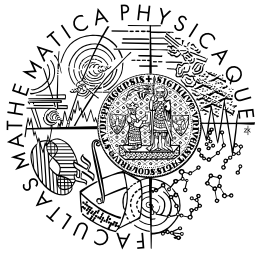


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Dominik Roháček

Improving probes in dynamic diffuse global illumination

Department of Software and Computer Science Education

Supervisor of the master thesis: Mgr. Tomáš Iser

Study programme: Computer science

Study branch: Computer Graphics and Game Development

Prague 2022

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to express deep gratitude to my thesis supervisor Mgr. Tomáš Iser for trust in my project and valuable pieces of advice. I would also like to express sincere thankfulness towards Erik Veselý for letting me work on the Fibix engine, for the numerous discussions we have had and for pushing me into more rigorous research in the area.

Attending the lectures of many teachers on MFF who lightened my interest in the field of computer graphics was a great honour. Especially doc. Jaroslav Křivánek and Dr Alexander Wilkie, with their enthusiasm for the field, inspired me to continue even in times of need.

My thanks go to Anna for her support and love during this work. Furthermore, I would like to thank my family for their support and my grandfather for always asking me about my studies even though every time I try to explain, it is hard for him to imagine why I am using all the strange words.

Thank you

Title: Improving probes in dynamic diffuse global illumination

Author: Bc. Dominik Roháček

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Tomáš Iser, Department of Software and Computer Science Education

Abstract: For a long time, real-time renderers typically only supported direct illumination. With recent technological advances, such as much faster GPU computations or the RTX platform, simulating more accurate global illumination in real time is now possible. This is especially important for rendering indoor scenes in the context of architectural visualization, as users can now add and modify illuminants in real time without waiting for a fully path-traced render.

In this thesis, we briefly describe the existing real-time solutions and investigate the Dynamic Diffuse Global Illumination technique in detail. We implement the solution to an existing real-time renderer with RTX support. We specifically describe several problems and artefacts that the method has and present our solutions to those. Mainly, we show an improved approach to probe placing and investigate the improvement it provides. We note that our implementation achieves a better visual quality as it avoids highly noticeable artefacts.

Keywords: global illumination real-time rendering indirect lighting

Contents

Introduction **3**
Motivation . . . 3
Overview of the Work . . . 3
Thesis outline . . . 4

1 Fundamentals **5**
1.1 Radiometric quantities . . . 5
1.2 Ray tracing . . . 6
1.2.1 Path tracing . . . 7
1.3 Nvidia GeForce RTX . . . 7
1.3.1 RT Cores . . . 8
1.3.2 Fibix engine . . . 8
1.4 Global Illumination . . . 9
1.5 Irradiation probes . . . 10
1.6 Texture atlas . . . 10

2 Related work **12**
2.1 Light mapping . . . 12
2.2 Sparse Voxel Octree Total Illumination . . . 13
2.3 Signed Distance Fields Dynamic Diffuse Global Illumination . . . 15
2.4 Global Illumination Based on Surfels . . . 16
2.5 Dynamic Diffuse Global Illumination . . . 18
2.5.1 Probe representation . . . 19
2.5.2 Probe update . . . 20
2.5.3 Indirect light sampling . . . 21

3 Implementation **23**
3.1 DirectX Raytracing . . . 23
3.1.1 Ray generation shader . . . 24
3.1.2 Any hit shader . . . 24
3.1.3 Closest hit shader . . . 25
3.1.4 Miss shader . . . 26
3.1.5 Intersection shader . . . 26
3.1.6 Callable shader . . . 26
3.1.7 Payload structure . . . 27
3.2 Dynamic Diffuse Global Illumination implementation . . . 27
3.2.1 Data storage . . . 27
3.2.2 Probe data . . . 28
3.2.3 Probe grid . . . 30
3.2.4 Smooth backface culling . . . 30
3.2.5 Ray-tracing . . . 31
3.2.6 Dircetion generation . . . 31
3.2.7 Indirect lighting sampling . . . 33
3.3 Problems . . . 34

3.3.1 Dead probes . . . 34
3.3.2 Depth samples rejection . . . 38
3.4 Controls . . . 38

4 Results **43**
4.1 Per-probe memory requirement . . . 43
4.2 Reduction of visual errors . . . 44
4.2.1 Dead probe artefact . . . 44
4.3 Depth maps resolution . . . 47
4.4 Light bleeding . . . 47
4.5 Maps resolution . . . 48
4.6 Video stability . . . 50
4.7 Times per algorithm stages . . . 50
4.7.1 Per probe rays time influence . . . 51
4.7.2 Depth resolution time influence . . . 52
4.8 Indirect lighting . . . 53

5 Conclusion **55**
5.1 Future work . . . 56

Bibliography **57**

List of Figures **60**

List of Tables **62**

List of Listings **63**

List of Abbreviations **64**

A Appendix A **65**

Introduction

This thesis aims to create and improve a complete real-time global illumination solution for a real-time renderer without the need for offline precomputation. Our goal is to support features such as time of day, dynamic geometry, animated scene and accurate light transport.

Motivation

Global illumination systems have been part of most game engines for years. Previously, global illumination was often pre-baked into lightmaps, but this time and the hardware-demanding process is less and less desirable. The industry is shifting more and more towards real-time dynamic global illumination. Those approaches allow developers to set time of day in their games with responsible lighting and shade their scenes more accurately and with the dynamic objects taken in accord.

This thesis builds on the work of Majercik et al. [2019] on Dynamic Diffuse Global Illumination (DDGI). This work utilises RTX technology introduced by NVidia. RTX technology allows developers to use real-time ray tracing natively supported by new NVidia cards from RTX 2000 and lately RTX 3000 series.

In the first chapter, we introduce the theoretical background of DDGI and all techniques used to implement this technique. Also, we will introduce the environment in which the thesis is implemented, called the Fibix engine. Using this production-ready engine allowed us to focus on the DDGI implementation and improvement.

Overview of the Work

The work presented in this thesis focuses on the development of a real-time global illumination solution for the existing engine. Our solutions uses previous work of Majercik et al. [2019] and futher improves it. The main goal is to allow users of the engine to set up a scene and lighting without distracting users by forcing them to adjust the properties of the system. The work-out-of-the-box property was crucial as the customers mainly use the engine to set up scenes in real-time view. They set up lighting, camera animation, scene animation and materials. The ability to see as accurate lighting as possible in this stage of work is crucial. The user's goal is to run offline raytracer on prepared scenes. Offline render usually takes a long time, and the ability to see lighting in scenes in advance could save considerable time.

Later the thesis introduces existing solutions to real-time global illumination. We have selected one of them and showed how we implemented it into the Fibix Engine. We also propose, implement and measure improvements to this solution. In the last section, this thesis provides results and measurements that show the benefits of our contribution.

Thesis outline

This thesis is structured as follows:

- **Fundamentals** We briefly examine the fundamentals of used technologies, including ray tracing, NVidia RTX and real-time ray tracing.
- **Related work** In Chapter 2 we will go through state-of-the-art global illumination solutions.
- **Implementation** We devote Chapter 3 to briefly describe the implementation of DDGI and improvements we made to the algorithm to fit the desired usage and scenario.
- **Results** In this chapter, We will provide measurements of our implementation of DDGI and analyze the results. This chapter also compares the ratio between memory, computation time, and accuracy that each technique achieved.
- **Conclusion** The last chapter discusses the achieved goals and proposes a future direction for improvements in the area.

1. Fundamentals

In this chapter, we walk the reader through the fundamentals needed to build a basis for further explaining the topic of this thesis. We start with ray tracing and path tracing. Then, we give an introduction to the RTX platform that our algorithm requires. In the end, we will discuss the topic of global illumination.

1.1 Radiometric quantities

The goal of this section to save readers confusion further on. We want to explain just two radiometric quantities, which are tightly connected.

The first quantity we would like to mention here is called *radiance*. This quantity describes the radiant flux emitted, reflected, transmitted or received by a given surface at a given location x to the solid angle ω. The Φ in this equation is radiant flux, an amount of energy per unit of time. The image 1.1 better explains the angles. Because the *radiance* is measured against a plane orthogonal to the given direction, we have to account for the projection. To include this projection, we need to use $\cos\theta$ factor.

$$L(x,\omega) = \frac{\mathrm{d}^2\Phi}{\cos\theta \mathrm{d}A\mathrm{d}\omega} \qquad [\mathrm{Wm}^{-2}\mathrm{sr}^{-1}] \quad (1.1)$$

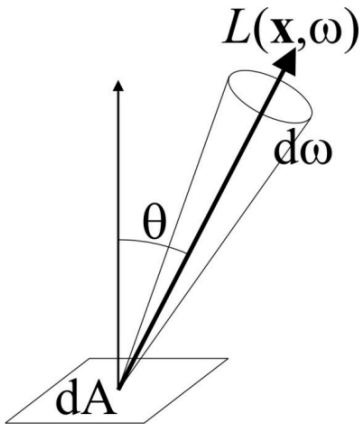


Figure 1.1: Visual explanation of the radiance. (Image source: Jaroslav Křivánek)

Irradiance is defined as radiant flux received by a surface at a point x per area. This concerns all the flux received from all directions.

$$E(x) = \frac{\mathrm{d}\Phi}{\mathrm{d}S} \qquad [\mathrm{Wm}^{-2}] \quad (1.2)$$

Both quantities are essential for this thesis, as this thesis will refer to them through this text and understanding the difference between them is crucial for the comprehension of parts of the algorithm.

The last note on this topic goes to irradiance calculation from a radiance. The Equation 1.3 formally describes the calculation.

$$E(x) = \int_{H(x)} L(x, \omega) \cos\theta \, \mathrm{d}\omega \tag{1.3}$$

The relation between the two quantities is evident from their definitions and their units. This integration considers all radiance incoming to the given point x from all the directions. Furthermore, those directions could only lie in the upper hemisphere; thus, we integrate over $H(x)$ denoting hemisphere around the point x. Also, we need to consider the mutual orientation of the rays and the surface. This is the reason why this equation uses the factor of $\cos\theta$. The cosine factor comes from well-researched work by Lambert [2001].

1.2 Ray tracing

The term ray tracing describes the algorithm where we cast rays into a 3D scene and trace them to find where they intersect scene geometry. This procedure is used in many applications, not only for image generation but also physics simulations. However, in the context of this thesis, we will focus only on ray tracing used for light transport.

The idea of this technique is to mimic the simplified behaviour of light. The light travels from light sources such as lights, sun, stars or hot enough objects in the form of electromagnetic waves. Camera sensors or our eyes then interpret different wavelengths of those waves as different colours. Thanks to wave-particle duality, we can view those waves also as particles. As Einstein said: "But what is light really? Is it a wave or a shower of photons? ... It seems as though we must use sometimes the one theory and sometimes the other, while at times we may use either." Einstein and Infeld [1961]. In this situation, it is usually more suitable for our needs to think of light as the cluster of particles travelling along with the rays in the form of energy. We refer to this energy as radiance.

Tracing each ray from the light source is unfortunately very inefficient as rays can end up going anywhere, but we are interested only in such rays that hit our eye or cameras sensor. Here we can make use of Helmholtz's reciprocity principle. Clarke and Parry [1985] This principle allows us to reverse our algorithm and trace rays starting from the camera or eye and tracing them back to the light sources. The simplest example of this is shown in Figure 1.2.

When the ray is cast from the camera, the algorithm tests the ray against the 3D scene representation for an intersection. As this procedure is computationally expensive, acceleration structures are usually employed in the process. From the most common one, we would like to mention k-d trees (Moore [1991]) and Bounding Volume Hierarchy (BVH) Meister et al. [2021]. Especially the latter is heavily used to allow ray tracing in real-time, as explained later in subsection 1.3.1.

Once the algorithm finds the nearest intersection, the "event" has to be processed. The ray is reflected or refracted based on the surface's properties. How much radiance the surface reflects in each direction is commonly modeled by a Bidirectional reflectance distribution function (BRDF). Unfortunately, only with a perfect mirror we can say that all photons will travel in one direction each time. So at each intersection, our ray has to split. For best results, the new

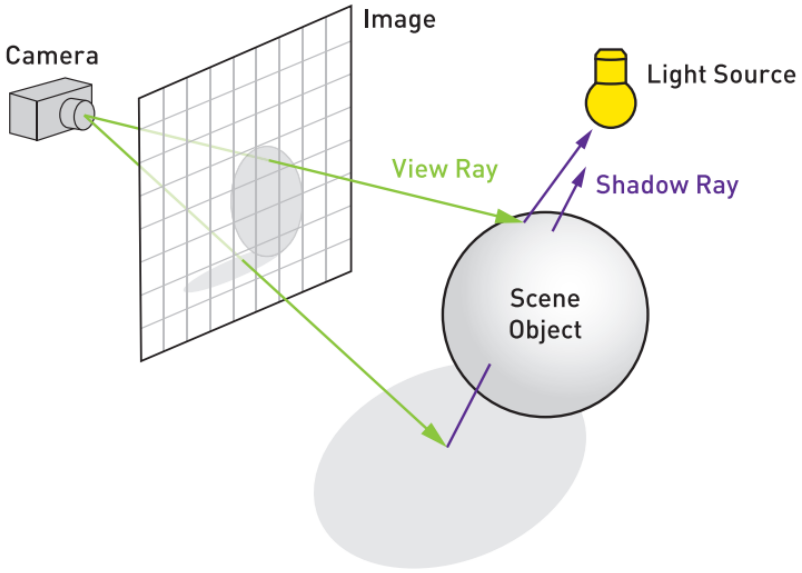


Figure 1.2: Ray casting technique. This picture shows how the algorithm generates the final image. (Image source: Haines and Akenine-Möller [2019])

directions should be sampled proportionally to BRDF, which is called importance sampling, but other direction sampling strategies work as well, but with a higher variance. This increases the number of rays exponentially with each intersection.

Although ray tracing is a computationally heavy approach for image synthesis, it gives us naturally many effects that are hard to achieve with other algorithms such as rasterization.

1.2.1 Path tracing

Path tracing is a type of ray tracing. It is a Monte Carlo approach to solve the rendering equation. Similarly to the ray-tracing described earlier, the rays are traced from the camera. Instead of branching the path at each intersection we choose one random direction from that point. A surface reflectance function then reduces the arriving illuminance from that direction.

1.3 Nvidia GeForce RTX

With the RTX 2000 series of graphics cards, NVidia introduced a new high-end computing platform [NVIDIA Corporation]. This technology was introduced together with the Turing architecture. There are four main areas of core technologies supported by this platform. Starting with simulations powered by CUDA cores through the PhysX, Flow, Flex APIs or directly Compute Unified Device Architecture (CUDA), you can achieve high-performance physical simulation. The second area of interest of the RTX platform is Artificial Intelligence (AI). The artificial intelligence of the RTX platform is aimed mainly towards visual computing and AI-augmented workflow. AI was allowed by the introduction of novel hardware units called tensor cores. Another part of RTX aims at rasterisation. In this area, RTX features improvements in the programable shading pipeline. To

enumerate advantages in the field rasterisation, we can name variable-rate shading, texture-space shading or multi-view rendering. The last one is an answer to the needs in the growing area of virtual reality. The last area, and the area this thesis is concerned with the most, is real-time ray tracing.

RTX ray-tracing brings typically offline techniques of ray-tracing into the real-time workflow. The RT Cores units allow that by accelerating ray intersection calculation.

NVidia introduced the RTX platform with the RTX 2000 series in 2018. The list of cards with RTX support enabled has increased since then. From the retail cards, there are already two series cards with RTX. Namely, we are speaking about RTX 2000 and RTX 3000 series. From professional cards there are RTX 5000, RTX 6000, RTX 8000, RTX A2000, RTX A3000, RTX A4000, RTX A4500, RTX A5000 and RTX A6000. The RTX platform has become the standard for professional cards. On the retail side, NVidia released GTX 16 series based on the Turing architecture but misses RT Cores and aims at entry-level to midrange customers. NVidia later announced RTX platform subset support for some GTX 16 series cards despite the lack of RT Cores and Tensor cores.

The RTX is officially fully supported on retail cards built on Turing, Ampere and Lovelace architecture. As you can see from the extent of support of RTX technology, it is realistic to expect that customers will have such a card available, especially as Fibix Engine aims toward architectural visualisations where professional cards can be expected.

1.3.1 RT Cores

RT Cores are specialised accelerator units included inside streaming multiprocessors. As described earlier in section 1.2 in the core of each ray tracing technology needs to be an intersection solver preferably employing some acceleration structure. And this is exactly where RT Cores comes to play. RT Cores are optimised for BVH traversal and ray versus triangle intersection. This way, RT Cores can perform visibility test on-demand from newly introduced shader types running on streaming multiprocessors.

The BVH traversal on RT Cores is autonomous and can be affected by programmers only at strictly defined control points. The BVH structure construction and refitting are handled purely by the driver. Figure 1.3 shows the architecture of the RT Cores.

At the current state of the technology, the two main APIs that support the described functionality are Microsoft's DirectX Raytracing API that is extending DirectX 12 and NVidia VKRay extension for Vulkan. In this work, we will employ DXR because at the time we have started this work DXR has better support and the Fibix engine already had DirectX 12 and DXR support.

NVidia provides more details on their website Emmett Kilgariff and Bell.

1.3.2 Fibix engine

The Fibix engine is a product of Fibix studio Fibix Studio. It is a rendering software, which specializes in architectural visualizations. This engine combines

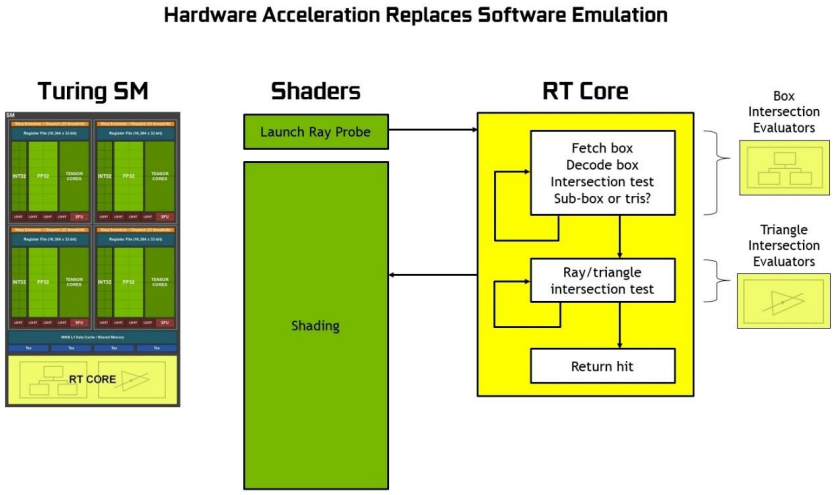


Figure 1.3: Turing Ray Tracing with RT Cores. (Image source: Emmett Kilgariff and Bell)

high-quality real-time visualization of the scene and accurate physically-based offline renderer for architectural visualizations.

We have chosen this engine as a base for the research. This choice provides us with all the functionality of constructing a scene, loading all the resources and other parts of the engine needed to implement a global illumination algorithm but not necessarily related to the core of our research. This way, we can focus more on the algorithm we are trying to improve than on technical stuff unrelated to the research.

1.4 Global Illumination

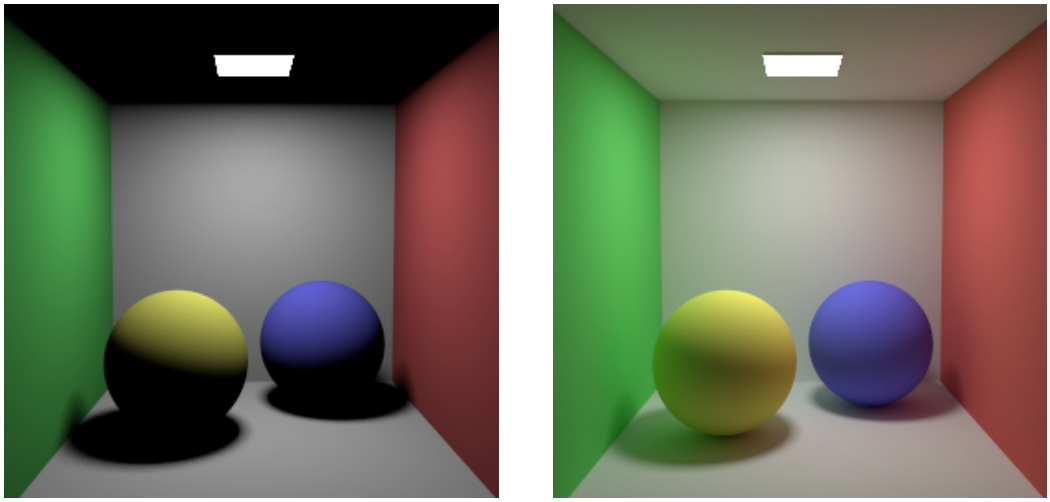
Under the term Global Illumination we understand algorithms that aim towards the realistic calculation of indirect light in the virtual scenes Dutré et al. [2006]. This account for both direct and indirect lighting. Direct lighting is simply the lighting that happens without any extra bounces between the light source and the surface where we observe the lighting. Any ray-tracing or path-tracing renderer that can simulate indirect light can also be used to compute direct lighting by simply limiting the maximum number of bounces to one.

Indirect lighting is a type of lighting that happens through multiple reflections, refraction, scattering, dispersion or other phenomena. This lighting is way harder to simulate as any light path between the camera and light source can bounce an indefinite number of times. The algorithm for accurate simulation of indirect lighting typically needs to have the full context of the scene geometry for the shading of each pixel in the final image. Ray tracing and path tracing are examples of algorithms capable of indirect lighting simulation. You can see the difference between direct-only and gobal (both direct and indirect) light simulation in Figure 1.4.

In real-time graphics, we usually need to use a more straightforward solution that simulates only parts of the indirect lighting or approximates it. As the capability of graphics cards increased, games started to use more and more complicated and more accurate solutions. Notwithstanding, we can call the shadow

mapping ([Roháček, 2018]) GI method as it simulates part of indirect lighting, which is not enough in terms of modern real-time rendering.

This creates a need for new and inventive ways to calculate GI in real time. The next chapter is dedicated to introducing the current state of the art of real-time GI methods.



(a) Only direct ligthing (b) Global illumination

Figure 1.4: This is an example of the difference between direct lighting only and full global illumination inside the Cornell box. (Image source: Jaroslav Křivánek, Supplemental resources for MFF CUNI course Computer Graphics III - Realistic image synthesis (NPGR010))

1.5 Irradiation probes

Many GI relies on some for of irradiance caching. Irradiance probe is one of the data structures used for irradiance caching in computer graphics. The probe stores prefiltered cache of irradiance for each direction. Key ability of this strucure is to be able to return irradiance for given direction quickly. In real-time graphics the irradiance probes are usually used for indirect lighting during the final shading stage [Landis, 2002].

There are many way we can store irradiance data. Most commonly used are cube maps and spherical harmonics.

1.6 Texture atlas

In computer graphics, we store image data in textures. To map those textures onto the 3D model, we use UV mapping. The texture space is defined with UV coordinates. Each vertex needs to define a UV coordinate to map the texture to the model. This coordinate defines how to project the polygon defined by vertices into the UV space. The definition of such coordinates for a model is called UV unwrapping. We usually need to split a model along some edges to unwrap a model. We can split the model into separate patches in UV space for simpler

unwrapping. By doing so, we create multiple islands of texture in UV space. We also can unwrap multiple models into one texture. The texture now can contain multiple images [Maillot et al., 1993].

The texture that contains multiple images is called *texture atlas*. The most straightforward texture division to an atlas is creating rectangular regions, each containing one image. This thesis will use this technique. An exaple of such atlas is Figure 1.5.

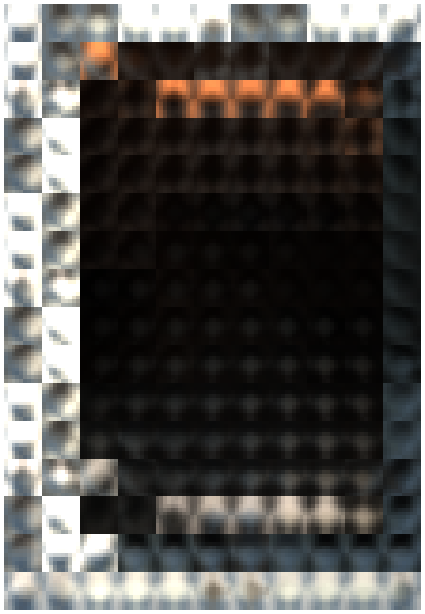


Figure 1.5: Texture atlas divided in a squares each representing one irradinace probe. (Image source: Thesis author)

2. Related work

This chapter provides an overview of previous work in the field of real-time global illumination.

2.1 Light mapping

Light mapping algorithms are a group of techniques whose idea is based on a technique called Photon mapping by Jensen [1996]. The photon mapping algorithm is a two-pass algorithm, and initially, it has been used to improve the performance of ray tracing.

Photon mapping uses a cache called *photon map*. This map is built during the first pass when rays are traced from the light sources and checked for intersection with the scene. When a "photon" intersects geometry, it is stored in a photon map. After storing, each photon is reflected, refracted or absorbed depending on the type of material and BRDF function that is being used. This algorithm builds a photon map that stores the distribution of stores the distribution of light energy across the scene. The "photons" in the context of this technique are rather clusters of energy than photon particles we know from physics.

In the second pass, rays are traced from the camera, and when they intersect with the scene, the photon map is used to calculate irradiance as interpolation of a predefined number of nearest photon samples.

Figure 2.1: Light mapping allows rendering of complex lighting phenomena such as caustics. (Jensen [1996])

Similar techniques are being used in the real-time rendering, where the main benefit of such techniques is decoupling the first calculation-heavy step from the second one. Those algorithms are called Light mapping, and usually, the

photon map is calculated on a server farm (cluster) before the game is shipped to customers. The second pass in this scenario is not using ray tracing; instead, the lightmaps are used in pixel shaders to calculate indirect lighting.

This technique allowed the rendering of complex phenomena such as caustics or reflections in real-time graphics. Unfortunately, the need for time-consuming precomputation of photon maps is a significant disadvantage of this technique.

The result of this technique is shown in Figure 2.1.

2.2 Sparse Voxel Octree Total Illumination

Sparse Voxel Octree Total Illumination (SVOTi) is voxel-based real-time global illumination introduced by Crassin et al. [2011] and improved in Sugihara et al. [2014]. Some sources also call this technique Sparse Voxel Octree Global Illumination (SVOGi). As the name suggests, the technique is based on a voxel octree. In the following paragraphs, we will break the process down and simplify it.

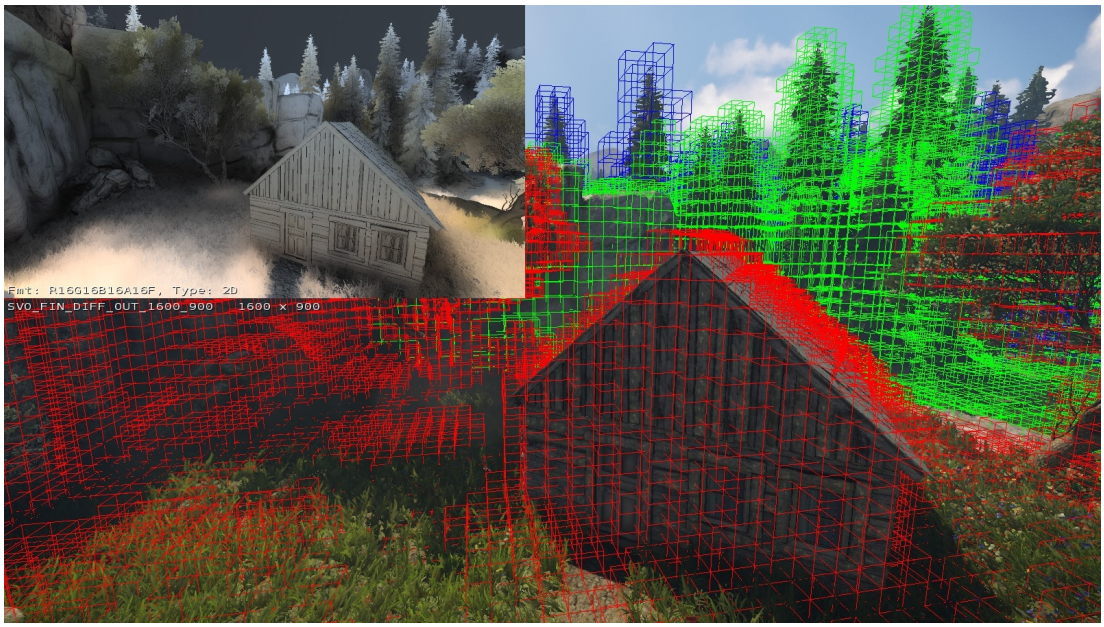


Figure 2.2: World voxelized for SVOTi with cascades color coded by voxel outline [Uyurkulak, 2021].

The first step of SVOTi is a discretization of the geometry into a sparse voxel tree data structure. Sparsity in this structure solves the problem of vast parts of empty spaces that would otherwise consume memory without any use. This voxel tree is stored in two 3D textures. One represents inner nodes, and the second one contains data for leaf nodes. The tree texture contains 2x2x2 bricks where each one represents one inner node with information about minimal and maximal coordinates of an Axis Aligned Bounding Box (AABB), eight slots to point to the child nodes and also information needed for proper streaming as the whole structure is being streamed in.

The leaf nodes represent the geometry itself. The primary information about geometry topology in each voxel is opacity along the cardinal axes. The reflectance information is separated into specular, diffuse and emitted colours. The

voxels also store information about the normal vector and the main incoming direction of the incoming ambient light.

The algorithm also uses a similar approach for optimizing screen space to memory consumption ratio, as we know from cascade shadow maps. This comes from the simple idea that one pixel across in screen space represents a more significant change in position in world space than a pixel containing a projection of a geometry nearer to the camera. This is discussed more in-depth in the paper by Xu and Lun [2010]. Visualization of the cascades in voxel tree fineness is visible in figure 2.2.

During the runtime, the first step of this algorithm is to render Layered Reflective Shadow Maps (LRSM) by Sugihara et al. [2014] that are used for sun shadows. Data from those shadow maps are then injected into the reflectance data in leaf nodes of the Voxel octree.

The algorithm starts with light propagation after the direct light injection into the voxel data structure. To achieve that, the cone tracing algorithm is used. The sparse voxel octree is well suited for this, and different octree levels could be used at different distances. When the cone hits opaque voxel, the opacity is saved for a hit, but the cone can continue until it is completely occluded. The cone tracing accumulates reflected radiance along its path, and this is propagated back. You can see those steps in the figure 2.3

Figure 2.3: Albedo colours, directl light injection and light propagation. (Uyurkulak [2021])

The cone tracing occurs on a small resolution viewport image. This has to be followed by image demosaic and upscaling steps.

Unfortunately, this technique introduces several problems. The most noticeable one is light leaking. The nature of the geometry representation introduces this error. For example, a thick wall aligned with a cardinal axis could leak the light. In such a scenario, when a ray or, in this case, cone travels along the wall, it can travel through the wall as the voxel could be partially transparent for the algorithm along the axis aligned with the wall.

Another problem arises from the temporal filtering involved in the algorithm that can cause phenomena called ghosting. Another problem could be a visible blockiness introduced by the upscaling step.

2.3 Signed Distance Fields Dynamic Diffuse Global Illumination

This technique by Hu et al. [2020] is also based on irradiance probes (Landis [2002] and Fernando [2004]) and is inspired by Dynamic Diffuse Global Illumination (DDGI) (section 2.5). The advantage of this technique is that it does not require RTX cards and, as the authors claim, should scale better for low-end hardware.

The first step of this technique is the calculation of Signed distnace field (SDF) Hart [1996] representation of the scene and its clustering into acceleration structure. The next step is probe placement. This is done using the SDF information to place probes. When SDF at probe location is smaller than the threshold, the gradient of SDF and by use of gradient descent method is moved to the acceptable sampling point.

The probe is updated after those preparation steps. From a given probe, random directions are being chosen to trace a ray using the clustered acceleration structure constructed in the first step. For each intersection point, radiance is calculated from Reflective Shadow Map (RSM) (Dachsbacher and Stamminger [2005]) and emission value that is stored inside the primitives for area light and self-emission support. This way algorithm calculates one bounce of ray in the scene. The consecutive bounces are calculated automatically because the irradiance is propagated through RSM as it uses the probes during the flux calculation.

To sample data acquired by the process described above, we first need to determine which probes to use. Similarly to DDGI this technique interpolates eight probes around the shaded point. On the other hand, unlike the DDGI this algorithm does not store depth information in the probes—instead, it uses SDF representation of the world.

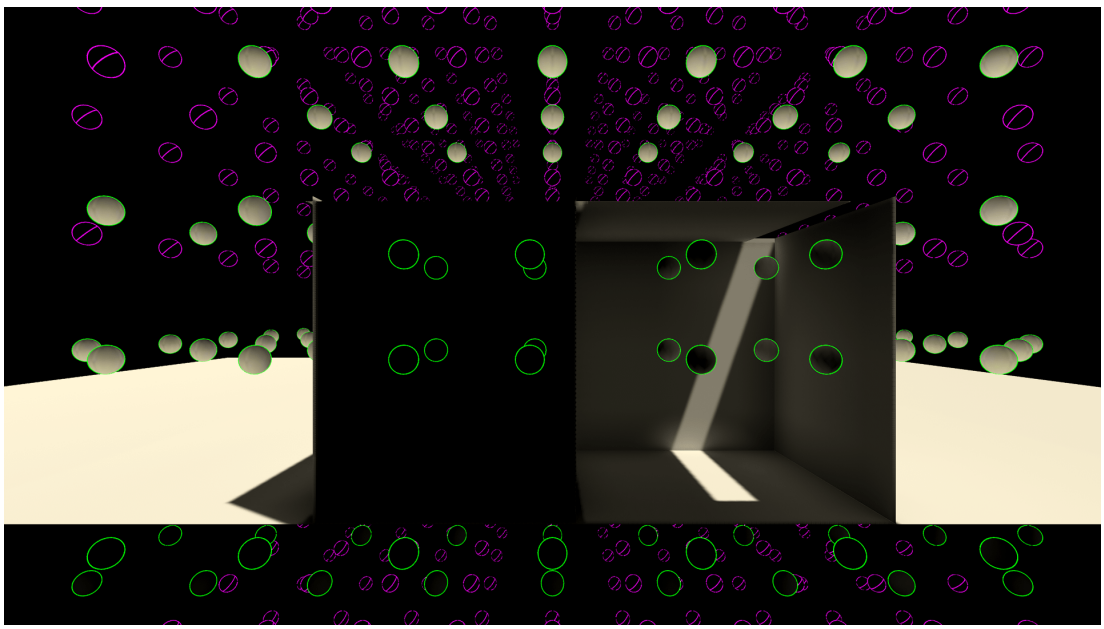


Figure 2.4: Signed distnace field Global Illumination uses also a grid of probes placed inside the scene. (Jensen [1996])

2.4 Global Illumination Based on Surfels

This technique by Stachowiak [2018] and further improved by Halen et al. [2021] is based on so-called surfels. Surfels are a way of discretization of the surface the same way we discretize images into pixels. The technique is using surfels for irradiance caching in the scene.

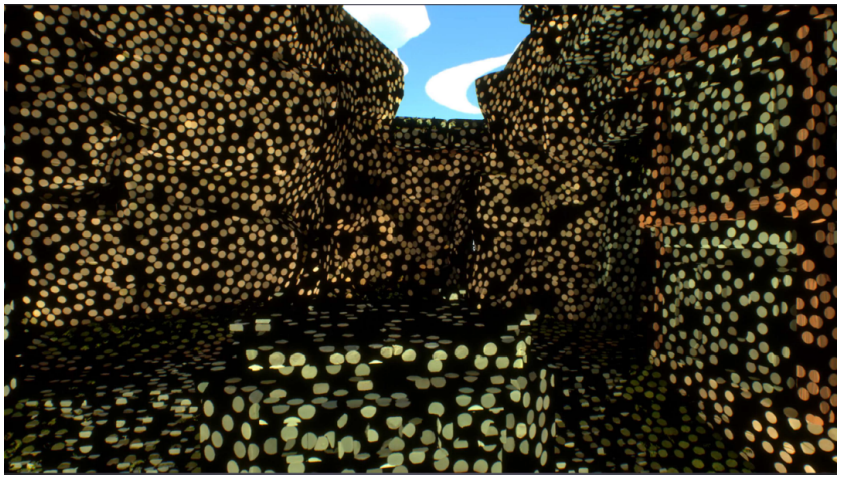


Figure 2.5: Discretization of the surface using surfels. (Halen et al. [2021])

First, we would like to explain the process of surfelization of the scene. What the technique is trying to achieve is to have enough surfels across the viewing frustum to restore indirect lighting in the final image credibly. For large scenes, it is impossible to have a dense system of surfels. Also, to be able to produce good GI in a dynamic scene, it is crucial to be able to do this process at runtime. For this purpose Stachowiak [2018] introduced an approach that uniformly covers the screen space with surfels that covers approximately uniformly. It uses a grid where each tile covers 16x16 pixels of screen space as shown in figure 2.6. The algorithm iterates over this grid and tries to cover gaps between surfels. This process is stochastic. This means that when a gap is found, it uses a threshold to determine whether to spawn a new surfel in this space. Also, to cover the screen space uniformly, it uses bigger surfels for distant surfaces and small for nearer geometry. When the camera moves towards a surfel, surfel shrinks, and this way, more surfels can be spawned on the given geometry which increases the resolution.

The algorithm also uses fixed budgeting for a whole scene and heuristics to despawn no longer needed surfels. By despawning a surfel, we are also throwing away the work. For that reason, the heuristic consists of the distance from the camera and time since the last usage of surfel for shading. This algorithm was further improved by Halen et al. [2021].

The surfels are spawned on the geometry. But in the case of skinned or animated surfaces, they need to be moved. This is accomplished by tracking a unique transformation identifier of their geometry. This also allows incorporating dynamic and skinned models and support correct indirect and emitted lighting on them.

The illumination for surfels is determined through the ray tracing algorithm. Several rays are shot each frame randomly from each surfel. When a ray misses the

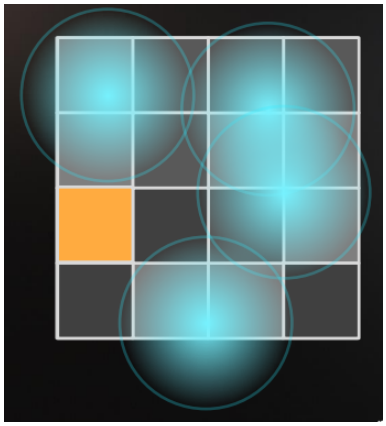


Figure 2.6: Grid used for uniform coverage of screen space. (Stachowiak [2018])

scene geometry, a skylight is evaluated. When a ray hits a geometry, the direct lighting is evaluated. Occlusion rays are used to achieve that, and if nothing occludes light from the surface point, then BRDF is evaluated as described in the first chapter. To optimize this process, the existing surfels around the contact point are used. The surfels had also accumulated irradiance so they could be reused. This way, the ray tracer does not need to follow long paths for each ray, and yet virtually infinite bounces could be achieved over time.

When using the surfels for shading, light bleeding could occur, especially when surfels are placed near thin geometry, which has high irradiance on one side and low on the other. To mitigate this Halen et al. [2021] propose the construction of depth function around each surfel. This depth function is initialized to the radius of the surfel. Each time a ray is shot from this surfel, and it ends up hitting geometry after a shorter distance than the radius of the surfel, the depth function is being updated.

The depth function is stored in the form of moving average of depth, and moving average of depths squared. When a fragment is being shaded we use the surfel the Chebyshev's inequality to detect occluded surfels [Chebyshev, 1867].

As surfels are being spawned using GBuffer data, it is not possible to consistently spawn surfels on transparent surfaces. Halen et al. [2021] proposes to use light probes instead for such situations.

The interesting part we would like to mention here is the use of importance sampling in the ray tracing to help the algorithm better utilize the given budget of rays. The algorithm is based on Müller et al. [2017] paper on path guiding. Instead of building quadtree, the iteration utilizes an octahedron representation (described in subsection 2.5.1) of the hemisphere above the surfel.

We can use inverse of the Cumulative Distribution Function (cdf) to generate samples according to Probability density function. First we need to draw random variable from uniform distribution between 0 and 1 and multiply it by the integral of function. Because our function is descrete we can replace the integral by the sum of the values sum(f). Trashold could be calcuated as $y = \text{rnd}(0,1) * \text{sum}(f)$ where rnd$(0,1)$ represents uniformly random distribution on the interval $< 0; 1 >$. Now we walk the function and accumulate the values as shown in Figure 2.7. Our pdf is 2D, but that does not present any problem as we can walk it, e.g. by rows. Once our accumulated value exceeds threshold we have our importance sampled

value proportional to its value. This way algorithm guides rays more toward the parts of the scene where most of the light comes from.

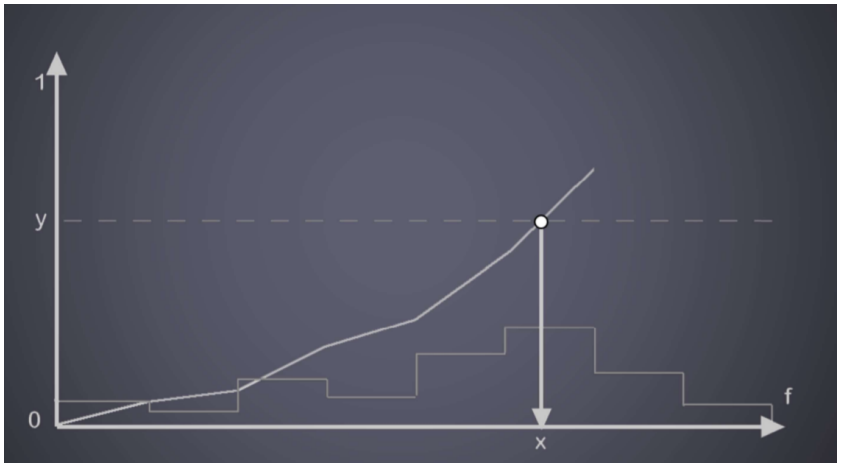


Figure 2.7: Importance sampling using inverse Cumulative Distribution Function. (Image source: Halen et al. [2021])

Another nice trick is budgeting according to the variance instead of using a fixed blend factor used in moving the average estimator. This blend factor defines whether the update will be reacting fast to the changes or whether the estimator will converge better over time. This approach is used in Dynamic Diffuse Global Illumination. Instead Halen et al. [2021] proposes a modified version where beside the long-term mean irradiance, the surfel also stores a short-term mean and variance. Those two values are used as blend factors for the long-term estimator. This allows fast reactions for changes in a scene and fast estimation of irradiance for new surfels spawned by the algorithm and reduces noise in the long term. This approach is more discussed in-depth by Barré-Brisebois et al. [2019]. The same values are used for ray budgeting. Surfels with higher variance receive more rays per frame for faster convergence to the correct irradiance and, as the product of this, lower the variance moving average.

As you can see from the previous explanation, the strength of this method is that it does not involve any offline precomputations. Another advantage of this method is that it supports dynamic and destructible scenes. This is possible thanks to dynamic surfelization and built-in budgeting. Unfortunately, this solution is not fit for our implementation in the Fibix engine as we aim to support transparent surfaces.

2.5 Dynamic Diffuse Global Illumination

Dynamic Diffuse Global Illumination is a full real-time global illumination solution based on a uniform grid of irradiance probes as seen in Figure 2.8. This solution promises an artefact-free method with no need for manual placement of probes in contrast to previous probe-based solutions. This was the main requirement for the solution for this thesis, and this is the reason we have chosen this one to explore further.

This technique utilizes RTX technology described in section 1.3 for Ray tracing as discussed in section 1.2. This limits usage on RTX enabled graphics cards, but

this was not considered a problem as the current workflow in the Fibix engine already contains RTX technology. This is why we will explain this technology more in-depth than previous ones.

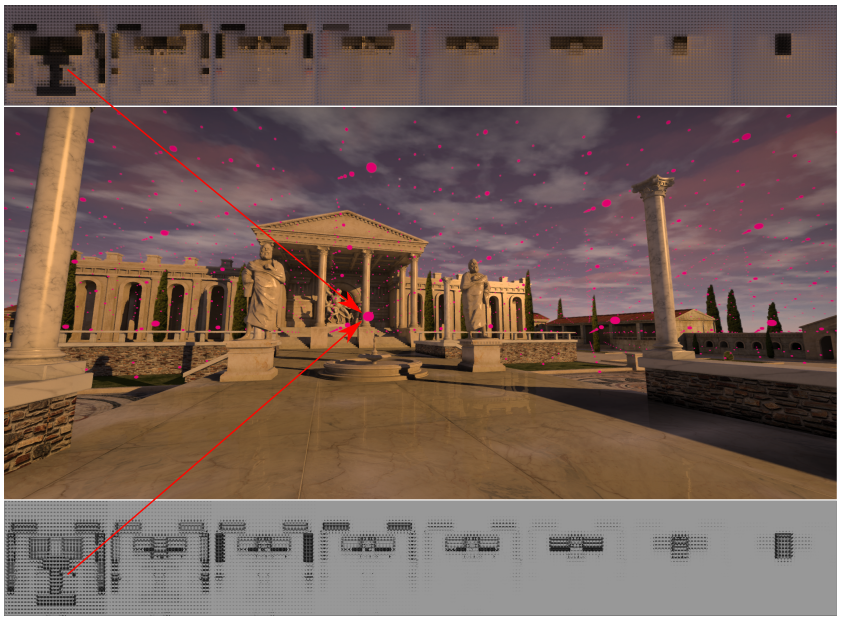


Figure 2.8: This figure shows the data stored for Dynamic Diffuse Global Illumination and placement of probes on uniform grid in a test scene. Image by Majercik et al. [2019]

2.5.1 Probe representation

As DDGI depends heavily on a great number of probes and also on the ability to update them continuously, we need to use a convenient representation of the probe data that fulfils those requirements. The most commonly used one is representation using spherical harmonics [Ramamoorthi and Hanrahan, 2001]. Unfortunately, this representation is not easily updatable [Green, 2003]. Instead, we will use one described by Cigolle et al. [2014] where we will map the sphere on an octahedron and then we will unwrap this octahedron into the UV space. 2.9

Gutter region

Described representation has to be sampled during the final shading of the scene. Because of the resolution of each probe, we need to use hardware-accelerated filtering. Bilinear filtering uses a 2x2 texel sampling region and applies bilinear filtering Sa [2014]. This would produce undesired artefacts near probe borders by sampling from adjacent probes. We can avoid this problem by introducing ”gutter regions”. We only need a one-pixel border around each probe for the given sampling area. In this region, we need to save the same values as the area that will be ”stitched” to this border in the final mapping.

The original paper also proposes additional padding around the image to align each probe top-left pixel to the 4x4 write boundaries.

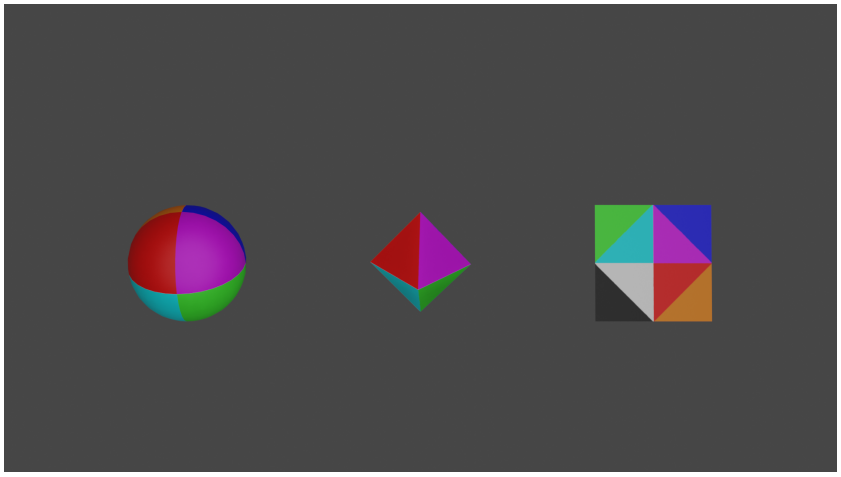


Figure 2.9: Process of unit vector mapping

2.5.2 Probe update

From the ray tracing, we received the nearest geometry's depth from the probe's view and radiance coming from the given direction. Now we need to update the data in texture atlases accordingly. The first problem is that we acquired radiance, but the probe atlas stores irradiance. As we explained in section 1.1 in order to calculate irradiance from radiance, we need to integrate over the hemisphere.

$$E(x) = \int_{H(x)} L(x, \omega) \cos\theta \, \mathrm{d}\omega \tag{2.1}$$

For our probe, this means that we need for each direction of our probe to integrate the whole upper hemisphere. As you can imagine, we can reuse radiance samples from one direction in a half hemisphere of the probe.

Our estimator is following.

$$E(p) = \frac{2\pi}{N} \sum_{i=1}^{N} L_i(p, X_i) * \max(\cos\theta, 0) \tag{2.2}$$

where X_i is a uniformly random direction from the given hemisphere. New term in the estimator $max(\cos\theta, 0)$ is here to satisfy the requirement on $X_i \in H(x)$.

Now we need to extend this estimator to create rolling mean estimator with use of hysteresis to smooth our estimation over time. As proposed by Majercik et al. [2019] we will make use of $\mathrm{lerp}(a, b, t)$ function.

$$\mathrm{lerp}(a, b, t) = (1 - t) * a + t * b \tag{2.3}$$

You can see the whole estimator in equation 2.4. E_j stands here for estimation in frame j with $E_0 = 0$. $hysteresis$ is a user defined value and defines how fast the estimator should react to the changes in lighting inside a scene. Note that the 2π factor disappeared here. We will add it back in the probe sampling.

$$E_j(p) = \mathrm{lerp}(E_{j-1}, \sum_{i=1}^{N} L_i(p, X_i) * \max(\cos\theta, 0), hysteresis) \tag{2.4}$$

We have made a little improvement to the estimator as follows. The *hysteresis* still stands for user defined value but we have added new term h(i) where i is frame number.

$$\mathrm{h}(i) = \begin{cases} 1 & i = 0 \\ hysteresis & otherwise \end{cases} \tag{2.5}$$

$$E_j(p) = \mathrm{lerp}(E_{j-1}, \sum_{i=1}^{N} L_i(p, X_i) * \max(\cos\theta, 0), h(j)) \tag{2.6}$$

This minor improvement in equation 2.6 allows us to have the closest estimation after the first frame we can get and then interpolate smoothly between frames.

The update of depth values in probes works similarly. We only use sharpening of those values. The filtering of the depth value into the whole hemisphere will smoothen all details away. The sharper the depth values are, the more precise the visibility term we get. On the other hand, using a too sharp depth map can lead to visible artefacts on edge between shadowed geometry and visible for the point of the probe.

2.5.3 Indirect light sampling

We will move to the final stage of DDGI. Now, when we have all the data gathered for the indirect lighting, we need to sample the indirect light.

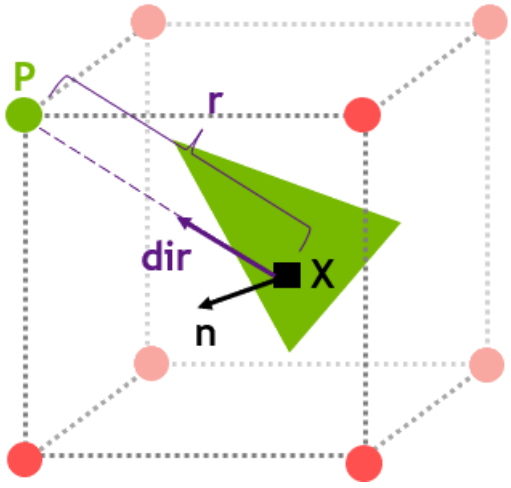


Figure 2.10: Shading of fragment X. Each fragment lies inside probe cage defined by corners of cage having integral coordinates in grid normalized space. Fragment X has normal facing in the direction n and we can compute direction dir to each probe P belonging to this cage and distance to this probe P denoted by r. (Image source: Majercik et al. [2019])

The algorithm limits sampling to the local part of the grid that we will further call probe cage. The probe cage is a cage of 8 probes around the shading point. We will use data from those eight probes to limit texture fetches. You can see the local setting of the cage in figure Figure 2.10.

In this cage, all eight probes are used to determine indirect lighting and weighted depending on the mutual orientation of shaded fragment and probe to backface culling and account for cosine factor coming from the Lambert law Lambert [2001].

Fragment offset

Majercik et al. [2019] propose to offset a shaded fragment "according to a bias proportional to the shading normal and the direction to the probes: this improves the robustness of the visibility-based interpolation weights by moving away from potential shadowed—unshadowed discontinuities". We have not found this solution improving our results in our implementation, so we decided to turn it off for final measurements.

3. Implementation

In this chapter we will explain what was all the steps needed to go through in order to correctly implement Dynamic Diffuse Global Illumination technique. We will start from the basics of new DXR API with focus on parts used in this work. Later we will explain in depth whole pipeline, some useful tricks and work my way towards improvements proposed by this thesis.

3.1 DirectX Raytracing

In this section we will focus on DXR API description. We will shorten this just to the basic concepts as We am not trying to match reference book but rather show how to start with DXR to the reader.

From high point of the view DXR introduced new pipline that you cen see in the Figure 3.1. From the diagram you can see the biggest difference from existing rasterization and compute pipelines. This pipeline is not strictly linear and can go through multiple loops before it ends. All the colored boxes are programatically customizable.

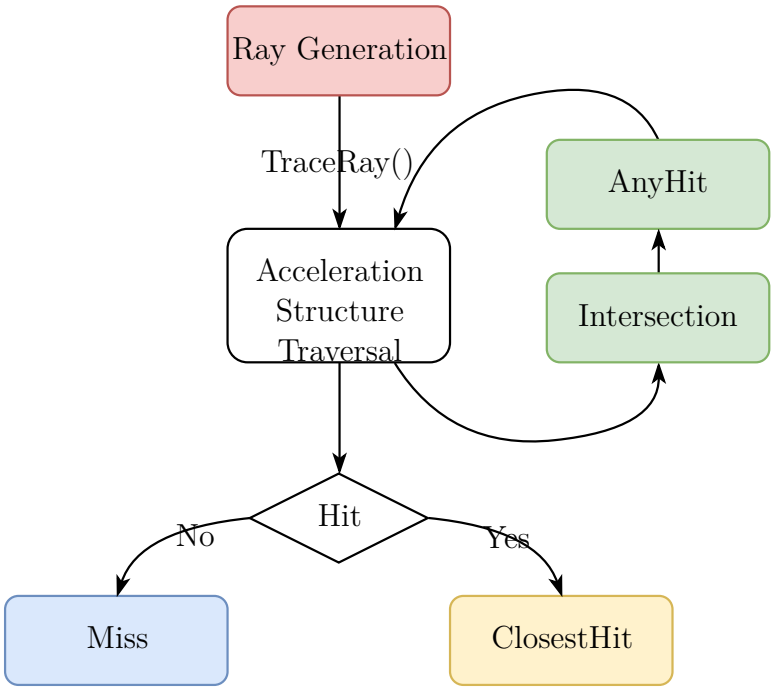


Figure 3.1: Overview of the RTX pipeline usable by DXR. (Image source: Lefrançois and Gautron)

One limitation that RTX ray tracer has that programmer have to define maximum recursion depth in advance. This create maximum stack size for given pipeline. The maximum recursion depth is 32 with ray generation shader is considered depth 0 and each *TraceRay()* call increases depth. Once the depth exceeds maximum depth the device going into removed state.

In the next paragraphs we will go through shader types, their usage and some of the syntax.

3.1.1 Ray generation shader

Ray generation shader is entry point of RTX ray tracing and it is being invoked by *DispatchRays()* function call. Ray generation shader is in some way similar to compute shader we know from graphics APIs. It knows its location in overall grid. The rays are being generated through *TraceRay()*. The specification does not guarantee execution order of threads.

```
[shader("raygeneration")]
float3 ComputeOrigin(uint3 gridPosition);
float3 ComputeRayDirection(uint3 gridPosition);
void raygen_main()
{
  const uint3 gridPosition = DispatchRaysIndex();
  const RayDesc Ray =
  {
    ComputeOrigin(gridPosition),
    SceneConstants.TMin,
    ComputeRayDirection(DispatchRaysIndex()),
    SceneConstants.TMax
  }

  Payload payload = {};

  TraceRay(
    accelerationStructure,
    0, //flags
    0, //InstanceInclusionMask
    0, //RayContributionToHitGroupIndex
    0, //MultiplierForGeometryContributionToHitGroupIndex
    0, //MissShaderIndex
    Ray,
    payload
  );

  RenderTarget[pixelCoord] = float4(payload.Radiance, 0);
}
```

Listing 3.1: Simple ray generation shader.

3.1.2 Any hit shader

A shader called once RT Cores found an intersection. This shader can either accept or ignore given hit. Also it is possible to modify intersection attributes and payload.

This shader can be useful in situations when parts of geometry are transparent. In this shader the ray can be accepted when opaque part is being hit. For example for shadow rays this can be used to update alpha value but ignore the hit to allow semi-transparent shadowing.

In situations when Any hit shader is not defined the default implementation simply accepts all hits.

IgnoreHit(). By this function you rejets hit and asks RT Cores to continue traversal of acceleration structure.

AcceptHitAndEndSearch() has very descriptive name. It allow RT Cores to end the traversal of acceleration structure and pass the info to the closes hit

shader.

Example of miss shader below in listing 3.2.

```
[shader("anyhit")]
void AnyHit(inout Payload payload, in Attributes attr)
{
  const float currT = RayTCurrent();
  const float3 hitLocation = ObjectRayOrigin()
    + ObjectRayDirection() * currT;

  const float alpha = computeAlpha(hitLocation, attr, ...);

  // the object is counted as opaque
  if (alpha > trashold)
      // register as a hit and pass to the closest hit shader
      AcceptHitAndEndSearch();

  // Add alpha to properlly calculate transmitance
  AddAlpha(payload, alpha, RayTCurrent())
  IgnoreHit(); // continue with acceleration structure traversal
}
```

Listing 3.2: Any hit short example that handles transparency for shadow rays.

3.1.3 Closest hit shader

Closest hit is envoked once RT Cores figures out the closes hit from the ray origin. This is the main point where your ray tracer is implemented. This is the point where to plug BRDF or multiple importance sampling.

It is guarenteed that all Any hit shaders along the path of the ray are being fully exeuted before the Closest hit shader. This allows programmer to freely change payload and intersection attributes without needs for synchronization.

In Closest hit shader is also possible to launch new rays up the the defined maximum of recursion or call callable shaders. Also in this shader you can store data to memory using Unordered Access Views.

```
[shader("closesthit")]
void closesHitShader(inout Payload payload, in Attributes attr)
{
  const float currT = RayTCurrent();
  const float3 worldRayOrigin = WorldRayOrigin()
    + WorldRayDirection() * currT;

  const flaot3 worldNormal = ...; // get normal from geomtry
  const float3 wo = ReflectRay(WorldRayDirection(), worldNormal);
  const RayDesc reflectedRay = {  worldRayOrigin, SceneConstants.
   Epsilon,
                                  wo,
                                  SceneConstants.TMax };

  TraceRay(
    accelerationStructure,
    0, //flags
    0, //InstanceInclusionMask
    0, //RayContributionToHitGroupIndex
    0, //MultiplierForGeometryContributionToHitGroupIndex
    0, //MissShaderIndex
```

```
    Ray,
    payload
  );

  RenderTarget[pixelCoord] = float4(payload.Radiance, 0);
}
```

Listing 3.3: Example of Closes hit shader starting new reflection ray and storing values to the memory.

3.1.4 Miss shader

If ray miss all the geometry the Miss shader is started. Miss shader can generate new rays and alter payload. The intersection attributes are not present but that is logical implication of no intersection happening.

This shader can be used for example to render sky dome or mark paths that ends up missing the scene.

Example of miss shader below in listing 3.4.

```
[shader("miss")]
void MissShader(inout Payload payload)
{
  CalculateSkyRadiance(...);

  TraceRay(...); // start new rays

  CallShader( ... ); // even call specific callable shader
}
```

Listing 3.4: Miss shader example with sky dome or callable shader call.

3.1.5 Intersection shader

The exception from the representation of the geomtry by triangles in accleration structure is bottom-level node containing AABB box that contains proceduraly generated primitive. The surface of such primitive is definde by running custom intersection shader. This shader is being evaluated when shader hits this AABB. Intersection shader has to define intersection attributes that are consequently passed to the any hit shader and susequent ones including T value. This type of shader was not crucial for my work so we are listing it only for completing the picture.

ReportHit(float THit, uint HitKind, attr_t Attributes) function serves to report a hit. The hit kind parametr is user defined and can be read from Any or closes hit shaders. Attribtues is user defined structure to pass info to hit shaders. Report hit returns true when hit gets accepted.

The search can end when first hit and end search is set or *AcceptHitAndEndSearch()* is called from any hit shader.

3.1.6 Callable shader

Callable shader is new shader that can be called from other shaders. This shader type is response to the exponential growth of shaders with all the permuations.

The main difference from calling a subroutine is that instead of having to run code of the subroutine on all streaming processors in local groups the call is being pushed to the queue for sheduler. The scheduler can schedule those shader runs separatelly from caller run once enought callable shaders are gathered. The caller is than resumed after callable shader is evaluated. The callable shader can also run callabe shader inside its body.

This opens new level of possible optimisations.

Although we have not used this shader in my implementation we think that it is fair to list it in this chapter as a new technology coming in graphics APIs with DirectX 12.

```
[shader("callable")]
void callableShader(inout Params params)
{
    // Run calculations
    CallShader( ... ); // or run another callable shader
}
```

Listing 3.5: Callable shader example.

In the Figure 3.2 you can see the whole lifecycle of a ray from *TraceRay()* function call to the Miss or Closest hit shader call.

3.1.7 Payload structure

Payload is user defined structure. This structure holds information about the path and is caried by DXR along the ray and passed to all shaders. Programmer has to pass it in *TraceRay()* function call as inout parameter. Each shader in the pipeline needs to specify matching payload structure as an argument even if it is not used inside the shader.

With given overview of DXR we can now discuss implementation details of my implementation of DDGI.

3.2 Dynamic Diffuse Global Illumination implementation

This section of this thesis should provide reader with enough informations to replicate same algorithm we have implemented. We will try to explain everything to sufficient detail. A short snippets of code will be provided with explenation but reader can find full code attached to this thesis.

3.2.1 Data storage

As we described earlier, each probe is stored in a square texture. Usage of one texture per probe would be highly limiting. So rather than using many textures or texture arrays, we will use two atlas textures. We explained the concept of texutre atlas in section 1.6. Atlas texture is a texture subdivided into separate regions each representing one ”sub-texture”.

Because our probe grid is a regular uniform grid we can simplify mapping from the world space coordinates to the grid coordinates and further into the

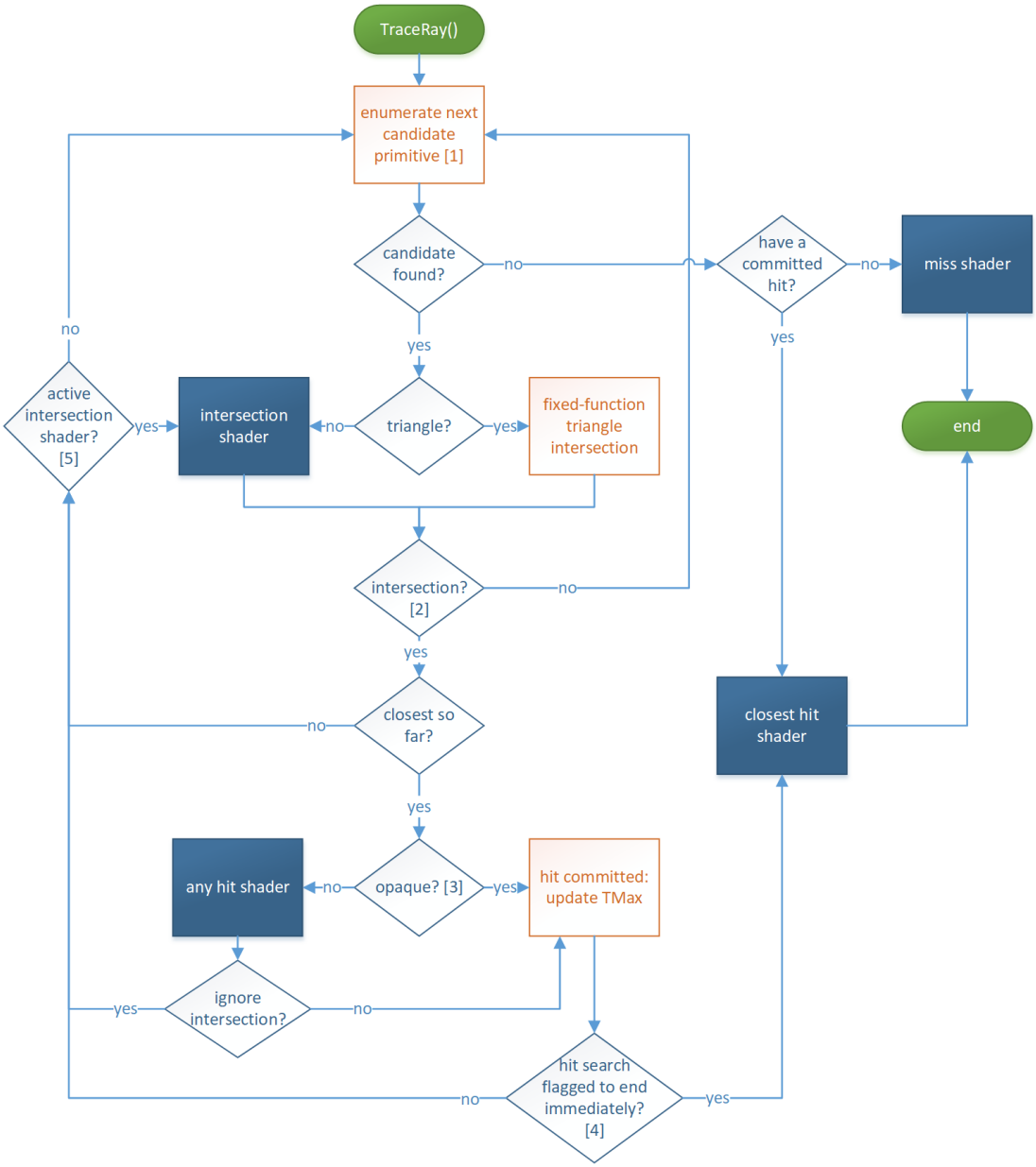


Figure 3.2: Full lifecycle of ray in DXR ray tracing pipeline. (Image source: Microsoft)

texture coordinates. We will discuss those mappings in the section dedicated to the mappings and their influence on performance.

3.2.2 Probe data

In each probe, we are going to store two types of information. The first one is usual irradiance as we know from irradiance mapping by Ramamoorthi and Hanrahan [2001]. The second one is depth information. The depth information allows us to calculate occlusion for our indirect lighting. To compress data and prevent sharp edges between occluded and lit parts of the geometry DDGI uses Chebychev's inequality (Chebyshev [1867]) as we know from variance shadow mapping Donnelly and Lauritzen [2006].

To store all those data we will use two texture atlases. This allows us to set resolution per probe separately for the irradiance and depth part. Also this allows us to set different resolution for depths and irradiance data. This will be more discussed in chapter Chapter 4.

It is even possible to use two different formats and we will make use of it. Reason being that dept data needs two channels of floating point format - one channel for mean distance and one for variance. On the other hand the irradiance stores RGB data.

Gutter region

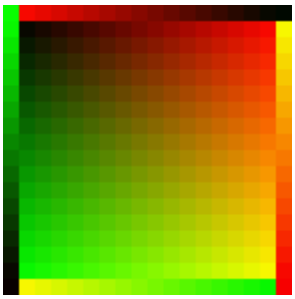


Figure 3.3: Visualization of gutter region UV mapping. (Image source: Thesis author)

Previously we have mentioned the need for a gutter region. The reasoning is in section 2.5.1. As you can see from the Figure 2.9 all we need to do is to reverse UV coordinates in this area to be mirrored along the adjacent border and move it one pixel outside. In Figure 3.3 we show the probe UV mapping, including the gutter border UV coordinates. In Listing 3.6 we provide a function to map atlas UV coordinates to individual probe UV mapping with gutter region correctly mirrored.

```
float2 GetInProbeUV(const uint2 texel, const uint
    perProbeResolution, const uint3 gridSize){
  const int probeSideWithBorder = perProbeResolution + 2;
  float u = (texel.x) % probeSideWithBorder;
  float v = (texel.y) % probeSideWithBorder;

  if(u==0)
  {
    v = probeSideWithBorder-v - 2;
  }
  else if(v==0)
  {
    u = probeSideWithBorder-u - 2;
  }
  else if (u == probeSideWithBorder-1)
  {
    u = perProbeResolution - 1;
    v = probeSideWithBorder - v - 2;
  }
  else if (v == probeSideWithBorder-1)
  {
    v = perProbeResolution - 1;
```

```
    u = probeSideWithBorder - u - 2;
  }
  else
  {
    u -= 1;
    v -= 1;
  }


  return (float2(u,v) + float2(0.5, 0.5)) * (2.0f / float(
   perProbeResolution)) - float2(1.0f, 1.0f);
}
```

Listing 3.6: This function maps the atlas UV coordinates to probe UV coordinates. This mapping also takes care of the gutter region correct translation. Can be found in *DDGIMappings.shader*

3.2.3 Probe grid

We have already explained that probes are placed in the world on a regular uniform grid. In order to simplify the algorithms used in this technique, we need to define some mapping function between the spaces we will use. Those mapping functions are going to be all bijections, and we will make use even of their inverse functions. To name the spaces, we will work with three names for simplification.

We will use world space coordinates. This space is simply identical to the world space the engine uses. The second space used will be grid normalized coordinates. This space is defined with $(0, 0, 0)$ at left bottom back of AABB of the grid.And each probe is in the basic algorithm placed on each, with all three coordinates being a positive integer. The last space used will be the UV space of the probe atlas texture.

3.2.4 Smooth backface culling

As the technique depends on a regular grid as a form of discretisation, there could be high-frequency changes on the borders between grid cells. One such change could be between probes that are front-facing and back-facing the shaded fragments. To avoid those sharp changes Majercik et al. [2019] suggests using smooth backface culling 3.2 instead of sharp culling 3.1 that is physically correct.

$$weight_{culling} = max(dot(\vec{N}, \vec{\omega_i}), 0) \tag{3.1}$$

$$weight_{culling} = max(dot(\vec{N}, \vec{\omega_i}) * 0.5, -1e4)^2 + 0.2 \tag{3.2}$$

The 0.2 offset is proposed by Majercik et al. [2019]. Unfortunatelly this made whole solution physically inaccurate and increased error in the resulting image. We have decieded to skip this part in my work and use smooth backface cullting but without this offset as seen in equation 3.3.

$$weight_{culling} = max(dot(\vec{N}, \vec{\omega_i}) * 0.5, -1e4)^2 \tag{3.3}$$

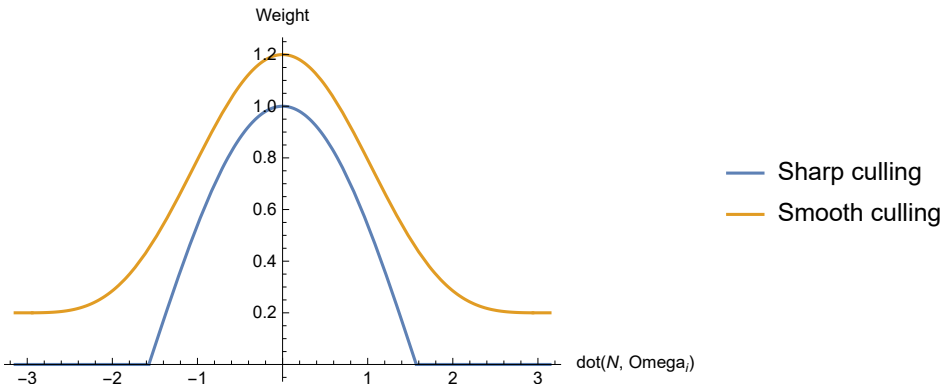


Figure 3.4: Comparison of smooth and sharp backface culling. (Image source: Thesis author)

3.2.5 Ray-tracing

We will use RTX technology to get radiance and depth data for our probes. The existing path-tracer code in Fibix has done an excellent service in this task. It allowed us to implement the whole solution faster and also as we will use this path tracer as ground truth in the upcoming Chapter 4 chapter.

The decision not to implement a part of DDGI with a feedback loop of probe data coming back into RTX part of the algorithm as we achieved quality results even without this part with good render times.

Because we skipped this part, we had to use path-tracing recursion. Not using this part could improve render times significantly as we will be able to prepare the workload for all RT Cores more predictable.

Algorithm

For data collection, we will make use of NVidia technology RTX. This will allow us to cast a huge number of rays in each frame. For each of the m probes, we are going to cast n ray every frame.

- Generate n -nth ray from Fibonacci spiral
- Rotate ray by random rotation matrix
- Cast ray
- Collect depth from first intersection
- Collect final radiance from ray tracer

3.2.6 Dircetion generation

For Monte Carlo estimation of irradiance integral(2.1), we rely on a uniform random number generator. Unfortunately, getting random numbers reliably in a heavily parallelized environment of graphics cards is not possible. Instead, we will use a different approach. We will first generate a uniform distribution of directions on the sphere, and then we will pass a random rotation matrix into a shader.

For uniform distribution of directions, we will use the Fibonacci spiral pattern Keinert et al. [2015]. For this spiral pattern, we first need to decide how many rays we will need and then pass each time number of total rays and the ray number. This spiral is deterministic, and it returns the same set of rays for the same total count of rays each time. This is why we need to pass the random rotation matrix into the ray generation shader. It is more convenient to generate this matrix on the CPU side as we need the same rotation for the whole probe, and also, GPU random number generation is not as straightforward as CPU.

For simplification, we have decided to use the same rotation for all probes in one frame. This should not affect the correctness of the algorithm as the Monte-Carlo estimators are separated for each probe.

Tracing the ray

We already described technical details of DXR previously in section 3.1. As we mentioned before we are using existing path tracer that is already present in Fibix engine. So we can narrow my explenation only to small code parts in Generation shader and than in Closest hit shader.

```
[ shader( "raygeneration" ) ]
void RaygenShaderSphere()
{
  const uint2 dispatchDim = ScreenOffsetCB.ViewportResolution;
  const uint2 pixelCoord = GetPixelCoords();

  const uint3 probeGridCoord = GetProbeGridCoordFromRelativeID(
   pixelCoord.y, uint3(RayTraceCB.DDGIGridSize));

  uint width, height;
  RenderTarget.GetDimensions(width, height);

  float4 dirRotated = normalize(mul(float4(
   SphericalFibbonaciGenerator(pixelCoord.x, width), 1.0),
   RandomRotation));

  float3 rayStart = GetProbeWorldOffset(probeGridCoord,
   RayTraceCB.CameraPosWS, RayTraceCB.DDGIProbeDistances);

  RayDesc ray;
  ray.Origin = rayStart;
  ray.Direction = normalized(dirRotated).xyz;
  ray.TMin = 0.0f;
  ray.TMax = 1000;

  PrimaryPayload payload; // prepare payload
  ...
  TraceRay( Scene, RAY_FLAG_NONE, 0xFFFFFFFF, hitGroupOffset,
   hitGroupGeoMultiplier, missShaderIdx, ray, payload );

  RenderTarget[ pixelCoord ] = float4(payload.Depth, 0, 0, 0);
  DDGIRadianceTarget[pixelCoord] = float4(payload.Radiance, 0.f);
  DDGIDirectionsTarget[pixelCoord] = float4(rayDir, 0.f);
}
```

Listing 3.7: Ray generation for Fibbonaci sampling used in DDGI. The shader is heavily simplified for better readablity. Can be found in *RTX.fx*

3.2.7 Indirect lighting sampling

In previous sections, We have described how to calculate the probe data, the way we store them. In this section, we will describe indirect lighting sampling used in the final fragment shading.

Given the world position of the point, we can determine its position within the probe grid. Each point lies within the probe cage defined by corners of cage having integral coordinates in grid normalized space. We can get a grid coordinate for a given point by subtracting the world offset of the grid and then dividing this position by grid spacing. This gives us the grid coordinate of the probe from this cage with the lowest x,y and z coordinate. You can see the code below in listing 3.8.

```
// returns base probe coord for given world pos
// Means the corner with  min(x) & min(y) & min(z)
GridCoord GetBaseProbeCoordinates(
    const float3 worldCoord,
    const float3 gridOffset,
    const float3 probeDistances)
{
  const float3 probeSpaceCoord = (worldCoord - gridOffset)/
   probeDistances;
  return uint3(probeSpaceCoord);
}
```

Listing 3.8: Mapping wolrd space position of the fragment to the base coordinates of its probe cage. Can be found in *DDGIMappings.shader*

For each of those eight probes, we will first perform back facing check. As we know the normal vector of the given fragment we can check whether the fragment faces the probe. We can ignore all probes behind the fragment because we are interested only in the upper hemisphere indirect lighting. You can see the full code in listing 3.9.

The listing also shows fragment offseting along the normal with the usage of view dependent factor as explained in section 2.5.3.

```
float weight = 1.0f;
float3 probeCoord = GetProbeWorldOffset(gridCoord, gridoffset,
   c_probeDistances);
float3 toProbe = probeCoord - worldPos;
float3 trueDirectionToProbe = normalize(toProbe);
#define DDGI_NO_NORMAL_OFFSET // turns off normal offsets.
#ifdef DDGI_NO_NORMAL_OFFSET
    const float3 pointToProbe = -toProbe;
#else
    FLOAT3 camPos = ...;
    float3 viewDir = normalize(camPos - worldPos);

    float3 pointToProbe = -toProbe + (normal + 3*viewDir) * 0.25;
#endif

float cosTheta = dot(trueDirectionToProbe, normal);
//weight = max(cosTheta, 0.000001); // sharp backface culling
weight = pow(max((cosTheta+1.0)*0.5, 0.000001), 2.0);
```

Listing 3.9: Smooth and sharp backface culling and view vector normal offset code snippet. Can be found in *PSShader.fx*

For the second check, we will make use of the depth data we gathered during the ray tracing phase. Using Chebyshev's inequality we can determine the probability of fragment being obstructed from the direction of the probe. You can also see that we have turned off the added variance in the term as this part of the calculations doesn't proved to improve the results significantly enough.

```
const float chebychevVarinaceBias = 0.0f;

float2 probeDepthData = SAMPLE_TEX2D(DDGIDepths, uv).xy;
float mean = probeDepthData.x + chebychevVarinaceBias;
float variance = abs(pow(mean, 2.0) - probeDepthData.y) +
   chebychevVarinaceBias;
float chebyshevWeight = variance / (variance + pow(max(distance -
    mean, 0.0), 2));
chebyshevWeight = max(pow(chebyshevWeight, 3.0), 0.0) /
  (1-chebychevVarinaceBias);

weight = max(0.00001, weight *
  (distance > mean?chebyshevWeight:1.0));
```

Listing 3.10: Chebyshev's inequality term used in visibility determination during the probe evaluation. Can be found in *PSShader.fx*

3.3 Problems

The usage of the grid-based method brings many advantages but disadvantages as well. One of the inherent problems grid methods introduces is the level of control over the placement of the probes. As described earlier, we are ignoring all the probes inside a geometry. However, due to the uniformity of the grid placement, some probes will end up inside a geometry. One of the worst-case scenarios is when a wall of a house is aligned with the grid. That case will remove half of the probes for cages on each hand of the wall. This not only costs us additional computation power to update such probes without getting any useful information but also can lead to fragments that will have no probe to get information from.

The other problem is that uniform placing of the probes can lead to situations when we have surfaces in our scene which doesn't have clear sight of view to any probe. This can happend due to unfortunate scene setup e.g. as you can see in Figure 3.5.

Those can seem like similar problems but actually they are not. This section addresses the first one but leaving second one stil intact.

3.3.1 Dead probes

The first problem we will try to solve is probe placement. We want to avoid probes inside a geometry. To detect those, we can modify our ray-tracer to detect back-face intersections. When a ray hits the back face of geometry, we will stop the path tracer and return a depth of zero. Just by this, we will avoid a lot of unnecessary calculations of the path tracer. Stopping the path-tracer will save computation time but not memory space, and we still have useless probes hidden somewhere inside the geometry. Such probes are guaranteed not to affect final

indirect light sampling as they appear occluded from all sides. Because of that, we will call those probes "dead probes".

Previous leads to the next improvement. We will try to detect such probes and find a better space. We can save information about average depth in the 2D array, and after the probe update phase, we can find those probes and change their positions. Now we have to be a little conscious about the real-time aspect of this technique. Moving probes after one frame of them being inside the geometry could lead to a flickering in the scene. We will accumulate information over multiple frames and move them after some time. We will achieve this delay by using hysteresis in the same manner as we are using in subsection 2.5.2.

Another step is finding the "good place" for such probes. The easiest solution is the movement along the grid axis by a small amount each time we detect the probe submerged in geometry. For a better search, we will cycle over the six possible directions and increase the distance from the original placement. This allows us to move the probe without a need for complex tests of surrounding geometry.

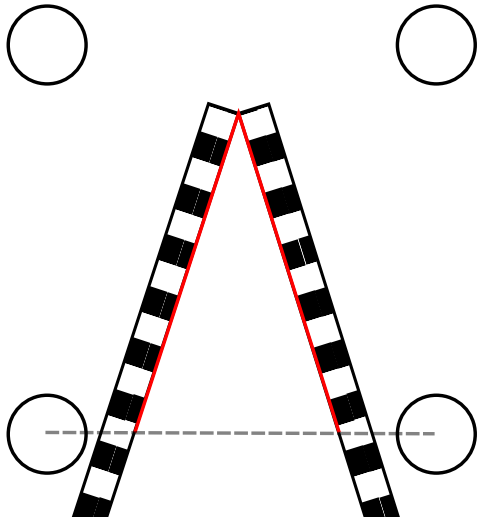


Figure 3.5: This drawing shows a situation when all probes are obscured by geometry. The dashed line shows the border of the probe cage. Fragments highlighted in red have no valid information about indirect lighting. (Image source: Thesis author)

This solution should follow the idea of shielding away artists from manually tweaking positions of the probes and allowing the technique to react to the changes in the scene dynamically.

The technical way of moving the probe is more complicated than it looks. First, we need to find probes to move. Then we need to store this information, and in the next step, we need to store the information about where we moved the probe. Lastly, we need to propagate information about the movement to the RTX.

Dead probe detection

To choose the probe we use information from the ray-tracing step. We introduced a new step in which we count the number of rays shot in the last frame that encountered the backface of geometry as a first event. In an ideal world where we would be able to make such constrain as forcing all the geometry to be manifold and does not intersect other geometry, we would simply check one ray and we would be able to tell whether such probe is submerged inside a geometry. Unfortunately, this is not the case in-game engines so we went for a more complicated but still forgiving solution. We mark only such probes that had more than half of the ray hit backface geometry. There is common practice that level artists submerges geometry inside another one in order to make it look like one piece of geometry e.g. wooden beam inside a wall in Figure 3.6.

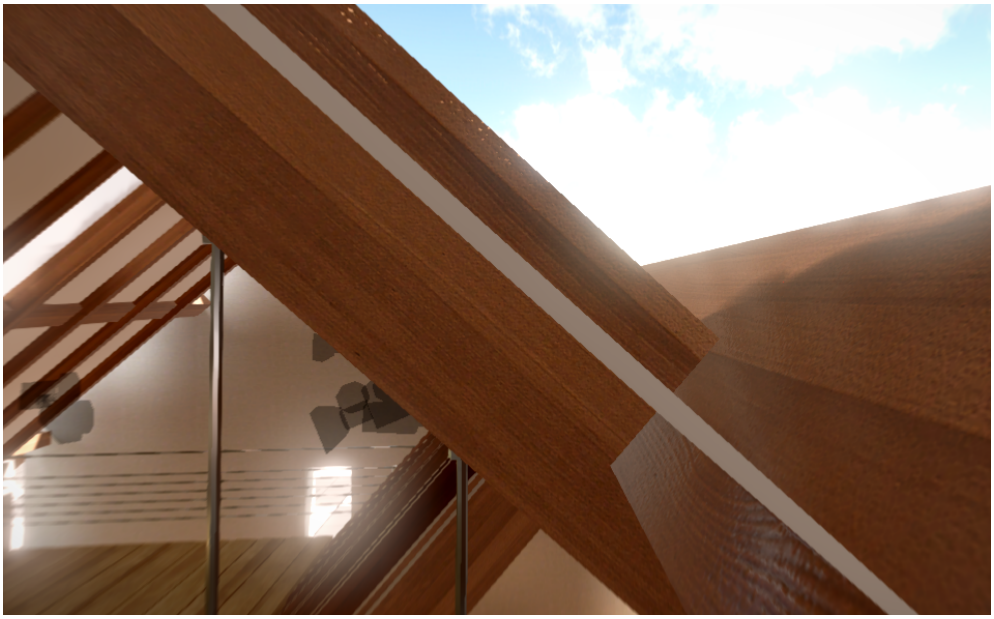


Figure 3.6: This image shows how submerging non-manifold geometry is used in 3D scenes. The white stripe in the middle represents the ceiling. (Image source: Thesis author)

Afterwards, we use this result to update the texture that marks "dead probes". We update this texture with adjustable hysteresis. The hysteresis in this case avoids the too aggressive movement of probes. If we would simply move every probe occluded in the given frame it could lead to forced move every frame when a character stands near the probe and frequently changes position.

Moving probe

From the previous step, we have a list of probes that are occluded. Now we need to move probes outside of geometry. This could be done in two main ways. The geometry aware or unaware approach is possible.

The geometry aware approach is easy to do from the initial position. We can write the distance to the backface hit in the raytracing step only with the negative sign and then analyze hits from the last frame in 6 directions along the axis of the grid. Then choose to move in the direction where the distance is shorter. But

this would mean more calculations in this step and if we want to preserve the constrain that all the probes lie at least on the lines of the grid we would have to complicate it even more or move the probe back on the intersection of the grid lines for one frame before moving it again.

On the other hand, we can move the probes completely unaware of the geometry simply by moving them in some pattern along the grid lines.

Trilinear filtering

The major part of light sampling affected is trilinear filtering. Original DDGI uses trilinear filtering to weight probes with the following Listing 3.11.

```
float3 filteringCoef = lerp(1-framgentPosition, framgentPosition,
    float3(probeCoord[i]));
```

Listing 3.11: Original trilinear filtering. All coordinates are in grid cage normalized space. Can be found in *PSShader.fx*

The positions in Listing 3.11 are all normalized to the grid cage space. This mean $coordinate \in < 0, 1 >^3$. The probe after the moving does not satisfy this condition. In order to have correct results from the indirect light sampling, the sum of weights need to be one, and each weight needs to remain on the $< 0, 1 >$ interval. However, if one of our probes moves, it will end up in half of the cages it belongs to with a position outside the cage. This can lead to subtracting the irradiance. A simplified case with two probes is shown on the left side of Figure 3.7. The right side shows our filtering adjusted for moved probes. Implementation is presented in Listing 3.12

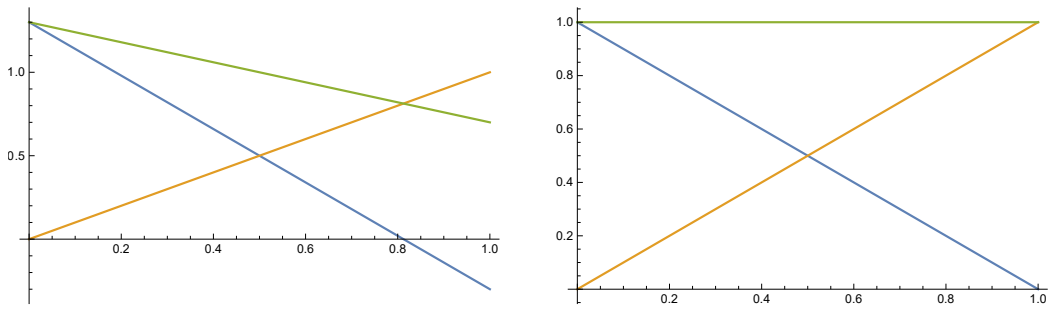


Figure 3.7: Comparison of smooth and sharp backface culling. (Image source: Thesis author)

```
float3 probeWorldOffset = GetProbeCustomOffset(gridPos, uint3(
    c_DDGI_gridSize));
float3 denom = (float3(1,1,1) - probeWorldOffset);


float3 filteringCoef = lerp(
  float3(1.0, 1.0, 1.0)-withinCageCoordNorm,
  withinCageCoordNorm,
  (float3(probeCage[i]) - probeWorldOffset) / denom
);
```

Listing 3.12: Improvedfiltering. All coordinates are in grid cage normalized space. Can be found in *PSShader.fx*

Propper sampling offset probes

In order to achieve correct sampling of moving probes, we also need to propagate this information into the Ray generation shader. This is done by sharing the offset texture back to the DXR part of the computation. In Listing 3.13 you can see how we have implemented it. The code is straightforward. The resulting offset is then added to the *rayStart* variable before passing ray descriptor into the *TraceRay()* function. The multiplication by 255 is here because we use the 8-bit uint format to store the offers, but the sampler returns numbers normalized to the interval $(0, 1)$.

```
Texture2D DDGIOffsets = Tex2DTable[RayTraceCB.DDGIOffsetsIdx];
uint widthOff, heightOff;
DDGIOffsets.GetDimensions(widthOff, heightOff); // for precies
    sampling

const float2 uv = GetProbeOffset(probeGridCoord, RayTraceCB.
    DDGIGridSize) + float2(1.0/widthOff, 1.0/heightOff);
const float OffsetID = DDGIOffsets.SampleLevel(PointSampler, uv,
    0.0f).x * 255;
float3 offset = GetProbeOffsetFromIDX(uint(OffsetID)) *
    RayTraceCB.DDGIProbeDistances;
```

Listing 3.13: Offset sampling code from ray generation shader for the dead probes offset improvement. Can be found in *PSShader.fx*

3.3.2 Depth samples rejection

One of the most significant issues we have encountered during the whole work was massive light leaking to the extent that we have considered a change of the selected technology to achieve our goals for Fibix. Right before the end of the thesis, we realized the source of the problems. The DDGI used depth samples from ray tracing.

As shown in Figure 3.8 if we use each depth sample, we can end up with depths that are missing geometry altogether and flying away into the infinity or just samples outside the cage. Adding those samples to the mean value saved inside the depth map lead to high variance. Afterwards, when we shade surfaces denoted by red and orange lines, those depths can lead to light leaking from this probe through a wall.

We have solved this issue by rejecting such samples. We are testing depths against the length of the main diagonal of the cage enlarged by the maximal offset distance of the probes as this is the worst-case scenario. This simple change has not changed the frame time cost of the whole solution and has a massive impact on the overall error of the technique.

We consider this solution to be logical, but it cannot be found in the reference code of DDGI nor the paper.

3.4 Controls

In the Fibix engine, we have included UI to ease usage of the global illumination solution. We tried to make controls simple to use but yet hard to misuse.

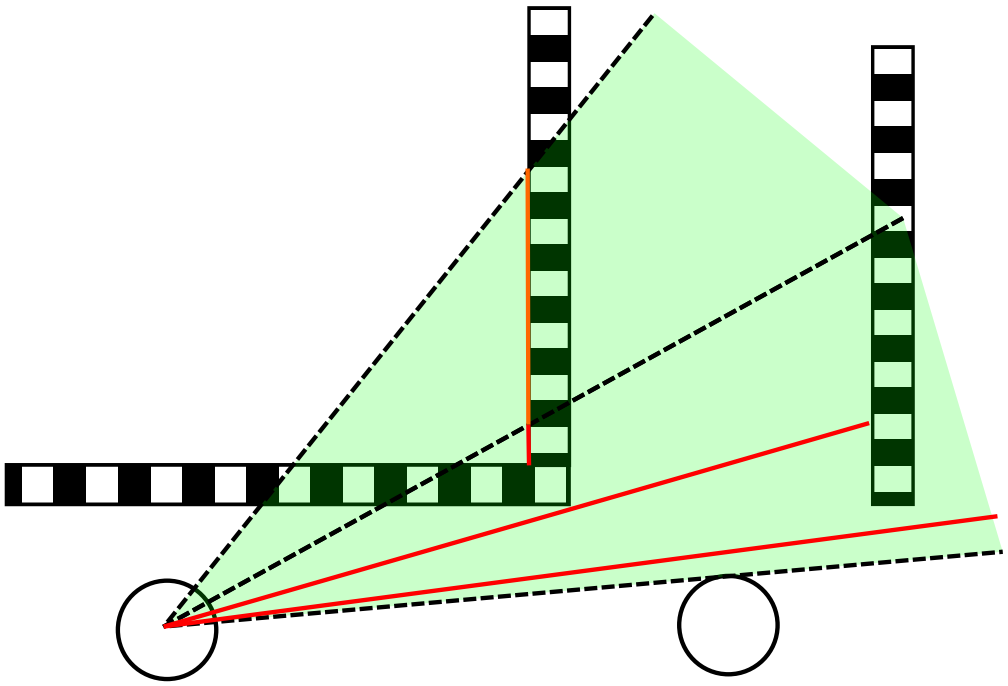
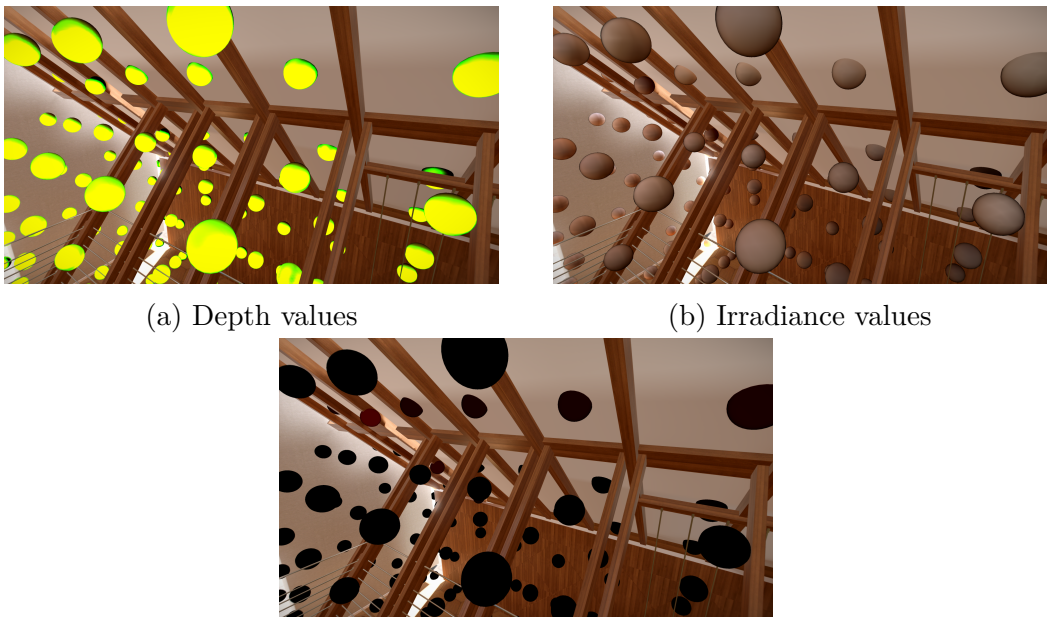


Figure 3.8: Light bleeding due to long depth samples outside of probe cage. (Image source: Thesis author)

The first field, “Show probes” is mainly supposed for debugging purposes, but environment artists can also use it to tweak lighting for a specific scene. It visualises probes positions and data stored inside them by displaying geometry inside 3D scenes. The example of this visualisation is shown in Figure 3.9.



(a) Depth values

(b) Irradiance values

(c) Offset indicies

Figure 3.9: A debug visualistaion on the probes

The next group of controls are "Num X/Y/Z probes" this defines number of probes inside a grid in the direction of the cardinal coordinate system of the world. These options could significantly affect frame time and memory cost and the visual quality of the resulting image. This option is also tightly connected with "Probe box size" as the combination of those settings defines the density of the probe grid. Probe box size defines the Axis Aligned Bounding Box (AABB) box in which you want to use DDGI probes. In order to move this box, we can move it relative to the world's origin (coordinate {0,0,0}) with "World offset".

The following section of settings deals with parameters of the original part of the DDGI algorithm. The first option of this group, "Hysteresis" defines the temporality of probe data. It defines how much of the previous value will be carried to the next frame. It affects the speed of update of both irradiance and depth textures. We recommend to set this value according to the speed of light changes you expect from your lighting setup in the scene. The reason for this hysteresis in the algorithm is described in depth in section 2.5.2.

The last parameter of the original DDGI is "Depth sharpness". The sharpness is described in section 2.5.2 and the implementation details in subsection 2.5.2. Generally speaking, a higher value means a sharper depth map but a higher possibility of visual artefacts.

The main control affecting the techniques speed is "Num samples per probe". This field defines how many rays should be shot from each probe every frame. More frames are used, the quicker and finer result technique provides, but at the same time, the more time it will take during RTX ray tracing. The effect on performance could be seen in Table 4.6.

Another group of controls is concerned with "dead probes". Those are explained in subsection 3.3.1. You can disable detection of such probes with "Detect dead probes" which will consequently disable also their offsetting. The offsetting of the probes can also be disabled separately with the "Offset dead probes" checkbox. To define how aggressively the algorithm moves such probes, we can adjust "Dead probes hysteresis". This value defines how much of the "certainty" that some probe is inside a geometry should be preserved from the previous frame. The value is defined in hundredths of percent. So a value of 0.95 means "carry 95% of certainty to the next frame". This can be useful when you know that your game will have a lot of fast-moving objects, and you do not want the lighting to flicker as the object flying through the probe's location is moving probes aside. In this situation, we would set this value close to 1. The last control affecting dead probes movement is the "Dead probe threshold". This number defines the smallest distance probe is supposed to have from the nearest object. Setting this value to zero means we only want the algorithm to move probes to the surface of the geometry. In most scenarios, this may not be enough because probes could end up in the small cavities or corners of geometry, and this would make them occluded from most of the directions. On the other hand, setting this value high could make it impossible for the algorithm to find the correct position for the probe.

The last section is dedicated to the debugging of the whole system. You can visualize there each texture used for final shading. This excludes RTX targets as those needs synchronization, and their data are not visually readable. So debug UI allows you to draw depth and irradiance texture atlas, texture called "delta

per probe" that contains previously mentioned "certainty" that probe is inside the geometry or too close (this depends on "Dead probe threshold"). The last texture is called "Delta position" and contains an index to the position offset of the probe. The second control used for debugging is the "Show debug layer" and controls which layer of grid you want to visualize. You can see how the debug is visualized in Figure 3.11.

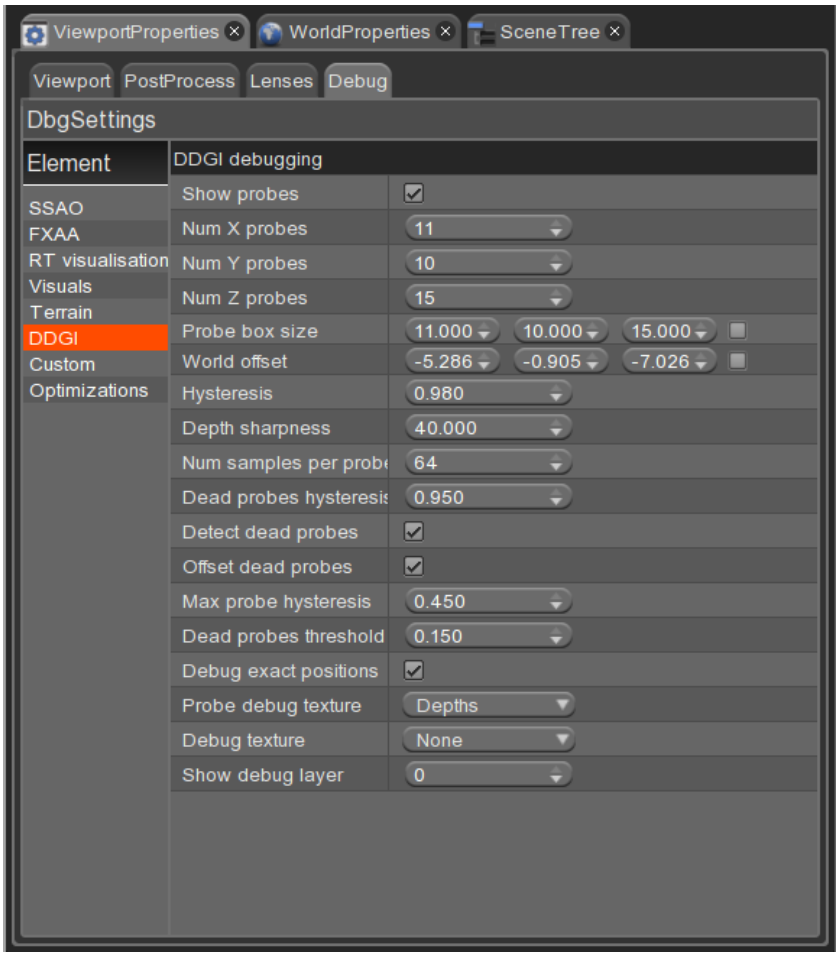


Figure 3.10: Application UI for DDGI settings. (Image source: Screenshot from Fibix engine)

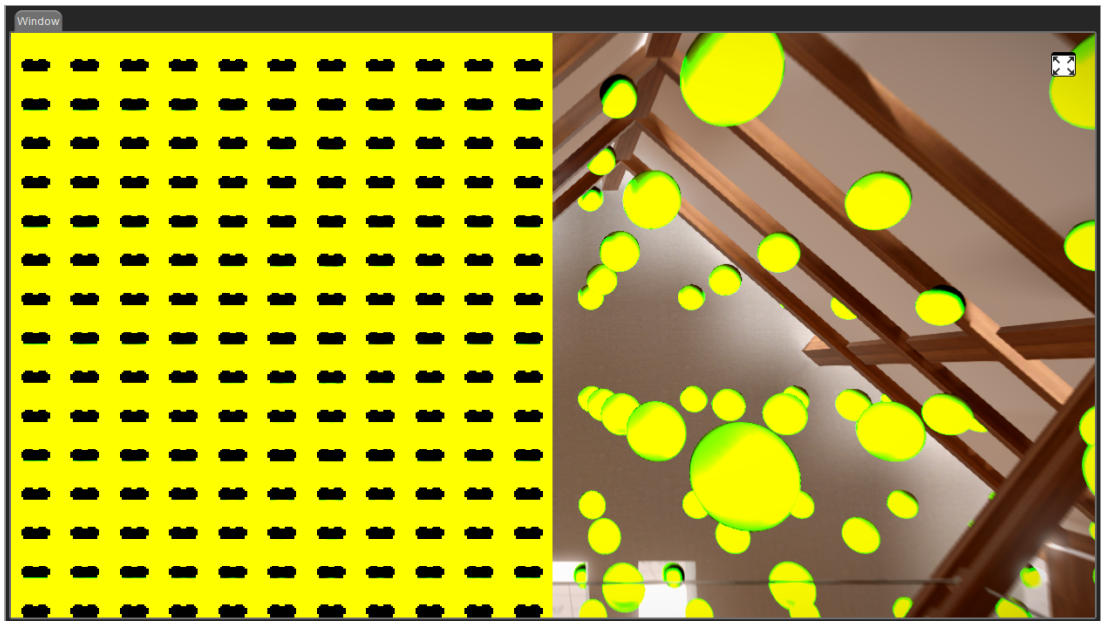


Figure 3.11: Debug of one layer of depths texture (Image source: Screenshot from Fibix engine)

4. Results

We tested our implementation of DDGI on the scene provided by the Fibix company as a representative baseline for scenes typically rendered in their engine. This chapter provides an overview of the achieved performance and overall accuracy of the rendered image compared to ground truth. We have obtained ground truth reference images from the offline path tracer implemented in the Fibix engine. The Fibix engine uses the offline path tracer to render realistic images.

4.1 Per-probe memory requirement

From probe atlas specification in subsection 3.2.2, we can calculate per probe memory requirements. Let res_i be a resolution of the irradiance probe and res_d resolution of depth probe. We also need to assume the used data format. For data formats, we define w_d and w_i as the memory size of one component of the used format in bytes. We can now calculate a memory demands of the used setup as

$$M = (res_d + 2)^2 * w_d * 2 + (res_i + 2)^2 * w_i * 4 \tag{4.1}$$

bytes. We are adding two pixels to the dap resolution due to the gutter region. Also, remember to multiply each map's pixel count by the number of components in a given pixel format, two for depth map and four for irradiance map. For better picture of how the data cost rises we provide table 4.1.

res_i	res_d	Irradiance format	Depth format	Memory[B]
8	8	ABGR16F	GR16F	1200
8	16	ABGR16F	GR16F	2096
8	32	ABGR16F	GR16F	5424
8	8	ABGR16F	GR32F	1600
8	16	ABGR16F	GR32F	3392
8	32	ABGR16F	GR32F	10048
16	16	ABGR16F	GR16F	3888
32	16	ABGR16F	GR16F	10544

Table 4.1: This table show how much resolution and format choice affect probes memory footprint.

While reading the table, bear in mind that DDGI depends on the usage of a grid of those probes that could contain hundreds or lower thousands of probes. This is why users should prefer 16-bit formats and keep the resolution down. We will further show how changes in resolution affect the image's visual quality.

4.2 Reduction of visual errors

In this section, we will focus on reducing visual errors. Those are errors that you can see, for example, in Figure 4.1. Those errors and their sources are explained in section 3.3. As those errors form the most undesired artefacts the original implementation had produced, it is logical that those were the ones we wanted to mitigate the most. Our evaluation consists of two metrics. One is the measurement of absolute difference in comparison to the ground truth. The second metric is the minimization of the visual appearance of the artefact, which we cannot measure, but we will provide a detailed explanation with each case.

To measure absolute error, we created a simple script that compares pixel values from the reference image with the corresponding pixel from the rendered image and returns the sum of the absolute difference per component. This resulting number lies on the interval $[0, 3]$ where 0 denotes completely equal values, and three indicates that one image contains an entirely black pixel on a place where the other shows a white pixel.

4.2.1 Dead probe artefact

Figure 4.1: Example of visual artefacts in original approach

For a demonstration of the problem presented in section 3.3 we have chosen the roof section of the house where the problem is most apparent. In Figure 4.1 you can see the error as a black section of the image.

The error is caused by a combination of multiple issues. Figure 4.2 shows the probe grid setup. The probe in the middle rendered in black is submerged into the wooden beam and, therefore, not useful for the irradiance sampling. The two probes, one to the left and one to the right, are obscured by other beams. The rest of the probe cage for the black area in the image is hidden behind the roof; thus, we end up with no probes that could be used for sampling. Moving the middle probe outside the beam could notably improve the described situation.

The following table shows absolute error for different settings of dead probes offsets in the view presented in Figure 4.1. The first line in the table represents only the baseline of direct illumination. You can see that error is almost peaking to the absolute difference between a black and white pixel. We have chosen this

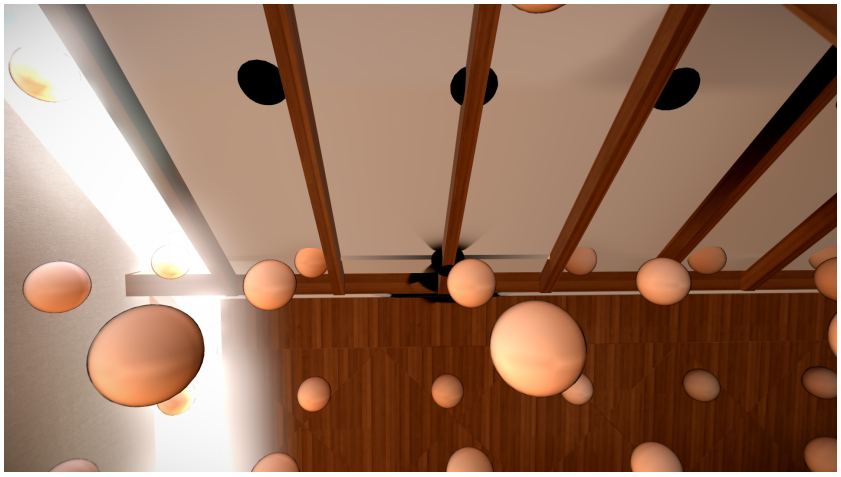


Figure 4.2: Placement of the probes used in Figure 4.1. Note that probe in the middle is submerged into the geometry.

Used variant	**Absolute error**	**Average**	**Max**
Only direct L_i	2294603	1.106580	2.925490
Simple DDGI	329610	0.158956	1.678431
Offset probes to surface	322392	0.155475	1.807843
Offset 15cm away from surface	322293	0.155427	1.701961
Improved filtering	326027	0.157228	1.647059

Table 4.2: Absolute error for different approaches to dead probes.

place precisely for this reason. Most of the view here is lit indirectly, which leaves most of the lighting on our solution.

The row denoted as Simple DDGI shows an error in the original implementation without dead probe offsetting. The drop in the error as well as maximum error is apparent, as we would expect. The next rows show the first error after probes are offset only to the nearest surface, then 15 cm away from the nearest surface. The last line shows an error with improved filtering used as described in section 3.3.1.

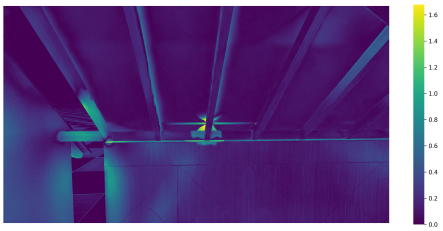
The table shows that offset of the probes to the surface slightly reduced error and was further cut down by offsetting probes away from the surface. On the other hand, the last line shows an increase of the error just below the plain DDGI implementation. This could look like a step back. But as you could see in Figure 4.4 the error has been spread more evenly across the image, and even the maximum error has dropped with the filtering.

The heatmaps in Figure 4.4 shows the spread of the error across the image. From the heatmaps, we can see the real advantage of our method. The error of the original algorithm is distinctly visible in Figure 4.4a dissolves by offsetting the probes to the surface of the geometry. The error border is smoothed out when we offset probes 15cm away from the surface. With improved trilinear filtering, we can achieve the distribution of the error to make the border of the original error barely visible.

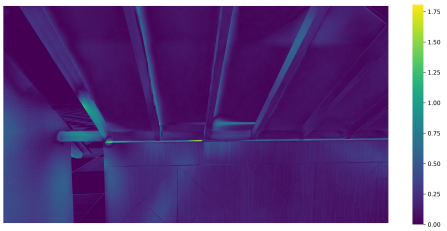
The improved distribution of the error has bigger impact on the visual qual-

ity of the image than reduction of the error. With the visible error as seen in Figure 4.1 immersion of user is being constantly disturbed. Figure 4.3 shows the improvement from the initial implementation. The error is still visible but greatly reduced.

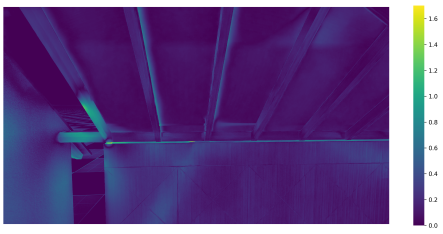
Figure 4.3: Final image with improved filtering.



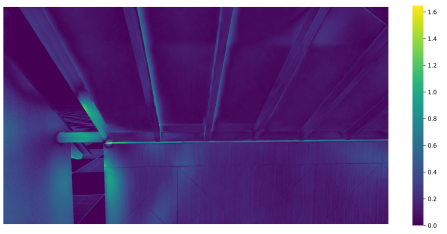
(a) Original DDGI algorithm



(b) All probes offset to the surface



(c) All probes offset at least 15cm from nearest surface



(d) Added correct trilinear filtering described in section 3.3.1

Figure 4.4: Heatmaps shows spread of the error across the image.

Depth resolution	Depth rejection	Average error	Max
8	no	0.24873	1.01569
16	no	0.23829	1.00392
32	no	0.21507	1.04706
64	no	0.18097	0.96078
16-16	yes	0.15507	0.88235

Table 4.3: This table shows how resolution of depth maps affects visual quality and how depth samples rejection changed the error.

4.3 Depth maps resolution

In this part, we would like to explore the influence of the resolution of the depth maps on reducing visual errors. There are two ways how the quality of depth maps can be changed. More pixels per probe can be designated to represent the depth map, or we can increase the precision of the pixel data. Our question for this measurement is how much we can reduce error compared to how much more we will pay in terms of memory usage. We chose the camera position for this measurement in which the visual error was most visible. This is not a representative case for a usual scene but worst case scenario. Because this thesis aims to create a general solution, we would like to mitigate exactly those situations as much as possible.

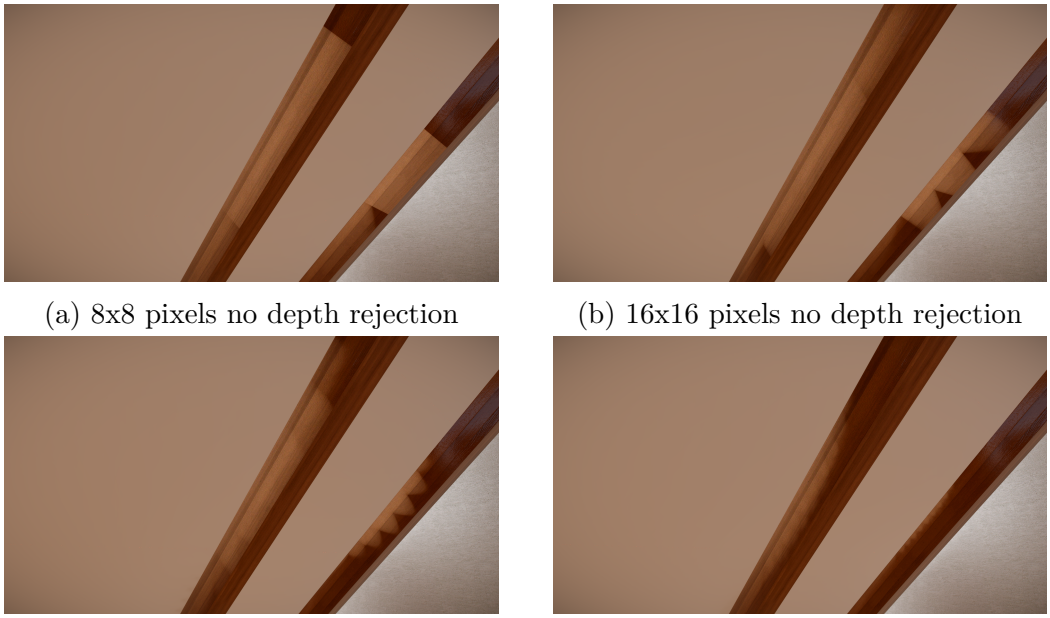
First we present results without depth rejection and afterwards the results with depth rejections. We would like to show how much this small change affected overall quality of the image.

4.4 Light bleeding

In this section we would like to address an error caused by light bleeding due to the problem described in subsection 3.3.2. The Table 4.3 shows how the resolution of depth map affects error. It shows we can bring the average error down by increasing resolution. That is expected as less samples from a given pixel could end up going towards distant geometry. We can see that even use of an enormous depth map of 64x64 pixels does not take the error down to the values from 16x16 pixels depth map that uses depth sample rejection. That is four times less memory with better results which we consider to be a huge improvement.

The visual error is still visible as shown in figures 4.5. The artefact is more disruptive in a running scene because it is not stable due to hysteresis. To quantize such error we have decided to render fifty frames of video and calculated average change of a pixel between consecutive frames. This is how table 4.4 was assembled. As the whole scene have been static during this test the ideal change between frames is 0 and higher means higher instabiltiy. Table shows that increasing depth map resolution have not affected stability significanly until the 64x64 pixels have been used (resonable setting is 16x16 pixels). Even increasing format precision have not helped much.

Figure 4.6 shows a dramatic increase of visual quality by using depth samples rejection explained in subsection 3.3.2. In Table 4.4 we can see the drop in average



(a) 8x8 pixels no depth rejection

(b) 16x16 pixels no depth rejection

(c) 32x32 pixels no depth rejection

(d) 64x64 pixels no depth rejection

Figure 4.5: Images taken without depth sample rejection and with different resolutions of depth maps.

error between the variants with and without the depth rejection. Again we have to bear in mind that we aim for better visual quality more than the reduction of numerical error.

4.5 Maps resolution

In this section we will explore the influence of the resolution of irradiance and depth maps on the final quality of the image.

Depth map size	**Precision [b]**	**Avg error**	**Max/min**
8	8	0.24058	10.19333/0.00090
16	8	0.23093	1.02153/0.00424
32	8	0.21207	1.00282/0.00333
64	8	0.17781	1.00078/0.00133
8	16	0.23795	1.01557/0.00204
16	16	0.21916	1.02145/0.00616
32	16	0.20792	1.00000/0.00424
64	16	0.18398	1.01533/0.00263
Depth rejection			
8	8	0.11866	0.71667/0.00200
16	8	0.12381	0.72020/0.00314

Table 4.4: Table shows error statistics for different resolutions for the detail view of error from Figure 4.5.

(a) 8x8 pixels with depth rejection (b) 16x16 pixels no depth rejection

Figure 4.6: Same camera view as in Figure 4.5 but with depth sample rejection used.

Depth resolution	**Irradiance resolution**	**Average error**	**Max error**	**Memory [kB]**
2x2	8x8	0.23159	2.41961	1392.2
4x4	8x8	0.22445	2.43529	1521.1
8x8	8x8	0.22374	2.43922	1933.6
16x16	8x8	0.22220	2.67451	3377.3
32x32	8x8	0.22430	2.67451	8739.8
64x64	8x8	0.22031	2.67451	29364.8
16x16	2x2	0.24147	2.67451	2294.5
16x16	4x4	0.22525	2.67451	2552.3
16x16	8x8	0.22225	2.67451	3377.3
16x16	16x16	0.22421	2.67451	6264.8
16x16	32x32	0.22420	2.67451	16989.8
16x16	64x64	0.22355	2.65490	58239.8

Table 4.5: This table shows how resolutions of each maps affects error in comparison to the ground truth

Figure 4.7: Ground truth view of the scene used for meassurements in section 4.6.

4.6 Video stability

One of the key attributes of Global Illumination is stability. For a still scene we want as stable video as possible. High changes between two consecutive frames could be disturbing for the user. In order to explore this property of our solution we have decided to set the metric for stability as average change between two consecutive frames for a short video. We are mainly interested in how different number of rays per probe affects the video stability. This could give us some insight on how we can use this parameter of the algorithm to increase the final quality of the solution.

This meassurement only make seanse for still scene without any animated objects or lights. Also, we have selected such view where most of the view is inside the DDGI AABB. The ground truth reference of this view is shown in Figure 4.7. The video sequance from which we have calculated the stability consists of fifty frames and we rendered it with six different settings.

The Figure 4.8 shows that our method of dead probe offseting did not influence the video stability. We expected this result as we have not changed how stable the calculation is we only moved the sample points.

The heatmaps in the Figure 4.9 shows that most of the instability comes from the parts of the scene with highest frequency of geometry change.

4.7 Times per algorithm stages

In this part we would like to provide some insight on how much would we pay for each stage of the pipeline.

We meassured scene from the shown in Figure 4.10 where most of the view is covered by surfaces inside DDGI AABB. We used grid of 11x10x15 probes. The irradiance map resolutin is set to 8x8 pixels and ABGR16F format is used. The images are rendered in Full HD resolution.

4.7.1 Per probe rays time influence

This test should show how the increasing rays per frame per probe affect the times of each individual step of the pipeline. We are doubling samples with each test case and measuring each pipeline step. We expected to observe doubling in the RTX step and an increase in the integration step as this step iterated over the samples for each frame. For the same reason, we would expect an increase in the dead probe detection stage because we iterate over the samples as well.

Sampels per probe	RTX	Integration	Dead probe detection	Probe offset	Shading	Total
8	0.60	0.01	0.10	0.13	0.30	1.14
16	0.84	0.09	0.05	0.09	0.37	1.44
32	1.50	0.04	0.10	0.08	0.41	2.13
64	2.71	0.02	0.05	0.10	0.48	3.36
128	5.15	0.25	0.03	0.10	0.41	5.94
256	9.56	0.79	0.09	0.11	0.48	11.01

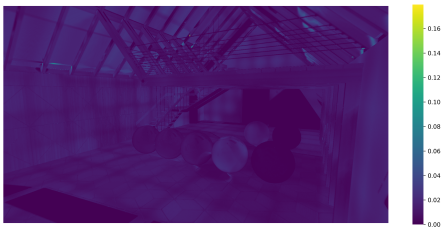
Table 4.6: This table shows the time of each step according to per probe ray shot per frame. Time values are measured in milliseconds.

We can read multiple information the Table 4.6. First, our expectation was not met. The length of the RTX step between eight and sixteen samples is not doubled. The reason is that we could not utilize all RT Core or fully employ graphic cards with so few rays. The expected increase starts to be noticeable between 32 and 64 samples.

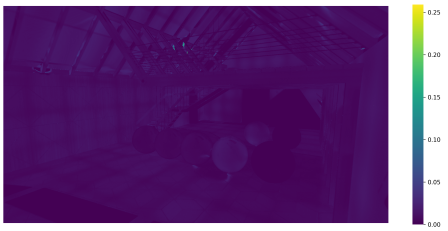
The second observation we can do here is that the integration nor dead probe detection step had not increased noticeably until 128 samples per frame. Furthermore, this increase is only visible in the integration step. This could also be caused by under-utilizing the graphic card. The integration step is doing more

Samples per probe	**Absolute change**	**Average change**	**Max change**
8	22647	0.0109	0.1796
16	16175	0.0078	0.2591
32	11443	0.0055	0.2827
64	7876	0.0038	0.2072
128	5420	0.0026	0.1339
256	3843	0.0019	0.0999
With dead probe offsets			
8	22996	0.0111	0.2151
16	15950	0.0077	0.1381
32	11411	0.0055	0.1314

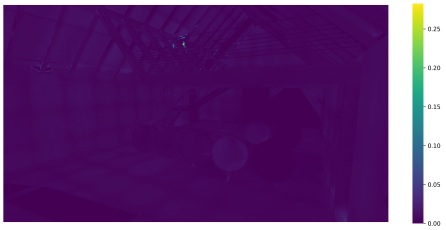
Figure 4.8: Influence of samples per probe on the stability of images. Meassured from 50 frames as difference between consecutive frames per pixel.



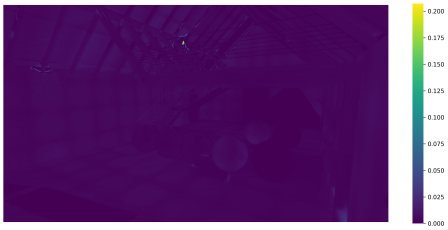
(a) 8 samples



(b) 16 samples



(c) 32 samples



(d) 64 samples

(e) 128 samples

(f) 256 samples

Figure 4.9: On those images you can see stability of the video according to number of samples per probe

work than dead probe detection. This could cause more work loaded on the GPU, which is why we probably see this dramatic increase.

Table 4.6 also shows that in most of the cases stages added by our improvements takes under 0.25ms.

4.7.2 Depth resolution time influence

This test should how the influence of depth map resolution on times individual stages of DDGI pipeline. Now we are doubling resolution of depth maps starting from 2x2 depth maps and maximum resolution being 64x64 pixels. We have not meassured bigger resolution due to hardware limitations.

Figure 4.10: Scene setup for measurements in this section.

Depth map resolution	RTX	Integration	Dead probe detection	Probe offset	Shading	Total
2	2.5	0.08	0.20	0.10	0.25	3.14
4	2.5	0.15	0.23	0.09	0.23	3.20
8	2.5	0.25	0.10	0.11	0.24	3.20
16	2.5	0.37	0.05	0.21	0.22	3.35
32	2.5	1.12	0.05	0.27	0.23	4.17
64	2.5	4.03	0.05	0.07	0.35	7.00

Table 4.7: This table shows the time of each step with increasing depth map resolution. Time values are measured in milliseconds.

4.8 Indirect lighting

Here we would like to show the magnitude of influence that adding indirect lighting into the renderer has on the final result. The Figure 4.11 shows the difference between a render with only direct lighting on the left and a scene with direct and indirect lighting on the right.

Figure 4.11: Direct illumination (left) in comparison with global illumination (right) computed using DDGI with automatic probe offseting

5. Conclusion

In this thesis, we aimed to build a real-time global illumination solution that does not need any offline precomputation. The requirements were to support time of day, dynamic geometry, and animated objects within a scene alongside a realistic lighting model. We set our goal to implement the method chosen based on the needs of the existing system and improve that solution to the state we will be able to use in a production-ready software.

This thesis has shown how to implement Dynamic Diffuse Global Illumination into a custom real-time renderer. We have investigated its properties and their influence on the quality of the resulting image. We have provided detailed measurements of this technique in Chapter 4.

We have proposed and implemented a solution to some of the observed problems based on initial measurements. This thesis has shown that our improvements positively influenced the quality of the synthesized image.

We have presented some issues with DDGI in section 3.3. We have proposed improvements to reduce those issues, also we have found some problems unsolvable.

Despite those improvements, our goal of a fully automatic global illumination solution for the Fibix engine has not been fulfilled. Our solution works in most cases and needs a manual setting of the properties in very few cases. The problem of global illumination is broad and solving all corner cases is too complex. We have proposed solutions for some of the problems in section 5.1. However, despite not meeting these criteria, we decided to use our solution and build on it in the future.

Our method of dynamically offsetting the probes improves the visual quality of DDGI in all test cases we tried. Improvements in the absolute error are not as significant as perceivable error, but we achieved a better spread of the error over the whole image.

The main improvement lies in the fact that the error is not immediately visible and noticeable without having a ground-truth path-traced reference for comparison. This means that users of the software will be more immersed in the virtual world, which is our goal. The fact that absolute error has not improved as much is not the issue for our use case. That is because the users of the Fibix engine will anyway use an offline path tracer to render a final correct image. Hence, our solution only serves as a real-time preview and needs to provide a good approximate understanding of the lighting inside the scene, which we believe it does well.

Our depth rejection method for depth samples beyond the useful distance has decreased visual glitches to the minimum.

We have proposed a solution that enabled the usage of the probes that would be not used with the original method. Such probes were either wholly submerged into the geometry or too close to the geometry to be optimally utilized. We have also shown that our method does not influence the video stability of the rendered result.

5.1 Future work

During our work on this thesis, we have realized that fully automatic global illumination is a problem too broad for the scale of this thesis. As a result, we surely have not explored all the possibilities to make our solution more robust. The list of possible ways we have considered follows.

Attaching the probes' AABB to the camera Currently, our system limited is to a predefined AABB box. This is not a problem for small scenes we aimed at for architectural visualization. However, for open-world games or bigger scenes with multiple buildings, it could be limited. This should not be a huge task. We will implement new mapping from world space to grid space aware of moving AABB.

Probe cascades Our method depends on a uniform, regular grid. This is convenient for world space to grid space normalized mapping, but it also means that for huge probe AABB boxes, we have just the same level of detail for irradiance even though we do not need the same detail a few kilometres away from camera as we need in the immediate surroundings of the camera. To achieve this, we can employ cascades similarly to those used in shadow maps Dimitrov [2007].

Ray budgeting Right now, our solution uses the uniform distribution of ray budget each frame between probes. This works well, but we could do better. We can come up with some heuristics for rays distribution. One solution could be manually feeding data about changes in the scene from the engine code. A better solution could be to think a little about where we want finer sampling. Those are such parts of a scene that encountered the biggest change of lighting in the last frames. We can use similar approach as proposed by Global Illumination Based on Surfels described in section Global Illumination Based on Surfels.

Dynamic hysteresis We can also change the speed of update by using a more informed way to set hysteresis. Right now, we use user-defined hysteresis. We could use the same approach as with ray budgeting and change hysteresis based on a variance of a short term mean-variance estimator proposed by Global Illumination Based on Surfels described in section Global Illumination Based on Surfels.

Bibliography

Colin Barré-Brisebois, Henrik Halén, Graham Wihlidal, Andrew Lauritzen, Jasper Bekkers, Tomasz Stachowiak, and Johan Andersson. Hybrid rendering for real-time ray tracing. *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs, Haines E., Akenine-Möller T.,(Eds.). Apress*, pages 437–473, 2019.

Pafnutii Lvovich Chebyshev. Des valeurs moyennes. *J. Math. Pures Appl*, 12(2): 177–184, 1867.

Zina H. Cigolle, Sam Donow, Daniel Evangelakos, Michael Mara, Morgan McGuire, and Quirin Meyer. A survey of efficient representations for independent unit vectors. *Journal of Computer Graphics Techniques (JCGT)*, 3 (2):1–30, April 2014. ISSN 2331-7418. URL `http://jcgt.org/published/0003/02/01/`.

F.J.J. Clarke and D.J. Parry. Helmholtz reciprocity: its validity and application to reflectometry. *Lighting Research & Technology*, 17(1):1–11, 1985. doi: 10.1177/14771535850170010301. URL `https://doi.org/10.1177/14771535850170010301`.

Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, and Elmar Eisemann. Interactive indirect illumination using voxel cone tracing. In *Computer Graphics Forum*, volume 30, pages 1921–1930. Wiley Online Library, 2011.

Carsten Dachsbacher and Marc Stamminger. Reflective shadow maps. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, I3D '05, page 203–231, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595930132. doi: 10.1145/1053427.1053460. URL `https://doi.org/10.1145/1053427.1053460`.

Rouslan Dimitrov. Cascaded shadow maps. *Developer Documentation, NVIDIA Corp*, 2007.

William Donnelly and Andrew Lauritzen. Variance shadow maps. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 161–165, 2006.

Philip Dutré, Kavita Bala, and Philippe Bekaert. *Advanced global illumination*. A K Peters, Wellesley, 2nd ed edition, 2006. ISBN 9781568813073.

Albert Einstein and Leopold Infeld. *The evolution of physics: the growth of ideas from early concepts to Relativity and quanta*. An Essandess paperback. 1961.

Nick Stam Emmett Kilgariff, Henry Moreton and Brandon Bell. Nvidia turing architecture in-depth. URL `https://developer.nvidia.com/blog/nvidia-turing-architecture-in-depth/`.

Randima Fernando. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education, 2004. ISBN 0321228324.

Fibix Studio. Fibix studio, 2021. URL https://fibix.eu/index.php.

R. Green. Spherical harmonic lighting : the gritty details. *Game developer's conference, 2003*, 2003.

Eric Haines and Tomas Akenine-Möller, editors. *Ray Tracing Gems*. Apress, 2019. http://raytracinggems.com.

Henrik Halen, Andreas Brinck, Kyle Hayward, and Xiangshun Bei. Siggraph 21: Global illumination based on surfels, 2021. URL http://advances.realtimerendering.com/s2021/index.html.

John C. Hart. Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12(10):527–545, December 1996. doi: 10.1007/s003710050084. URL https://doi.org/10.1007/s003710050084.

Jinkai Hu, Milo K. Yip, Guillermo Elias Alonso, Shihao Gu, Xiangjun Tang, and Xiaogang Jin. Signed distance fields dynamic diffuse global illumination. *CoRR*, abs/2007.14394, 2020. URL https://arxiv.org/abs/2007.14394.

Henrik Wann Jensen. Global illumination using photon maps. In Xavier Pueyo and Peter Schröder, editors, *Rendering Techniques '96*, pages 21–30, Vienna, 1996. Springer Vienna. ISBN 978-3-7091-7484-5.

Benjamin Keinert, Matthias Innmann, Michael Sänger, and Marc Stamminger. Spherical fibonacci mapping. *ACM Transactions on Graphics*, 34:1–7, 10 2015. doi: 10.1145/2816795.2818131.

Johann Heinrich Lambert. *Photometry, or, On the measure and gradations of light, colors, and shade*. Illuminating Engineering Society of North America, 2001. Original "Photometria sive de mensure et gradibus luminis colorum et umbra" translated to English by David L. DiLaura.

Hayden Landis. Production-ready global illumination. *Siggraph course notes*, 16 (2002):11, 2002.

Martin-Karl Lefrançois and Pascal Gautron. Dx12 raytracing tutorial - part 2. URL https://developer.nvidia.com/rtx/raytracing/dxr/dx12-raytracing-tutorial-part-2.

Jérôme Maillot, Hussein Yahia, and Anne Verroust. Interactive texture mapping. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 27–34, 1993.

Zander Majercik, Jean-Philippe Guertin, Derek Nowrouzezahrai, and Morgan McGuire. Dynamic diffuse global illumination with ray-traced irradiance fields. *Journal of Computer Graphics Techniques (JCGT)*, 8(2):1–30, June 2019. ISSN 2331-7418. URL http://jcgt.org/published/0008/02/01/.

Daniel Meister, Shinji Ogaki, Carsten Benthin, Michael J. Doyle, Michael Guthe, and Jiří Bittner. A survey on bounding volume hierarchies for ray tracing. *Computer Graphics Forum*, 40(2):683–712, 2021. doi: https://doi.org/10.1111/cgf.142662. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.142662.

Microsoft. Directx raytracing (dxr) functional spec. URL https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html.

Andrew W. Moore. An intoductory tutorial on kd-trees, 1991.

Thomas Müller, Markus Gross, and Jan Novák. Practical path guiding for efficient light-transport simulation. In *Computer Graphics Forum*, volume 36, pages 91–100. Wiley Online Library, 2017.

NVIDIA Corporation. Nvidia rtx technology, 2021. URL https://www.nvidia.com/en-us/design-visualization/technologies/rtx/.

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*, SIGGRAPH '01, page 497–500, New York, NY, USA, 2001. Association for Computing Machinery. ISBN 158113374X. doi: 10.1145/383259.383317. URL https://doi.org/10.1145/383259.383317.

Dominik Roháček. Shadow rendering using view frustum splitting methods in opengl, 2018.

Yang Sa. Improved bilinear interpolation method for image fast processing. In *2014 7th International Conference on Intelligent Computation Technology and Automation*, pages 308–311, 2014. doi: 10.1109/ICICTA.2014.82.

Tomasz Stachowiak. Stochastic all the things: Raytracing in hybrid real-time rendering. *SEED, Digital Dragons*, 2018.

Masamichi Sugihara, Randall Rauwendaal, and Marco Salvi. Layered reflective shadow maps for voxel-based indirect illumination. In *High Performance Graphics*, pages 117–125, 2014.

Umut Uyurkulak. Voxel-based global illumination (svogi), 2021. URL https://docs.cryengine.com/pages/viewpage.action?pageId=25535599.

Fan Zhang Hanqiu Sun Leilei Xu and Lee Kit Lun. Parallel-split shadow maps for large-scale virtual environments. 2010.

List of Figures

1.1 Visual explanation of the radiance. (Image source: Jaroslav Křivánek) 5
1.2 Ray casting technique. This picture shows how the algorithm generates the final image. (Image source: Haines and Akenine-Möller [2019]) 7
1.3 Turing Ray Tracing with RT Cores. (Image source: Emmett Kilgariff and Bell) 9
1.4 This is an example of the difference between direct lighting only and full global illumination inside the Cornell box. (Image source: Jaroslav Křivánek, Supplemental resources for MFF CUNI course Computer Graphics III - Realistic image synthesis (NPGR010)) . 10
1.5 Texture atlas divided in a squares each representing one irradinace probe. (Image source: Thesis author) 11

2.1 Light mapping allows rendering of complex lighting phenomena such as caustics. (Jensen [1996]) 12
2.2 World voxelized for SVOTi with cascades color coded by voxel outline [Uyurkulak, 2021]. 13
2.3 Albedo colours, directl light injection and light propagation. (Uyurkulak [2021]) 14
2.4 Signed distnace field Global Illumination uses also a grid of probes placed inside the scene. (Jensen [1996]) 15
2.5 Discretization of the surface using surfels. (Halen et al. [2021]) . . 16
2.6 Grid used for uniform coverage of screen space. (Stachowiak [2018]) 17
2.7 Importance sampling using inverse Cumulative Distribution Function. (Image source: Halen et al. [2021]) 18
2.8 This figure shows the data stored for Dynamic Diffuse Global Illumination and placement of probes on uniform grid in a test scene. Image by Majercik et al. [2019] 19
2.9 Process of unit vector mapping 20
2.10 Shading of fragment X. Each fragment lies inside probe cage defined by corners of cage having integral coordinates in grid normalized space. Fragment X has normal facing in the direction n and we can compute direction dir to each probe P belonging to this cage and distance to this probe P denoted by r. (Image source: Majercik et al. [2019]) 21

3.1 Overview of the RTX pipeline usable by DXR. (Image source: Lefrançois and Gautron) 23
3.2 Full lifecycle of ray in DXR ray tracing pipeline. (Image source: Microsoft) 28
3.3 Visualization of gutter region UV mapping. (Image source: Thesis author) 29
3.4 Comparison of smooth and sharp backface culling. (Image source: Thesis author) 31

3.5 This drawing shows a situation when all probes are obscured by geometry. The dashed line shows the border of the probe cage. Fragments highlighted in red have no valid information about indirect lighting. (Image source: Thesis author) 35
3.6 This image shows how submerging non-manifold geometry is used in 3D scenes. The white stripe in the middle represents the ceiling. (Image source: Thesis author) 36
3.7 Comparison of smooth and sharp backface culling. (Image source: Thesis author) 37
3.8 Light bleeding due to long depth samples outside of probe cage. (Image source: Thesis author) 39
3.9 A debug visualistaion on the probes 39
3.10 Application UI for DDGI settings. (Image source: Screenshot from Fibix engine) 41
3.11 Debug of one layer of depths texture (Image source: Screenshot from Fibix engine) 42

4.1 Example of visual artefacts in original approach 44
4.2 Placement of the probes used in Figure 4.1. Note that probe in the middle is submerged into the geometry. 45
4.3 Final image with improved filtering. 46
4.4 Heatmaps shows spread of the error across the image. 46
4.5 Images taken without depth sample rejection and with different resolutions of depth maps. 48
4.6 Same camera view as in Figure 4.5 but with depth sample rejection used. 49
4.7 Ground truth view of the scene used for meassurements in section 4.6. 50
4.8 Influence of samples per probe on the stability of images. Meassured from 50 frames as difference between consecutive frames per pixel. 51
4.9 On those images you can see stability of the video according to number of samples per probe 52
4.10 Scene setup for measurements in this section. 53
4.11 Direct illumination (left) in comparison with global illumination (right) computed using DDGI with automatic probe offseting . . . 54

List of Tables

4.1 This table show how much resolution and format choice affect probes memory footprint. 43
4.2 Absolute error for different approaches to dead probes. 45
4.3 This table shows how resolution of depth maps affects visual quality and how depth samples rejection changed the error. 47
4.4 Table shows error statistics for different resolutions for the detail view of error from Figure 4.5. 48
4.5 This table shows how resolutions of each maps affects error in comparison to the ground truth 49
4.6 This table shows the time of each step according to per probe ray shot per frame. Time values are measured in milliseconds. 51
4.7 This table shows the time of each step with increasing depth map resolution. Time values are measured in milliseconds. 53

A.1 This table explains name coding of the measured scenarios 65
A.2 This table explains contents of code files in ./Code/ folder. 66

List of Listings

3.1 Simple ray generation shader. 24
3.2 Any hit short example that handles transparency for shadow rays. 25
3.3 Example of Closes hit shader starting new reflection ray and storing values to the memory. 25
3.4 Miss shader example with sky dome or callable shader call. 26
3.5 Callable shader example. 27
3.6 This function maps the atlas UV coordinates to probe UV coordinates. This mapping also takes care of the gutter region correct translation. Can be found in *DDGIMappings.shader* 29
3.7 Ray generation for Fibbonaci sampling used in DDGI. The shader is heavily simplified for better readablity. Can be found in *RTX.fx* 32
3.8 Mapping wolrd space position of the fragment to the base coordinates of its probe cage. Can be found in *DDGIMappings.shader* . 33
3.9 Smooth and sharp backface culling and view vector normal offset code snippet. Can be found in *PSShader.fx* 33
3.10 Chebyshev's inequality term used in visibility determination during the probe evaluation. Can be found in *PSShader.fx* 34
3.11 Original trilinear filtering. All coordinates are in grid cage normalized space. Can be found in *PSShader.fx* 37
3.12 Improvedfiltering. All coordinates are in grid cage normalized space. Can be found in *PSShader.fx* 37
3.13 Offset sampling code from ray generation shader for the dead probes offset improvement. Can be found in *PSShader.fx* 38

List of Abbreviations

AABB Axis Aligned Bounding Box. 13, 26, 30, 40, 50, 56

AI Artificial Intelligence. 7

API Application Programming Interface. 7, 8, 24, 27

BRDF Bidirectional reflectance distribution function. 6, 7, 12, 17, 25

BVH Bounding Volume Hierarchy. 6, 8

cdf Cumulative Distribution Function. 17, 18, 60

CUDA Compute Unified Device Architecture. 7

DDGI Dynamic Diffuse Global Illumination. iii, 3, 4, 15, 18, 19, 21, 23, 27, 28, 37, 38, 40, 41, 43, 45, 46, 50, 52, 54, 55, 60, 61, 65, 66

DXR DirectX Raytracing. 8, 23, 27, 28, 32, 38, 60

GI Global Illumination. 9, 10, 15, 16, 50, 60

GIBS Global Illumination Based on Surfels. 56

LRSM Layered Reflective Shadow Maps. 14

pdf Probability density function. 17

RSM Reflective Shadow Map. 15

SDF Signed distnace field. 15, 60

SVOGi Sparse Voxel Octree Global Illumination. 13

SVOTi Sparse Voxel Octree Total Illumination. 13, 60

UAV Unordered Access View. 25

A. Appendix A

Electronic attachments

Attached to this thesis in electronic form is our implementation and examples of resulting renders.

The following directory tree shows structure of the attachement:

```
/
├── Code ...  Our implementation of DDGI
├── Videos ...  Videos comparison of
│               path-traced reference
│               and our technique
└── img ...  Content of this folder
             is divided into
             subdirectories with
             images with identical
             view but different
             settingsDetails on
             settings are described
             in Table A.1
```

File name	Contains
GT	ground truth
DirLi	image with only the direct illumination
Original	The original DDGI technique
Offset0	The DDGI technique with dead probes pushed to the surface as described in section 3.3.1
Offset*XX*	The DDGI technique with dead probes pushed *XX* cm away from the nearest surface as described in section 3.3.1
DepthClamp/NoClamp	Those files shows difference between depth sample rejection turned on and off
betterfiltering	This image shows image with our improved sampling described in section 3.3.1.

Table A.1: This table explains name coding of the measured scenarios

The code directory contains code written in C++ and HLSL languages. This is a crucial part of the algorithm needed to reproduce our implementation. Because the Fibix engine is the intellectual property of the Fibix studio, we could not publish the whole code base, but we have presented all work crucial for our research. You can find explanation for those files in Table A.2.

File name	Contains
DDGIAdjustProbes.fx	This is compute shader for probe offsetting.
DDGIDeltaTexture.fx	This file calculates whether probe is submerged in geomtery. Here we apply the dead probe hysteresis.
DDGIMappings.shader	This file contains all mapping between spaces described
DDGIProbeUpdate.fx	This file contains all mappings between the spaces we us. Description of those spaces is in subsection 3.2.3
OctahedronMapping.h	The octahedral mappings. This mappings originates from Cigolle et al. [2014].
DepthClamp/NoClamp	Those files shows difference between depth sample rejection turned on and off.
DDGIRenderer.h/cpp	This file contains controller of the whole DDGI technique.
PSShader.fx	This file contains indirect lighting sampling.
RTX.fx	This file only contain ray generation RTX shader as the only RTX shader specific to our solution.

Table A.2: This table explains contents of code files in ./Code/ folder.