumer hardware, these advancements were primarily made on the hardware level. In short, while GPUs are often faster at path tracing than their CPU counterparts, that is largely thanks to their vastly superior memory bandwidth and not because they are inherently a more suitable architecture. The path-tracing algorithm used in current GPU implementations has not really been rethought to make it more optimal for a massively parallel processor. Worse still, the path-tracing algorithm is somewhat of a worst-case scenario for a GPU. Indirect lighting rays are scattered all over the scene, possibly intersecting with many different materials, which have different processing needs, leading to very divergent branches of computation. These characteristics of the algorithm critically deteriorate both *task* and *data parallelism*, for which the GPU is essentially optimized.

Video games are a primary application of real-time ray tracing, but with emerging new technology new applications also arise. Examples of these are virtual experiences such as virtual concerts, chat rooms and conferencing in the so-called *metaverse*. Another upcoming phenomenon is that of *digital twins*, or the virtual duplication of a physical environment or system. Physical environments are most often dynamic. Real-time ray tracing can improve the simulation capabilities of these digital twins in dynamic environments. For example, assume a digital twin of a giant greenhouse is made to simulate and monitor the incoming amount of light energy based on sensor data. In this way, artificial light can be deployed less wasteful and more targeted. Natural lighting is dynamic, so a real-time path-tracing algorithm is required to make this simulation feasible. Another observation that we have made is that historically, almost all computer programs have transitioned from native applications to cloud-based service models. It is therefore an interesting question to ask why this has not happened for computer graphics applications yet, such as the majority of video games. The reason could be that most rendering algorithms are designed to render viewpoints for a single user. For example, the classic path-tracing algorithm traces light rays from the camera backwards into the scene, until it hits a light source. The result of this is the rendered viewpoint of one camera corresponding to one user. Using this approach, the cloud provider essentially needs to run a separate physical system per user, due to the computational workload behind these algorithms. In scenarios with many users, this model is not scalable. Assume the worldwide gaming community makes the transition to cloud gaming. With the current approach, it makes more sense that each user owns their own hardware. Existing cloud gaming implementations are based on the aforementioned model, but they have not yet gained much traction over the years. Instead, it would be more interesting if providers of cloud-based graphics applications needed a constant amount of computation given a scene of a certain complexity. By adopting such a model, results of computation are shared between users and each part of information necessary to generate a 3D viewpoint is computed not more than once. Although the total computational workload to calculate all the values required to generate any arbitrary viewpoint is a superset of the workload to generate just one viewpoint, it potentially is a model that scales better with a large amount of users. The contrast between these two different ideas of how to organize path tracing in a multi-user graphics application is illustrated in Figure 1.0.1.

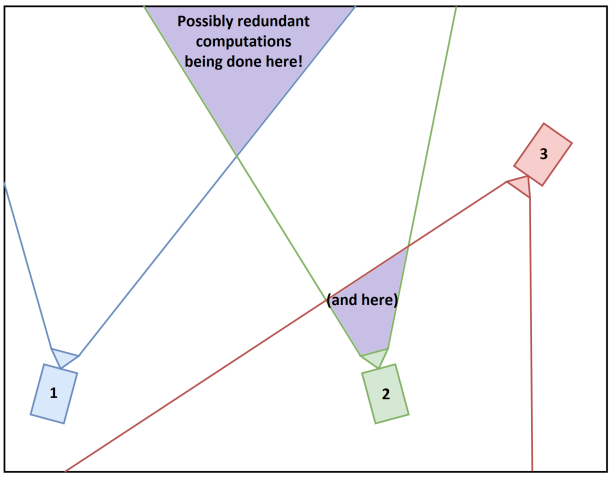
In this thesis we will experiment with an approach to *Monte Carlo path tracing* that makes extensive use of the GPU's cache memory by exploiting the coherence of rays in the order that they are processed and by limiting the space that they traverse. We develop a radiance communication scheme that relies on spatial properties of scene geometry to propagate indirect

radiance throughout the scene in a more coherent manner. We do this by spatially subdividing the scene into cells and calculating the indirect irradiance for the inner geometry of each cell independently. We follow an incremental approach that handles each indirect lighting bounce separately. This is done for two reasons. One reason is to eliminate additional data dependencies between concurrent threads. They work on the same lighting bounce, all requiring the data of the previous lighting bounce. Another reason is to hypothetically limit the amount of divergence within the batch of rays that is being processed. If we work over multiple lighting bounces simultaneously, there is a larger probability that the rays access incoherent locations due to the additional recursive scattering that is introduced. Direct lighting consists of primary rays that are rather coherent by nature, so the implementation handles this part of the rendering equation separately. For each indirect bounce, we march over the cell grid to compute the indirect irradiance for each radiance cell. Our proposed algorithm is originally biased. The hypothesis is that this biased implementation is fast, in exchange for physical inaccuracies. Therefore we also provide an unbiased approach, which we expect to be slower but estimates the rendering equation without introducing bias. The biased approach involves a hybrid rendering procedure that makes use of both path tracing and approximation by radiance probes. For nearby indirect lighting, rays are traced to determine the irradiance at a certain surface point. Distant indirect lighting is approximated by reading from the most nearby radiance probe. To determine whether to make use of distant or nearby lighting, a threshold value called the *tracing range* is configured. The unbiased approach essentially only makes use of path tracing, which is equivalent to setting the tracing range to infinity. A second focus of this thesis is to make the algorithm ideal for the specific use case of virtual environments with many users, such as multiplayer games. We aim to centralize the calculation of the scene's light distribution such that the information required to generate the union of the viewpoints of all users is calculated no more than once. The work performed to calculate this information should be minimized, that is, results will be reused between users and no redundant calculations will be made. We plan to do this by caching the scene's rendered irradiance in a data structure from which irradiance can later be sampled to generate arbitrary viewpoints for all users in the scene. To effectively handle dynamic scenes, it will be necessary for us to demonstrate the scalability of our algorithm. Specifically, our algorithm should maximize parallelism to efficiently utilize server resources, thereby achieving real-time frame rates in proportion to the allocated resources.

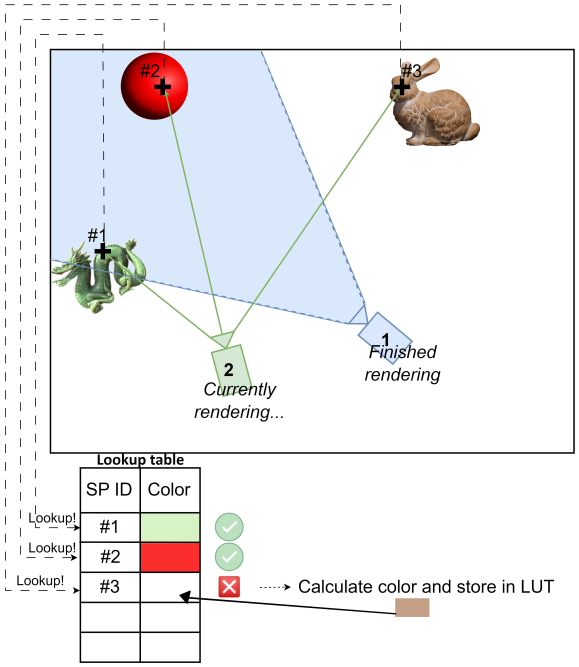
## 1.1 Problem areas and challenges

There are several areas in which challenges will have to be faced. A challenge central to this thesis is that we are aiming to improve the memory coherence of Monte Carlo path tracing on the GPU, to ultimately benefit the performance of the algorithm. This means that we will have to develop a modified version of the algorithm without introducing any overhead that would outweigh the benefits gained by a higher ray coherence. A second closely-related challenge is that the resulting implementation should minimize its memory usage. The more compact the used data objects are, the more of them fit in cache memory and therefore the probability of a cache hit is increased. Simultaneously, the size of the data objects possibly impacts speed of the implementation. For example, one could choose to compute a set of values once and store them in memory, essentially using extra memory but saving on computation time. Another option could be to recompute the values each time they are needed. This is known as the trade-off between memory complexity and time complexity that often recurs in programming in general. To maximize the positive impact of our implementation we need to very carefully think through this trade-off process.

Another challenge that will be encountered is the need to tweak the implementation's parameters. These parameters encompass various aspects, ranging from the scene's subdivision granularity and resolution of the textures used to store irradiance, to the tracing range of rays and beyond. Not only do these parameters shape the appearance of the rendered image, they govern the internal rendering process and therefore impact performance. The correct choice of




**(a)** Multi-user scene with redundant viewpoint calculations




**(b)** Multi-user scene with lookup table (LUT) data structure

**Figure 1.0.1:** Illustration of two models that can be used to render viewpoints in a scene with multiple users. For this example we only consider diffuse global illumination. In Figure 1.0.1a each user renders their own viewpoint. This is the traditional model used in current cloud gaming solutions, but as can be seen, there is overlap between viewpoints. For viewpoint independent illumination effects, such as diffuse global illumination, this means the computations done for the overlapping areas (shaded in purple) are redundant and could theoretically be shared between users. Contrary to the previous model, in Figure 1.0.1b, all computations done are stored into an arbitrary *lookup table* (LUT) data structure. The consequence of this is that other users can later make use of it and do not have to recompute information unless necessary. The lookup table in this toy example stores an ID for each surface point (SP), together with the diffuse irradiance at that point (color that the camera perceives). Camera 1 has already finished rendering its viewpoint, while camera 2 is still rendering. We look at 3 rays in more detail. The rays that intersect with SP 1 and SP 2 do a lookup in the LUT and see that it already contains a diffuse color for that SP, so the user does not have to make any additional calculations for that ray. SP 3 falls outside of camera 1's viewpoint, so has no associated diffuse color stored in the LUT yet. User 2 (or a centralized server) therefore computes the diffuse irradiance for this SP and stores it in the LUT.

parameters is crucial, and the ideal set of parameters could differ between scenes.

Given the fact that we use an incremental approach to calculate resulting irradiance, meaning that every lighting bounce is handled separately, it is necessary to have a data structure that can be accessed efficiently to store the calculations done in each lighting bounce. Different representations are possible and will be discussed further in this thesis (section 3.11), but it is already worth mentioning that the problem of choosing the right representation is far from trivial. Each representation has its own advantages and disadvantages and some are harder to implement than others. The underlying representation is interchangeable, but brings some time-consuming subtleties that need to be addressed in the implementation of the ray tracing kernels. Therefore, it is crucial to weigh the pros and cons of each representation to determine whether their downsides are tolerable and to prevent them from causing additional bottlenecks.

Possibly the most challenging part of this thesis is the constraint of run time. To prove that our algorithm has potential, it needs to be able to achieve real-time frame rates. Together with all the challenges described above, this will require many non-trivial optimizations and trade-offs to be made. In case it is not possible to achieve interactive frame rates on a single GPU, it will remain valuable to demonstrate the feasibility of achieving such frame rates using multiple GPUs. This holds true as long as the number of GPUs needed is smaller than what is typically required with the traditional approach, particularly when considering a large user base.

To conclude this section, we define the objective of this thesis. **The objective of this thesis consists of 2 parts. One part is to define an approach to Monte Carlo path tracing that benefits from a higher ray coherence than traditional Monte Carlo path tracers. The ideas behind our algorithm will be evaluated and demonstrated via a proof-of-concept implementation that should be representative for medium to large scenes in video games. For the second part, we aim to confirm the hypothesis that an algorithm that calculates the full scene's irradiance scales better for many-user scenarios. We attempt to make this tangible by making an explicit comparison between the workloads performed by our algorithm and a reference GPU Monte Carlo path tracer.**

## 1.2 Global illumination

The reason why path-tracing algorithms are employed instead of traditional rendering techniques is to achieve physically-accurate global illumination effects. In addition to the camera's position and the illuminated object's position in relation to the light source, global illumination models additionally take into account the positions and material properties of other objects. In other words, light energy interactions between all surfaces in the scene are being accounted for, in contrast to local illumination models, where the rendering of an object is only affected by the camera's position and direct sources of light. Global illumination algorithms are able to reproduce realistic lighting effects as they occur in the real world, such as shadows, *diffuse* interreflections and *specular* or glossy reflections and refractions. Several algorithms to simulate global illumination exist, some of which will be explored in Chapter 2.

When speaking of global illumination, we mainly consider two types of effects: diffuse and specular. Diffuse effects appear when the scattering of light by an object happens in a uniform manner and is proportional to the cosine of the angle between the surface's normal and the outgoing radiance direction. In other words, light is distributed uniformly across the surface regardless of viewing angle. Examples of diffuse materials are chalk, matte paint and plaster. In contrast, specular effects appear when the reflection of light is concentrated into a narrower cone of directions. This is also known as a *glossy* reflection. If the light is only reflected into one direction, we call the material *perfectly specular*. An example of a perfectly specular object is a mirror (although a perfect mirror does not exist in the real world). An example of a

**(a)** Diffuse material

**(b)** Perfectly specular material (reflection + refraction)

**Figure 1.2.1:** Example renders of the dragon model using a standard diffuse material (a) and a perfectly specular material that is reflective and refractive (b). Due to the light being uniformly reflected on the surface's hemisphere, we observe that the diffuse material has a matte appearance. In contrast, specular highlights are visible on the perfectly specular material. Renders from [31].

glossy material is plastic. Figure 1.2.1 shows a comparison between an object rendered with a diffuse material and a perfectly specular material. This thesis will mainly focus on the diffuse component of global illumination given the fact that it is more easily reused accross varying viewpoints. Additionally, the scattering behavior of diffuse surfaces is a worst-case scenario when it comes to ray coherence. Extensions to the ideas presented in this thesis that also support specular effects can be considered as future work.

## 1.3 The rendering equation

Light transport throughout a scene can be described by an integral equation. This insight was discovered in the field of optics [39], long before the advent of computer graphics. In 1986 this concept was introduced to the field of computer graphics by J. Kajiya with the rendering equation [18]. The rendering equation is an integral equation that describes the outgoing light in a particular direction at a certain surface location in the scene, under steady-state conditions. In other words, it describes the light transport between surfaces in the scene, assuming there are no surrounding media (vacuum). The rendering equation goes as follows:

$$L_o(\mathbf{x}, \omega) = L_e(\mathbf{x}, \omega) + L_r(\mathbf{x}, \omega) \tag{1.1}$$

where $\mathbf{x}$ represents the surface location and $\omega$ the outgoing light direction. $L_e$ stands for the emitted radiance at $\mathbf{x}$ in direction $\omega$. $L_r$ is the reflected radiance at point $\mathbf{x}$ in direction $\omega$ and is described by the following equation:

$$L_r(\mathbf{x}, \omega) = \int_{\Omega^+} L_i(\mathbf{x}, \omega_i) f_r(\mathbf{x}, \omega_i \rightarrow \omega) \langle N(\mathbf{x}) \ , \ \omega_i \rangle^+ \, d\omega_i \tag{1.2}$$

Here $L_i$ is the incident radiance along direction $\omega_i$, which is computed by ray casting to determine which other surface location emits and/or reflects this radiance. This is where the recursive nature of global light transport becomes apparent, through indirect lighting. Furthermore we have $f_r$, which represents the bi-directional reflectance function (BRDF). The BRDF represents the characteristics of an opaque surface material and determines, given an incident radiance direction $\omega_i$ and outgoing radiance direction $\omega_o$, the ratio of outgoing radiance exiting along $\omega_o$. Finally, the dot product $\langle N(\mathbf{x}) \ , \ \omega_i \rangle^+$ represents *Lambert's law*, which says that the radiant intensity observed from a diffuse surface is directly proportional to the cosine of the

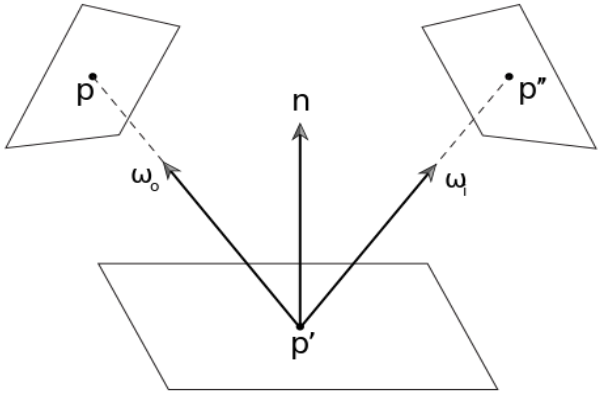



**Figure 1.3.1:** The surface area form of the rendering equation integrates over the domain of points on surfaces in the scene, rather than over directions in the (hemi)sphere. In this example, radiance comes in from **p"** and is reflected onto **p** due to interaction with the surface point **p'**. Figure from [31].

angle $\theta$ between the observer's line of sight and the surface normal.

There exists another, equivalent form of the rendering equation that integrates over surface area instead of incoming directions. This form can be useful in some scenarios, when it is more straightforward to sample surfaces instead of directions. A concrete example of this is the computation of direct lighting caused by area lights. Given some basic information about the area light source, such as its position, orientation and dimensions, it is generally easier to generate random points on this light than to generate random directions that go towards this light source. The surface form of the rendering equation goes as follows:

$$L_r(\mathbf{p'}, \mathbf{p}) = \int_A L_i(\mathbf{p'} \to \mathbf{p}) f_r(\mathbf{p"} \to \mathbf{p'} \to \mathbf{p}) G(\mathbf{p"} \leftrightarrow \mathbf{p'})\, dA(\mathbf{p"}) \tag{1.3}$$

In this equation, **p** represents the point for which we would like to compute incoming radiance, **p'** is a surface point that reflects radiance, and **p"** is a point on the surface area over which we integrate. Intuitively, this can be read as: *radiance coming from **p"** is reflected onto **p** via **p'***. The surface area version of the rendering equation is illustrated in Figure 1.3.1. The additional term $G(\mathbf{p"} \leftrightarrow \mathbf{p'})$ is called the *geometric coupling term*. This term can be fully written as:

$$G(\mathbf{p"}, \mathbf{p'}) = V(\mathbf{p"} \leftrightarrow \mathbf{p'}) \frac{|\cos\theta||\cos\theta'|}{||\mathbf{p"} - \mathbf{p'}||^2} \tag{1.4}$$

$V(\mathbf{p"} \leftrightarrow \mathbf{p'})$ is a fundamental term in rendering in general, and is called the *visibility term*. The visibility term determines, as its name implies, whether two given points are visible to each other. It is a binary function, taking the value of 1 when the two points are mutually visible, and 0 in the case of an occlusion. The $\cos\theta$ comes from Lambert's law, as was mentioned above. The other factor, $\frac{|\cos\theta'|}{||\mathbf{p"}-\mathbf{p'}||^2}$ comes from the transformation that relates solid angle to surface area. The transformation will not be derived here, but intuitively this factor serves as a compensation for the orientation and distance of the surface over which we integrate, relative to the point for which we compute incoming radiance.

## 1.4 Monte Carlo Integration

The rendering equation mainly consists of a very complex integral, that is essentially summing together the contributions of light rays. Due to integrand's high dimensionality and the recursive term within, it is impossible to find an analytical solution for most practical scenarios. Therefore, it is necessary to use a numerical method instead. Classic numerical integration techniques such as the trapezoid rule or Gaussian quadrature have the upper hand when it comes to low-dimensional, smooth integrals. Unfortunately, their convergence rate for high-dimensional and discontinuous integrals makes them unsuitable for physically-based rendering. The scattering of light in fact is a function of infinite dimensions, due to the recursive term. Furthermore, because of visibility discontinuities (i.e. occlusion boundaries) the function is the opposite of smooth. [31]

Monte Carlo integration uses a stochastic approach to approximate an integral, that provably converges to the correct solution with a convergence rate independent of the dimensionality of the integrand. It is a very powerful technique in the sense that to estimate the value of the integral $\int f(x)\,dx$, all that is needed are samples at arbitrary points in the domain of the integrand $f(x)$. This makes the method suitable to apply to discontinuous functions. The Monte Carlo estimator is described by the following expression:

$$F_N = \frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)} \tag{1.5}$$

Intuitively, we take $N$ random samples $X_i$ from an arbitrary distribution $p(x)$, evaluate the integrand $f(x)$ in these samples and inverse proportionally weigh them by the the probability that this sample was taken. It can be proven that this estimator returns the correct value of the integral on average:

$$\begin{aligned} E[F_N] &= E\left[\frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)}\right] \\ &= \frac{1}{N}\sum_{i=1}^{N}\int_a^b \frac{f(x)}{p(x)}p(x)\,dx \\ &= \frac{1}{N}\sum_{i=1}^{N}\int_a^b f(x)\,dx \\ &= \int_a^b f(x)\,dx \end{aligned} \tag{1.6}$$

To apply this to light transport simulation, we can estimate the rendering equation by taking samples in the domain of the integrand. For example, in the version of the rendering equation described in 1.3, for each indirect lighting bounce we integrate over directions in the upper hemisphere at a surface point to determine the reflected radiance in a certain direction $\omega$. In this case, the domain of the integrand consists of all directions in the upper hemisphere. Therefore, to estimate the value of this integral we can sample an arbitrary amount of directions in the upper hemisphere at that surface point and feed them to the Monte Carlo estimator $F_N$. By taking more and more samples, the result that the estimator returns converges to the real value of the integral, that is, the amount of reflected radiance in direction $\omega$.

The disadvantage of Monte Carlo integration is its very slow convergence rate of $O(n^{-\frac{1}{2}})$. This implies that to halve the variance, it is required to take four times as many samples. This introduces a large computational cost, because each sample corresponds to tracing a ray into the scene, testing for intersections, interacting with a material, etc. . Nevertheless, because its convergence rate is independent of the integrand's dimensionality, Monte Carlo is the only

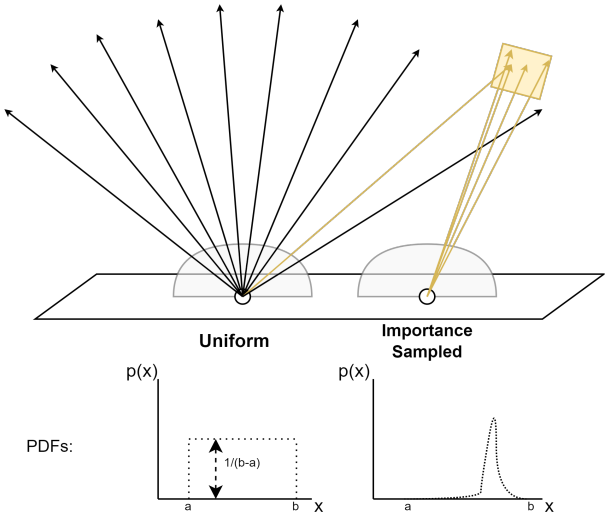



**Figure 1.4.1:** Illustration of importance sampling. On the left we see a point that samples the upper hemisphere of directions according a uniform distribution. Many samples miss the light source. Averaging these out and weighing them equally will lead to a noisy result. On the left a point is importance sampled according to a PDF that resembles the reality's light distribution much better. Samples that contribute a lot of useful information have a high probability to be chosen, while samples that do not contribute in the end are chosen with a low probability.

practical numerical integration method for high-dimensional integrals. The convergence rate of Monte Carlo is fixed, there is nothing we can do to lower it. However, it is possible to lower the variance from the start by sampling in a more clever way. This is called *importance sampling*. The idea behind importance sampling is that we sample from a distribution that makes it more likely to take samples that will contribute a lot. In other words, the distribution $p(x)$ from which we sample should be similar to function $f(x)$ that is the integrand. For example, imagine a surface that is illuminated by one small area light source. Assume we want to calculate the reflected radiance for a point on this surface, and therefore we could integrate over the upper hemisphere of directions at that point. However, many of these directions will not hit the area light source, so if the distribution from which we sample would be uniform, a lot of work will be wasted. This is because the samples that do not hit the light source will not contribute to the final result. Although this sample does not contribute useful information to the end result, it will count as a sample taken and its weight $\frac{1}{p(x)}$ will be taken into account. In this simple scenario, the reality is that all the useful information comes from the area light source, all radiance reflected onto the camera plane has its origin there. However, the PDF $p(x)$ does not give an accurate representation of this reality, it weighs and treats all samples equally (because it is a uniform distribution). This mismatch essentially causes a dark sample to contribute equally to the information in the end result as a sample that hits the light source. It therefore introduces variance which is manifested as noise in the resulting image. By choosing a different $p(x)$ that better resembles reality, this variance is reduced. For this example it would mean that distribution $p(x)$ has high values for the directions that go towards the light source, and low values for directions that would not hit the light source. The example is also illustrated in Figure 1.4.1.

# Chapter 2

# Literature study

## 2.1 Existing algorithms in physically-based rendering

### 2.1.1 Whitted ray tracing

Recursive ray tracing was first introduced to the computer graphics community in 1979 by T. Whitted [38]. This first attempt reshaped the rendering process behind computer generated imagery from being a surface visibility test (rasterization) to being a light transport simulation. It uses the idea of ray tracing to account for global lighting effects. Whitted's algorithm only supports perfectly specular reflection and transmission. For example, it can accurately simulate the reflections of a perfect mirror, or the refraction of light through a glass window. At the time, the realism of lighting effects in synthetic images produced by this approach were unseen before. This raised enthusiasm within the community.

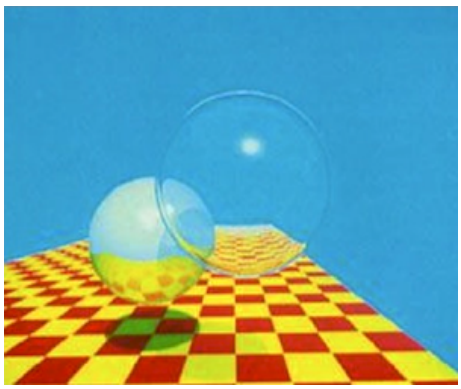


**Figure 2.1.1:** Original image produced by T. Whitted for his 1979 paper that introduced recursive ray tracing. Note the perfectly specular (mirror) reflections on the metallic sphere and the refraction of light on the glass sphere.

### 2.1.2 Monte Carlo path tracing

Monte Carlo path tracing [18] took physically-based light transport simulation to the next step. Not only did he introduce the rendering equation (discussed in section 1.3), Kajiya also demonstrated how to use Monte Carlo integration (section 1.4) to approximate this equation. The equation models the radiance transport between surfaces in the scene, therefore being able to account for all kinds of global illumination effects, such as diffuse interreflections and (glossy) specular reflections and refractions. For each pixel in the image, a ray is sent out from the camera into the scene. This ray will interact with the scene's geometrical surfaces and recursively change its trajectory based on the material properties of the surface. At each

intersection, the wavelengths (color) and amount (intensity) of radiance being transferred along the next path segment is also governed by these materials. This process is recursively repeated until a path segment strikes a light source, or the maximal recursion depth is reached.

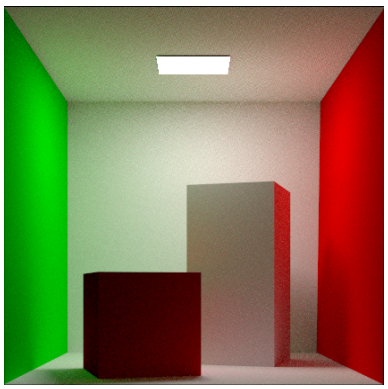


**Figure 2.1.2:** Cornell box scene rendered with pbrt-v3 [31], a Monte Carlo path tracer.

### 2.1.3 (Progressive) photon mapping

*Photon mapping* [17] is a biased, importance-driven approach to simulate global illumination. It consists of two passes, namely the *photon pass* and the *eye/camera pass*. The classic photon mapping algorithm starts with the photon pass. During this pass, photons are emitted from all light sources and scattered around the scene. When the photon hits a surface, it is either absorbed or reflected with a random probability. At each point where the photon is reflected, its properties (position, radiant energy, incoming direction) are stored in a data structure called the *photon map*. During the second pass, the photon map is used to render indirect lighting at the points visible to the camera. Primary rays are sent out into the scene. At each intersection with a surface, the photons in the neighborhood around the intersection point are combined to determine the indirect radiance. This is done by searching the photon map for photons that traveled a similar path to the path being traced. The photon map is typically stored in an acceleration structure such as a *kd-tree*, which makes the search faster. Direct lighting can be handled separately, via the usual approach where light sources are importance sampled. Photon mapping is illustrated in Figure 2.1.3. Classic photon mapping is limited by memory constraints, since the amount of photons that can be stored in memory puts a limit on the quality that can be achieved. If the memory is full, no more photons can be added to further increase the quality.

*Progressive photon mapping* [15] addressed this issue by making some adjustments to the classic photon mapping algorithm. Progressive photon mapping switches the order of the passes and instead starts with the camera pass. It sends primary rays into the scene, which recursively change trajectory upon intersection. The intersection points found along these rays can be called *light path vertices* or *visible points*. In the context of the proposed optimization mentioned in section 4.7.1, these visible points are most relevant to this thesis. The properties of the visible points are stored in each pixel of the camera's viewport that generated the path (since a primary ray was sent out for each pixel). Then during the second pass, photons are randomly emitted from the light sources. When a photon hits a surface nearby a visible point, it contributes to the reflected radiance estimate for that point. The final contribution of a visible point to the incoming radiance at the film plane is determined by taking the sum of the products of the contributing photons' weights and the path throughput weight $T$ (the throughput weight $T$ can be considered as the product between the BRDFs at each intersection point along the light path). Now the photons no longer need to be stored, which is a major improvement in terms of memory usage. Instead, the information about the visible points does need to be stored. In most cases, this does not form a great problem because the amount of visible points is relatively small. Progressive photon mapping is illustrated in Figure 2.1.4.

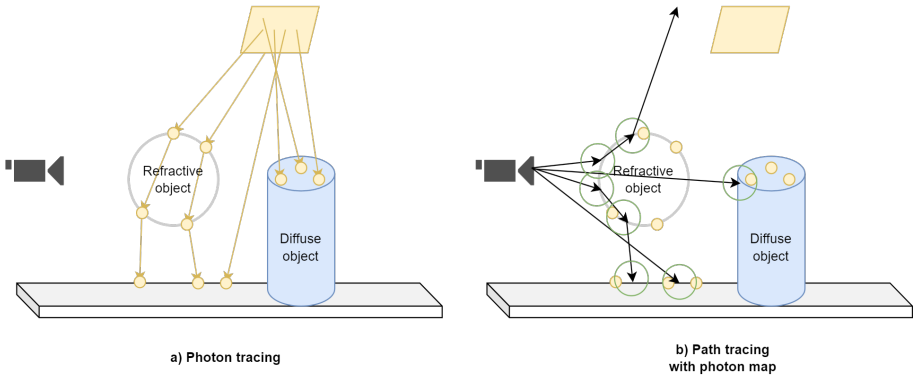



**Figure 2.1.3:** Classic photon mapping. On the left, the photon tracing pass in which photons are scattered into the scene. On the right, the actual render pass in which nearby photons at each ray intersection are used to compute indirect lighting. Photons are represented by yellow dots.

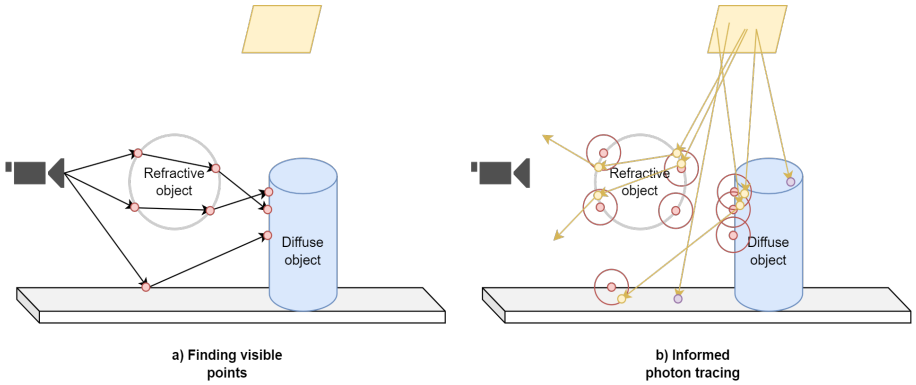



**Figure 2.1.4:** Progressive photon mapping. On the left, visible points are generated by tracing paths from the camera. Visible points are represented by red dots. On the right, photons are traced in random directions starting from the light source. Only photons that end up nearby visible points contribute. These are represented by yellow dots. The non-contributing photons are represented by purple dots. Only visible points need to be stored.

## 2.2 The 3D visibility complex

The visibility problem (i.e. the mutual visibility between 2 surface points) is a well-known and frequently encountered problem not only in physically-based rendering, but in computer graphics in general. *The 3D visibility complex* is a data structure developed by *Durand et al.* [10] which captures the global visibility information of the scene. This means that with this data structure, we are able to determine for any pair of points in the 3D scene whether they can "see each other". This is a powerful property that many solutions could profit from. In the domain of global illumination and ray tracing, we are constantly calculating whether light rays intersect with geometry, or whether one object occludes another. The 3D visibility complex can serve as some kind of oracle here that tells us in real time whether two points are connected by an uninterrupted ray. The free space between two points that are mutually visible is called a *maximal free segment*. A maximal free segment is illustrated in Figure 2.2.1. Visibility is a topic that returns several times throughout this thesis, in the context of difficulties with radiance probes (Figure 2.5.2), as well as in future work proposed in section 6.1. Since the majority of computations in a light transport algorithm consists of intersection tests, the 3D visibility complex is expected to lead to significant performance gains.

The details of the 3D visibility complex are rather abstract and require several geometrical insights to understand, but briefly summarized the visibility complex subdivides the scene into discrete volumes which consist of points that are "able to see each other". A disadvantage of this structure however is that it has a very high memory footprint ($O(n^4)$, with $n$ the number of objects), as described in the theoretical paper. A more practical exploration of the 3D visibility complex by *L. Glaves* [12] reports a contradiction however, with an expected growth rate of $O(n^2)$. A closely related data structure, called *the visibility skeleton* [9], even reports a linear growth rate for most scenes. Although this last discovery was promising, to this date, no known fully practical implementation of the 3D visibility complex exists to our knowledge. The reported space and time complexities are therefore only predictions. More research on this topic is necessary to determine whether it can serve as a feasible solution to accelerate light transport algorithms. We draw inspiration from the visibility complex in the sense that it exhibits high locality in memory space for neighboring bundles of rays. If the maximal free segments can be predicted for bundles of coherent rays then the performance will improve.

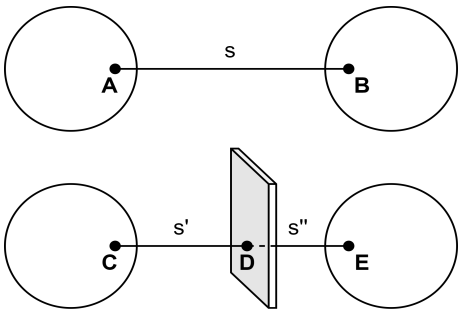



**Figure 2.2.1:** The concept of a maximal free segment, visualized. On top, we see two points $A$ and $B$ that are mutually visible. In other words, the linear space between the two points is unoccluded. The unoccluded line segment $s$ between the two points is therefore called a maximal free segment. On the bottom, points $C$ and $E$ cannot "see" each other. The line segment $CE$ is occluded by an object. Therefore, the linear space between points $C$ and $E$ is split into two maximal free segments, namely $s'$ and $s''$.

## 2.3 Spherical harmonic lighting

### 2.3.1 Introduction

Because incoming radiance from (distant) light sources at a point can come from an arbitrary amount of directions, it can be useful to represent this as a *spherical function*. A spherical function $S(\theta, \phi)$ is a 2D function parameterized by an elevation angle $\theta$ and an azimuthal angle $\phi$ respectively. An example of a spherical function is the lighting function shown in Figure 2.3.1. The function represents incoming light at a point coming from 2 monochromatic light sources. This is just one example, but intuitively we can assign any point on the sphere an arbitrary value, allowing for an infinite amount of possible spherical functions.
What is more special is that there exists a special kind of spherical function that can be used to approximately reconstruct any other spherical function. We call these special functions *spherical basis functions*. Different types of spherical basis functions exist, such as *spherical harmonics [13], spherical gaussians and spherical radial basis functions*. In this section we will discuss spherical harmonics in more detail given they are a suitable candidate to reconstruct smooth signals like diffuse indirect lighting. Spherical harmonic basis functions form a complete set of *orthogonal functions*, which makes them an *orthonormal basis*. Intuitively this can be thought of as a set of basis functions that cover the whole domain, but have no overlap between each other. In other words, the product of two basis functions that are the same results in a constant value $c$. In contrary, the product between two basis functions that are not the same equals 0. There exists a great analogy between the *Fourier transform* and spherical harmonics

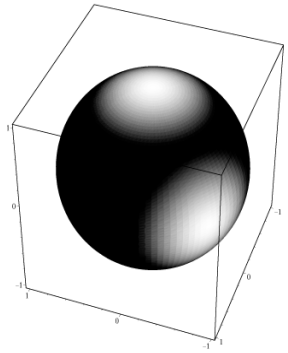


**Figure 2.3.1:** An example spherical lighting function representing the incoming light at a point $P$. While $P$ stands for the center of the sphere, the function value $S(\theta_i, \phi_i)$ can be seen as the incoming radiance at point $P$ along direction $(\theta_i, \phi_i)$. Lighter points represent higher values, while darker points represent lower values. Figure from [13].

(SH). While the Fourier transform decomposes an arbitrary function (signal) into sine wave frequency components (the Fourier basis), SH decomposition decomposes an arbitrary spherical function into SH basis functions.

### 2.3.2 Basis functions

SH basis functions are small pieces of a spherical signal with which we can make a linear combination to reconstruct an approximation to the original function. Each basis function $B_i$ is weighted by its coefficient $c_i$. How close the approximation is to the actual function depends on the amount of SH bands used. An SH band is denoted by $l$, which is also called the *degree* of the spherical harmonic. This terminology originates from the *associated Legendre polynomials* [23], which serve as the foundation for the mathematical definition of spherical harmonics. Associated Legendre polynomials, denoted by $P_l^m(x)$, are the canonical solutions of the *general Legendre equation* [2]. The general Legendre equation is encountered in the solution of *Laplace's equation* for spherical coordinates. The theoretical details and derivation of these principles will not be covered in this thesis, since they are rather tedious and unnecessary to get a good intuition of the concepts presented here. We can think of the SH band $l$ as a set of angular frequencies that are described by this band. The higher the number of the band, the higher the angular frequency that is related to it. Again we can make the analogy to Fourier decomposition, where an arbitrary periodic function can be approximated by a weighted linear combination of sine wave basis functions, each with a different frequency. Intuitively this corresponds to the basis functions 'explaining' the original function's content of its frequency. Assume for example a basis function $B_i$ that is weighted by coefficient $c_i$. The quantity of information with the same frequency as $B_i$ in the original function is proportional to the value of $c_i$. The set of SH basis functions for the first 5 SH bands are presented in Figure 2.3.2.

### 2.3.3 Projection and reconstruction

Calculating how much we need to weigh each basis function in the linear combination that forms the reconstruction is called *projection*. Conceptually this is quite simple, we project the original function $f(x)$ onto each basis function $B_i(x)$ by calculating the product $f(x)B_i(x)$. By integrating this product over the full domain of $f$ we obtain the coefficient $c_i$ to weigh $B_i$. Intuitively, the integral of the product between the original function $f(x)$ and basis function $B_i(x)$ over the full domain of $f$ gives us a metric that represents the fraction $\eta$ of $f$ that is 'covered' by $B_i$. In other words, $B_i$ 'explains' fraction $\eta$ of $f$, or contains fraction $\eta$ of $f$'s information. Conversely, we can reconstruct the original function approximately by scaling each basis function $B_i$ by its coefficient $c_i$, and summing the results. An illustrated example

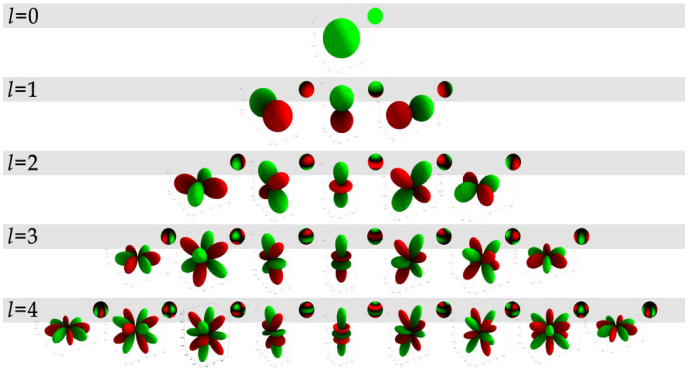



**Figure 2.3.2:** Spherical basis functions for the first 5 SH bands. Green points represent positive values and red points represent negative values. We can observe the angular frequency increasing as the band number increases. Figure from [13].


**(a)** Projection **(b)** Reconstruction

**Figure 2.3.3:** a: We calculate basis function coefficients $c_i$ by integrating the product of the original function $f$ and each basis function $B_i$ over the full domain of $f$. b: We approximate the original function $f$ by summing over each basis function $B_i$ scaled by $c_i$. Figure from [13].

of projection onto basis functions and reconstruction from basis coefficients is shown in Figure 2.3.3.

### 2.3.4 Calculating the basis function projection of an unknown function

The formula $\int_a^b f(x)B_i(x)dx$ with $[a, b]$ the full domain of $f$ is useful for an analytical approach to integration, i.e. when we know the specifics of each term and the inner term of the integral is integrable. However, for many interesting problems that we wish to solve this is not the case. Later in this thesis we will see an example of an unknown lighting function that we wish to project onto basis functions. Although we are unable to solve the integral for unknown terms analytically, numerical solutions are available. One powerful method is Monte Carlo integration, which was introduced in section 1.4. Since Monte Carlo integration makes use of samples and can be used to approximate the integral of an arbitrary function, we can also use it to estimate the basis function projection of an unknown spherical function of which we can only take samples.

## 2.4 Denoising

Due to the stochastic nature of the numerical Monte Carlo approximator, variance is inevitable. This variance manifests itself as noise in the resulting image. By taking more samples, the variance can be reduced, however this implies a heavier computational workload that goes beyond the budget of real-time computer graphics. Together with dedicated hardware designed for ray-tracing operations, what really made real-time ray tracing possible are the impressive results


**Figure 2.4.1:** Example scene that illustrates the performance of NVIDIA's real-time denoiser (NRD). On the right we can see the input image containing a severe level of noise. In the middle the reconstructed image by NRD. On the left the reference image rendered with a very high amount of samples per pixel.

achieved by global illumination denoisers. A remarkable advancement in this area is *spatiotemporal variance-guided filtering (SVGF)* [36]. Denoisers implement roughly three distinct techniques (or combinations of them): spatial filtering, temporal accumulation and reconstruction via deep learning. Spatial filtering works on local image regions and uses spatial information of similar neighbours to reconstruct the denoised image. Side effects of spatial filtering are blurriness, muddiness and temporal instability artifacts when used on its own. Temporal accumulation takes into account the information from (the) previous frame(s) in order to correct temporal instabilities between frames. It therefore reduces temporal artifacts. Finally, image denoising via deep learning trains a neural network to reconstruct the denoised signal. The neural network is trained to minimize reconstruction loss on a dataset containing a wide variety of noisy images together with their ground-truth denoised image. Modern commercial denoisers such as ReLAX and ReBLUR [21, 40], developed by NVIDIA, need no more than 1 light path sample per pixel to produce plausible images in many scenarios [24]. NVIDIA OptiX (section 3.14) version 5 also introduced a learning-based denoiser that achieves interactive frame rates and makes use of autoencoders [7].

## 2.5 Related work

### 2.5.1 Ray binning

The incoherent nature of light transport causes rays to be scattered all over the scene during the process of path tracing. This divergence between rays typically hurts performance because cache and memory bandwidth are put under extra pressure. One technique to improve ray coherence is *ray binning*. In ray binning, rays are grouped (binned) by direction and origin. In general, two rays have a good chance of intersecting the scene at a similar location when their origin and direction are similar. Implementations of ray binning can be found in several state-of-the-art applications. It has been used to accelerate ray-traced reflections in the video game *Battlefield V (2018)* and an implementation was added to the commercial game engine *Unity* [6]. Ray binning is illustrated in Figure 2.5.1. Note that although ray binning generally improves performance, it is certainly not free. Depending on the application, it should be verified whether gains made by increased coherence outweigh the overhead introduced by the binning of the rays.

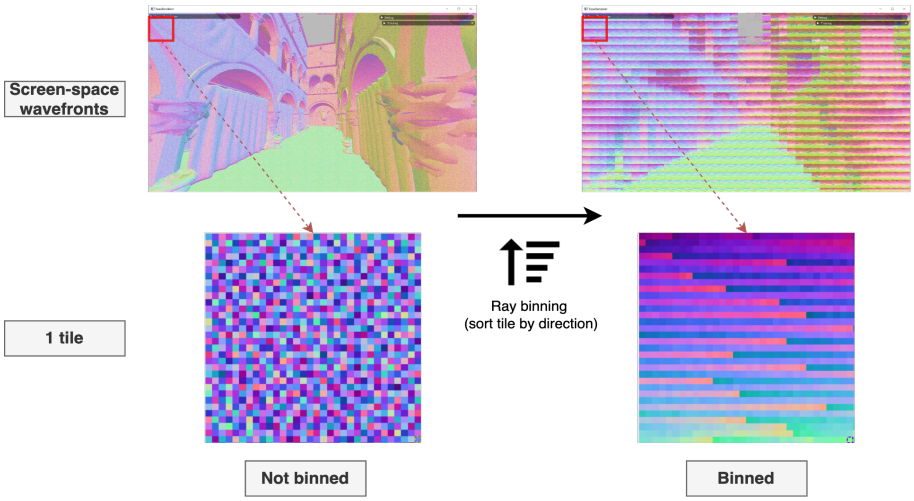
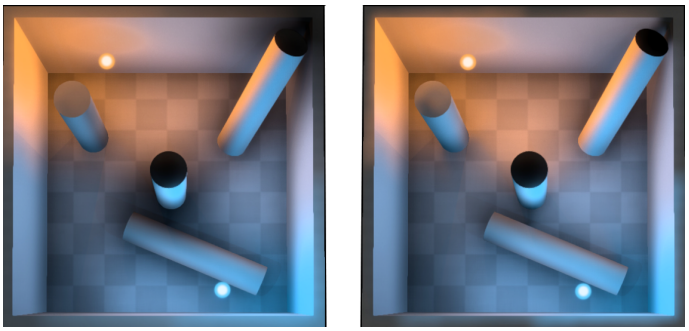



**Figure 2.5.1:** Visual overview of the ray binning implementation used in Unity game engine. In the top row we see the directions of each ray rendered into screen space. The screen space is subdivided into tiles. In the bottom row we see a zoomed view of 1 tile. The layout before ray binning is shown on the left, while the layout after is shown on the right. Rays are sorted by direction for each tile. Before the execution of ray binning, two subsequent ray directions in a tile can be very different. This can cause incoherent memory accesses when these rays are processed sequentially. We observe that after ray binning is done, the wavefronts in screen space have a more coherent layout. Rendered images from Kosta Anagnostou's blog (Interplay of Light).

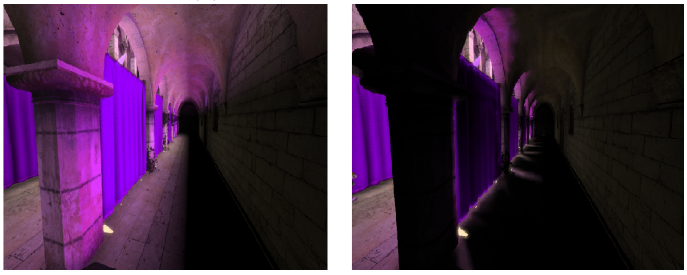
### 2.5.2 Precomputed light field probes

A recent approach to increase the ray coherence in path tracing and achieve real-time global illumination is the use of *precomputed light field probes* [25]. Light field probes sample a scene's light field and visibility at a certain point in space. The method presented by McGuire et al. [25] uses ideas from screen-space ray tracing and voxel cone tracing to efficiently sample the light field probe data structure with awareness of visibility. A common problem with techniques based on radiance probes is their placement. Wrong or insufficient placement of light probes can lead to light and shadow-leaking artifacts or misaligned reflection effects. More specifically, this happens when the mutual visibility assumption between a light probe and the surface it is contributing to is violated (e.g. because a probe is placed inside of an object, or an area between probes contains high geometrical complexity). An example of these artifacts is illustrated in Figure 2.5.2. McGuire et al. solve these issues by prefiltering visibility. They use a sparse regular grid of light field probes distributed over the scene. For each of the probes in the scene they precompute an *octahedral irradiance map*, which is in essence a *3D-slice of the 5D-plenoptic function*. In other words, the idea is that the light field is being queried at the 3D spatial position of each light probe, over *all* incoming directions. In addition to this, they also compute filtered maps of radial distance and squared radial distance between the probe's position and surrounding geometry. The radial distance is a representation of visibility information: indeed, it can be used to find the *maximal free segment* between 2 points in space. Following this interpretation, by taking the union of information that all light probes provide, we get a good notion of global 3D visibility information in the scene (provided that a sufficient amount of probes were placed). Because the underlying cause of the aforementioned artifacts was the violated assumption of mutual visibility, this visibility information is what empowers the authors to place radiance probes at arbitrary locations in the scene, without having to worry about introducing artifacts related to the light field probes.

The most relevant part of this previous work to this thesis is how it increases memory coherence. The authors developed the *light field ray tracing* algorithm, which allows to trace



**(a)** Shadow leak



**(b)** Light leak

**Figure 2.5.2:** (*a*): "Darkness leaks" around the pillar in the center and the pillar in the upper right corner. The cause of this is a radiance probe being inside the pillar's geometry, therefore being completely isolated from any incoming radiance. When queried, the isolated probe contributes its null radiance values to surrounding geometry, resulting in the undesired shadow leaks. On the right, the same scene rendered with prefiltered visibility, solving this problem. (*b*): Radiance is "leaked" through the curtain's geometry, leading to undesired and unrealistic lighting effects. This is caused by highly varying spatial lighting conditions around the placement of the radiance probe. On the right we can see again that visibility information helps to resolve this issue. Figures from [25].

world-space rays via texture traversal. This is achieved by transforming a world-space line segment in $R^3$ to a polyline in $R^2$ via *spherical octahedral projection*. Since each light probe contains an octahedral texture for incoming radiance, radial distance and surface normals for surrounding geometry, this corresponds to mapping the 3D world-space rays onto the light probe's 2D representation of the 3D light field slice. The trace of the ray can then entirely be processed in 2D octahedral texture space of the light probe (switching between light probes is necessary if there is insufficient visibility information), which is extremely coherent. Nevertheless, the method also has its limitations. For example, the light field probe ray tracer produces unacceptable aliasing for *primary eye rays*. Another limitation is its high memory footprint caused by the octahedral probe representation of the light field slices.

### 2.5.3 Ray classification

In another attempt to find and exploit coherence in ray tracing, J. Arvo and D. Kirk [3] discovered that the global volume of rays in a scene can be subdivided in 5D space. To enlighten this, each ray can be represented by its origin (a 3D vector) and its direction (a 2D spherical coordinate). Instead of subdividing the scene spatially, the 5D rays are grouped into 5D hypercubes, each of which represents an equivalence class. A 5D hypercube can be seen as a bounding volume in 5D space, which contains the set of rays that are located between the minimum and maximum coordinates along each axis. Rays that are contained within the same hypercube are coherent by nature of this subdivision. This method is called *ray classification*. The rays are classified hierarchically, using *lazy-evaluation* to build the hypercube tree, meaning that child nodes are only generated when necessary. This prevents the solution from allocating memory for nodes that will never be accessed. To each hypercube the authors assign a candidate set. This set represents the scene objects that could be intersected by the rays in the paired hypercube. The data structure thus serves as a lookup table, which takes in a ray, and looks up the set of objects that that ray could possibly intersect. This can also be interpreted as the set of objects that are 'visible' via a set of rays. Compared to the 3D Visibility Complex this is a slightly weaker, nevertheless very useful property.

A great advantage that follows from the coherence that exists within the data structure is the opportunity to exploit caching. The method allows us to check unprocessed rays against cached hypercubes. If it is located within a cached hypercube, we can instantly return the candidate set, without having to traverse the hierarchy. Since rays that are near each other in 3D space are also likely in the 5D data structure, this method of caching can significantly speed up computation. A disadvantage of the classic ray classification method is that it does not support dynamic scenes.

### 2.5.4 Shader execution reordering

A very recent development made by NVIDIA is the concept of *shader execution reordering* (SER) [8], available on the Ada Lovelace generation of NVIDIA GPUs (RTX 40 series). This innovation, illustrated in Figure 2.5.3, is similar to the work presented in this thesis in the sense that it performs reordering of GPU threads that perform similar work to reduce both data and execution divergence. The reordering is implemented in hardware and available as an API to the programmer. The core functionality can be called with a single function: *void ReorderThread(key)*. This function call can be called from any GPU thread, and it will ask the system to reorder all the threads that called the function. After the execution of this function, threads that are scheduled together will be more coherent (i.e. suffer less from data or execution divergence) than before *ReorderThread* was called. The higher coherence is achieved by sorting the threads based on the parameter *key* that is passed to the function. In the use case of ray tracing, the key parameter is a *HitObject*, representing the hit information of a ray that was sent out into the scene. Based on this hit information, the system can decide on a specific ordering of the threads, before starting any shading calculations for which additional data is necessary. The system could for example group threads that have a similar 3D hit location,

because they are likely to use the same data for shading operations. This maps directly to the relation between ray coherence and memory coherence, which is one of the two topics of focus for this thesis. However, there are differences between SER and the approach presented in this thesis that are worth mentioning. First of all, SER aims to do a fine-grained reordering of threads to increase both instruction and memory coherence. It does this by applying a hardware-accelerated key-based sorting algorithm on the threads ready for execution. It is important to note that this sorting step introduces additional overhead. Whether this extra overhead improves overall performance depends on the situation. On the other hand, our approach aims to perform a more high-level, coarse-grained reordering of ray calculations via a coherent radiance communication scheme and limiting ray traversal. Although it will not deliver a perfect and deterministic ordering, our coarse-grained ray sorting attempts to increase memory coherence without introducing any additional overhead.

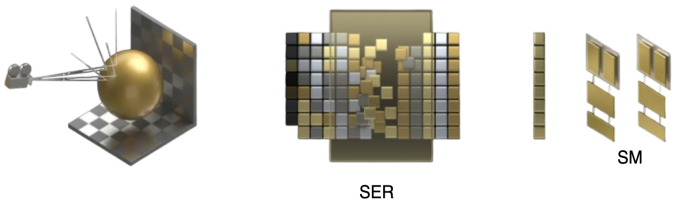
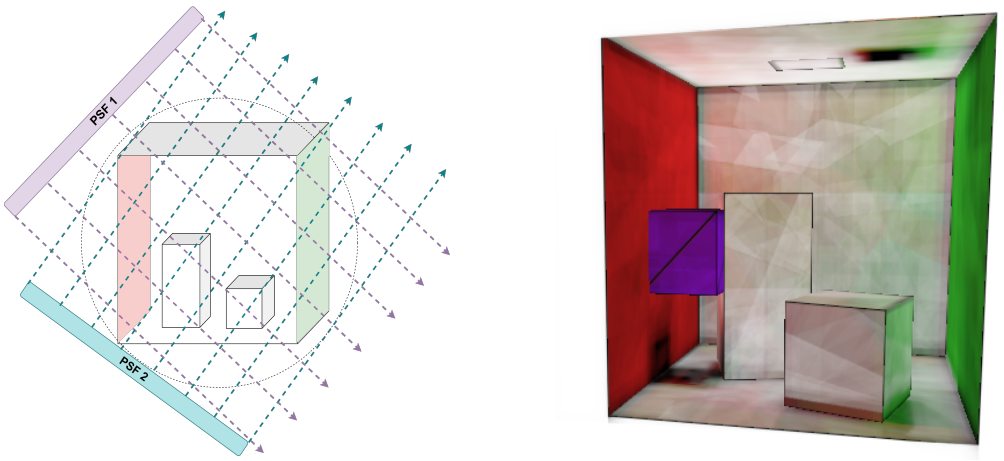



**Figure 2.5.3:** Illustration of the concept behind Shader Execution Reordering (SER). On the left we see rays being scattered into the scene, off to hit very different spatial locations that contain various materials. In the center it is visible that the threads are initially very disorganized due to the data incoherence caused by the incoherent nature of path tracing. The SER interface reorders the threads, leading to coherent warps that are ready to be executed on a Streaming Multiprocessor(SM). Figure source: [8]

### 2.5.5 A virtual light field approach to global illumination

Another valid approach to achieve global illumination is the use of *virtual light fields* (VLFs) [37]. This idea was introduced by J. Mortensen in his PhD thesis [27]. The essential idea behind this method is the transformation of the ray tracing problem into a rasterization problem. Parallel bundles of rays (called *parallel subfields*) are traced throughout the scene by making an orthogonal projection from different viewpoints around the scene. This maps directly to rasterization, where each pixel can be considered as one ray in the bundle of parallel rays. Visibility is handled by performing a clever trick with the available GPU hardware and traditional render techniques. Depth layers for each pixel (ray) are rendered into multiple samples via multisampled rendering and turning off the depth test (*z-buffering*). The distance between 2 depth layers are then the *maximal free segments* (cf. The 3D visibility complex, section 2.2) along that direction. The rasterization hardware available on consumer grade GPUs can efficiently be employed to solve this problem. A great benefit of this technique is extreme coherence achieved via the grouping of rays in parallel bundles. Some disadvantages of this approach are that it is less suitable for small area light sources, as the number of required sampling directions increases drastically in this case. The method also introduces bias, especially for specular viewpoint-dependent effects, where interpolation between discrete samples is needed. Furthermore, a high number of sampled directions is necessary in order to prevent discretization artifacts. Finally, with the availability of hardware-accelerated ray tracing, this method might be less relevant as of today. A simplified illustration of a virtual light field for the *Cornell box* scene together with an example render is shown in Figure 2.5.4.




**(a)** Simplified schematic **(b)** Example render

**Figure 2.5.4:** (*a*): Shown in the illustration are 2 parallel subfields (PSFs), which consist of a parallel bundle of rays along a certain direction, encapsulating the whole scene. The union of all PSFs that can be created via orthogonal projections on the scene's bounding sphere is the virtual light field (VLF) that represents the scene's light volume. A light volume is the collection of all possible rays in a volume (all ray directions are covered in each 3D point). (*b*): Diffuse global illumination rendered for the Cornell box scene using the VLF approach with 384 PSF directions. Discretization artifacts are clearly visible, due to the low amount of directional samples and the only emitter being a small area light.

### 2.5.6 The irradiance volume

The visual appearance of a point on a diffuse surface can be described by the product of the Lambertian BRDF and the radiometric quantity *irradiance* at that surface point. Irradiance at a point can be computed by a cosine-weighted integral over the hemisphere of directions at that point, with the integrand being the incident radiance function $L_i(\mathbf{x}, \omega_i)$. The irradiance function itself is defined over all surface points of the scene, making it a computationally intensive function to compute. Greger et al. defined a novel volumetric approximation of the irradiance function, called the *irradiance volume* [14]. The irradiance volume estimates irradiance at surface points by trilinearly interpolating between discrete samples, taken from *irradiance probes*. Irradiance probes are organized in a multi-level 3D grid (regions containing geometry are sampled more densely), forming the irradiance volume data structure. The method can handle *semi-dynamic* scenes. A scene is semi-dynamic when most of the surfaces are stationary and the dynamic objects are small relative to the rest of the environment. The irradiance volume is built from the static geometry during a preprocessing stage and can then be queried to illuminate dynamic objects with indirect illumination. Direct illumination is not supported and should be handled separately. A visualization of the irradiance volume is shown in Figure 2.5.5.

### 2.5.7 (Cascaded) light propagation volumes

A *light propagation volume* (LPV) [19] is a data structure used to approximate indirect illumination in dynamic scenes, with support for participating media. The method embeds the scene into two grids, one that stores the light distribution (the LPV) and one that stores a volumetric approximation of the scene geometry (the *geometry volume* or GV). The directional distribution of light is represented using low-order spherical harmonics. From a high-level perspective, the algorithm to build the data structure consists of three subsequent steps. The first step is to initialize the the LPV with surfaces causing indirect lighting and low-frequency direct light (area light sources). The idea behind this step is that the low-frequency lighting in the scene is

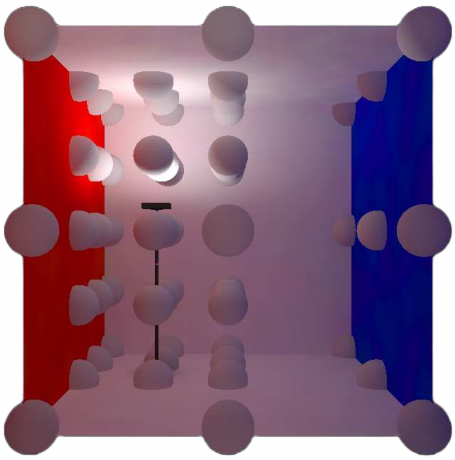


**Figure 2.5.5:** Visualization of the irradiance volume. Irradiance probes are represented by spheres. The color of the sphere shows the irradiance in each direction at the probe's position. We observe a red glow for the directions pointing towards the red wall, while the directions pointing towards the blue wall have a blue appearance. Notice the multi-level grid of irradiance probes. Regions with more geometry are sampled more densely. Figure from [14].

converted into a set of *virtual point lights* (VPLs) [20]. The VPLs are subsequently transformed into spherical harmonics (SH) representation. The SH form is a directional representation of the VPL's intensity. The SH representations of the VPLs are then 'summarized' into the LPV's grid structure by adding up the SH coefficients to form one common SH probe per LPV cell. In the second step, the scene geometry is 'summarized' by the geometry volume. This is done by a method called *fuzzy occlusion*. In fuzzy occlusion, the directional blocking of light going through a GV grid cell is determined by the accumulation of the blocking potentials of all surface elements inside this grid cell. The directional blocking potentials are also transformed into SH space, and combined. The scene geometry proxy is therefore modeled as a grid of spherical harmonics, each of which represents directional probabilities that light going through the cell in the corresponding direction is blocked. The third and final step is that light is iteratively propagated throughout the LPV grid. This is done by taking the initial state of the LPV as input, where each cell stores a SH probe that represents the directions in which it is sending out radiance. This outgoing radiance is then communicated to its 6 neighbors (4 axial directions). The exchange of radiance between neighbors is iteratively repeated for a number of iterations depending on the resolution of the grid. An overview of the light propagation is visualized in Figure 2.5.6.
This work greatly inspired us during the early stages of our design. It gave us the idea to group work locally per cell and make use of a propagation scheme to increase coherence. Furthermore, the compression of low-frequency lighting with VPLs, represented by a grid of spherical harmonics, motivated us to make use of radiance probes.

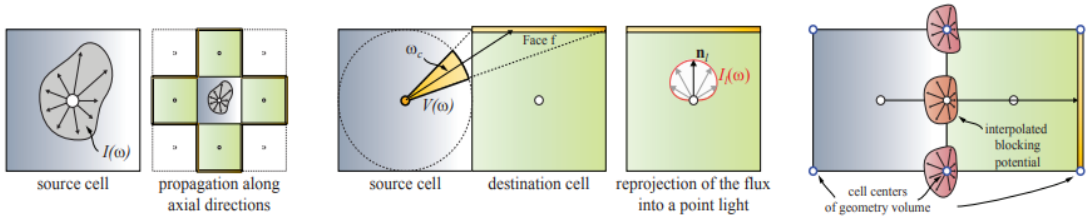



**Figure 2.5.6:** Radiance propagation between neighbors and reprojection into point lights, two ideas that greatly inspired us during the design phase of the algorithm proposed in this thesis. Figure from [19].

# Chapter 3

# Implementation

In this thesis we attempt to address two problems. First of all, we want to find an adaptation to Monte Carlo path tracing that is more suitable to a cloud-based rendering system. Secondly, we attempt to minimize incoherent memory accesses during the computations made by this algorithm. In order to achieve this, we developed a data structure and rendering algorithm that primarily operate on spatially local data. To enable for diffuse global illumination effects, it implements a per-bounce radiance transfer communication scheme. In this chapter we discuss the technical details of the concepts that form our system and design choices that we made.

## 3.1 Overview of the system

The core component of the proposed system is the *radiance cell*. The scene is divided into a regular grid of radiance cells, each of which represents a local spatial volume that keeps track of its local geometry and other local scene data. Besides a grid of radiance cells, the system also introduces a regular grid of *radiance probes*. These radiance probes capture the omnidirectional incoming (indirect) radiance at a certain point in space. By combining the concepts of both radiance cells and indirect radiance probes, we can transfer radiance throughout the scene in a flexible but rather coherent way. In particular, we set a threshold value that determines for each indirect lighting ray sample how far it can traverse space. Indirect light sources that fall outside of this range are considered distant and therefore they contribute a form of *distant indirect lighting*. This distant indirect lighting is approximated by the information absorbed by the radiance probes. Our hypothesis is that this hard limit on a ray's traversal will benefit memory coherence. From a high-level perspective, the rendering algorithm consists of 3 render passes: the *direct lighting* pass, the *radiance probe gathering* pass and finally the *radiance cell scattering* pass. The latter two passes can be repeated as many times as desired, depending on how many indirect lighting bounces the user requires. A visual overview of the system is provided in Figure 3.1.1.

## 3.2 Initializing the data structure

Before the actual render process can start, some preprocessing steps are required to initialize the underlying data structure. When we talk about memory in this chapter, we will make the distinction between host and device memory. Host memory represents the computer's main memory, while device memory is the GPU's dedicated memory. First, the scene geometry is loaded into host memory and normalized. Next, the light source data (emitters) is initialized. Once this data is in host memory the radiance cell grid can be built. The resolution of the grid can be altered depending on the density requirements of the scene.

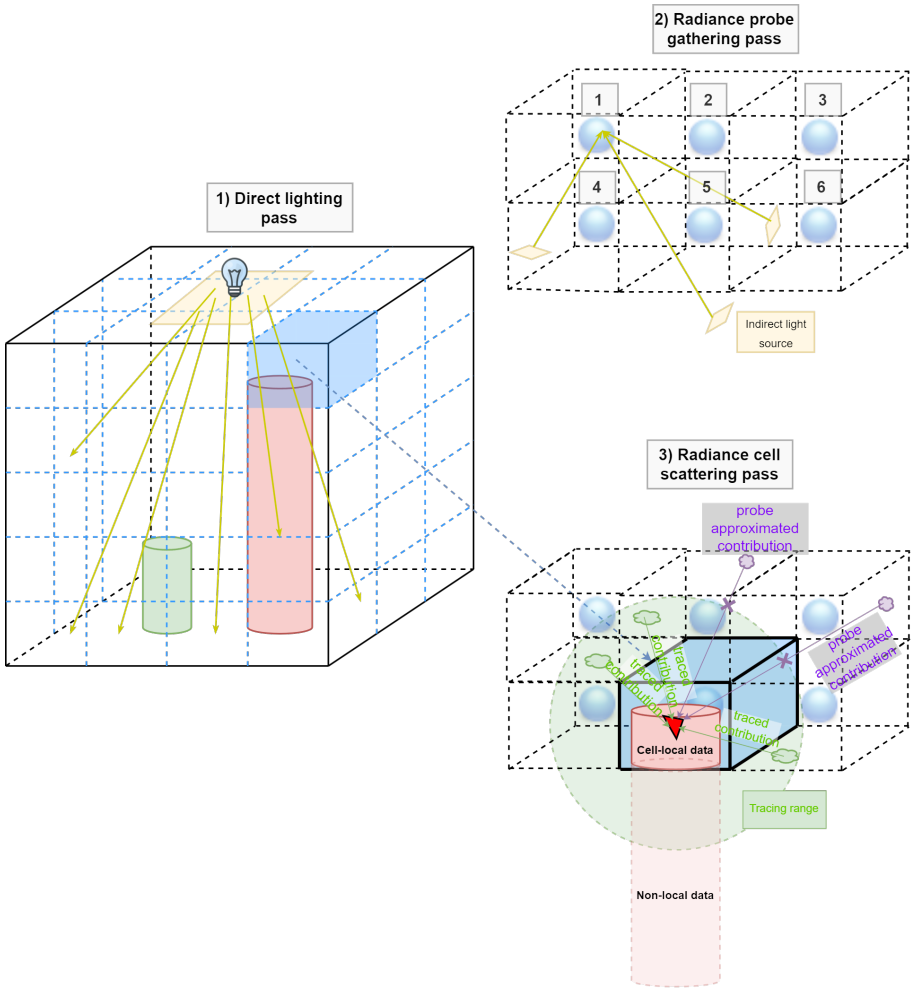



**Figure 3.1.1:** High-level overview of the thesis' proposed rendering algorithm and used data structures. The scene is subdivided into a regular grid of radiance cells. The algorithm starts with the direct lighting pass, in which direct radiance starting from emitters is propagated throughout the scene. Secondly, during the radiance probe gathering pass, indirect radiance from the previous lighting bounce is gathered into radiance probes. After this pass each radiance probe contains an "omnidirectional view" of the incoming indirect radiance at that exact location. Finally we have the radiance cell scattering pass, in which indirect radiance is transferred onto the local geometry of each radiance cell. In other words, the scene's surfaces are illuminated with indirect radiance for the current lighting bounce. For nearby indirect radiance transfer, we have a threshold value called the tracing range that determines for each surface point the maximal range for which we perform path tracing. If we do not find an intersection within this range, we make the assumption that the incoming radiance along that direction is distant. We approximate the real distant radiance by using the information in the most nearby radiance probe. Note that pass 2 and 3 together deliver us one bounce of indirect light. We can repeat these passes as many times as we want, depending on the amount of lighting bounces we desire.

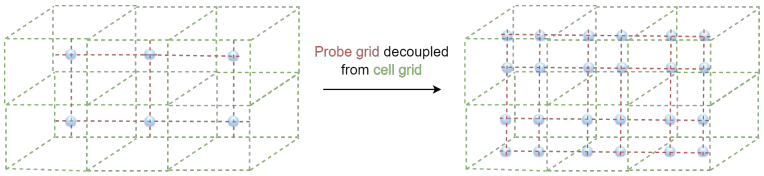



**Figure 3.4.1:** The radiance probe grid is decoupled from the radiance cell grid. On the left we see the configuration that is featured in our implementation, which has 1 radiance probe centered in each radiance cell. On the right a more dense configuration is used. A motivation to use more dense radiance probe grids could for example be the presence of high frequent lighting effects.

## 3.3 Initializing device memory

Since the majority of the algorithm's calculations are performed on the GPU, there are several buffers that need to be prepared before the render passes can use them. First of all the algorithm requires a buffer to store the contribution for a lighting bounce. For ease of explanation, assume that we allocate a separate buffer for each lighting bounce. To improve memory usage, it is possible to combine the contributions of multiple lighting bounces into one accumulator texture. Furthermore, to compute incoming radiance onto the surfaces, we need the world positions and normals of the surface points where we are gathering radiance. Assuming the scene is static, this data can be precomputed in a preprocessing procedure. The world data of dynamic objects will need to be updated each time the object undergoes a transformation. Finally, we need to provide a buffer for the radiance probe data. The size of this buffer depends on the type of radiance probe that is used. More details on radiance probes are discussed in the next subsection.

## 3.4 (Indirect) radiance probes

The radiance probe grid is a data structure that is used for further high-level optimization of our algorithm. It should be noted that although the use of radiance probes introduces bias, it *limits* the range in which intersection tests need to be done, potentially eliminating a significant amount of calculations and increasing the coherence of the memory accesses being made. For an unbiased approach, see section 3.10. Our implementation provides a radiance probe in the center of each radiance cell for generality purposes and ease of implementation, however other configurations might be desired depending on the scene layout and materials present in the scene. It should furthermore be noted that the radiance probe grid is decoupled from the radiance cell grid; that is, in principle the radiance probe grid can have any configuration, regular or irregular, as long as the radiance cell scattering algorithm accesses the probes in a correct way. This is illustrated in Figure 3.4.1.

Another important thing to discuss is the type of radiance probes that are used. We are interested in techniques to represent incoming radiance from all directions at a certain point. A straightforward representation to achieve this with is an *environment cubemap*. A cubemap is a texture that exists of 6 layers, each of which represents one face of the cube. What is interesting is that there exists a direct mapping such that a cubemap can be queried using 3D direction vectors. We can thus use a cubemap to store and query the incoming radiance along a certain direction at a given point in space. A cubemap has a few drawbacks however, the most important of which is its high memory cost for our use case. Pseudocode to convert a 3D direction vector to cubemap space is given in Algorithm 1. An example of a cubemap radiance probe that we used is presented in Figure 3.4.2
Another valid type of radiance probes for our use case are spherical harmonic probes. In section 2.3 we introduced the necessary theory behind spherical harmonics to understand this discussion. Because of their ability to accurately reconstruct functions that contain low frequencies

**Algorithm 1** 3D direction vector to cubemap UV mapping

maxExtent = Find max extent of direction vector **d** (X,Y,Z)
**if** The X component is the max extent **then**
    Set face index to 0 or 1, $uc$ to -Z or Z and $vc$ to Y, depending on whether X is positive
**end if**
**if** The Y component is the max extent **then**
    Set face index to 2 or 3, $uc$ to X and $vc$ to -Z or Z, depending on whether Y is positive
**end if**
**if** The Z component is the max extent **then**
    Set face index to 4 or 5, $uc$ to X or -X and $vc$ to Y, depending on whether Z is positive
**end if**
$u \leftarrow 0.5 * ((uc/\text{maxExtent}) + 1.0)$
$v \leftarrow 0.5 * ((vc/\text{maxExtent}) + 1.0)$ **return** $u$, $v$, faceIndex

**Figure 3.4.2:** Radiance probe cubemap that absorbs the omnidirectional incoming indirect radiance at a certain point in space. The indirect radiance that is shown here is for the second lighting bounce (the indirect radiance that is caused by diffuse reflections after the direct lighting pass).

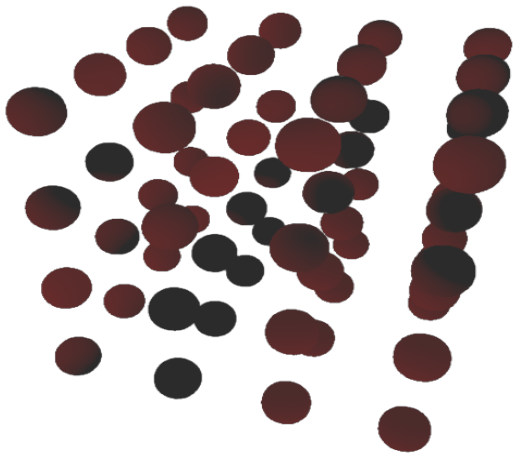


**Figure 3.4.3:** Visualized example grid of spherical harmonics radiance probes. In this example only one color channel is visualized. In practice, a vector of spherical harmonics coefficients is provided for each color channel.

(no sudden changes in function values), they are a great candidate to serve as a probe for indirect radiance. The reason behind this is that subsequent illumination bounces are considerably smoother due to the scattering behavior of the BRDF, which serves as a low-pass filter. This is especially true for diffuse surfaces. Although in this thesis we will focus on diffuse surfaces, a SH representation will struggle to reconstruct indirect radiance coming from perfectly specular surfaces, because their scattering behavior is anything but smooth. What makes a SH representation very attractive for our use case is its low memory requirements. With just 9 coefficients per RGB channel we are able to represent a 3rd degree SH. This requires only 27 floating point numbers (108 bytes) per radiance probe, compared to the cubemap representation which would already consume double the amount of space (216 bytes) if we would use 3x3 textures on each face of the cube. Smaller memory size has several advantages, it allows us to cache more radiance probe instances simultaneously, reducing the probability of a cache miss. Furthermore, it can be very interesting to reduce the required bandwidth for real-time streaming of scenes using this representation. An example visualization of a grid of spherical harmonic radiance probes is shown in Figure 3.4.3.

For glossy and perfectly specular surfaces the reflected radiance is very viewpoint dependent, so a radiance probe using a spherical harmonic representation might not be able to effectively capture parallax effects caused by these surfaces. In these situations it might be necessary to use a second radiance probe representation (e.g. spherical wavelets, spherical Gaussians...) to better handle these types of materials. A denser grid of radiance probes might also be desirable in this case. Although our proposed method does not directly support specular surfaces, we believe the above-mentioned ideas could enable support for it.

## 3.5 The direct lighting pass

During the direct lighting pass the direct contributions from emitters in the scene are gathered onto the scene geometry. The points for which we gather the contributed irradiance will need to be represented by some data structure. This is the topic of section 3.11. During the direct lighting pass we iterate over the entries of this data structure. In practice, many parallel threads are started for each entry in this data structure. The steps that each thread performs are listed in Algorithm 2. After this pass we have a scene that is irradiated with contributions from direct emitters only (no interreflections). A visualization is shown in Figure 3.5.1.

**Algorithm 2** Algorithm to calculate direct irradiance at a surface point. *threadIndex* is the index of the launched thread running this procedure.

```
entryWorldPos ← worldPositionBuffer[threadIndex]          ▷ World pos. of surf. point
entryWorldNormal ← worldNormalBuffer[threadIndex]         ▷ World normal at surf. point
diffuseBRDF ← diffuseColorBuffer[entryIndex]              ▷ BRDF of surf. point
for e in emitters do
    stratifyWidth ← e.properties.height / stratificationResolution
    stratifyHeight ← e.properties.height / stratificationResolution
    irradianceAccumulator ← {0.0, 0.0, 0.0}               ▷ RGB irradiance
    numSamples ← 0
    for x ← 0 to stratificationResolution.x - 1 do        ▷ Loop over stratification cells
        for y ← 0 to stratificationResolution.y - 1 do
            lightSamplePos ← random point in stratification cell (x, y)
            rayDir ← lightSamplePos - entryWorldPos
            Shoot ray and check for occlusion between entryWorldPos and lightSamplePos
            if light ray is not occluded then
                cosineWeightHitpoint ← ⟨hitNormal, rayDir⟩        ▷ Cosine weight hitpoint
                cosineWeightLight ← −⟨e.normal, rayDir⟩           ▷ Cosine weight light source
                lightArea ← ‖e.du × e.dv‖                         ▷ Light source area
                distanceToLightSample ← length(lightSamplePos - entryWorldPos)
                totalWeight ← (cosineWeightHitpoint * cosineWeightLight * lightArea) / (π * distanceToLightSample²)
                irradianceAccumulator += e.power * totalWeight * diffuseBRDF
            end if
            numSamples ← numSamples + 1
        end for
    end for
end for
irradianceAccumulator /= numSamples
```

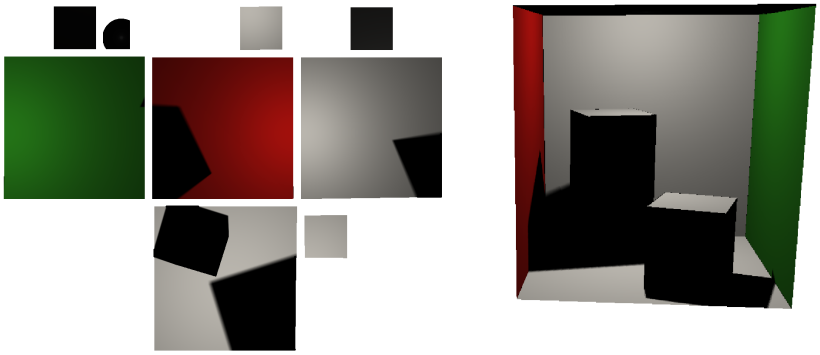


**Figure 3.5.1:** Result of the direct lighting pass, shown for the Cornell box scene.

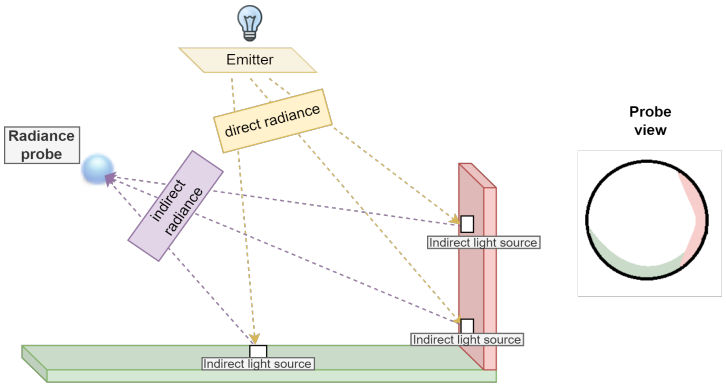



**Figure 3.6.1:** Simplified illustration of indirect radiance gathering for one radiance probe. The emitter emits direct radiance. When a direct radiance ray interacts with the scene it is reflected by the surface of the object it intersects with. This reflection can be seen as indirect radiance. It is exactly this indirect radiance that is captured by the radiance probes that we use. On the right side of the figure an approximated omnidirectional 'view' is shown for this radiance probe. It should furthermore be noted that we handle each lighting bounce separately.

## 3.6 The radiance probe gathering pass

The second render pass of the algorithm is the *radiance probe gathering* pass. During this pass, as its name implies, indirect radiance is gathered into radiance probes. We march through the grid of probes and launch a kernel for each probe separately. By doing this, each probe builds its own omnidirectional 'view' of the incoming indirect radiance that is the result of the previous lighting bounce. An illustration of this concept is provided in Figure 3.6.1. The view of indirect radiance can be seen as a spherical function, that returns a color for each direction on the sphere.

Possible representations for radiance probes were already briefly discussed in section 3.4. In this section, we assume this representation to be abstract: it is a collection of *entries* that collect incoming indirect radiance per direction. The render pass sends out a ray for each of those entries, according to the direction that belongs to the entry. At its first intersection, each ray reads the irradiance value stored during the previous lighting bounce. This irradiance value represents the diffuse appearance of the point, and thus the indirect radiance that it emits during the current lighting bounce. The value is therefore written to the entry for which the ray was traced. At the end of the render pass, the entries together form a discretized omnidirectional representation of the incoming indirect radiance at the probe's position. Note that for a continuous representation, such as spherical harmonics, a bit more work is required. For SH probes, we can take omnidirectional samples and use these to compute the weights of the basis functions, as described in section 2.3.4.

## 3.7 The radiance cell scattering pass

Once the radiance probes are initialized, we can use them to compute the irradiance of surface points inside a radiance cell. This is done during the *radiance cell scattering* pass. During this pass we march through the radiance grid and process cell by cell. We adopt a hybrid approach where we switch between tracing rays and approximating distant incoming indirect radiance with the help of the radiance probes. The *tracing range*, a configurable parameter, determines which of the two is selected. For each cell, we launch a number of parallel threads equal to the amount of surface points inside that cell. Each thread estimates the indirect irradiance for the surface point it was launched for and does this by sending out a predefined number of rays. This predefined number of rays is the configurable amount of hemisphere samples that we

want to take for indirect lighting (it is important to mention this considering the experiments discussed in chapter 4). If the ray intersects any geometry within the tracing range, we look up the indirect radiance that the hitpoint emits as a reaction to the energy it gathered during the previous lighting bounce. If no intersection is found within the tracing range, we assume the incoming indirect radiance is *distant* and the radiance probes are used to approximate this. To get the correct value from the probe, we project the ray that caused the probe lookup onto the scene's bounding box. This gives us a distant point from which we assume the indirect radiance is coming. We can then do a lookup in the radiance probe's data, provided the direction vector towards that point. This idea is illustrated in Figure 3.7.1. The complete algorithm that each thread executes is listed in Algorithm 3.

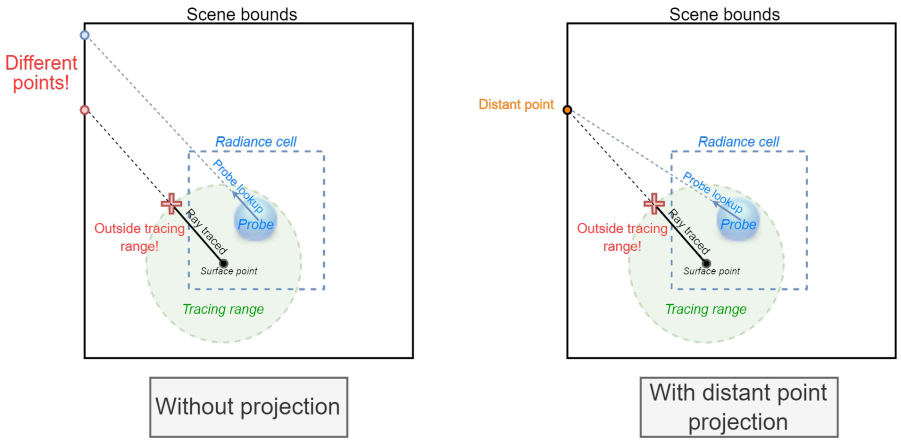



**Figure 3.7.1:** Illustration of the distant point projection algorithm. Projecting a surface point onto the scene's bounding box is necessary to ensure the correct radiance value is sampled from the probe. On the left an example is shown where no distant point projection is done. When the ray encounters no intersection within the tracing range, the most nearby radiance probe is sampled in the exact same direction as the ray. As the figure shows, this causes the wrong point to be sampled. We are estimating the incoming distant radiance along the ray's direction, which corresponds to the radiance coming from the red point on the left. By sampling the radiance probe in the same direction as the ray, we are instead sampling the radiance coming from the blue point on the left (assuming there are no closer intersections). By projecting the surface point for which irradiance is calculated onto the scene's bounding box and sampling the radiance probe in the direction towards the projected point, the ray and radiance probe are "looking towards" the same point and the problem is omitted.

## 3.8 The camera ray pass and tone mapping

The final stage of our proposed rendering algorithm consists of a *camera ray* pass and *tone mapping* pass. To form an image that can be presented to the screen, we send out a primary camera ray for each pixel of the framebuffer. At the first intersection that each camera ray encounters, we look up the outgoing radiance values (product of irradiance at that point and the surface's BRDF) that we have computed during previous passes. Each lighting bounce has its own computed outgoing radiance value, but because irradiance is additive, we can add them all up. This gives us the amount of radiance that arrives at each pixel of the virtual camera.
However, because light transport involves continuous quantities that can take on various values within a wide range, it is best that we work with *high dynamic range* (HDR) images. For floating point images, RGBA tuples are encoded as tuples over the domain $[0; 1]$. As an example, assume we have two colors A and B. Color A has value $(10.0, 10.0, 10.0, 1.0)$ and color B has value $(1000.0, 1000.0, 1000.0, 1.0)$. In a standard floating point image, both of these colors would

**Algorithm 3** Calculation of indirect irradiance at a surface point in a radiance cell. Threads are scheduled in parallel per radiance cell to process all surface points inside that cell. The *t_max* value of a ray limits the range where the intersection routine searches for intersections. *threadIndex* is the index of the launched thread running this procedure.

```
1: radianceCellIndex ← launchParams.currentCellIndex
2: entryIndex ← entriesInsideCell[threadIndex]                ▷ Global index of surf. point
3: entryWorldPos ← worldPosBuffer[entryIndex]                 ▷ World position of surf. point
4: entryWorldNormal ← worldNormalBuffer[entryIndex]           ▷ World normal at surf. point
5: diffuseBRDF ← diffuseColorBuffer[entryIndex]                        ▷ BRDF at surf. point
6: irradAccumulator ← {0.0, 0.0, 0.0}                          ▷ RGB indirect irradiance
7: numSamples ← 0
8: for s ← 0 to NUM_SAMPLES_HEMISPHERE) do   ▷ Nr. of samples for indirect lighting
9:     randomDir ← random direction vector on normal-oriented hemisphere at surface point
10:    Shoot ray into the scene with t_max value of TRACING_RANGE
11:    if distance to first intersection < TRACING_RANGE then
12:        cosineWeight ← ⟨entryWorldNormal, randomDir⟩          ▷ Lambertian cosine weight
13:        incomingRadiance ← indirectRadianceBuffer[hitpoint.index]
14:        irradAccumulator ← irradAccumulator + (cosineWeight * incomingRadiance)
15:    else
16:        distantPoint ← project surface point along randomDir onto scene's bounding box
17:        probe ← launchParams.probes[radianceCellIndex]
18:        probeSampleDir ← distantPoint - probe.pos
19:        incomingRadiance ← sampleProbe(probe, probeSampleDir)
20:        irradAccumulator ← irradAccumulator + (cosineWeight * incomingRadiance)
21:    end if
22: end for
23: irradAccumulator ← irradAccumulator∗diffuseBRDF / NUM_SAMPLES_HEMISPHERE∗π
```

appear as plain white. It should be clear that this is problematic for graphics applications that attempt to simulate the physical reality of light transport, because in reality color B is 100 times more intense than color A! The tone mapping operation solves this by compressing the range of luminance values in the image, while preserving the relative differences in brightness and contrast between different parts of the image. Various tone mapping algorithms exist, but since they are not the focus of this thesis, our implementation uses the simple but effective *Reinhard tone mapping* algorithm [33]. The *Reinhard* tone mapping operator is a non-linear function that is denoted as follows:

$$L_d(x, y) = \frac{L(x, y)}{1 + L(x, y)} \tag{3.1}$$

In equation 3.1, $L_d$ stands for the resulting luminance (the resulting color value for a certain light bundle, represented by a pixel $(x, y)$), whereas $L$ stands for the scaled luminance value for a pixel $(x, y)$. This means that for great luminance values, the equation will converge to 1, whereas for smaller luminance values the equation will converge to 0. In this way, the resulting luminance is guaranteed to be in displayable range. An example result of the camera and tone mapping pass rendered with our method is shown in Figure 3.8.1.

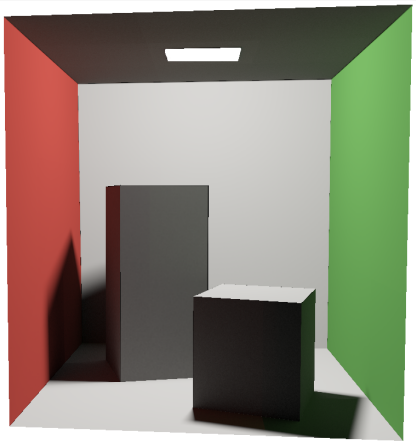


**Figure 3.8.1:** Example rendering of the Cornell box scene that is the result of the camera ray and tone mapping pass. Notice the indirect lighting effects, such as color bleeding on the side panels of the boxes.

## 3.9 Scattering vs. gathering

Because our proposed algorithm is an example of forward path tracing (starting from the light source), our first intuition was to scatter light rays with random directions from the light source into the scene. However, this led to a very noisy image. The reason for this is that when we scatter rays from the light source, we have to shoot an infinite amount of rays to cover all geometry in the scene. Given that the sampling resolution of the scene's irradiance data structure is relatively high, the probability to hit a certain surface point is extraordinary low. Many entries in the irradiance data structure will therefore not be 'filled', and some might be oversampled, resulting in a noisy output. An example of this is presented in Figure 3.9.1. It is instead better to follow a gathering approach when it comes to direct lighting. This is the approach described in section 3.5. By doing this, we make sure that each entry in the irradiance data structure is sampled equally, and all rays that are traced will provide useful information (in the scattering approach it happens that rays are sent out into the void, which is a waste of effort).

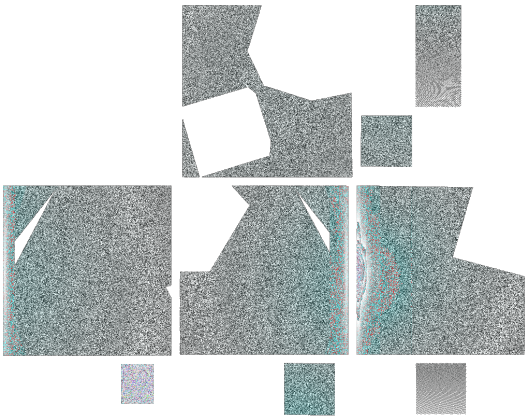


**Figure 3.9.1:** Output of the direct lighting pass that contains a lot of noise, due to the use of the scattering approach. In addition, we observe areas in the irradiance texture that are undersampled, while other areas are oversaturated.

## 3.10 The unbiased approach

Recall that in our previously provided overview of the system the algorithm uses indirect radiance probes as an optimization to set a hard limit on the *tracing range* for which ray tracing operations need to be performed. We could also modify our algorithm and remove this optimization. This would be equivalent to setting the tracing range to infinity which means that we will no longer make use of the radiance probes. This also gives the benefit that it removes the bias that was introduced with the radiance probes. The unbiased approach is described and illustrated in Figure 3.10.1.

The figure exists of 2 parts, with in the left part a theoretical design and in the right part a practical design. The theoretical design represents our initial idea of how to organize the unbiased radiance cell scattering. The idea behind this design was that by tracing indirect radiance per cell by cell and keeping the indirect light source fixed, we could considerably boost coherence. The algorithm would then loop over the indirect light sources and repeat this to account for all indirect radiance. For each 'batch' of rays that is traced, we thus have a pair consisting of a radiance cell and an indirect light source. The envisioned benefit by doing this is that we increase coherence in two ways. Firstly, because we march through the grid cell by cell and handle each cell separately, the rays have a similar origin, which increases the probability that we are writing to coherent memory locations (although this needs to be handled carefully, see section 4.3.4). We will call this coherence in *write space*. Secondly, because marching through the cell grid is done per indirect light source, the rays processed in order have a similar destination. This means that they will read surface data such as reflected radiance and material properties from coherent memory locations. We will call this coherence in *read space*. The theoretically presented algorithm thus has the advantage of being coherent in both write space and read space. However, when we started to implement this we realized that the approach sounds good on paper, but is intractable in practice. This is due to the large amount of indirect light sources that are present in most scenes. With a large amount of indirect light sources, the amount of rays that need to be traced with the approach presented in the left part of the figure rises drastically. In the worst case, where each surface point is an indirect light source, this would lead to a quadratic amount of rays with respect to the amount of surface points. The performance that we would gain from the increased coherence would be greatly outweighed by the amount of work that we would need to do extra compared

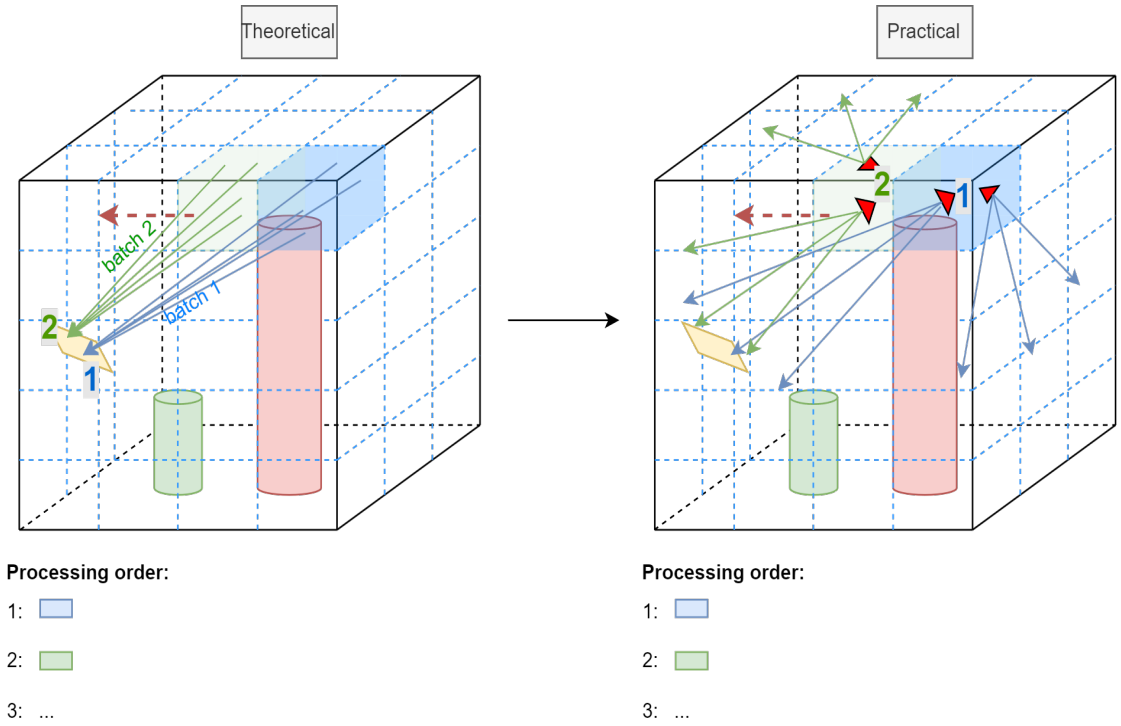



**Figure 3.10.1:** High-level illustration of how the unbiased approach was designed. Note that this figure only describes the transfer of indirect radiance, the direct radiance is still handled by the direct lighting pass, just like in the biased approach. On the left, our initial theoretical design that marches through the grid of radiance cells per indirect light source (shaded in yellow). A batch of rays is formed between the surface points inside a radiance cell and an indirect light source. Rays within a batch are processed in parallel. By processing batch per batch, we trace highly coherent rays in parallel. This theoretical idea sounds promising because it achieves high coherence in both "write space" and "read space" without having to perform extra pre-processing tasks such as sorting the rays. Nevertheless, this concept is intractable in practice because the number of rays that needs to be traced is quadratically proportional to the number of surface points. This forces us to fall back on a more practical approach, presented on the right, where a fixed amount of random samples is taken on the hemisphere of each surface point (in this figure only a few surface points are drawn in red to maintain simplicity). The fixed amount of samples makes the approach tractable, but the directional randomness causes us to lose coherence in read space.

to a traditional path tracer. In practice, we want to take several samples and estimate the irradiance at a surface point using the Monte Carlo estimator instead. This is illustrated in the right part of Figure 3.10.1. We are still calculating irradiance per cell, but are no longer marching over the cell grid per indirect light source. Instead, a fixed amount of samples is taken on the hemisphere of each surface point in the cell. This makes the amount of rays traced scale linearly with respect to the amount of surface points, which is acceptable. The downside of this is that we partially lose coherence. Although the ray origins of the rays being traced are spatially nearby, the rays might have varying directions, which can lead to intersections scattered over the scene. We therefore only stay coherent in write space, but no longer in read space.

The unbiased render pipeline differs from the biased one in 2 ways. Firstly, the radiance probe gathering pass is omitted, because no radiance probes are used in the unbiased approach. Additionally, the radiance cell scattering pass is no longer hybrid (tracing a ray vs. reading from a probe), but always performs ray tracing.

## 3.11 Data structure to store irradiance

In the previous sections, we represented the data structure in which to store the irradiance of the scene's surface points as an abstract container. The underlying representation of this data structure can be arbitrary, but implementation details might change depending on which representation is chosen. In this thesis we experiment with two representations, namely octree textures (section 3.12) and per-object lightmaps (section 3.13). As will become clear in the corresponding sections, each representation has its own benefits and downsides. It should be noted that the representations that we experimented with are not necessarily the most suitable and several other representations, such as *surfels* [16, 30], exist. The representations do not directly contribute to the novelty of our approach, but are the crucial glue that brings all the computations together and allows us to combine them to generate a final image.

## 3.12 Octree textures on GPU [22]

Our original implementation stored irradiance values into a single 2D lightmap texture that represents the whole scene. For a simple example scene like the *Cornell box*, this is a feasible approach. The geometric surface of this scene is very limited in size and not complex at all. However, if we want to render larger scenes, a single lightmap for the entire scene introduces a lot of problems and further complexity. It is necessary to find a technique that can handle complex and possibly large scene geometry in an efficient way without distortion or other artifacts. Given these requirements, the *octree texture* seems to be a good candidate. We included an implementation of GPU octree textures into our renderer, given that a good description of its implementation is available in literature [22].

### 3.12.1 Building the octree

Before we can use the octree to store irradiance values, we need to build it up according to the scene's geometry. The core idea behind the octree is providing more detail (finer granularity) where it matters, that is where the geometry is, while skipping large parts of space where no geometry is present (coarser granularity). Starting from this idea, a straightforward procedure to build the octree is listed in Algorithm 4 and 5. A visualization of a resulting octree is presented in Figure 3.12.1.

**Algorithm 4** Octree construction

```
root = OctreeNode(scene.boundingBox)        ▷ Create root node that encapsulates scene
root→recursiveSplit(0)
```

**Algorithm 5** recursiveSplit(*currentLevel*: int)

```
if currentLevel >= maxDepth then
    return
end if
for o in objects do
    for t in o.triangles do
        if t intersects with this.boundingBox then   ▷ Check if any triangle intersects with octree cell
            Create 8 child nodes by splitting along each axis
            Call recursiveSplit(currentLevel + 1) on each child return
        end if
    end for
end for
```

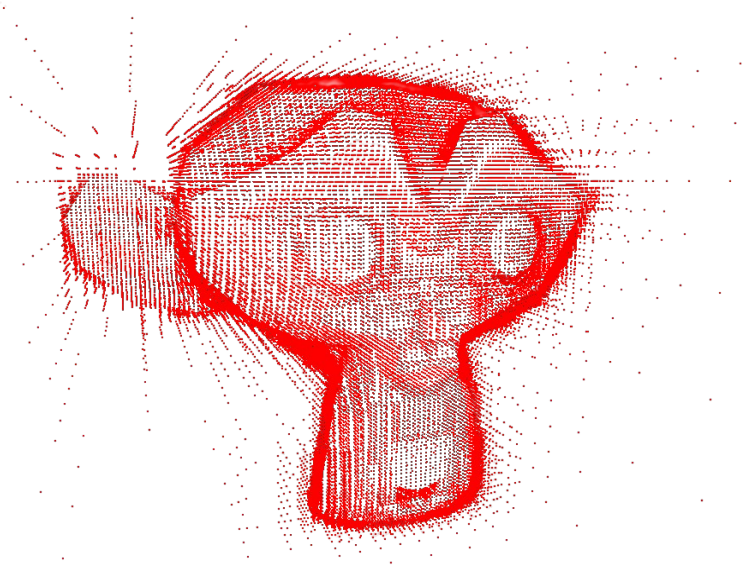


**Figure 3.12.1:** Visualization of octree built for the Blender Monkey model, constructed with Algorithm 4 and 5. Vertices of the octree cells are drawn in red. Note that the spatial areas that contain geometric detail are densely populated with small octree cells, while empty areas contain a sparse collection of larger octree cells.

### 3.12.2 Storing the octree texture on GPU

To make use of the octree data structure on the GPU there is still a problem that needs to be solved. Instead of pointer types that are used in the CPU-based approach, we have access to a large contiguous block of memory in which values can be stored that can be randomly accessed. Therefore, we simulate pointers by storing indices (offsets to the start of the memory block) as values. An octree node either has 8 children, or no children. 8 "slots" are available for each node to hold a (color) value or, in the event that the child will be further subdivided, a pointer to the corresponding child. The 8 slots that represent a node is also known as the *indirection grid*. Pointers to the child nodes can be represented by the aforementioned indices. Since we are working with RGB values to represent the 3 color channels, the octree texture can be represented as a 3D texture of which each "texel" can be addressed by an integer RGB value. Recall that each node features 8 slots, corresponding to 8 texels. Each of these texels contains a RGBA value. The fourth channel (A) is used to distinguish between the types of the node containing the slot. We use the convention that an A value of 1.0 means that the node containing the slot is a leaf node that is non-empty and the slot contains a color value. An A value of 0.5 means that the node containing the slot is an internal node and the slot thus contains a pointer to a specific child of that node. Lastly, an A value of 0.0 means that the node containing the slot is a leaf node, but the slot is still empty (has not been written to yet). Pointers to child nodes are, as mentioned before, represented by the first 3 channels of the RGBA value. As an example, an RGBA value of $(4, 5, 0, 0.5)$ means that this slot belongs to a node that is an internal node, and thus points to one of its children. The data of this child begins at offset $(4, 5, 0)$ into the 3D texture. Offset $(4, 5, 0)$ can then be transformed into a linear index for the memory block by taking into account the amount of indirection grids per row and column of the 3D texture. For our example, assume we store 6 indirection grids per row of the 3D texture, each indirection grid contains 8 slots and each slot contains 4 values (RGBA). The linear index for $(4, 5, 0)$ can then be calculated as follows: $(0) + (5 * 6 * 8 * 4) + (4 * 8 * 4) = 1088$.

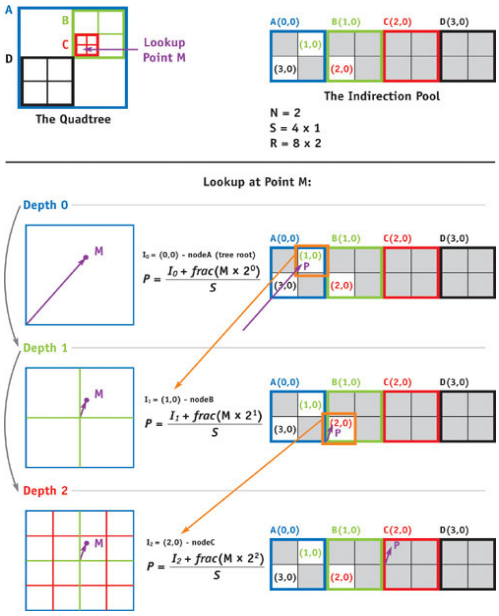



**Figure 3.12.2:** In the top part of the figure, an illustration of the 2D equivalent of the octree (quadtree) texture representation on GPU. An example of how to access the quadtree is worked out in the bottom part of the figure. Figure from [22].

### 3.12.3 Read-write access to the GPU octree texture

The access to the GPU representation of the octree is quite straightforward. Consider a 3D point in space of which we want to look up the color. Assuming the root indirection grid is stored at offset $(0, 0, 0)$, we find this color by following successive child pointers until a leaf node is found. The 3D point's location within the node *octants* (3D equivalent of quadrant) determines which child pointer should be followed at each node. In case of write access, the A-channel value of the slot being written to should also be updated from 0.0 to 1.0 in case the slot was empty.

### 3.12.4 Sampling strategies for the octree texture implementation

Because we are no longer storing the irradiance in a 2D surface parameterization of the scene's mesh (a UV map) but in a volumetric data structure, we need a different approach to take samples in this data structure where we will gather irradiance. Analogous to 2D texture maps, where irradiance is gathered for each texel, we ideally take a sample in each leaf node of the octree. Since the octree leafs are not tightly bound to the mesh surface, unlike 2D texels are, it is unclear where exactly to take the sample within the leaf node. The samples need to be placed exactly on the mesh surface to avoid false ray intersections in the upcoming ray tracing routines. For our implementation we have come up with two approaches. The first one generates samples by looping over all of the scene's objects, looping over each of the object's triangles, and generating uniformly distributed random points in each triangle. The amount of points that are generated per triangle are heuristically determined by the area of the triangle. This method is very simple to implement, however has the disadvantage of being less robust. Depending on the octree's resolution and the geometry's triangle sizes, there is a probability that some octree leaf nodes will not be covered because no samples landed in that area. This results in a noisy output, with black spots (unsampled leaf nodes) distributed throughout the octree's highest level of detail. This problem is illustrated in Figure 3.12.3. An example render containing these artifacts is shown in Figure 3.12.5.

A second method is to generate exactly one sample per octree leaf node. The 3D position

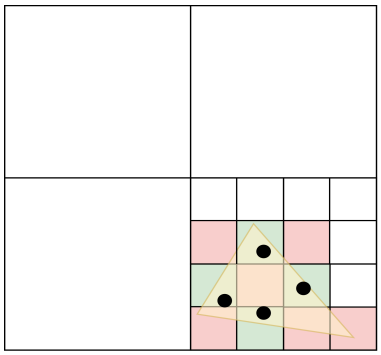


**Figure 3.12.3:** Illustrated example of the unsampled leaf nodes problem that occurs when we generate uniformly distributed random sample points in each triangle. This illustration shows the 2D quadtree equivalent of the problem. One triangle of the triangle mesh is visualized in yellow. The quadtree's quadrant on the right bottom is subdivided by the triangle up to leaf level (the maximal depth is 3 here). On the triangle, four random points are chosen according to a uniform distribution to serve as surface point samples. We observe that the triangle intersects with 10 leaf nodes, but only four samples are taken. In the case of this example, only four octree leafs will be filled with an irradiance value (shaded in green), while six stay empty (shaded in red). This results in a noisy output.

of this sample within the leaf node is then found by considering the triangle that intersects with the leaf node. The center of the leaf node can be projected onto this triangle along its normal. Note that this requires an intersection with exactly one triangle for each leaf node. In practice this is usually impossible, because along object edges it is impossible to divide the adjacent triangles between two leaf nodes, unless the object edge is perfectly aligned with one of the three axes that define the dimensions of the octree. This idea is presented in Figure 3.12.4. A simple workaround for this is to randomly choose one of the triangles to project the point onto. Another consequence of this approach is that the depth of the octree should be large enough so that each leaf only intersects with at most one triangle (except at object edges). To accurately capture all the detail in the scene this is a minimal requirement anyways, but in a situation when available memory or construction time is limited, it can be desirable to limit the resolution of the octree. Multiple triangles combined with taking one random sample per leaf node would lead to unexpected sudden color changes. This can be mitigated by taking multiple samples and averaging them out, very similar to how mipmapping works for an octree texture. Triangles that intersect each other could also lead to artifacts (e.g. consider when the sample point would be taken on the secant), but we can assume that the majority of graphics applications do not include these.

### 3.12.5 Examples and limitations

Although octree textures form a robust alternative to store irradiance values, they introduce several problems that need to be addressed. A first one is texture filtering. In order to reduce the visual impact of aliasing artifacts, texture filtering is an important element to consider when sampling from textures. Texture filtering for standard 2D and 3D textures is hardware accelerated on consumer-grade graphics hardware. Sparse GPU octree textures do not have hardware support for texture filtering, due to the manner in which they are stored. Although it is possible to filter octree textures via the means of trilinear interpolation, this needs to be implemented in software and therefore introduces significant overhead. To enable trilinear interpolation, 8 samples need to be taken, which corresponds to 8 octree lookups. Given our whole algorithm is trying to reduce overhead by increasing memory coherence, this is not an attractive characteristic. Another thing to mention is the construction time of the octree. In

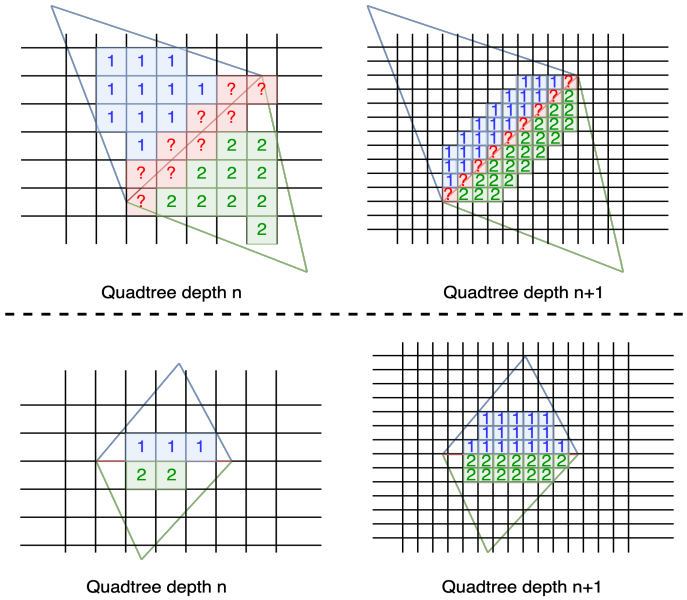



**Figure 3.12.4:** Figure to illustrate that an infinite quadtree (idem for octrees) resolution is required at triangle edges to make sure each cell only contains one triangle. This is illustrated in the top row of the figure. No matter how high the resolution of the cell grid is, there will always be cells that intersect both triangles. These are marked in red in the figure. An exception on this rule is when the edge between two triangles is exactly aligned with the cell grid. This is illustrated in the bottom row of the figure.

our implementation we have provided a version of Algorithm 5 on CPU. For scenes with a larger geometric complexity this is extremely slow, and for each depth level we add to the octree, the algorithm run time increases with a factor 8 in the worst case. To put this into perspective, the octree used to render the scene in Figure 3.12.5 took over 8 hours to construct. It is clear that the resolution of this octree is still by far too low to produce a photorealistic scene. Taking into consideration that this is an octree of only depth 9, and ideally an octree of at least depth 13 is required to provide sufficient detail for this scene, it is safe to say that the CPU algorithm is too slow. Even if we assume a large part of the scene stays static so we only have to compute the octree once and can then store it in a file, it would still take days to generate the file with the current approach. After this observation, we have 2 valid options to choose from. One of them is to find a largely improved octree texture construction algorithm, implemented on GPU. The second option is to fall back to 2D texture mapping, which implies that we should find a workaround for larger scenes. Given the other limitations that octree textures introduce, we have decided to implement the second option.

## 3.13 Parameterizing 3D mesh surfaces in 2D via lightmapping

In order to render more representative scenes using the 2D texture mapping approach, finding a unique mapping for each 3D point on the surface mesh in 2D space is required. This can be achieved via the approach of *lightmapping* [5]. Lightmapping is directly available in commercial render engines such as *Unity Game Engine* and *Blender*. For this thesis, Blender was used to generate unique UVs for the scene's surfaces. Our first attempt was to pass the mesh file of the full scene into the lightmap packing tool. Although this theoretically fulfills the requirement of generating a unique mapping in 2D space, the problem with this approach is that we are trying to map a very large surface space into one UV map, which leads to many surface areas being mapped into a miniscule 2D area. This not only creates many 'gaps' (points that lie

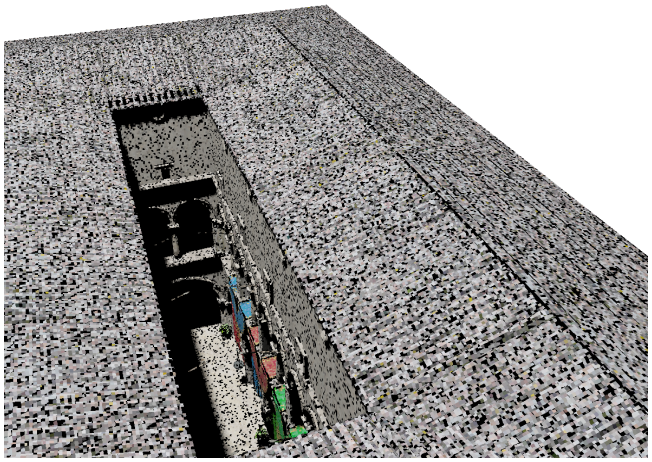


**Figure 3.12.5:** An example render of the Crytek Sponza scene that uses the rendering algorithm presented in this thesis and GPU octree textures to store irradiance. The octree used here has a maximum depth of 9. Only direct illumination was enabled for this render. Notice the "pepper noise", caused by octree leaf nodes that did not get a sample due to the random sampling of the triangles.

close to each other in 3D surface space but are distant in 2D texture space), but also requires a very high texture resolution to capture sufficient detail for each triangle in the surface mesh. This is inconvenient for two reasons, the first of which is that more gaps in 2D space makes the render process making use of this texture more susceptible to *texture seam* artifacts (see section 3.15.1). The second reason is the extremely high memory requirements introduced by the fine granularity of unique mappings in the single UV map.

To overcome this, we divide the scene's mesh file into objects and generate a unique mapping in 2D space for each object separately. In this way, each object receives its own 2D irradiance map, leading to coarser granularity of the mapping and less gaps in most cases. An example of such an unwrapping is presented in Figure 3.13.1. Depending on the geometric complexity of the object, the required resolution of the irradiance map might still be relatively high. Another thing to take into consideration is the surface area of the mesh surface. For large surface areas, it is required to have an equally matched irradiance texture resolution. This is not the case for diffuse texture maps that are often used in games, because those can make use of *texture repeating* (e.g. GL_REPEAT in OpenGL). For irradiance textures there is a unique 2D point for each surface point, meaning that the larger the surface area is, the larger the area in 2D space is. If we downsample the surface's mapped area in 2D space this results in the loss detail, which leads to a 'blocky' appearance of lighting effects. The unique 2D mapping will not only be used for the storage of irradiance, but also geometrical surface point data such as world positions and world normals, necessary for the light transport calculations. This geometrical data can be thought of as a global equivalent for *G-buffers*.

## 3.14 NVIDIA OptiX

Our entire implementation is built on top of the NVIDIA OptiX ray tracing SDK. OptiX is implemented in CUDA, which is a parallel programming model and computing platform for general purpose GPU computing. CUDA employs an architecture called *Single-Instruction, Multiple-Thread (SIMT)*, which allows massive parallelism between large amounts of threads executing the same instructions. SIMT enables us as a programmer to write thread-level code that should be executed in parallel for each scheduled thread. This section will briefly discuss the workflow that OptiX uses, together with its features.

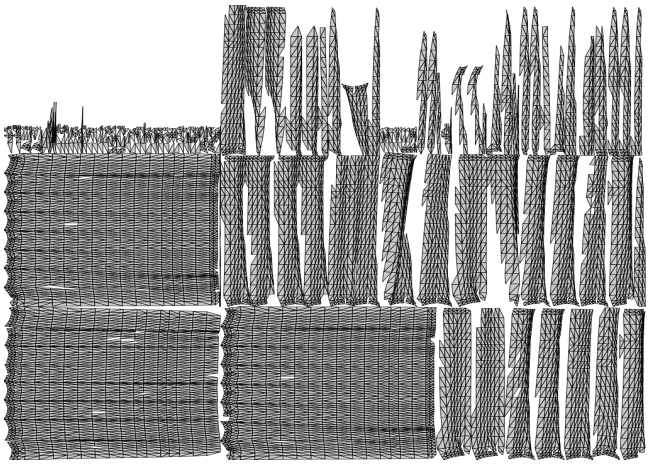


**Figure 3.13.1:** Lightmap UV-space of the curtains in the Crytek Sponza scene, visualized in Blender. Each point on the object's 3D surface gets a unique mapping in 2D UV-space.

### 3.14.1 Pipeline architecture

OptiX implements a single-ray programming model, meaning we have to define the pipeline of programs that will be traversed for each ray. The ray-tracing pipeline in OptiX is governed by a collection of 8 shader programs, that have to be implemented by the programmer (some are optional however). The most important of these eight are the *ray generation* program, the *closest-hit* program, the *any-hit* program and the *miss* program. The relationships and flow between the programs are presented in Figure 3.14.1. The execution of a pipeline can be started in host code with the call *optixLaunch*, which takes several arguments, such as which pipeline to launch and the amount of threads that should be executed. Data required for computations can be passed via *launch parameters*. Large blocks of data should first be copied to device memory, pointers to this memory are then passed to the launch parameters. We now provide a brief description for each of the programs presented in Figure 3.14.1:

1. *Ray generation*: The entry point to the ray tracing pipeline. This is invoked for each thread that is launched with the *optixLaunch* call. Blocks of threads are scheduled onto streaming multiprocessors (SMs), each of which executes warps of threads in parallel. This program sends out rays into the scene that will be tested for intersections based on a given origin and direction.

2. *Any-hit*: This program is called whenever an intersection is found.

3. *Closest-hit*: This program is called when the first intersection along the ray is found.

4. *Miss*: This program is called when no intersections are found for this ray.

5. *Direct callable*: Similar to a normal callable CUDA function. Should be named with prefix *__direct_callable__* and invoked via their index in the shader binding table. Direct callables are called immediately, but cannot make a call to *optixTrace*, which corresponds to sending out a recursive ray.

6. *Continuation callable*: Variant of direct callable that can make calls to *optixTrace*. This introduces the additional overhead that continuation callables need to be scheduled before execution. Should be named with prefix *__continuation_callable__* and invoked via their index in the shader binding table. An example where continuation callables are used are functions where recursive rays need to be traced, e.g. a closest-hit shader for primary rays that sends out recursive shadow rays to compute direct lighting.

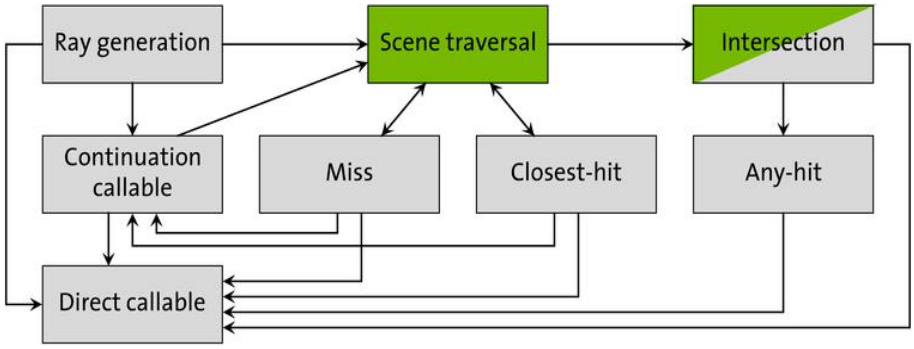



**Figure 3.14.1:** Flowchart showing the relationships between the shader programs that implement the ray tracing pipeline of OptiX. Green nodes represent fixed-function hardware-accelerated operations, while gray nodes represent user programs.

### 3.14.2 Shader binding tables

The *shader binding table (SBT)* is a data structure containing information about the location and parameters of shader programs. The SBT needs to be built per pipeline and is stored in device memory. Elements of the SBT data structure are called *records*. A record consists of a header and a data block. The header content is non-customizable. The user passes an *OptixProgramGroup* to the API function *optixSbtRecordPackHeader*, and given this information the function will build up the header that contains the necessary data to identify and invoke shader programs during ray-scene traversal. The data block can contain arbitrary data that is accessible in the shader programs. For example, for a *hitgroup record* (for closest-hit, any-hit shaders), the data block can contain the associated primitive's texture coordinates or diffuse texture colors. The SBT records need to be built on the host and uploaded to device memory before making a call to *optixLaunch*. Different SBT records can be assigned to different geometric primitives in the scene, based on their material properties for example.

### 3.14.3 Acceleration structures

An *acceleration structure* (AS) is a data structure used to accelerate the search for intersections between rays and scene geometry. It is based on the concept that when a ray intersects with a geometrical volume, it must also intersect with every possible *bounding volume* of that volume. This means that to find an intersection between a ray and an object, we can first check whether the ray intersects with a bounding volume of the object. The key is that the geometry of these bounding volumes is much simpler, therefore allowing much quicker intersection tests. If the ray misses the bounding volume, all objects inside that volume can be culled. This is much more efficient than the brute-force approach of looping over all triangles in the scene and testing for an intersection with each of them. There are several types of acceleration structures, but the most well-known are probably *Bounding Volume Hierarchies (BVHs)* [26] and *KD-Trees* [11]. NVIDIA OptiX uses BVHs and hides all implementation details from the user, instead providing an API that allows the user to define the geometry for which to build an AS (via *OptixBuildInput* structs). OptiX then internally handles GPU-accelerated building, compression and traversal of the AS.

OptiX makes a distinction between two types of acceleration structures: *instance acceleration structures (IASes)* and *geometry acceleration structures (GASes)*. A GAS is an abstraction for the actual acceleration structure that contains the geometry. It allows to perform hardware-accelerated intersection tests and optimizes the representation of the underlying geometry. In contrast, an IAS allows us to efficiently traverse and intersect rays with complex scenes consisting of multiple instances. It references to a GAS object, and can then represent multiple instances of this geometry by keeping track of transformation matrices. Briefly said, IASes are

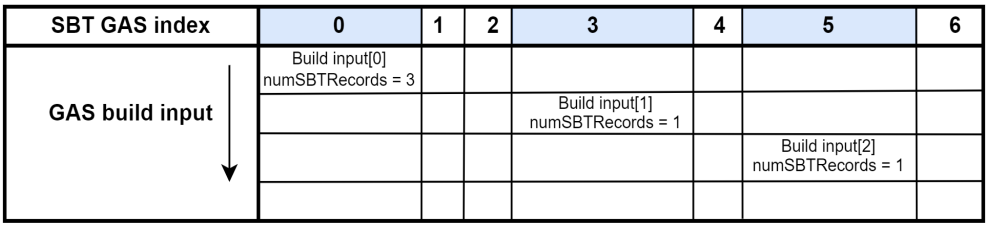



**Figure 3.14.2:** Example of "Sbt-geometry-acceleration-structure-index" assignments for 3 given build inputs. The indexing starts from zero, incrementing with the amount of SBT records that each build input references. The corresponding SBT GAS index for each build input is shaded in blue.

used to organize the scene hierarchy and geometry instances, while GASes represent individual geometric objects and provide accelerated and optimized ray tracing calculations for these objects.

Which shader program is executed at a ray-geometry intersection is determined by the following formula: *sbt-index = sbt-instance-offset + (sbt-geometry-acceleration-structure-index + sbt-offset-from-trace-call)*. *sbt-instance-offset* is only applicable to IASes that inherently contain multiple GASes. For scene traversables that exist of one single GAS *sbt-instance-offset* is zero. *Sbt-geometry-acceleration-structure-index* is an index assigned per *build input* and stands for the index of the first SBT record that belongs to that build input. For example, take a GAS that exists of 3 build inputs. Build input 1 has three SBT records associated with itself (this can be the case when the geometry described by this build input exists of 3 different materials, for example). Build input 2 and 3 both reference two SBT records. Then the *sbt-geometry-acceleration-structure-index* for build input 1 is 0, the index for build input 2 is 3, and the index for build input 3 is 5. The SBT GAS indexing is visually shown in Figure 3.14.2. Finally, the *sbt-offset-from-trace-call* can be set when casting a ray into the scene via the *optixTrace* call. This finer grained offset can be used to distinguish between different ray types.

## 3.15 Limitations

### 3.15.1 Artifacts

A disadvantage of storing irradiance in 2D textures are texture seam artifacts that are introduced, as was previously mentioned. The amount of seam artifacts that appear heavily depends on the quality of the 3D to 2D parameterization. The likelihood of seam artifacts decreases as the mapping becomes more tightly woven (less cuts). An example of seam artifacts rendered with our approach is shown in Figure 3.15.1. Seams are caused when the 3D surface space is 'cut up' in 2D texture space. When texture filtering is applied, non-textured background areas of the texture map will be taken into account on the edges of these cuts. This causes dark seams to appear on the cutting line. In most practical cases it is inevitable that cuts will be introduced when mapping a 3D surface to 2D, but minimizing the amount of cuts helps to reduce seam artifacts. Furthermore, techniques to suppress these artifacts exist, e.g. an extrapolation technique named *pull-push* [34]. Because the main purpose of this thesis is to verify the coherence of the proposed light transport algorithm and whether it is suitable for multi-user scenarios, these techniques have not been included in our baseline implementation.

### 3.15.2 Memory usage

Another disadvantage is the large memory usage of the approach. Besides irradiance, world positions, normals and diffuse texture coordinates also have to be stored into textures. Because we are working with physical quantities that are part of a large continuous space, it is necessary

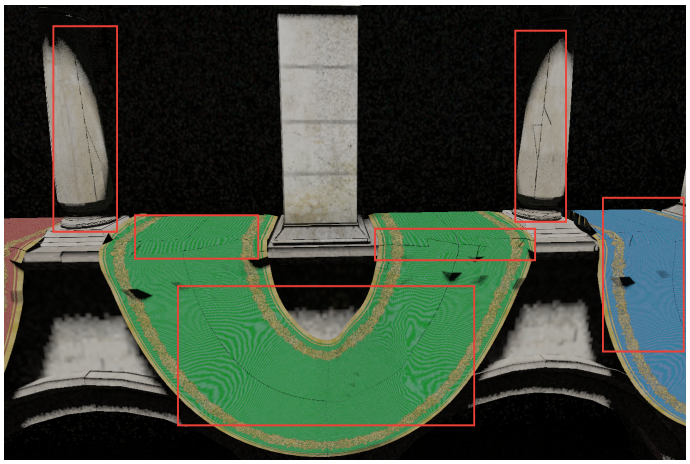


**Figure 3.15.1:** Examples of texture seam artifacts, visible as thin black lines on the textured geometry.

that we make use of 32-bit HDR textures. Without taking compression into account, this means that we use 128 bits per texel. Given that high-resolution textures are required for large surfaces, the memory usage rises drastically. However, because we have separate textures per object, the resolutions can be set in a more flexible manner. Furthermore, for indirect lighting bounces it is not necessary to sample at the same high frequency as direct lighting (see section 4.7.2). An alternative optimization that ensures a limit on memory usage is proposed in section 4.7.1.

### 3.15.3 No support for specular effects

Our implementation solely focuses on the support for diffuse global illumination, for two reasons. The first reason is that diffuse reflections are the worst-case scenario when it comes to the spatial incoherence of light rays. Belief was held that improving coherence here would yield the most significant impact. A second reason is that diffuse global illumination is viewpoint-independent. In other words, the diffusely reflected radiance at a surface point is perceived identically in every direction. Therefore, it is the ideal candidate to be computed on a centralized system and shared among many users in multi-user applications. In contrast, specular global illumination effects depend on the user's viewing direction. This makes it harder to compute specular global illumination for all users in the scene and cache it so it can be reused. It is a task that is exhaustive both in terms of memory as well as processing resources. A possible solution could be to use a dense grid of radiance probes to gather specular indirect radiance and interpolate between probes. However, for perfectly specular materials this approach is still problematic. In fact, for multi-user cloud gaming environments it could be more beneficial to follow a hybrid approach where specular global illumination is path traced on client side, in case the client's hardware is capable of doing so. Otherwise, approximations such as *screen-space reflections (SSR)* [1] and/or reflection maps [32] can serve as alternative solutions.

# Chapter 4

# Implementation evaluation and analysis

## 4.1 Experimental setup

All of our experiments were conducted on a system with the following specifications:

- CPU: Intel i7-12700K @ 3.61 GHz
- Main memory: 32GB DDR5 @ 6400 MHz
- GPU: NVIDIA RTX 3090 (24GB VRAM)

## 4.2 Additional notes on the experiments

Before discussing the experimental data described in this section, there are several things worth noting. Unless mentioned differently, the baseline implementation (without optimizations) that is described in section 3 was used in all experiments. The scene is subdivided into objects that each have their own irradiance map to store the irradiance per lighting bounce. All experiments were performed on two test scenes: *Crytek Sponza* (ca. 260K triangles) and *Rungholt* (ca. 4M triangles). The *Crytek Sponza* scene was subdivided into several objects, each having their own respective irradiance texture size. These objects and their respective texture sizes are listed in Table 4.1. The *Rungholt* scene was subdivided into 15 equal-sized tiles, each with a texture resolution of (4096 x 4096). The same texture resolutions were used for all lighting bounces. We disregard the additional run time of a denoising algorithm in all experiments, as this is irrelevant to the comparisons we make.

To make a comparative analysis with the traditional path tracing approach, we additionally implemented a reference Monte Carlo path tracer on GPU that accounts for diffuse global illumination. The reference implementation was also built on top of NVIDIA OptiX. It renders frames at a resolution of 1920x1080. In all experiments, we have traced 1 direct lighting bounce and 2 indirect lighting bounces for both the reference implementation and our proposed algorithm.

## 4.3 Parameter impact

### 4.3.1 Radiance cell size

A first interesting parameter to investigate is the size of the radiance cells. Assuming the scene is normalized, the radiance cell size determines in how many radiance cells the scene is subdivided.

| Object name | Texture resolution |
|---|---|
| *arcs* | 2048 x 2048 |
| *ceiling* | 2048 x 2048 |
| *walls* | 4096 x 4096 |
| *thorns* | 2048 x 2048 |
| *flags* | 2048 x 2048 |
| *fire pit* | 1024 x 1024 |
| *curtains* | 2048 x 2048 |
| *doors* | 1024 x 1024 |
| *lion* | 1024 x 1024 |
| *lion backplate* | 1024 x 1024 |
| *pillars* | 1024 x 1024 |
| *upper pillars* | 2048 x 2048 |
| *upper pillars 2* | 1024 x 1024 |
| *plants* | 1024 x 1024 |
| *floor* | 2048 x 2048 |
| *pots* | 1024 x 1024 |
| *roof* | 4096 x 4096 |
| *spears* | 2048 x 2048 |
| *chains* | 256 x 256 |
| *side panel* | 256 x 256 |
| *hanging vase* | 1024 x 1024 |

**Table 4.1:** Objects and their respective texture resolutions for the Crytek Sponza scene.

It represents the length of the sides of the cube volume that forms a radiance cell. For example, a radiance cell size of 0.5 implies that the scene will be subdivided into 2 x 2 x 2 cells. The simple formula $ceil(\frac{1.0}{radianceCellSize})$ gives the amount of radiance cells in each dimension. To determine the impact of this parameter, we keep all other scene variables constant (8 stratified light samples for direct lighting, 10 hemisphere samples for indirect lighting, tracing range of 0.3 for the biased approach). For this experiment we have averaged out the total algorithm run time over 100 runs for each cell size that we tested. We have made measurements for radiance grid sizes including and between 1 x 1 x 1 and 7 x 7 x 7, for both the biased approach (using radiance probes) and the unbiased approach. The measurements for the *Crytek Sponza* scene are summarized in Figure 4.3.1. For the *Rungholt* scene very similar results were observed. These results are shown in Figure 4.3.2 for the purpose of completeness.

It becomes immediately clear that the impact of the radiance cell size on the total algorithm run time seems rather limited, for both approaches. The average algorithm run time decreases very slightly with smaller cell size (a speedup of about 50 ms), up until the size of 0.25 (grid of 4 x 4 x 4), after which it starts increasing again. To recall what the concept of a radiance cell was used for, it divides the scene into independent spatially local batches. The algorithm marches through the grid of cells, processing cell by cell separately. Our hypothesis was that this would improve spatial coherence of the rays being traced. The results in the graph show that the algorithm run time only slightly improves with smaller radiance cell size (up to a certain point). Why is the impact not greater? First, it is worth mentioning that the direct lighting pass of the algorithm does not make use of the radiance cells, so the impact on algorithm run time is solely decided by the radiance probe gathering and radiance cell scattering passes. The radiance cell size decides the dimensionality of the radiance grid and therefore also decides how many radiance probes are placed into the scene (in our implementation we place one radiance probe in the center of each cell). For the biased approach, this means that by decreasing the radiance cell size, the amount of work for the radiance probe gathering pass increases. The extra work introduced by the radiance probe gathering pass also explains the gap in algorithm

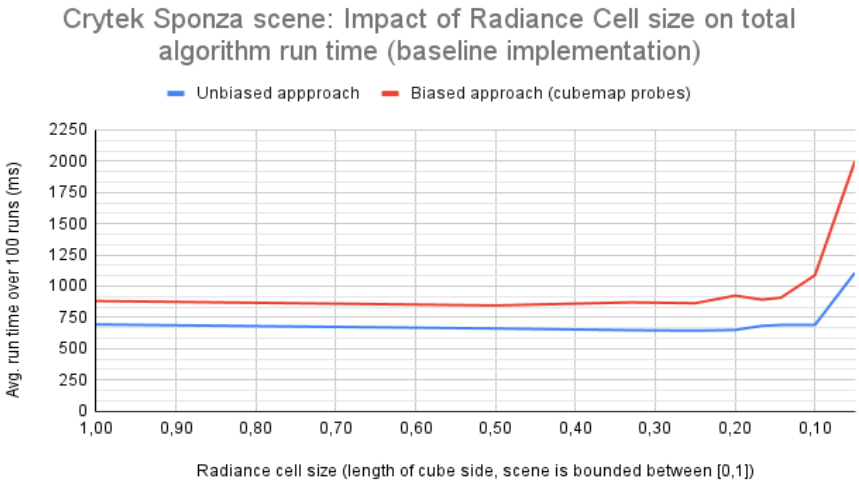



**Figure 4.3.1:** Graph showing the impact of the radiance cell size on average algorithm run time over 100 runs for the Crytek Sponza scene. 8 light samples were taken at each surface point for direct lighting, together with 10 hemisphere samples for indirect lighting. The tracing range was kept constant at 0.3. The scene is normalized, so a radiance cell size of 0.1 corresponds to a radiance grid of 10 x 10 x 10. We performed tests for both the unbiased and biased approaches, both of which show similar behavior. As we decrease the cell size up to 0.25 (4 x 4 x 4 grid), we see that the average run time slightly benefits from this (about a 50 ms improvement). Further decreases in cell size lead to a severe performance loss however, suggesting this is caused by the additional overhead introduced by a larger radiance grid.

run time between the biased and unbiased approach. For the radiance cell scattering pass, in both the biased and unbiased approach, the radiance cell serves as an origin for the ray. Samples are taken on the hemisphere during this pass, so unless a limiting tracing range is used like in the biased approach, we cannot fully assure coherence. However, with smaller cell sizes, we expect that the rays that are processed per cell have similar origins, and would therefore demonstrate more coherent behavior (similar to primary rays). The small increase in run time performance up to a certain size could possibly be explained by this, but more metrics such as cache hit ratios are needed to confirm an increase in ray coherence. Furthermore, given the small impact it seems like the advantages gained from this higher coherence are countered by overhead that is introduced by the larger radiance grid. To make this concrete, a separate CUDA kernel needs to be started for each cell in the grid. Launching and returning from these kernels introduces additional overhead, depending on several parameters such as the size of the kernel. Decreasing the cell size also decreases the number of threads that are launched per cell (each cell now contains less geometry), which negatively impacts performance when the GPU's occupancy becomes too low. For these reasons combined, when the radiance grid becomes too large (larger than 4 x 4 x 4), the additional overhead appears to start outweighing the small benefits that were gained.

### 4.3.2 Tracing range

A second configurable parameter is the tracing range threshold. The tracing range is only relevant for the biased approach, since the unbiased approach essentially uses an infinite tracing range. It governs whether incoming radiance at a surface point is determined using ray tracing or whether it is approximated by a radiance probe. Furthermore, it serves as a hard limit of spatial data that can be accessed. Our hypothesis is that this hard limit leads to an increase of coherence, because it guarantees that no data lookups will be done for data that falls outside of the tracing range. For this experiment we keep all other scene parameters constant. We

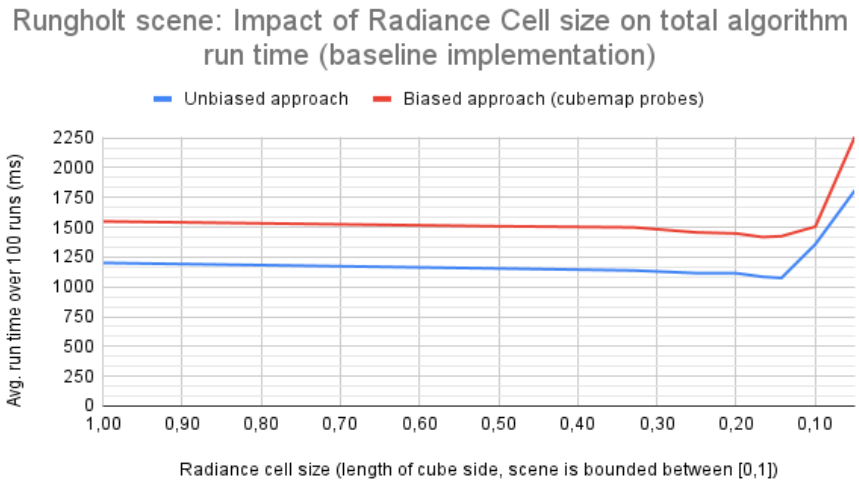



**Figure 4.3.2:** Similar trends as in Figure 4.3.1 can be observed for the Rungholt scene.

used a radiance cell size of 0.5 and took 8 light samples for direct lighting and 10 hemisphere samples for indirect lighting at all surface points. For each test, we have averaged out the total algorithm run time over 100 runs. We observe that the tracing range initially does not have a lot of impact, up until a tracing range of 0.4. A plausible explanation for this is that for larger tracing ranges, the majority of intersections are found within the tracing range. For these large tracing ranges, there is still a large chance to make incoherent accesses. For example, a tracing range of 0.5 still allows us to use ray tracing across half the scene. A lot of incoherent data can be encountered within this range. For smaller tracing ranges, below 0.1, the impact is more significant. We also went to the extreme where we set the tracing range to 0, which essentially corresponds to only making use of light probes. Decreasing the tracing range below 0.1 leads to a total decrease in algorithm run time between 150 - 200 ms. Again, similar trends for both test scenes were observed. The results are presented in Figure 4.3.3 and Figure 4.3.4.

Another interesting performance indicator to look at is the cache hit ratio at both L1 and L2 level. This gives us a good idea about the coherence of the memory accesses that were made throughout the render process. We have made a comparison for the hit ratios of L1 and L2 cache memory between the biased approach using a tracing range of 1.0 (making no use of radiance probes) and a tracing range of 0.0 (only making use of radiance probes). The results are listed in Table 4.2.

It is important to note that by making use of the tracing range we are switching to a biased approach. This biased approach will use radiance probes to make approximations based on the assumption that all radiance incoming from outside the tracing range can be considered as distant lighting. Using a biased approach has the disadvantage that it is no longer physically accurate and can therefore also introduce artifacts. For this experiment we have noticed dark leak artifacts that start appearing once the tracing range is set below 0.10. This is illustrated in Figure 4.3.5. Dark leaks are caused by radiance probes falling inside geometry, therefore not catching any radiance. A solution to this problem is a more clever strategy for probe placement, however this is regarded as future work.

### 4.3.3 Accurate distant point projection vs. approximation

During the analysis of our implementation, we observed that for the biased approach the distant point projection that is required to determine the direction in which to sample the radiance probe is causing overhead. This is evidenced by Figure 4.3.6. We see that a large amount of

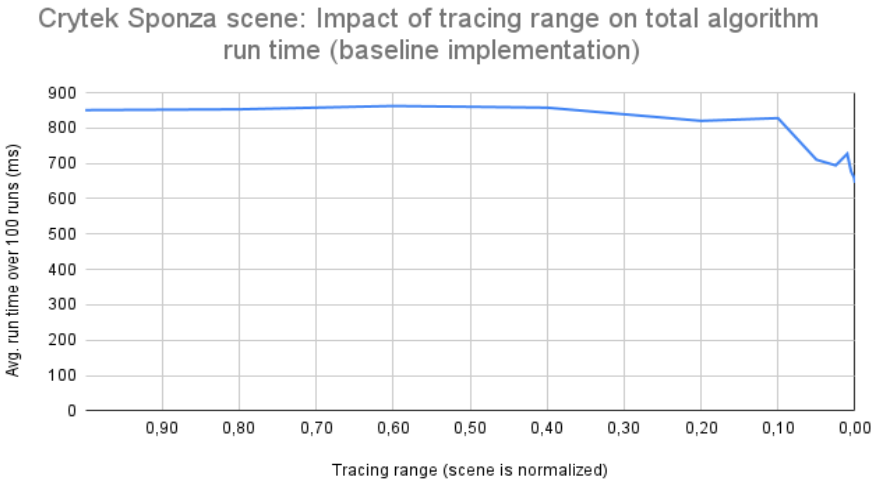



**Figure 4.3.3:** Graph showing the impact of the tracing range on average algorithm run time over 100 runs for the Crytek Sponza scene. 8 light samples were taken at each surface point for direct lighting, together with 10 hemisphere samples for indirect lighting. The radiance cell size was kept constant at 0.5. The scene is normalized, so a tracing range of 0.1 corresponds to a radius with length equal to $\frac{1}{10}$ of the scene's size.

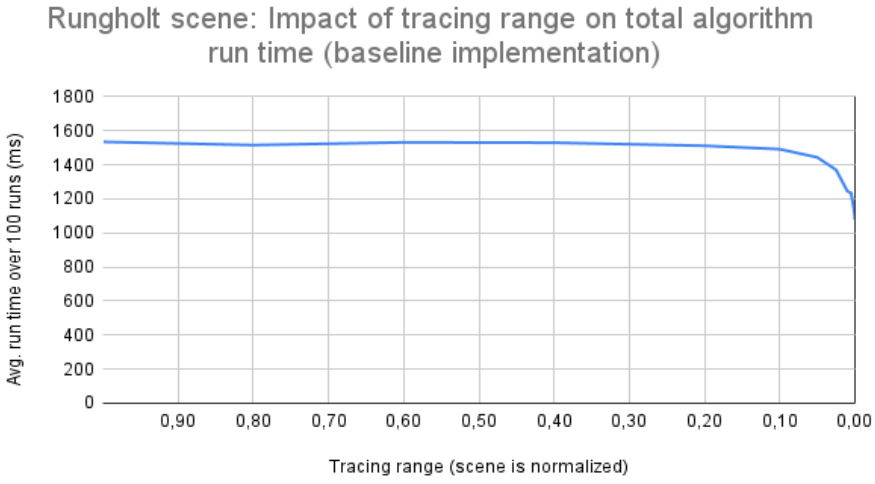



**Figure 4.3.4:** Similar trends as in Figure 4.3.3 can be observed for the Rungholt scene.

| Impact of tracing range on cache hit ratios (Crytek Sponza) | | |
|---|---|---|
| | L1 hit ratio (Avg. %) | L2 hit ratio (Avg. %) |
| Tracing range 1.0 | 62.83 | 71.71 |
| Tracing range 0.0 | 60.69 | 87.79 |

**Table 4.2:** Cache hit ratios of L1 and L2 caches averaged over all render passes, measured with Nsight Compute as an indicator of coherence. We observe that a lower tracing range has a positive impact on the L2 cache hit ratio, which confirms our hypothesis. A lower tracing range imposes a hard limit on the incoherence in read space (everything outside this range is approximated by the radiance probe belonging to the same radiance cell). At L1 level there is less noticeable impact. The L1 cache is limited to only 128 kB per SM. It is therefore mainly used for caching local memory when register spillovers happen, as well as instructions that are frequently needed by the SM. It can also cache frequently used texture data, but due to its limited capacity it is less suitable for larger texture data.

**(a)** **(b)**

**Figure 4.3.5:** Dark leak artifacts appearing in renders of the Crytek Sponza scene (8 samples for direct lighting, 10 samples for indirect lighting) using the biased version of our algorithm, with a tracing range set to 0.025.

warps that should ideally run in parallel are stalled after the function call. This is because the function introduces a severe amount of branching, which is in violation with the SIMT execution model. To resolve this, we experimented with an approximation to the distant projection. The distant projection projects the surface point along the sampled ray's direction onto the scene's bounding box. In contrast to this, the approximation instead just projects the surface point along the sample ray's direction over a distance of 1.0. We found that this gives results that are visually similar and solved the problem of warps being stalled. This minor optimization resulted in a maximal speedup of 46 ms (at a tracing range of 0.0) on the total algorithm's run time.

### 4.3.4 Calculating indirect radiance per object vs. per cell

While experimenting we found that rather large memory transfers were being made between the L1 and L2 caches. We decided to further investigate this and noticed a 'flaw' in our current architecture's design. The approach that we currently follow works per cell when it calculates the indirect radiance during the radiance cell scattering pass. This design originated from the idea that rays with a spatially proximate origin tend to be more coherent. In addition it is a flexible manner to divide the workload in the scene. However, it is possible that inside a

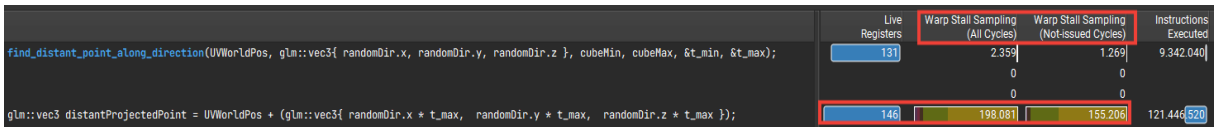



**Figure 4.3.6:** Screenshot of Nsight Compute. We see a significant amount of warp stalls after the distant point projection function, leading to a sub-optimal performance. The large number of warp stalls is caused by a major amount of branching within the function.

cell many different objects are present. Recall that our irradiance data structure consists of a collection of textures, one for each object. This means that although the rays themselves are more spatially coherent, the memory writes that we make could be much less coherent, because we store irradiance in a texture per object. This is visualized in the left part of Figure 4.3.7. Instead we could also gather indirect radiance per object. We would then run a kernel per object which launches a thread for each sampled surface point on the object. This essentially means we are launching a thread for each texel of the texture in which the irradiance for the corresponding object is stored. All concurrent threads are therefore writing to the same texture, which leads to more write-space coherence. However, one should keep in mind that for larger objects, the rays being traced concurrently from the surface points may now no longer have spatially proximate origins, and might therefore make less coherent accesses when intersecting with other geometry. The per-object approach is illustrated in the right part of Figure 4.3.7

We have also verified this theoretical explanation experimentally. After analysis with Nsight Compute we observed that for the per-object approach, the average cache hit ratio is increased by 3.5% for L2 cache and 3% for L1 cache. This is nothing too significant, and as we have mentioned before, by calculating indirect radiance per object the spatial coherence of concurrent rays may be decreased, in particular for larger objects. What is more significant however is the difference in memory traffic between the two caches. For the per-cell approach, the memory traffic between caches is about 10 to 15 times higher than for the per-object approach. This is presented in Figure 4.3.8. When we compared the algorithm run time for both approaches there was little to no difference, which might come as a surprise, given the significant difference in memory traffic. The reason for this is that the setup we used for the experiments supports a very high memory bandwidth (936.2 GB/s). For systems with lower bandwidth or scenes that require more bandwidth, we do expect to see a difference in performance.

In conclusion, switching to a per-object approach did not change the performance of our implementation for this experimental setup, but nevertheless it is important to document these findings because it can have a significant impact on other configurations. Furthermore, which approach is ideal to use depends on the scene, amount and size of objects, type of data structure in which irradiance is stored, and is likely a trade-off that should be made. Because we are storing irradiance in textures, the per-object approach might be more suitable to reduce memory traffic and increase coherence for memory write operations. In contrast, if we would use another irradiance data structure, such as surfels, the per-cell approach can be better. Briefly said, it depends on how the irradiance data is stored in memory.

## 4.4 Comparison with reference implementation

In this section we will compare the performance of our algorithm to the reference path tracer. It is likely that our algorithm in its totality will run slower than the reference implementation. This is because the standard path tracer just traces paths that contribute to a single viewpoint, while our algorithm calculates the irradiance for the whole scene. In other words, our algorithm performs more work than a reference path tracer, but with the advantage that the result obtained from this work can be used to generate any arbitrary viewpoint in the scene

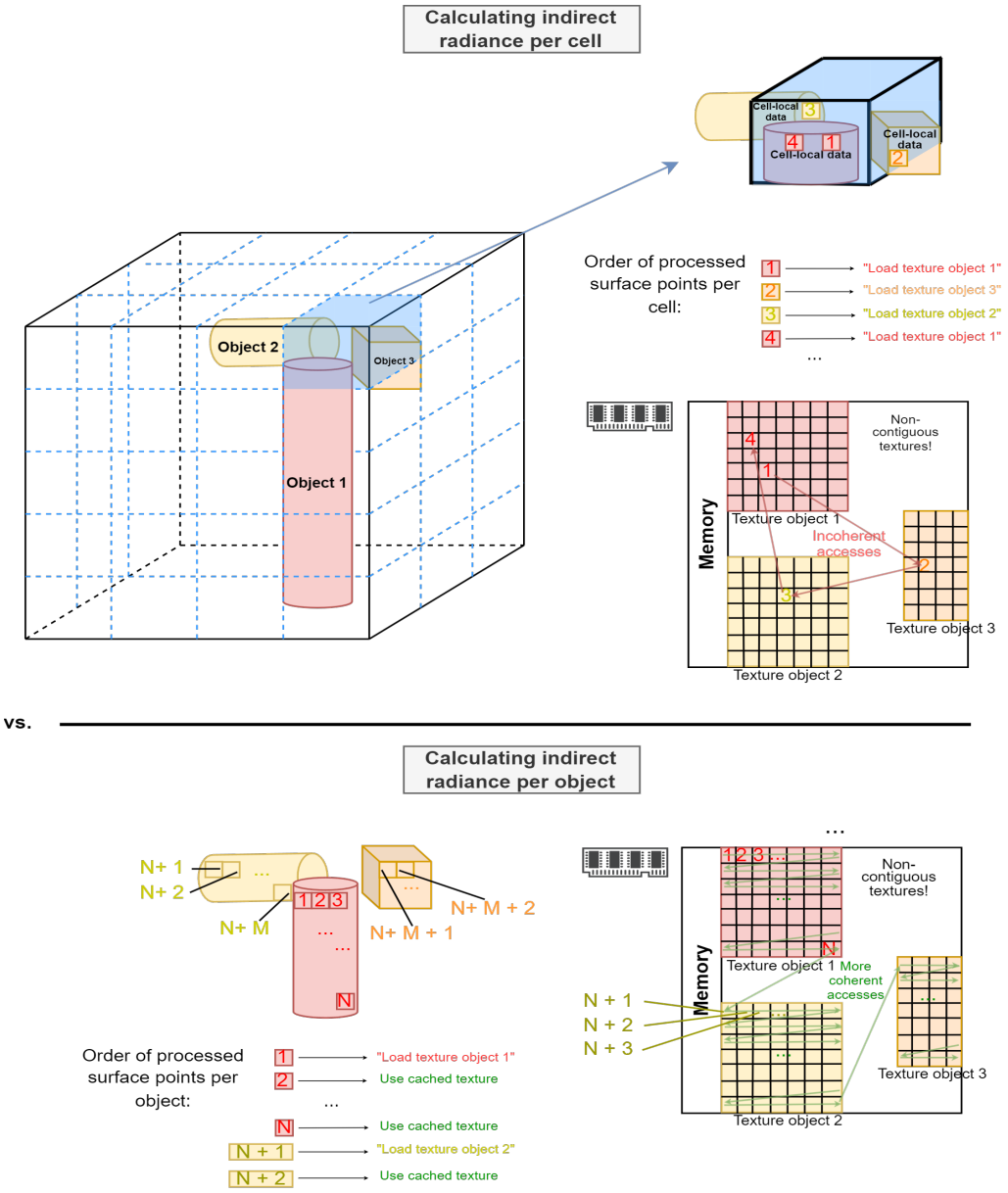
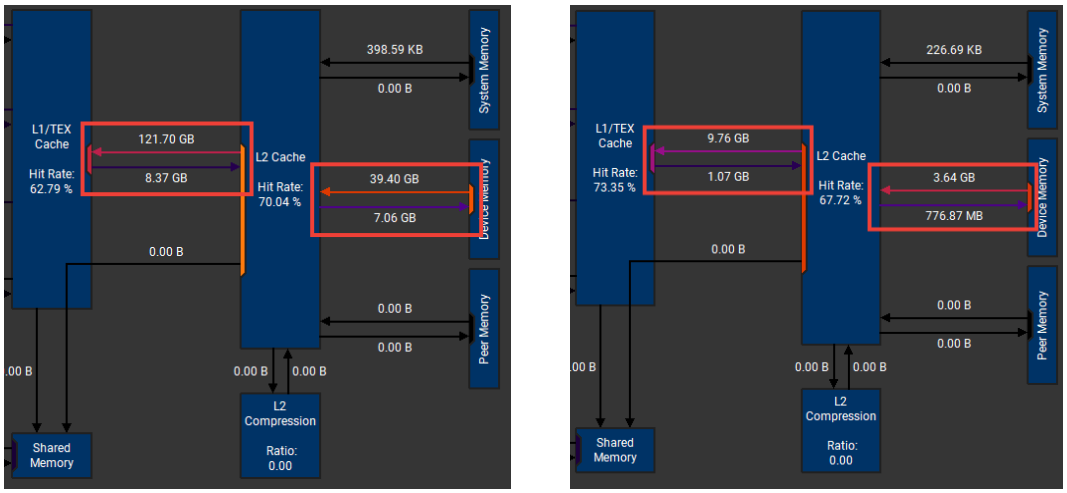



**Figure 4.3.7:** Side-by-side illustration of the workings of both the per-cell (left) and per-object (right) computations of radiance. We first discuss the per-cell computation. The radiance cell for which indirect radiance is currently being calculated is shaded in blue. We observe that 3 different objects are partially inside this cell. Four surface points within the cell are denoted on the figure as an example. Assume that the threads responsible for these surface points are scheduled in the same warp to be executed in parallel. The threads running in parallel need to access different textures, because the irradiance for each object is stored in its own texture. Therefore, many incoherent accesses are made, introducing a significant amount of additional memory traffic. In contrast we look at the per-object computation. Here, instead of grouping surface points per cell, we group surface points per object. The indirect radiance is calculated for all sampled surface points on the object before moving on to the next object. For instance, assume object 1 has $N$ sampled surface points. Then a kernel with N threads is launched for object 1. The threads scheduled to run in parallel are only writing to locations within the same texture, which is far more coherent in write space.




**(a)** Unbiased approach, per-cell computation

**(b)** Unbiased approach, per-object computation

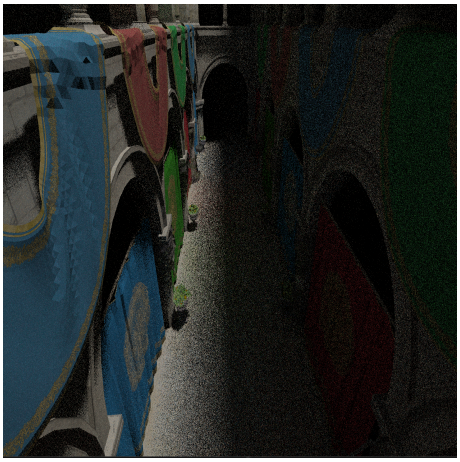
**Figure 4.3.8:** Memory transfer sizes between caches for a certain render pass, measured with Nsight Compute. We see that the per-cell approach transferred much more data between the caches than the per-object approach did. This can be explained by the fact that in the per-cell approach, concurrent threads are making memory writes to different textures due to different objects intersecting with the cell. Each time these textures are accessed, they have to be loaded in entirely. With the per-object approach this is avoided, because each object has its own texture. We launch a kernel per object, with the amount of threads corresponding to the resolution of the object's texture. Concurrent threads are therefore writing to the same texture, resulting in less memory traffic.

by just sending out primary rays and looking up the irradiance for each ray. This corresponds to the idea presented in Figure 1.0.1 and is mostly relevant for multi-user applications. An interesting question that is still unanswered however is: *"What is the amount of work that our algorithm is doing in addition to the work done by the reference path tracer?"*, or in other words, *"How many users are required to make it beneficial to use our algorithm over a reference path tracer?"*. An answer to this question could be of interest to providers of cloud gaming services, to minimize cost. To answer this question, we have set up an experiment that finds the cross-over in algorithm run time between our algorithm and the reference implementation. Both render a fixed number of viewpoints. The cross-over point represents the turning point where our algorithm starts to be more beneficial to use than the reference path tracer (in terms of work done per frame). Note that the timings in this experiment represent the work done per frame, preprocessing steps only have to be done once and are not considered here. We assume that our algorithm runs in a server environment, therefore running once per frame to serve all clients. We also assume the server calculates the irradiance at each point in the scene, but it is up to the client to use this information to render their own viewpoint. This is a valid assumption in most cases, because contemporary client-side systems often contain graphics hardware capable of tracing primary rays and doing a texture lookup for each ray. For the sake of completeness, we have measured the time it takes to compose a viewpoint once the irradiance data is computed. We found that the composition of a 1920 x 1080 viewpoint frame takes 4 ms on average, using the camera and tone mapping pass presented in section 3.8. This is a relatively small fraction of the overall workload of our algorithm.

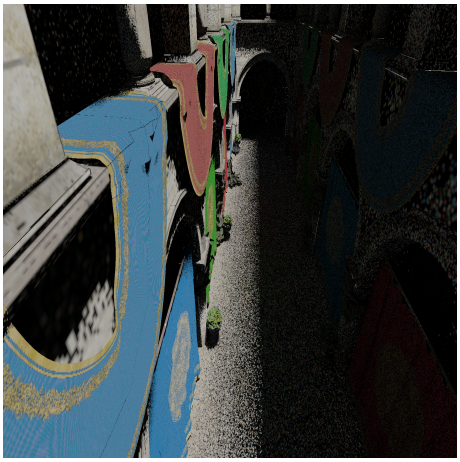
To conduct the experiment, we generate a fixed number of viewpoints for the reference path tracer. We did this by generating positions that are uniformly distributed across each dimension within the scene bounds. All viewpoints are looking towards the center of the scene. It is necessary to generate different viewpoints, because some viewpoints will lead to very complex paths being traced, whereas other might lead to more simple paths (although since we only

consider diffuse materials this is less of a concern). Furthermore, to ensure a fair comparison, it is important that the amount of noise in the outputs of both our algorithm and the reference tracer is similar. We ran the experiment twice for both test scenes. In our first attempt we used the reference path tracer with 1 sample per pixel, and our algorithm with 1 sample for direct lighting and 1 sample for indirect lighting. This run is summarized in Figure 4.4.1 for the *Crytek Sponza* scene. We can see that the cross-over point is located at a user count of 98. For large open-world applications, like MMORPGs, this is an optimistic number. In the other run of the experiment we used the reference path tracer with 5 samples per pixel, and our algorithm with 8 samples for direct lighting and 10 samples for indirect lighting, as we found that these parameters resulted in a similar amount of noise in the output images. The second run is summarized in Figure 4.4.2. Just by making this small change to the sample counts, the the cross-over point drastically shifts to the left, to a user count of 27. Looking at the data of both runs in more detail, we see that our algorithm only achieves a 1.81 times speed-up when we reduce the amount of samples from 10 direct lighting and 8 indirect lighting samples to just 1 sample for both direct and indirect lighting. This in contrast to the reference tracer, which gains a 6.27 times speed-up when we reduce the amount of samples from 5 samples per pixel to 1 sample per pixel. This observation suggests that our algorithm's workload is mostly dominated by the number of surface points for which it must compute irradiance data (we compute irradiance data for the whole scene) and less by the number of samples taken per surface point. This is a promising finding, given our experiments were performed on the baseline implementation. Many optimizations are still possible when it comes to reducing the amount of surface points that are being traced. The same trend can be observed in the data obtained by running the same experiment on the significantly larger *Rungholt* scene. When we increase the scene's size, the amount of surface points that we have to compute irradiance for also increases (assuming that we intend to maintain the same sampling density). This implies that the algorithm has to perform more work, which makes the total run time longer. For completeness, the experimental data for the experiment on the *Rungholt* scene is shown in Figures 4.4.3 and 4.4.4.

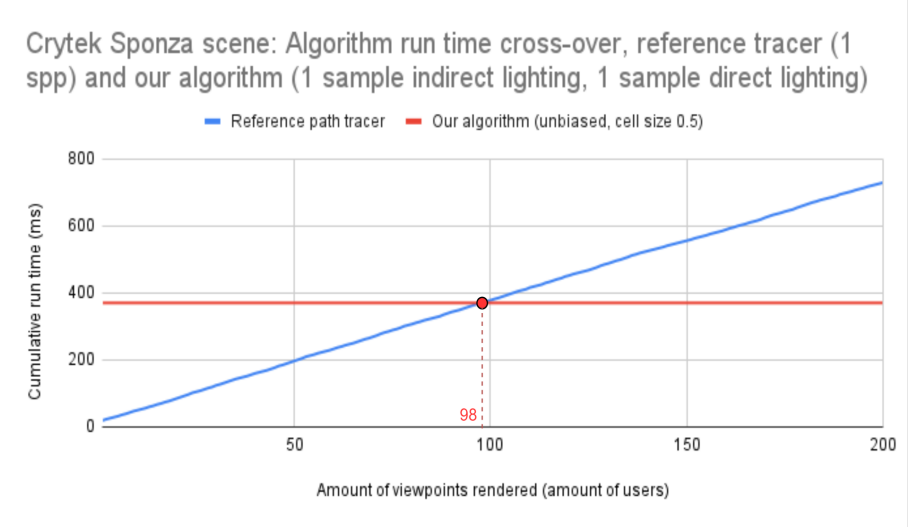
Although doing both runs of the experiment gave us valuable insights, the most relevant run was the first one (Figure 4.4.1). This is because in practice, modern denoising algorithms achieve good results in real time on images generated with only 1 sample per pixel in most cases. Given this fact, it would not make sense to take more than 1 sample per pixel with the reference tracer. Nevertheless, this does not make our approach irrelevant. The cross-over point in Figure 4.4.1 just shows where the amounts of work being done (in terms of run time) by both algorithms intersect. Making the optimistic assumption that to halve the run time of our algorithm we need to double the amount of GPUs working in parallel, a cross-over point at $X = 98$ indeed tells us that at least 98 GPUs would be required to achieve the same frame rate as the reference implementation with 1 sample per pixel. But a good question to ask is if a frame rate this high is a critical requirement. If we look at the data of the first run of this experiment, we see that it takes 3.6 milliseconds on average for the reference path tracer to render one frame of the *Crytek Sponza* scene (1 spp). This corresponds to a frame rate of 277.7 frames per second. For the majority of entertainment applications this is excessive. A frame rate between 30 and 60 FPS is already acceptable for many games, especially when they are fully path traced. To reach a frame rate of 60 FPS, the baseline implementation of our algorithm should be 22.21 times faster than it currently is. This is again without taking into account the composition of a viewpoint for a certain user using the computed irradiance data. Following the same optimistic assumption as we made above, about 23 GPUs in parallel would be required to achieve this. Our approach's key strength however is that regardless of how many people join, these 23 GPUs will continue to be adequate. This in contrast to the traditional approach using the reference path tracer, which has to allocate a separate GPU for each user. **The previous observations ultimately imply that the required resources for the approach presented in this thesis scale based on the size of the scene to be rendered, instead of the amount of users that want to render the scene.**



**(a)** Reference path tracer, 1 spp

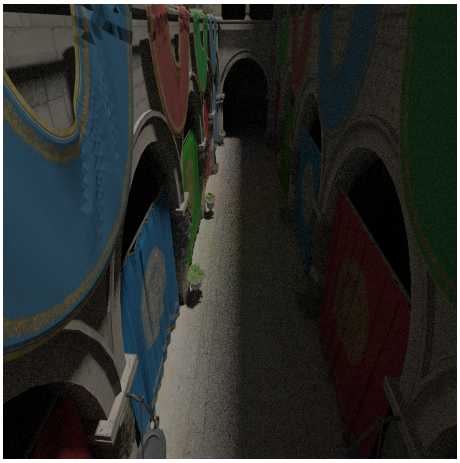


**(b)** Our algorithm, 1 sample direct, 1 sample indirect



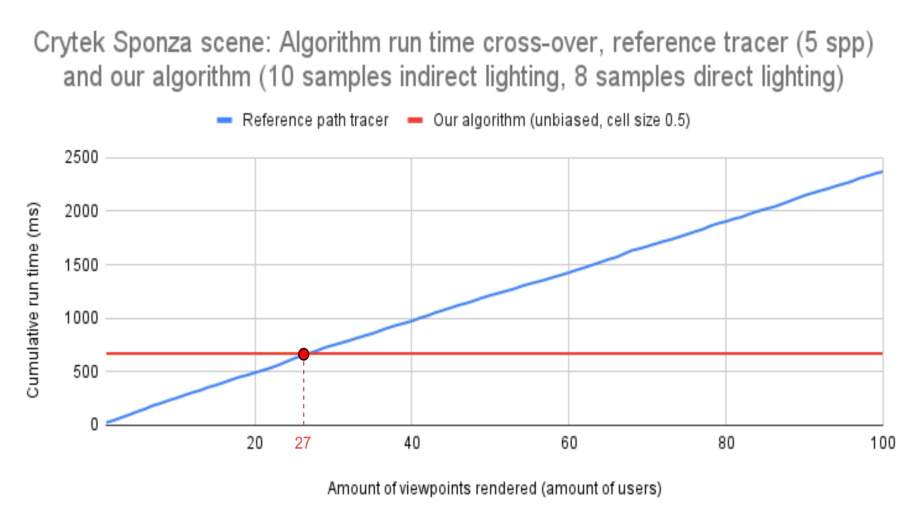

**(c)** Graph with experimental data

**Figure 4.4.1:** Summary of the first run of our run time cross-over experiment for the Crytek Sponza scene. Subfigures 4.4.1a and 4.4.1b show the visual quality of both the reference path tracer and our algorithm with the parameters used for this run, while subfigure 4.4.1c shows the graph containing the cross-over point. For this experiment, we assume that our algorithm is ran in a server environment, the clients use the irradiance information calculated by the server to compose their own viewpoint image. Our algorithm calculates the irradiance of the whole scene just once, and can then use this information to generate any arbitrary viewpoint. Therefore, the work done by the server to generate viewpoints with our algorithm remains constant when we increase the amount of viewpoints. We observe that the cross-over point is located at $X = 98$. This indicates that, starting from 98 simultaneous users, our algorithm takes less time than all reference path tracers combined to render 1 frame for each user's viewpoint. Beyond this point, the amount of work that is saved by using our algorithm scales linearly with an increasing amount of users.



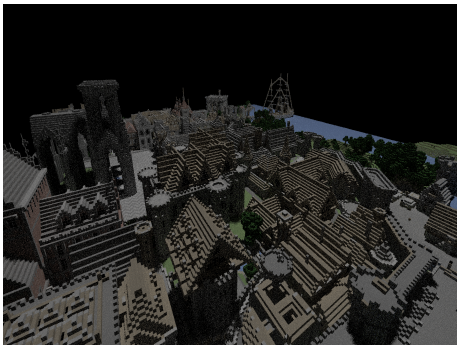
**(a)** Reference path tracer, 5 spp

**(b)** Our algorithm, 8 samples direct, 10 samples indirect

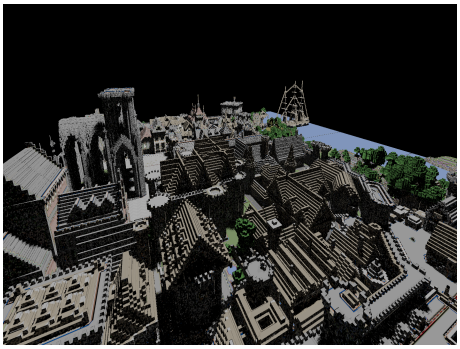



**(c)** Graph with experimental data

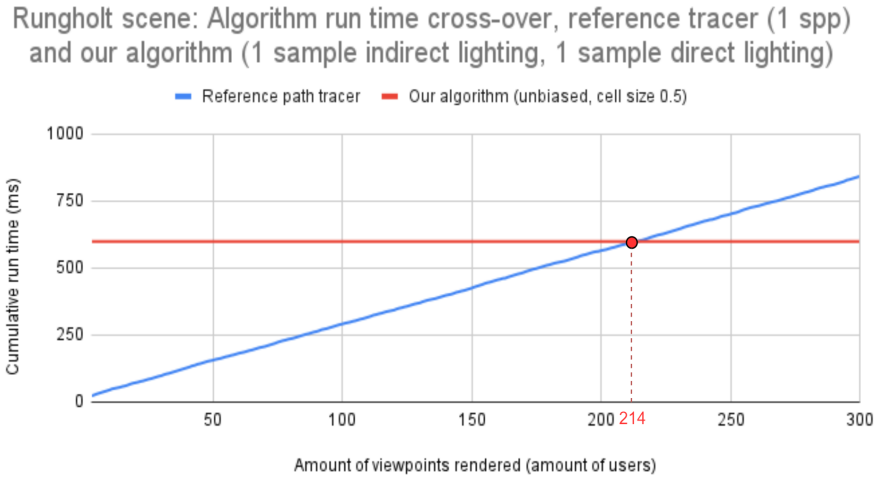
**Figure 4.4.2:** Summary of the second run of our run time cross-over experiment for the Crytek Sponza scene. We observe that the cross-over point is now located at $X = 27$.



**(a)** Reference path tracer, 1 spp

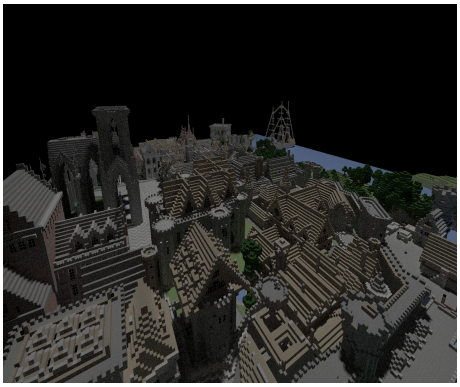


**(b)** Our algorithm, 1 sample direct, 1 sample indirect

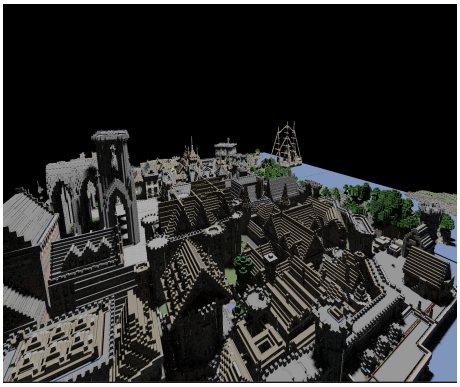



**(c)** Graph with experimental data

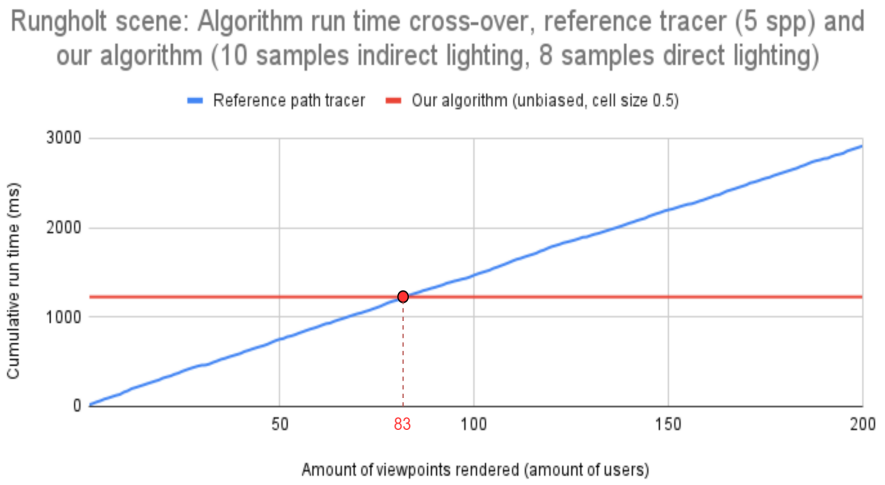
**Figure 4.4.3:** Summary of the first run of our run time cross-over experiment for the Rungholt scene. It can be observed that the accumulated run times per user of the reference path tracer do not change a lot, suggesting that the larger scene size does not impact the traditional Monte Carlo path tracer that much. Instead, its workload is dominated by the amount of samples per pixel that are taken. This is in contrast to our algorithm, whose run time increases significantly with an increase in scene size. Because of this, the cross-over point is shifted further to the right. It is now located at $X = 214$. This suggests that with our approach resources should be scaled based on the scene's size, in contrast to the reference implementation, where resources should be scaled based on the amount of users rendering the scene.



**(a)** Reference path tracer, 5 spp



**(b)** Our algorithm, 8 samples direct, 10 samples indirect




**(c)** Graph with experimental data

**Figure 4.4.4:** Summary of the second run of our run time cross-over experiment for the Rungholt scene. We observe that the cross-over point is now located at $X = 83$.

## 4.5 Scalability

The use of radiance cells in the proposed algorithm nicely subdivides the scene into workload chunks that are comparable in size. This makes the method attractive for higher-lever parallelism. We tested our baseline implementation on a system with one GPU. This implies that we currently parallelize the computation of irradiance at surface points within a radiance cell, but the radiance cells are processed sequentially. Because the algorithm is designed for a cloud-gaming application that runs in a server environment, we can assume that multiple GPUs are available there. The higher-level parallelism that our approach then allows is to divide the radiance cells among the available GPUs. The sole bottleneck present with this approach is that the contributions for a lighting bounce have to be computed before moving on to the next. Therefore, if a GPU has finished processing the cells that were assigned to it, it has to wait until the other GPUs finish with their batch of cells too. This illustrates the importance of the manner in which the division of cells between GPUs is done. It is crucial that the total workload is evenly distributed between GPUs. Fortunately, a good prediction of the workload inside a cell can be made beforehand, because we know the amount of sampled surface points inside each cell. Based on this prediction the cells can be distributed between the available GPUs. In case we do not work per cell but rather per object (see section 4.3.4), the equal distribution of work can be done in a similar manner, e.g. by dividing objects between GPUs.

In conclusion, we expect that, given the possibility to flexibly subdivide of the total workload, the proposed algorithm scales well with an increasing amount of GPUs. In an ideal world, the workload would scale linearly ($O(n)$) with the size of the scene to be rendered (a larger size implies more surface points). With the proposed algorithm this workload can be easily distributed among the available GPUs. An inverse proportional scaling (i.e. the run time is halved when we double the number of GPUs) is likely unrealistic, because in most cases it will be impossible to divide the workload perfectly equally among the available processors. However, given a reasonable distribution of the workload, we expect the algorithm's scaling to closely resemble the inverse proportional behavior. This has to be experimentally verified, and additionally it would be another interesting experiment to explore different strategies for dividing the workload among GPUs. Due to the limited time frame of this thesis we did not include a multi-GPU implementation. We believe that our experimental data verifying the performance of our algorithm on a single GPU is more valuable in the current stage.

## 4.6 Networking

This thesis fully focuses on the rendering aspect that is part of multi-user graphics applications. However, additional efforts in the field of networking are required in order to make our system practical. The networking architecture that would enable such a system is out of the scope of this thesis, but we have kept this in mind while we designed our approach. From a high-level perspective, we propose two options of how to organize the networking aspects of the system. A first option is to handle viewpoint rendering on server side. Using this method, the server composes the viewpoints of the users by looking up the required irradiance values in its previously computed and stored irradiance data. The rendered viewpoints can then be streamed over the network as a real-time video stream (cfr. contemporary cloud gaming).

A second possiblity is that we stream the computed irradiance data to the client. The client then only needs (low-end) graphics hardware to compose their viewpoint to from this data. This could be done via rasterization with texture mapping, or by implementing our camera ray pass on client side. Obviously one will need to carefully consider which irradiance data to stream to each client. It is impossible to send the full scene's irradiance data in real-time to the client, so a selection will have to be made based on what the client can see. This data will need to be encoded in such a way that on client side the irradiance data can still be accessed correctly.

## 4.7 Proposed optimizations

The previous findings and conclusions made in this chapter served as an inspiration to propose several optimizations for the framework presented in this thesis.

### 4.7.1 Tracing and caching of only the visible light paths

The proposed rendering algorithm is dominated by the large amount of surface points for which we need to compute irradiance, rather than the amount of directional samples being taken. We could therefore attempt to prune the collection of surface points for which which we do these computations. Instead of computing irradiance for the whole scene, another option is to only trace paths that are visible to the union of cameras present in the scene. This is similar to the *surfel recycling* idea used in *Global Illumination Based on Surfels (GIBS)* [16]. In addition, this idea aligns closely with the first pass in progressive photon mapping (section 2.1.3), where *visible points* are determined to later gather contributions from nearby photons. For this proposed optimization, there is a fixed *caching budget* that limits the amount of irradiance data that can be cached. This in contrast to our current approach that computes and stores the irradiance for the entire scene. We start from a given set of cameras, project the primary rays coming from their viewpoints onto the scene, this gives us all points that are directly visible from the union of cameras. For these points, we then have to calculate direct and indirect lighting. This concept is visualized in Figure 4.7.1. As long as the caching budget allows it, irradiance data for processed surface points can be stored (cached). Once the caching budget is depleted, stored data will have to be recycled. This can be done heuristically, for example based on the distance to the camera, the viewing angle, the time since the surface point was inside of a camera's viewing frustrum, and so on. This approach not only saves memory, it also scales better to large scenes (the total memory usage is limited). Additionally, this would let the server perform the minimum amount of work that is required to serve all users in the scene. It is desirable to use a more flexible data structure to store irradiance values instead of per-object lightmaps with this approach, because textures will make it difficult to save memory space. Even though this "optimization" may instead be viewed as a whole new approach considering the number of modifications to the current implementation that it requires, we believe it was worthwhile to mention. It is not necessarily combinable with the other optimizations proposed here, but rather provides another interesting look at the problem of path tracing for multi-user environments.

### 4.7.2 Lower texture resolution for indirect bounces

Another way to reduce the amount of surface points that have to be processed is by reducing the resolutions of the irradiance textures. Our current texture-based approach allows us to do this in a flexible way and sample indirect lighting bounces at a lower resolution. This sounds reasonable because diffuse indirect lighting effects manifest themselves as smooth, low frequencies, which can be approximated well by low-resolution textures. The problem with this however is that the larger the surface is, the more visible the usage of low resolution textures becomes. Especially for surfaces that are solely illuminated by indirect lighting, this can become unacceptable. Therefore, the correct way to implement this optimization depends on the scene and one should verify experimentally whether the used texture resolutions are visually acceptable. We found that by lowering the texture resolutions listed in section 4.2 by a factor of 4 in both dimensions, we still obtain results that are visually acceptable. This is with some exceptions in areas where highly-detailed diffuse textures were used on large surfaces only illuminated by indirect lighting. By doing this, we reduce the amount of surface points for which we need to compute indirect irradiance by a factor of 16. An example rendering that shows the visual quality after lowering texture resolutions for indirect lighting is shown in Figure 4.7.2.

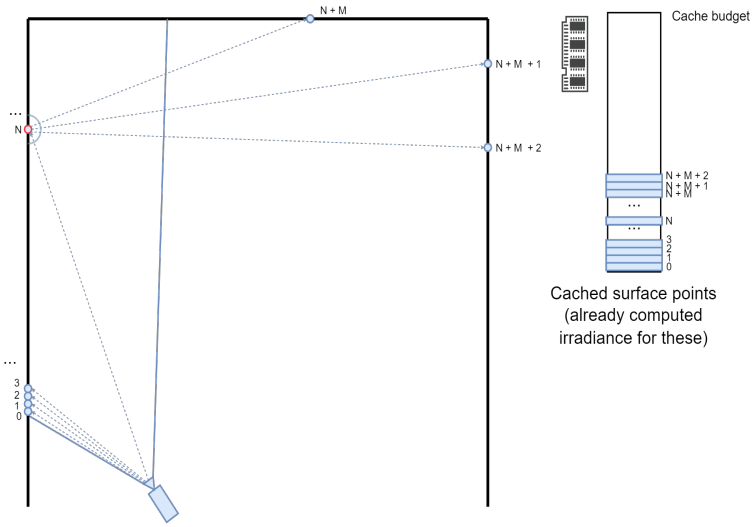
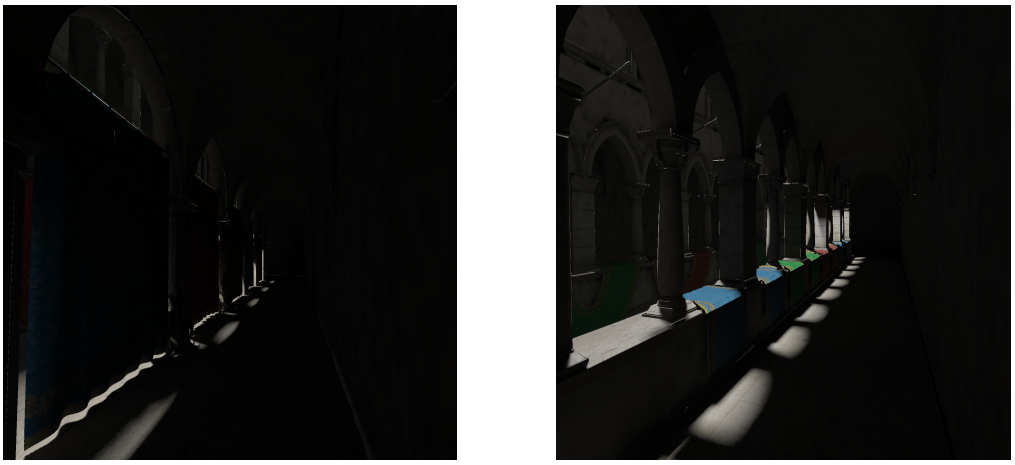



**Figure 4.7.1:** Computing and caching irradiance only at surface points along visible paths. The primary rays coming from the blue camera are projected onto the scene. The intersections between these rays and the scene geometry form the set of all surface points visible to this camera. Examples are points 0, 1, 2, 3 and $N$. We call these the "primary surface points". This is the minimal set of surface points for which irradiance has to be calculated in order to be able to serve all users. Direct lighting is straightforward to calculate for these points by taking light samples and tracing shadow rays from the surface point to the light samples. Because we also want to account for indirect lighting, we need to find incoming radiance from other surface points. The indirect irradiance can be described by the integral over the upper hemisphere of incoming directions at a primary surface point, which can be estimated with a few samples via Monte Carlo integration. We can therefore just take a fixed amount of samples (shown as the 3 recursive rays on the figure) which will lead to an extra set of points for which we have to calculate irradiance (points $N+M$, $N+M+1$ and $N+M+2$ in the figure). This process can be repeated until the desired number of lighting bounces is reached. All points for which irradiance has been computed are cached so they can be reused later when cached surface points appear in the camera's view space. This simple example only shows one camera, but this idea can be expanded to multiple cameras for multi-user applications.



(a) (b)

**Figure 4.7.2:** Parts of the scene where the majority of radiance comes from indirect sources. The textures in which indirect irradiance is gathered have a resolution that is $\frac{1}{4}$ of the resolution used for direct illumination (the textures are 16 times smaller). Thanks to the smooth, low-frequent nature of diffuse indirect lighting, we observe that the impact of these lower resolutions on the visual quality is acceptable.

### 4.7.3 Gradually updating of irradiance in dynamic scenes

The baseline implementation requires that the irradiance in the entire scene is recalculated when the lighting distribution in a dynamic scene changes. This might be an excessive amount of work that is done at once. It should be intuitive that a moving object has a larger radiative impact on some regions of the scene than others. For example, if a red sphere moves underneath a light source, its shadow will move along. When it moves closer to a white wall, we should see the effect of *color bleeding* on that wall. The areas where the shadows and color bleeding (dis)appear are affected the most. Because our scene is subdivided into radiance cells, we could choose to first update the cells that are expected to be affected the most, and queue the rest of the cells to be updated later when no urgent cells are in the queue. A cell can be greatly impacted by a moving object in three ways. It can either be the cell containing the moving object, be very close to it, or intersect with a direct light path towards the moving object. This is illustrated in Figure 4.7.3. It should be noted however that the implementation details of this proposed optimization should be handled with care. That is, the algorithm that decides which cells should be updated first and the management of the queue of cells to be processed should be as lightweight as possible. The run time of this algorithm added to the time it takes to update the selected urgent cells should be less than the run time of the current baseline algorithm. The viability of this needs to be verified experimentally and is regarded as future work.

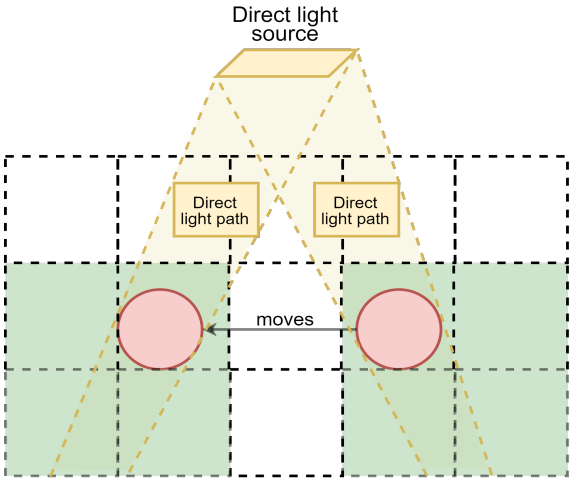



**Figure 4.7.3:** Illustration that visualizes the priority of certain cells when it comes to updating the irradiance in a dynamic scene. This simple example contains one direct light source. A red sphere moves from right to left. The direct light paths to the sphere in both states are shaded in yellow. The areas that lie along the direct light paths after intersection with the moving object will be impacted mostly, and should therefore be updated first. These cells are marked in green. The rest of the cells will not be much affected by this dynamic change, and can therefore be updated a little later without making it noticeable to the user.

# Chapter 5

# Discussion and conclusions

The framework proposed in this thesis has potential benefits that were experimentally verified. Although it certainly has its limitations (section 3.15), it seems plausible that most of these can be overcome. Our main focus was to centralize Monte Carlo path tracing for multi-user scenarios in a server environment and do this in a coherent manner. We aimed to investigate whether a centralized approach could have advantages for applications in industries as cloud gaming. Contemporary implementations of cloud gaming services run separate instances for each client, which does not scale well for applications with many simultaneous users. Our experiments have analyzed the difference in terms of total executed workload between our proposed algorithm and a group of reference path tracers. The question that we wanted to answer was: *"Starting from how many users does our algorithm perform less work than the union of reference path tracers allocated for all users?"* This was done by comparing the run time of our algorithm on a single GPU with the accumulated run time of an increasing amount of instances of a reference GPU path tracer (section 4.4). Although the workload of our algorithm would normally be shared between multiple GPUs, the execution time on a single GPU is a decent indicator for the overall work being performed by the algorithm. With an increasing amount of users, we observe that our algorithm takes the upper hand starting from a certain cross-over point. This means that beyond this point, utilizing our algorithm essentially requires less total work per frame than running a separate path tracer for each user. In other words, each GPU that is allocated per joining user beyond this point can be considered wasteful, because our algorithm can achieve the same performance with a constant amount of GPUs. Where the cross-over point is located depends on the scene and the amount of samples being taken, although our experiments show that for realistic amounts of users our algorithm is already beneficial to use. Furthermore, the cross-over point represents the point where the total algorithm run time between our algorithm and all instances of the reference is the same. Suppose that the cross-over point is located at 60 users. If we assume that we want our algorithm to achieve the same frame rate as the reference path tracer, this means that on average our algorithm needs to be 60 times faster than it currently is. Assuming a linear scaling, this implies that we would need at least 60 GPUs. This may appear to be a lot. However, the frame rates achieved by the reference path tracer are often excessive for most applications. We have observed frame rates over 240 FPS for the reference tracer in our experiments. Network delays would form a bottleneck for a frame rate that is this high. In addition, it is in most applications acceptable to have a frame rate between 30 and 60 frames per second. The number of GPUs that the cross-over point "prescribes" can therefore be considered as an indicative upper limit of GPUs that are actually required for our algorithm to achieve an interactive frame rate. The number of GPUs required to achieve acceptable frame rates is most often expected to be significantly lower. This in contrast to the traditional approach, where there is no other option than to allocate a separate GPU per user, regardless of the frame rate that should be achieved.

Although we have included an implementation of both a biased and unbiased approach, it currently does not make sense to use the biased approach. It not only introduces bias, but

is also slower. We designed the biased algorithm with memory coherence in mind. It was indeed confirmed by our experiments that the biased approach leads to a higher percentage of cache hits (section 4.3.2). However, the performance gained by this increase in coherence seems to be outweighed by the overhead caused by the additional render pass (the radiance probe gathering pass, section 3.6) that the biased approach introduces. The extra work of updating the radiance probes for each frame is simply too expensive. If we make the assumption that the scene is semi-dynamic, we could precompute the content of the radiance probes (cfr. the irradiance volume, section 2.5.6). This introduces other limitations however, such as missed occlusions between dynamic objects. A possible solution for this that can be explored in future work is to trace additional rays from the dynamic objects to the surrounding geometry. This seems like a workable strategy if the number and size of the dynamic objects are reasonable.

For our implementation, we observed that computing the irradiance per object led to a more coherent execution than by computing it per radiance cell (all the sampled surface points inside the radiance cell). This is due to the fact that we store irradiance in a separate texture for each object. Warps of threads that are scheduled to execute in parallel can therefore write to the same texture, leading to higher coherence in *write space*. It should be noted that coherence in write space is important to our algorithm, but much less for a traditional path tracer. A traditional path tracer sends out rays for each pixel in a camera's viewport. These pixels are stored into the same texture, which implies that it is automatically coherent in write space. On the contrary, the algorithm proposed in this thesis computes irradiance in global scene space. With this knowledge, it should become clear that given the large amount of memory writes being done, these should ideally be processed in a coherent manner. Based on our experimental observations, we expect that the ideal approach to achieve this will differ per irradiance caching data structure that is chosen. For future work that takes inspiration from our work, it is therefore important to ensure that irradiance samples that will be stored in close proximity to one another in memory are scheduled to be executed in parallel together.

Our approach for caching irradiance in textures is likely not the best choice. Although it has its advantages (texture memory is coherent, hardware-accelerated lookups, ...), the additional problems that texture mapping and unique mesh surface parameterization introduce both in terms of quality (artifacts) as well as pre- and post-processing requirements are far from ideal. Instead, we believe that a more flexible method where samples can be placed freely (e.g. surfels) could be more suitable to cache irradiance. This has the advantage that the sampling rate can be dynamically adapted based on the region in the scene, and no explicit mapping from 3D to 2D space is required. Furthermore, it would allow to store the irradiance cache in a more flexible manner and even set a hard limit on the memory budget for irradiance caching (cfr. section 4.7.1).

A multi-GPU implementation of the algorithm proposed in this thesis is considered as future work. Given the scalable design, we expect the algorithm run time to scale nearly inverse proportionally with an increasing amount of GPUs. Our experimental findings allowed us to propose several optimizations that could be explored in the future to improve this framework (section 4.7). Although we were not able to provide ground-breaking discoveries, the knowledge and concepts that we gained from this thesis will be helpful as we continue to research similar topics in the future. One promising idea is that of *visibility propagation*, discussed in more detail in chapter 6.

# Chapter 6

# Future work

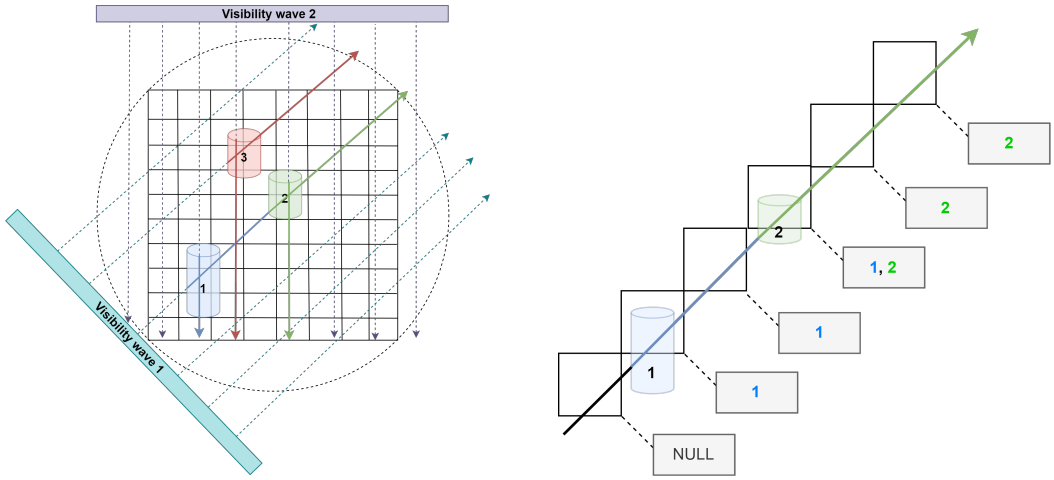
First of all, the optimizations proposed in section 4.7 can be considered as future work. Additionally, it would be interesting to experiment with other representations for the storage of computed irradiance. More specifically, we believe that surfels and *neural graphics primitives* [28] are suitable candidates. However, the most promising idea that the observations in our experiments gave us is probably that of *visibilty propagation*. This idea is further explained in the remainder of this chapter.

## 6.1 Propagating visibility instead of radiance

The work proposed in this thesis started from the idea of propagating radiance throughout the scene, in the hope this would increase local memory accesses and therefore coherence. Through experimentation it was noticed that radiance data might not be the best choice for propagation, given the fact that it can influence the rest of the scene and therefore has to be updated with each change in illumination. For interactive applications, such as video games, this would mean that in practice the radiance propagation has to be done for each frame. Furthermore, only the biased version of the proposed algorithm ensures increased coherence in both read and write space, the unbiased version only increases coherence in write space. However, the biased version is still slower than the unbiased version, because the radiance probes have to be updated with each change in illumination as well. It is possible to select and update only the probes that will see changes, but this comes down to testing visibility which is an expensive step to perform every frame.

### 6.1.1 Propagating visibility

The idea of *propagating visibility information* seems to be a promising solution to this. This concept is presented in Figure 6.1.1. The figure shows an algorithm we have envisioned to propagate visibility in a discrete manner. The data structure resulting from this propagation can be seen as a discrete form of the 3D visibility complex [10]. The scene is again divided into a grid of cells, which we will name *visibility cells* in this discussion. Each visibility cell contains a list of pointers to the objects that are visible from that cell. If there is a single point in cell X that can see point Y, then Y should be added to the pointer list of X. Because visibility not only depends on position but also has a directional component, we propagate visibility in the form of directional waves. These *visibility waves* can be seen in Figure 6.1.1a and are in structure very similar to parallel subfields in the context of virtual light fields that were discussed in section 2.5.5. A visibility wave is a bundle of parallel visibility rays. For a fixed amount of directions, visibility waves are propagated throughout the scene. Every time a visibility ray encounters an object, it gets "marked" by that object. The ray can only be marked by one object at a time, meaning that the mark will be overwritten once the ray intersects a new object. For each visibility cell that the visibility ray encounters, the object that currently marks the ray is added to the pointer list of that cell. The object that the ray intersects with should also be added




**(a)** High-level overview **(b)** Propagation illustrated for one visbility ray

**Figure 6.1.1:** Illustration of the envisioned visibility propagation algorithm. In Figure a), two visibility waves that propagate throughout the scene are shown. A visibility ray is given the color of an object after intersecting with it, meaning that it is "marked" by that object. In Figure b), the propagation process along one visibility ray is shown in more detail. The list of pointers per cell represents the objects that are visible from that cell.

to the list of the cell in which the intersection occurs. This process is shown in Figure 6.1.1b. After visibility is propagated, each visibility cell in the scene knows which objects are visible to it. The space that the ray travels between intersecting two distinct objects corresponds to the maximal free segment between those objects. Recall that a maximal free segment is the linear space between two points that are mutually visible. We are in essence storing a discretized form of these maximal free segments.

### 6.1.2 Specular surfaces

The algorithm described above does not take into account specular surfaces. It only propagates *direct visibility*, that is, when two points in space are connected by an unoccluded straight line. With specular surfaces, *indirect visibility* is introduced. In other words, an object can be visible via the reflection of a specular object's surface. To address this, we can cast additional *reflected visibility rays* at encounters with specular surfaces according to the surface's BRDF. This is illustrated in Figure 6.1.2 for a perfectly specular surface. For glossy surfaces, the BRDF can be sampled a given amount of times and multiple reflected visibility rays can be casted.

### 6.1.3 Rendering with visibility information

Once the *visibility grid* is filled with discrete visibility information, it can be used as a guiding data structure for importance sampling during rendering. Traditional path tracing is used to render a viewpoint, but instead of taking random samples according to the surface's BRDF at an intersection, the computed visibility information can be used to take samples in a more clever way. By consulting the list of objects that are visible to the cell in which the intersection occurred, directional samples towards these objects can be taken. This corresponds to importance sampling (in)direct light sources. An illustration of how the visibility grid can be used for importance sampling is shown in Figure 6.1.3. This is just one use case, but a discretized approximation of visibility information could be useful to solve other problems as well. Other use cases will have to be explored in future work.

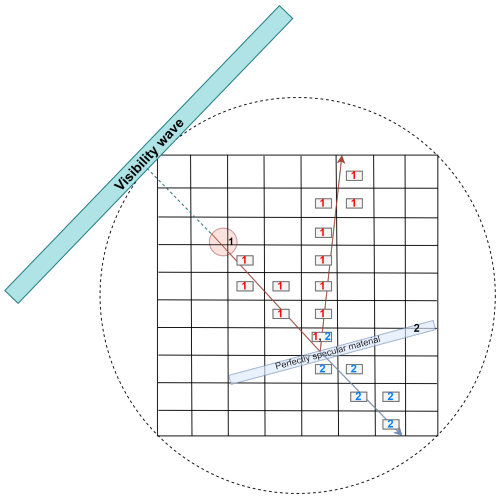



**Figure 6.1.2:** Besides casting visibility rays that capture direct visibility, additional "reflected visibility rays" can be casted at specular surfaces to account for indirect visibility.

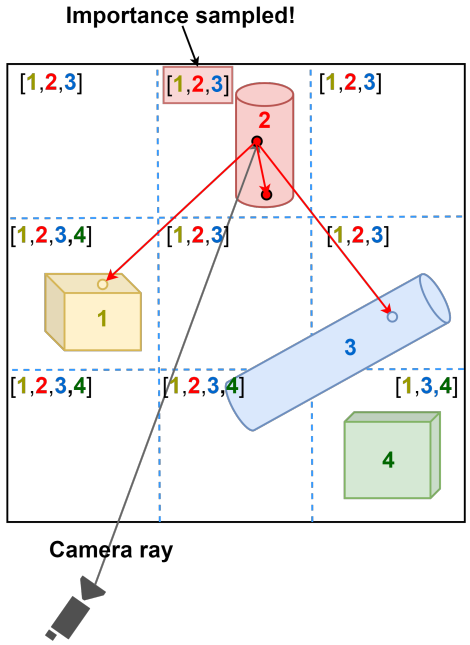



**Figure 6.1.3:** Using visibility information to importance sample visible objects for indirect lighting. A camera ray intersects with object 2. The list of objects visible to the cell in which the intersection occurred is consulted to determine where samples should be taken.

### 6.1.4 Updating the visibility grid

A core advantage of the concept of visibility propagation applied to importance sampling is that we do not need to update the complete visibility data structure in real-time to use it for rendering. Because we are using it to guide rays for importance sampling, we can update it gradually without losing correctness. We will only introduce additional bias when the visibility information becomes obsolete. However, to reduce the effect of this, objects that have a large impact can be updated in real time by adopting an inside-out strategy. This means that we send out visibility rays from the object into the scene, in contrast to the visibility waves that adopt

an outside-in strategy. Examples of objects with a large impact are light sources, reflective objects, fast moving objects...

### 6.1.5 Object priorities

It is also possible to assign priorities to certain objects when they are added to a cell's pointer list, e.g. based on nearness or direction that the visibility was propagated along. It is expected that this further improves the importance sampling. Additionally, handling visibility on a per-object basis is not a requirement. Objects could also be further subdivided into patches. Instead of adding object pointers to a cell's visibility list, pointers to patches could then be added. This will increase the memory usage, in exchange for more accurate visibility information.

# Appendix A

# Nederlandstalige samenvatting

## Introductie

Doorheen de geschiedenis hebben we gemerkt hoe de meerderheid van entertainment applicaties de transitie naar de cloud gemaakt heeft. Een interessante vraag om te stellen is waarom dit nog niet gebeurd is voor computer graphics applicaties, zoals video games. In *cloud gaming* wordt de infrastructuur voor gaming voorzien door een centrale cloud provider. Cloud gaming haalt daarbij de werklast van het renderen en managen van een video game weg van de client. In de plaats daarvan worden deze taken toegewezen aan een server, die de resultaten streamt naar de client. Dit laat toe om games on-demand te spelen op praktisch eender welk toestel dat beschikt over een display, invoer en internetverbinding. Alhoewel er reeds oplossingen voor cloud gaming bestaan, worden deze nog niet veel gebruikt. De meest gangbare verklaring hiervoor is dat de huidige oplossingen niet goed opschalen naar grote gebruikersaantallen, vooral op het vlak van multiplayer applicaties.

Met de opkomst van hardware-accelerated ray tracing is het mogelijk geworden om lichttransport in real-time te simuleren op consumer hardware. Dit laat toe om zeer realistische belichtingseffecten in video games te integreren. Desondanks vergen deze simulaties zeer veel rekenkracht. Ze simuleren met fysische accuraatheid hoe een beeld genomen van een bepaald standpunt eruit zou zien. Omdat bestaande lichttransport algoritmen individuele standpunten renderen, schalen ze niet optimaal op naar meerdere gebruikers. Dit komt doordat de onderliggende algoritmen in essentie een aparte rekeneenheid per gebruiker nodig hebben, gegeven de computationele intensiteit van de achterliggende werklast. Daarom is het op dit moment logischer dat gebruikers hun eigen hardware aankopen, in plaats van dat een cloud provider deze hardware moet voorzien voor iedere gebruiker apart.

Een ander probleem dat inherent is aan simulaties van lichttransport, is dat van ray-incoherentie. Wanneer lichtstralen worden getraceerd in een scène, interageren ze met verschillende materialen, wat resulteert in een verandering van hun traject. Dit impliceert dat een groep stralen mogelijk sterk uiteenlopende richtingen en/of oorsprongen heeft. We noemen deze stralen incoherent. Om real-time simulatie van lichttransport mogelijk te maken, worden veel stralen in parallel verwerkt op een massaal parallelle processor. Als incoherente stralen parallel worden verwerkt, kunnen ze mogelijk veel verschillende geheugenlocaties bereiken. Dit verspilt het potentieel van het snelle cachegeheugen van de GPU, omdat telkens wanneer er een cache-miss optreedt, gegevens moeten worden opgehaald uit het hoofdgeheugen, dat vele malen trager is. Daarom vormt geheugenincoherentie nog steeds een groot knelpunt in hedendaagse path-tracing algoritmen [4].

# Objectieven

Deze thesis richt zich op twee doelstellingen. Een eerste doelstelling is het definiëren van een aanpak voor Monte Carlo path tracing die profiteert van een hogere ray-coherentie dan traditionele Monte Carlo path tracers. De ideeën achter ons algoritme zullen worden geëvalueerd en gedemonstreerd via een proof-of-concept implementatie die representatief moet zijn voor middelgrote tot grote scènes in video games. Een tweede onderwerp van focus zal zijn om de hypothese te bevestigen dat een algoritme dat de belichting van de volledige scène berekent beter schaalt voor veel gebruikers. We proberen dit tastbaar te maken door een expliciete vergelijking te maken tussen de werklasten die worden uitgevoerd door ons algoritme en een referentie-GPU-Monte Carlo path tracer. Bovendien moet worden vermeld dat we ons zullen richten op de berekening van diffuse globale belichtingseffecten, om twee redenen. Ten eerste zijn diffuse belichtingseffecten de meest relevante kandidaat voor caching, vanwege hun vertoning die onafhankelijk is van de kijkrichting. Ten tweede zijn ze het worst-case scenario als het gaat om ray-incoherentie, omdat diffuse oppervlakken lichtstralen verstrooien op een uniforme manier. Om deze redenen verwachten we de grootste impact te hebben met ondersteuning voor diffuse belichtingseffecten.

# Implementatie

We stellen een algoritme voor dat de irradiantie voor de hele scène berekent en opslaat. De berekende gegevens kunnen vervolgens worden gebruikt door een willekeurig aantal gebruikers om een beeld voor hun standpunt samen te stellen. Dit werkt voor diffuse globale verlichting omdat het visuele uiterlijk van een punt op een diffuus oppervlak kan worden beschreven door het product van de Lambertiaanse BRDF en de irradiantie op dat punt. We beschrijven eerst kort de componenten van ons systeem. We maken onderscheid tussen directe en indirecte belichting. We behandelen directe belichting in een aparte render pass. Voor indirecte belichting wordt de scène onderverdeeld in zogenaamde *radiance cellen*. Deze radiance cellen werken onafhankelijk van elkaar op hun lokale geometrie. Met andere woorden, indirecte belichting wordt cel per cel berekend, waarbij iedere cel zijn lokale oppervlakken behandelt. We hanteren een hybride radiantie-propagatieschema voor indirecte belichting, wat betekent dat nabije indirecte belichting wordt geraytraced en verre indirecte belichting wordt benaderd. Deze benadering wordt gedaan door gebruik te maken van een nabije *radiance probe*. In onze implementatie hebben we een radiance probe geplaatst in het midden van iedere radiance cell. Of er ray tracing of een benadering wordt gebruikt, wordt bepaald door een threshold parameter die de *tracing range* wordt genoemd. Verder werkt het algoritme per lichtbotsing. Het resultaat van een lichtbotsing wordt gebruikt als input voor de volgende lichtbotsing. Een overzicht van ons algoritme wordt gepresenteerd in Figuur A.0.1.

## Unbiased aanpak

Het algoritme dat wordt gepresenteerd in Figuur A.0.1 is *biased*. Naast dit algoritme leveren we ook een *unbiased* aanpak waarbij we de drempelwaarde voor de tracing range parameter op oneindig hebben gezet. Dit betekent in essentie dat we geen gebruik meer maken van benaderingen met radiance probes voor indirecte belichting die veraf is, maar in plaats daarvan alles raytracen. Omdat we geen gebruik meer maken van de radiance probes, kunnen we ook de tweede render pass overslaan. De unbiased aanpak voert daarom slechts twee render passes uit: de *directe belichting pass* en de *radiance cell scattering pass*.

## Datastructuur om irradiantie in op te slaan

Er werden verschillende datastructuren overwogen om de berekende irradiantie in op te slaan, om er later samples uit te kunnen nemen. In het bijzonder werden *GPU octree texturen* en *per-object lightmaps* geïmplementeerd en getest. GPU octree texturen waren in eerste instantie aantrekkelijk vanwege hun robuustheid en het ontbreken van de noodzaak om 3D punten op de

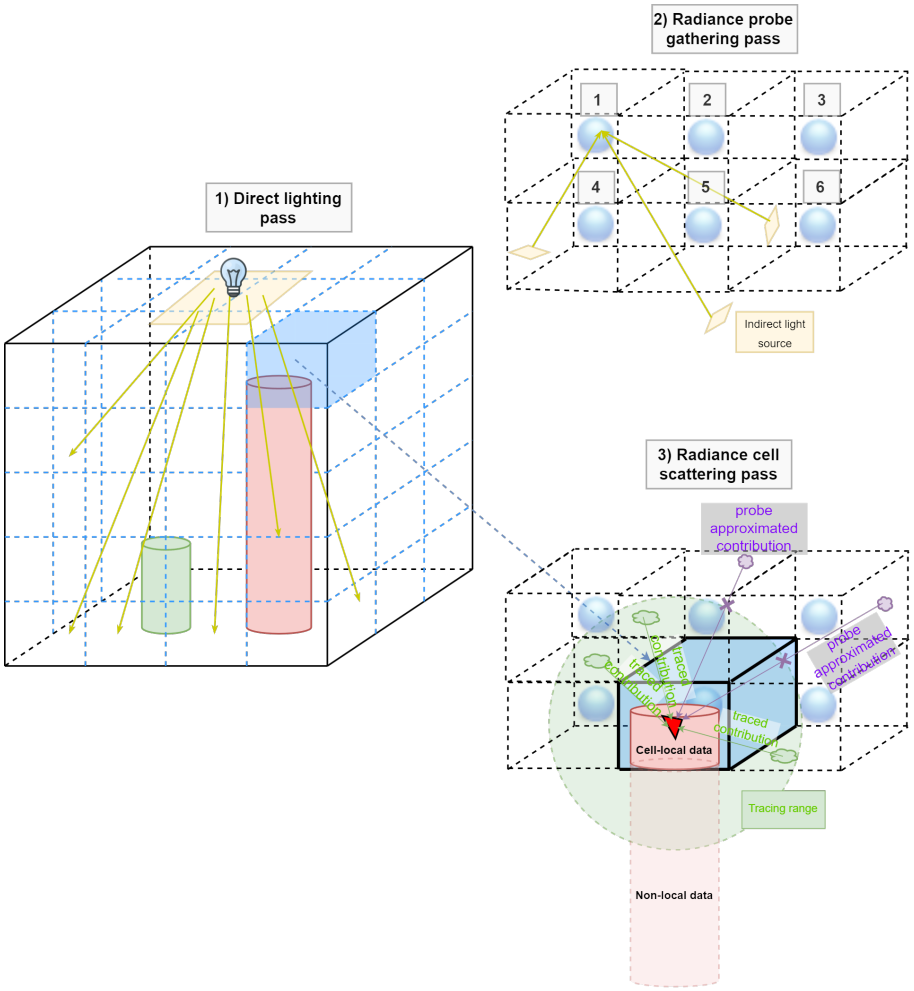



**Figure A.0.1:** Overzicht van het voorgestelde renderalgoritme en gebruikte datastructuren in de thesis. De scène is onderverdeeld in een regelmatig raster van "radiance cellen". Het algoritme begint met de directe belichtingsfase, waarin directe lichtstralen vanaf lichtbronnen doorheen de scène propageren. Vervolgens, tijdens de "radiance probe gathering" fase, wordt indirecte radiantie vanuit de vorige lichtbotsing verzameld in "radiance probes". Na deze fase bevat iedere radiance probe een omnidirectioneel beeld van de binnenkomende indirecte radiantie op hun positie. Ten slotte, in de "radiance cell scattering" fase, wordt indirecte radiantie overgedragen naar de lokale geometrie van iedere radiance cel. Met andere woorden, de oppervlakken van de scène worden belicht met indirecte radiantie voor de huidige lichtbotsing. We maken onderscheid tussen nabije en verre indirecte radiantie op basis van een drempelwaarde genaamd de "tracing range". We handelen nabije binnenkomende indirecte radiantie af met ray tracing en benaderen verre indirecte radiantie door gebruik te maken van de radiance probes.

geometrische oppervlakken naar 2D-ruimte te transformeren. Later hebben we echter besloten om hiervan af te stappen, voornamelijk vanwege het gebrek aan ondersteuning voor *hardware texture filtering*. Dit zou de snelheid van textuur zoekoperaties aanzienlijk vertragen, terwijl ons doel was om minimale overhead te introduceren. Daarentegen bieden per-object lightmaps wel ondersteuning voor hardware acceleration, hoewel ze te kampen hebben met andere beperkingen, zoals een hoog geheugengebruik en textuur artefacten. Het is verder belangrijk op te merken dat hoewel de onderliggende datastructuur verwisselbaar is, iedere representatie heeft bepaalde kleine nuances waar rekening mee moet worden gehouden in de implementatie van de renderkernels.

### Limitaties en optimalisaties

Enkele beperkingen waar de implementatie last van heeft zijn artefacten, hoog geheugengebruik en geen ondersteuning voor speculaire belichtingseffecten. De eerste twee worden voornamelijk veroorzaakt door de gekozen onderliggende datastructuur voor belichting. *Texture seam*-artefacten zijn meestal onvermijdelijk in de praktijk, omdat er een mapping van 3D naar 2D-ruimte moet worden gemaakt. Het algoritme dat deze mapping uitvoert, moet op sommige plaatsen 'scheuren' maken om het 3D-oppervlak 'uit te vouwen'. Deze scheuren manifesteren zich als *texture seams* wanneer textuurfiltering wordt toegepast. Hoewel deze beperkingen momenteel aanwezig zijn in onze implementatie, geloven we dat ze kunnen worden overwonnen, gemotiveerd door het feit dat er al oplossingen bestaan voor deze beperkingen. Helaas vielen deze oplossingen buiten het bestek van deze thesis en hebben we ze daarom niet opgenomen.

Hiernaast hebben we verschillende optimalisaties voorgesteld die als toekomstig werk beschouwd kunnen worden. Een eerst observatie is dat het algoritme momenteel de irradiantie voor de hele scène berekent. Het zou intuïtief moeten zijn dat dit onnodig is voor het doel dat we willen bereiken. Het doel is namelijk om alle berekeningen die nodig zijn voor alle gebruikers in de scène te centraliseren en die berekeningen tussen gebruikers te hergebruiken. Daarom legt een van de voorgestelde optimalisaties uit dat het voldoende zou zijn om de belichting te berekenen voor de unie van zichtbare punten voor alle gebruikers. Een andere optimalisatie stelt een vorm van asynchroon renderen voor, waarbij de belichting eerst wordt bijgewerkt op de plaatsen waar het het belangrijkst is en later in gebieden die minder visueel worden beïnvloed door de dynamische veranderingen in de scène.

## Evaluatie en analyse van de implementatie

Allereerst hebben we geëxperimenteerd met de configureerbare parameters van onze implementatie. Meer specifiek was onze hypothese dat door het verlagen van de tracing range drempelwaarde, we de coherentie zouden vergroten. De redenering hierachter is dat door de ruimte die een lichtstraal kan doorkruisen te beperken, we ook de geheugenruimte beperken die de straal kan bereiken. Om dit te verifiëren, hebben we ons algoritme uitgevoerd voor verschillende waarden van de tracing range, waarbij alle andere parameters constant werden gehouden. We hebben zowel de uitvoertijd van het algoritme als de cache hit ratio's voor de L1- en L2-caches gemeten. Onze experimenten bevestigden inderdaad dat het verlagen van de tracing range een positieve invloed heeft op zowel de algehele prestaties van het algoritme als de L2 cache hit ratio (coherentie). Het is echter belangrijk op te merken dat het te laag instellen van de tracing range artefacten zoals *light leaks* kan introduceren, omdat de veronderstelling van *distant lighting* wordt geschonden.

Op dezelfde manier werd de impact van de grootte van de radiance cel op de prestaties van het algoritme gemeten. De impact van de grootte van de radiance cell was minder significant. Wanneer de grootte te klein wordt ingesteld, begint de prestatie van het algoritme te lijden onder de extra overhead die wordt geïntroduceerd. Deze overhead ontstaat doordat er per radiance cel een aparte kernel moet worden gestart. Als de radiance cellen te klein en talrijk worden, wordt

de werklast van elke radiance cel ook te klein om de kracht van de GPU volledig te benutten.

Het tweede uitgevoerde experiment vergelijkt het totale werk dat wordt verricht door ons algoritme met de werklast van een traditionele Monte Carlo-path tracer op de GPU. Het kan interessant zijn voor aanbieders van clouddiensten om een idee te hebben vanaf welk aantal gebruikers het voordelig zou zijn om een algoritme zoals het onze te gebruiken in plaats van een traditionele aanpak, waarbij het standpunt van elke cliënt onafhankelijk wordt geraytraced. Om dit mogelijk te maken, hebben we een referentie path tracer geïmplementeerd die diffuse globale belichting berekent voor 3 lichtbotsingen (net als ons algoritme). We hebben onze vergelijking gemaakt in termen van uitvoertijd van het algoritme. Ons algoritme hoeft slechts één keer per frame te worden uitgevoerd om alle gebruikers te bedienen, als we de tijd die nodig is om een standpunt samen te stellen uit onze gecachte irradiantie data buiten beschouwing laten. Dit in tegenstelling tot de referentie path tracer, die volledig opnieuw moet worden uitgevoerd voor elk standpunt dat moet worden gerenderd. Voor de referentie path tracer hebben we een uniforme reeks standpunten gegenereerd en de totale uitvoertijd berekend door de rendertijd voor elk van deze standpunten op te tellen. We hebben deze geaccumuleerde uitvoertijd vergeleken met de uitvoertijd van ons algoritme en hebben het *cross-over* punt bepaald waar het voorgestelde algoritme minder werk begint te kosten dan de traditionele aanpak.

Dit experiment heeft ons twee inzichten opgeleverd: Ten eerste toont het het cross-over punt waar ons algoritme minder werk begint te verrichten dan het totale werk dat wordt verricht door alle onafhankelijke referentie path tracers samen. Dit punt geeft ons ook de factor waarmee ons algoritme versneld moet worden om dezelfde frame rate te behalen als de referentie path tracer. Let op dat deze versnellingsfactor het worst-case scenario vertegenwoordigt: voor 1 sample per pixel behaalt de referentie path tracer frame rates van meer dan 250 FPS, wat meer dan genoeg is voor de meeste applicaties. Voor games is het acceptabel om frame rates tussen 30 en 60 te hebben, vooral wanneer het spel volledig geraytraced is. Ten tweede toonde dit experiment ons ook dat de werklast van ons algoritme voornamelijk wordt bepaald door de hoeveelheid punten op de geometrische oppervlakken waarvoor het de irradiantie moet berekenen. Dit in tegenstelling tot de referentie path tracer, waarbij de werklast voornamelijk wordt bepaald door de hoeveelheid samples die per pixel worden genomen. Dit impliceert dat wanneer we de scène vergroten, de uitvoertijd van het algoritme langer wordt en we meer middelen moeten inzetten om een interactieve framerate te bereiken. Maar dit is precies wat we in de eerste plaats wensten: vanuit het perspectief van een cloud provider is het veel interessanter om rekenkracht te schalen op basis van de scène die moet worden gerenderd, in plaats van de hoeveelheid gebruikers die de scène willen renderen.

# Conclusie en toekomstig werk

Het doel van deze thesis bestond uit twee subdoelen. Ten eerste wilden we het Monte Carlo path tracing algoritme aanpassen om het schaalbaarder te maken voor multi-user toepassingen. Het tweede subdoel was gericht op het verbeteren van de coherentie van Monte Carlo path tracing op GPU. Er werd een algoritme voorgesteld dat zowel rekening houdt met coherentie als schaalbaarheid. De schaalbaarheid van het algoritme werd experimenteel geverifieerd door een vergelijking te maken tussen de totale werklast van het voorgestelde algoritme en die van een referentie GPU-padtracer. De belangrijkste bevinding uit dit experiment is dat, in tegenstelling tot de traditionele aanpak, het voorgestelde algoritme de rekenkracht schaalt op basis van de grootte van de te renderen scène, in plaats van het aantal gebruikers dat de scène wilt renderen.

Ten tweede ondersteunden experimentele resultaten onze hypothese dat het verlagen van de tracing range parameter de geheugencoherentie ten goede zou komen. Kort gezegd, bij de biased aanpak merken we een verbeterde geheugencoherentie op. Daarentegen wordt bij de unbiased aanpak de tracing range ingesteld op oneindig. Dit betekent dat de door de lichtstraal afgelegde afstand niet langer beperkt is, met als gevolg dat de verbetering in geheugencoher-

entie verloren gaat. Desondanks toonden onze experimenten aan dat de unbiased aanpak nog steeds sneller is dan de biased aanpak. De reden hiervoor is dat de biased aanpak een extra render pass moet uitvoeren om de probes bij te werken telkens wanneer de lichtverdeling in de scène verandert, wat simpelweg te duur is. Deze observatie bracht ons op het idee dat *visibility propagatie* interessanter zou kunnen zijn dan *radiantie propagatie* als het gaat om het verbeteren van coherentie. Bij visibility propagatie bouwen we een datastructuur op die discrete visibility informatie bevat (informatie over of twee punten elkaar 'kunnen zien'). Deze data structuur zou mogelijks in verschillende toepassingen kunnen worden gebruikt, waarvan één importance sampling voor indirecte belichting is. Een implementatie voor visibility propagatie wordt beschouwd als toekomstig werk.

# Bibliography


[1] *SIGGRAPH '15: ACM SIGGRAPH 2015 Courses*, New York, NY, USA, 2015. Association for Computing Machinery.

[2] V Aboites. Legendre polynomials: A simple methodology. In *Journal of Physics: Conference Series*, volume 1221, page 012035. IOP Publishing, 2019.

[3] James Arvo and David Kirk. Fast ray tracing by ray classification. *ACM Siggraph Computer Graphics*, 21(4):55–64, 1987.

[4] Krste Asanović, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[5] Rui Bastos, Michael Goslin, and Hansong Zhang. Efficient radiosity rendering using textures and bicubic reconstruction. In *Proceedings of the 1997 symposium on Interactive 3D graphics*, pages 71–ff, 1997.

[6] Anis Benyoub. Leveraging real-time ray tracing to build a hybrid game engine, 2019.

[7] Chakravarty R Alla Chaitanya, Anton S Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn, Derek Nowrouzezahrai, and Timo Aila. Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder. *ACM Transactions on Graphics (TOG)*, 36(4):1–12, 2017.

[8] NVIDIA Developer. Shader execution reordering, whitepaper. 2022.

[9] Frédo Durand, George Drettakis, and Claude Puech. The visibility skeleton: A powerful and efficient multi-purpose global visibility tool. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 89–100, 1997.

[10] Frédo Durand, George Drettakis, and Claude Puech. The 3d visibility complex. *ACM Transactions on Graphics (TOG)*, 21(2):176–206, 2002.

[11] Jerome H Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software (TOMS)*, 3(3):209–226, 1977.

[12] Leslie Glaves. An exploration of the 3d visibility complex. *Master's thesis, Polytechnic University, Brooklyn, NY*, 2007.

[13] Robin Green. Spherical harmonic lighting: The gritty details. In *Archives of the game developers conference*, volume 56, page 4, 2003.

[14] Gene Greger, Peter Shirley, Philip M Hubbard, and Donald P Greenberg. The irradiance volume. *IEEE Computer Graphics and Applications*, 18(2):32–43, 1998.

[15] Toshiya Hachisuka, Shinji Ogaki, and Henrik Wann Jensen. Progressive photon mapping. In *ACM SIGGRAPH Asia 2008 papers*, pages 1–8. 2008.

[16] Henrik Halen and K Hayward. Global illumination based on surfels. In *Proc. ACM SIGGRAPH Symp. Interactive 3D Graph. Games*, pages 1399–1405, 2021.

[17] Henrik Wann Jensen. Importance driven path tracing using the photon map. In *Rendering Techniques' 95: Proceedings of the Eurographics Workshop in Dublin, Ireland, June 12–14, 1995 6*, pages 326–335. Springer, 1995.

[18] James T Kajiya. The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 143–150, 1986.

[19] Anton Kaplanyan and Carsten Dachsbacher. Cascaded light propagation volumes for real-time indirect illumination. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pages 99–107, 2010.

[20] Alexander Keller. Instant radiosity. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 49–56, 1997.

[21] P Kozlowski and T Cheblokov. Relax: A denoiser tailored to work with the restir algorithm. gpu technology conference, 2021.

[22] Sylvain Lefebvre, Samuel Hornus, Fabrice Neyret, et al. Octree textures on the gpu. *GPU gems*, 2:595–613, 2005.

[23] Taweetham Limpanuparb and Josh Milthorpe. Associated legendre polynomials and spherical harmonics computation for chemistry applications. *arXiv preprint arXiv:1410.1748*, 2014.

[24] Edward Liu. Low sample count ray tracing with nvidia's ray tracing denoisers. *Real-Time Ray Tracing, SIGGRAPH Courses*, 2018.

[25] Morgan McGuire, Mike Mara, Derek Nowrouzezahrai, and David Luebke. Real-time global illumination using precomputed light field probes. In *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 1–11, 2017.

[26] Daniel Meister, Shinji Ogaki, Carsten Benthin, Michael J Doyle, Michael Guthe, and Jiří Bittner. A survey on bounding volume hierarchies for ray tracing. In *Computer Graphics Forum*, volume 40, pages 683–712. Wiley Online Library, 2021.

[27] Jesper Mortensen. *Virtual light fields for global illumination in computer graphics*. PhD thesis, UCL (University College London), 2011.

[28] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant neural graphics primitives with a multiresolution hash encoding. *ACM Trans. Graph.*, 41(4):102:1–102:15, July 2022.

[29] Eihachiro Nakamae and Katsumi Tadamura. Photorealism in computer graphics—past and present. *Computers & graphics*, 19(1):119–130, 1995.

[30] Hanspeter Pfister, Matthias Zwicker, Jeroen Van Baar, and Markus Gross. Surfels: Surface elements as rendering primitives. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 335–342, 2000.

[31] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically based rendering: From theory to implementation*. Morgan Kaufmann, 2016.

[32] Ravi Ramamoorthi and Pat Hanrahan. An efficient representation for irradiance environment maps. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 497–500, 2001.

[33] Erik Reinhard, Michael Stark, Peter Shirley, and James Ferwerda. Photographic tone reproduction for digital images. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 267–276, 2002.

[34] Pedro V Sander, John Snyder, Steven J Gortler, and Hugues Hoppe. Texture mapping progressive meshes. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 409–416, 2001.

[35] VV Sanzharov, AI Gorbonosov, VA Frolov, and AG Voloboy. Examination of the nvidia rtx. In *CEUR Workshop Proceedings*, volume 2485, pages 7–12, 2019.

[36] Christoph Schied, Anton Kaplanyan, Chris Wyman, Anjul Patney, Chakravarty R Alla Chaitanya, John Burgess, Shiqiu Liu, Carsten Dachsbacher, Aaron Lefohn, and Marco Salvi. Spatiotemporal variance-guided filtering: real-time reconstruction for path-traced global illumination. In *Proceedings of High Performance Graphics*, pages 1–12. 2017.

[37] Mel Slater. A note on virtual light fields. *University College London*, 2000.

[38] Turner Whitted. An improved illumination model for shaded display. In *Proceedings of the 6th annual conference on Computer graphics and interactive techniques*, page 14, 1979.

[39] Ziro Yamauti. The light flux distribution of a system of interreflecting surfaces. *JOSA*, 13(5):561–571, 1926.

[40] Dmitry Zhdan. *ReBLUR: A Hierarchical Recurrent Denoiser*, pages 823–844. 08 2021.