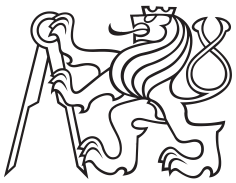
Master Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Dynamic Diffuse Global Illumination

Michal Hvězda

Supervisor: doc. Ing. Jiří Bittner, Ph.D.
Field of study: Open Informatics
Subfield: Computer graphics
May 2022

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Hvězda** Jméno: **Michal** Osobní číslo: **459888**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačové grafiky a interakce**

Studijní program: **Otevřená informatika**

Specializace: **Počítačová grafika**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Dynamické Difuzní Globální Osvětlení

Název diplomové práce anglicky:

Dynamic Diffuse Global Illumination

Pokyny pro vypracování:

Zmapujte existující metody pro výpočet globálního osvětlení v reálném čase. Implementujte metodu využívající reprezentaci osvětlení ve scéně pomocí prostorových sond osvětlení vypočítaných pomocí metody sledování paprsků [1]. Soustřeďte se na efektivní začlenění metody vrhání paprsků v reálném čase do implementace metody. Implementujte režimy využívající výpočet přímého osvětlení pomocí vrhání stínových paprsků a pomocí rekonstrukce ze sond osvětlení. Vyhodnoťte vlastnosti implementované metody z hlediska kvality výsledného obrazu a rychlosti výpočtu na nejméně třech testovacích scénách. Navrhněte a zdůvodněte možná vylepšení metody.

Seznam doporučené literatury:

[1] 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), vol. 8, no. 2, 1-30, 2019
[2] Zander Majercik, Thomas Mueller, Alexander Keller, Derek Nowrouzezahrai, and Morgan McGuire. Dynamic Diffuse Global Illumination Resampling. In ACM SIGGRAPH 2021 Talks (SIGGRAPH '21). Article 24, 1–2, 2021.
[3] Ari Silvennoinen, Jaakko Lehtinen. Real-time Global Illumination by Precomputed Local Reconstruction from Sparse Radiance Probes. ACM Transactions on Graphics, Vol. 36, No. 6, Article 230, 2017.
[4] Peter-Pike Sloan, Jan Kautz, and John Snyder. Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments. ACM Transactions on Graphics, Vol. 36, No. 6, Article 230, 2017.
[5] Jaroslav Křivánek, Kadi Bouatouch, Sumanta Pattanaik, and Jiří Žára. Making radiance and irradiance caching practical: Adaptive caching and neighbor clamping. In Rendering Techniques 2006 (Proc. of Eurographics Symposium on Rendering), pages 127–138, 2006
[6] Tobias Ritschel Carsten Dachsbacher Thorsten Grosch Jan Kautz. The state of the art in interactive global illumination. In Computer Graphics Forum (Vol. 31, No. 1, pp. 160-188), 2012.
[7] Šimon Sedláček. Real-Time Global Illumination using Irradiance Probes. Diplomová práce, ČVUT FEL, 2019.

Jméno a pracoviště vedoucí(ho) diplomové práce:

doc. Ing. Jiří Bittner, Ph.D. Katedra počítačové grafiky a interakce

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **09.02.2022** Termín odevzdání diplomové práce: **20.05.2022**

Platnost zadání diplomové práce: **30.09.2023**

doc. Ing. Jiří Bittner, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. _____
Datum převzetí zadání

_____
Podpis studenta

Acknowledgements

I would like to thank my supervisor doc. Ing. Jiří Bittner, Ph.D. as he was supportive during my work on this thesis and encouraged me to continue in this field of study beyond the scope of this thesis.

Declaration

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškerou použitou literaturu. V Praze, 20 May 2022

Abstract

We present our implementation of a recent real-time global illumination method based on irradiance fields, which uses ray tracing to approximate multiple bounces of light transmission in a virtual scene. Then we present our method, which repurposes the global illumination algorithm to compute global ambient occlusion in real time. We also present our extensions to the algorithm, such as field cascades and geometry-aware probe culling. Lastly, we discuss possible further extensions, such as adaptive ray generation and probe placement.

Keywords: DDGI, Real-time global illumination, ray-tracing, ambient occlusion

Supervisor: doc. Ing. Jiří Bittner, Ph.D.

Abstrakt

Představujeme naši implementaci algoritmu pro výpočet globálního osvětlení v reálném čase založený na světelných polích z nedávno vydaného článku. Tato metoda využívá sledování paprsků pro aproximaci několika násobného odrazu světla ve virtuální scéně. Dále představujeme naši metodu, která využívá konceptů implementovaného algoritmu pro výpočet globálního okolního zastínění v reálném čase. Představujeme také námi navrhnuté rozšíření algoritmu jako kaskády světelných polí a selektivní aktualizace sond v závislosti na okolní geometrii. Nakonec, diskutujeme nad možnými dalšími rozšířeními jako adaptivní množství generových paprsků a umístění sond.

Klíčová slova: DDGI, Globální osvětlení v realném čase, trasování paprsků, zastínění okolím

Překlad názvu: Dynamické Difůzní Globální Osvětlení

Contents

Figures

Tables

Chapter 1

Introduction

Figure 1.1: Sponza scene using environment mapping (left) and DDGI (right).

Accurate illumination in a virtual scene is often the most crucial visual effect in computer graphics. Correct light transfer simulation blurs the line between an actual image and an artificial one. Furthermore, the result can yield stunning images even in stylized graphical applications. However, the computation of correct light propagation is costly, and thus global illumination is often only approximated for real-time applications.

With the rise of the ray-tracing GPUs in recent years, much development into algorithms that try to leverage the potential of the hardware has spurred up. We implemented one such algorithm called Dynamic Diffuse Global illumination (DDGI) from a recent paper by Majercik et al. [MGNM19] which complements traditional rasterization with a ray-traced irradiance field to approximate a diffuse global illumination for dynamic illumination and geometry. We also describe our extensions to the algorithm, which focus on improving the technique's performance and the quality of the final image.

We were intrigued by how DDGI handles shadowing, so we tried to repurpose the algorithm to approximate global ambient occlusion and shadows. We detail our changes to the DDGI method to capture occlusion and describe our implementation.

As a starting point for our real-time application, we used G3D Innovative Engine [MMM17] which supports GPU raytracing using NVidia's OptiX framework [PBD+10]. The DDGI method was implemented as part of our custom renderer, which uses some of the G3D's features: a physically-based illumination model, deferred shading, and shadow mapping.

The thesis is structured as follows: Chapter 2 gives an overview of the

relevant work in the computer graphics field. Namely, we explain the theory behind the rendering equation and how it is approximated using Monte Carlo integration which is used by global illumination techniques. Then, we also give an overview of some of the methods which are used today to approximate global illumination in a virtual scene.

In Chapter 3 we describe some of the techniques our rendering pipeline uses which are relevant to DDGI but not directly linked to it. Furthermore, Chapter 4 will detail the DDGI method to approximate diffuse global illumination in a fully dynamic scene. In Chapter 5 we present our modifications to the DDGI algorithm to compute global shadowing. Also, in Chapter 6 we discuss our proposed extensions to the algorithm, which are also used by our ambient occlusion modification of DDGI. Our results are detailed in Chapter 7, where we compare the quality of the resulting images to a path-traced reference. Last but not least, Chapter 8 discusses the drawbacks and possible further improvements to the global illumination method.

Chapter 2

Related Work

Since the implemented algorithm belongs to a class of algorithms for real-time global illumination, we separated this chapter into multiple sections. First, in Section 2.1 we detail the theory behind the light transport with no surrounding media described by the rendering equation. Then in Section 2.2 we give an overview of existing solutions for global illumination methods used in real-time applications.

2.1 Rendering Equation

Rendering equation [Kaj86] states that the outgoing radiance L_o from a point on a surface $\boldsymbol{x}$ in a direction $\boldsymbol{\omega}$ is a sum of emitted radiance L_e and reflected radiance L_r [ŽBS+05]:

$$L_o(\boldsymbol{x}, \boldsymbol{\omega}) = L_e(\boldsymbol{x}, \boldsymbol{\omega}) + L_r(\boldsymbol{x}, \boldsymbol{\omega}) \tag{2.1}$$

where the reflected radiance L_r is computed as:

$$L_r(\boldsymbol{x}, \boldsymbol{\omega}) = \int_{\Omega} f(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{\omega}_i) L_i(\boldsymbol{x}, \boldsymbol{\omega}_i) cos\Theta \, d\boldsymbol{\omega}_i \tag{2.2}$$

where Ω is the upper hemisphere, $f(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{\omega}_i)$ is the bidirectional reflectance distribution function - BRDF, $L_i(\boldsymbol{x}, \boldsymbol{\omega}_i)$ is the incident radiance and Θ is an angle between $\boldsymbol{\omega}_i$ and a normal vector $\boldsymbol{N}$ of the surface [ŽBS+05]. In the original formulation of the rendering equation the incident radiance is formulated as [Kaj86]:

$$L_i(\boldsymbol{x}, \boldsymbol{\omega}_i) = L_o(\boldsymbol{x}, -\boldsymbol{\omega}_i) \tag{2.3}$$

It can be seen that the rendering equation is one of the Fredholm equations of the second kind. The Fredholm equations introduce a complication that the unknown lies on both sides of the equation [ŽBS+05].

Alternatively, the reflection integral in L_r can also be formulated as an integral over the whole scene's surface S instead of directions [RDGK12]. Therefore if one interprets L_r as such the resulting rendering equation would look like [ŽBS+05]:

$$L_o(\boldsymbol{x}, \boldsymbol{\omega}) = L_e(\boldsymbol{x}, \boldsymbol{\omega}) + \int_S f(\boldsymbol{x}, \boldsymbol{x'} \rightarrow \boldsymbol{x}, \boldsymbol{\omega}) L_i(\boldsymbol{x'} \rightarrow \boldsymbol{x}) V(\boldsymbol{x}, \boldsymbol{x'}) G(\boldsymbol{x'}, \boldsymbol{x}) \, dA' \tag{2.4}$$

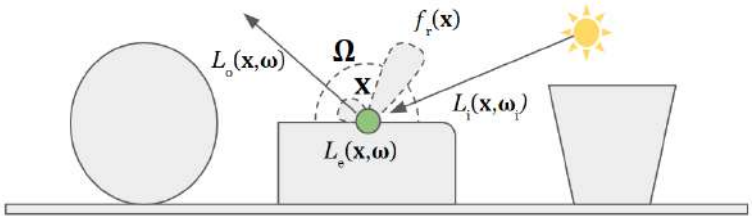


Figure 2.1: Spherical representation of the rendering equation

where S is the surface of the entire scene, $V(\boldsymbol{x})$ is a visibility term which says that if location x is visible from point x' and vice versa [ŽBS+05]:

$$V(\boldsymbol{x}, \boldsymbol{x'}) = \begin{cases} 1 & \boldsymbol{x} \text{ is visible from } \boldsymbol{x'} \\ 0 & \text{otherwise} \end{cases} \tag{2.5}$$

Geometry term $G(\boldsymbol{x}, \boldsymbol{x'})$ contains coefficients which express projections of emitted radiance after the ray leaves point $\boldsymbol{x'}$ and after it hits point $\boldsymbol{x}$.

To summarize, the surface formulation of the rendering equation says that to compute outgoing radiance from location $\boldsymbol{x}$ in the direction $\boldsymbol{\omega}$, the contributions from the whole scene's surface need to be taken into account. Specifically, the visibility term of all points $\boldsymbol{x'}$ and geometry term is computed if a point $\boldsymbol{x}$ is visible. The geometry term is then multiplied together with the outgoing radiance at the point $\boldsymbol{x}$ and the BRDF. The goal of GI algorithms is to compute $L_o(\boldsymbol{x}, \boldsymbol{\omega})$ for a given scene, materials and lighting L_e.

2.2 Global Illumination Methods

Since global illumination is costly to compute, GI methods usually resorted to some trade-off to calculate the effect. The most notable is the use of a fully static scene. For scenes where only the camera is dynamic, we can use any of the well-established offline methods, e.g. *path-tracing* [Kaj86]. Using such techniques, the light transfer would be precomputed and stored it in textures beforehand for later rendering. This process is often called light baking.

However, for dynamic or at least partially dynamic scenes, the real-time global illumination problem becomes more difficult. The paper by Ritschel et al. [RDGK12] separates the classical approaches of interactive global illumination into multiple categories and this chapter makes an overview of some of these approaches.

2.2.1 Finite Elements

The first category would be Finite Elements (Radiosity) methods [GTGB84], whereas the name implies the scene is discretized into a finite number of surface elements. The approach completely omits the camera view from its calculations and only considers the light transfer between the surface patches.

The original method is suitable for diffuse light transfer and later was extended [ICG86] by accounting for glossy materials. The quality of the global illumination is dependent on the number of patches which introduces the problem of scalability. Thus, due to the quadratic nature of the light transfer calculation, the original approach does not scale well. However, P. Hanrahan et al. [HSA91] introduced a hierarchical approach to solving the light transfer thus reducing the complexity of the problem.

2.2.2 Photon Mapping

Another approach would be *photon mapping* [Jen96]. The idea behind photon mapping is to emit a large number of photons from the light source and let them bounce inside the scene. At each hit point, the photon is stored inside a photon map. During rendering, the photon map is used to determine the irradiance of each fragment based on the photon density in the area.

The method can be adapted to achieve interactive rates under some conditions. Purcell et al. [PDC+05], and Ma and McCool [MM02] described an approach where they used spatial hashing instead of the nearest neighbor search required in density estimation, which better fits GPUs.

2.2.3 Monte Carlo Ray Tracing

A possible approach to solving GI is the use of Monte Carlo techniques [Kaj86]. These techniques produce A high number of directional samples, which are evaluated for incoming light, and the mean of the results converges to the correct solution.

In order to evaluate a sample, the incoming light from one direction has to be computed. This computation is usually done by ray-tracing. A ray is sent, and the light emitted from the first hit point is computed, potentially again computing a solution of the rendering equation at the point.

Generating random samples that potentially have a low contribution to the final value is rather inefficient. Instead the sample generating high contribution samples to accelerate the convergence to the correct solution would be more desirable. To do so, many techniques exist such as importance sampling [RDGK12].

Instead of blindly sending rays everywhere, rays are focused where the rendering function integral yields the highest values. This is easy for direct illumination. However, indirect illumination is more complicated because the origins of rendering function's highest values are not known beforehand. Furthermore, combining the factors of the rendering equation integrand, such as BRDF and incoming light L_i into one importance sampling approach is difficult already for direct illumination. One solution to this problem presents Bi-directional pathtracing from the paper by Lafortune et al. [LW93].

The Monte Carlo integration is a complex problem for millions of dynamic lights. The paper by Bitterli et al. [BWP+20] introduces an algorithm called ReSTIR that renders such lights interactively. They achieve this by repeatedly resampling a set of candidate light samples and leveraging information from

relevant nearby samples by further spatial and temporal resampling. Later Majercit et al. [MMK+21] proposed to combine ReSTIR's shadowing with DDGI. Combining these two algorithms outperforms hardware accelerated path tracing in both runtime and noise.

Silvennoinen et al. [SL17] presented a radiance field method for mostly static scenes with dynamic lights, cameras, diffuse, and emissive materials. Their approach features minor light leaking due to their algorithm for faithfully interpolating incidence radiance captured at a sparse set of low-frequency radiance probes to nearby receiver points.

2.3 Shadowing Methods

In recent years many shadowing methods have been presented. Most use a single sample hard shadow maps together with some smart filtering method to compute perceptually or even physically accurate soft shadows. One such technique is called variance shadows [DL06] which creates physically correct shadows by reducing aliasing using approximation of how shadows soften on their edge.

On the other hand there is the class of algorithms that calculate physically correct soft shadows by taking and combining many light samples. Paper by D. Sherzer et al.[SSMW09] proposes a method which samples the light source over multiple frames. Then they make use of temporal coherence and spatial filtering to create correct and fast soft shadows.

Another class of shadowing algorithms is ambient occlusion. Ambient occlusion is a cheap but effective approximation of high frequency global illumination. Methods such as screen-space ambient occlusion (SSAO) [BS08], which sample the framebuffer as a discretization of the scene geometry, have become rather popular for real-time rendering. SSAO however suffers from bluring and noise artifacts due to strong spatial filtering. A paper on temporal SSAO (TSSAO) by Mattausch et al. [MSW10] solves this issue by caching and reusing previously computed SSAO samples.

Chapter 3

Rendering Pipeline

Figure 3.1: A diagram of steps which our renderer takes in each frame to compute the final image.

For rendering, we used a custom rendering pipeline implemented using the G3D innovative engine. Our primarily OpenGL renderer uses many of the G3D's features, such as shadow mapping and deferred rendering. However, for this thesis, the essential feature of G3D was its implementation of Nvidia's Optix raytracing framework and its integration with OpenGL.

In this chapter, we mainly describe parts of our rendering pipeline that are not directly part of the DDGI algorithm but are analogous to its concepts. Therefore we do not go into greater detail on how we sample indirect illumination or how the fields are updated since those two topics are described in later chapters. However, in Section 3.1 we describe what deferred rendering is and how it uses a structure called GBuffer to compute direct illumination for the final image efficiently. Furthermore, since our renderer uses G3D's shadow maps, we also explain what those are and how they are used in Section 3.2. Lastly, in Section 3.3 we describe how ray casting works in G3D and how we gather and store rays.

3.1 GBuffer and Deffered Rendering

To better understand what deferred rendering is and why it is used, we first need to give a bit of background on graphics pipeline and forward rendering. After that, we describe what deferred shading is and when it is a good idea to use.

There are numerous methods to render a scene, such as forward rendering, deferred rendering, or extensions of these two, for example, forward plus rendering [HMY12]. The main difference between the forward and deferred rendering [HH04] is when the direct illumination from light sources in the scene is computed.

Forward rendering is the standard most modern rendering engines use to compute direct illumination in the scene. The principle behind it is direct illumination is computed at the same time as the the rendered object into the scene.

The basic idea is simple. The geometry of the object is sent to the GPU. On the GPU, vertices are either directly projected to the screen or in more complex pipelines; they are first sent to a geometry shader (or tessellation shader). Consequently, the GPU decides if each fragment is visible and, therefore, if it should be drawn. If the GPU decides to draw the fragment onto the screen, it is shaded by the direct illumination from a light source. On top of that, this is usually done in one or more passes based on the number of lights in the scene.

This leads to a problem. The whole graphics pipeline needs to be called for every object for possibly every light in the scene. It would be much more efficient to store the data about the portion of the scene the viewer sees once. Later the direct illumination would be computed using only the data already stored in the structure. This is precisely what deferred shading aims to do.

Deferred shading decouples the geometry from the shading routine utilizing a structure called GBuffer. GBuffer is best to imagine as a buffer of textures with the exact resolution as the final image. The GBuffer computation is done using a framebuffer with multiple render targets for each of the scene's attributes, see Figure 3.2.

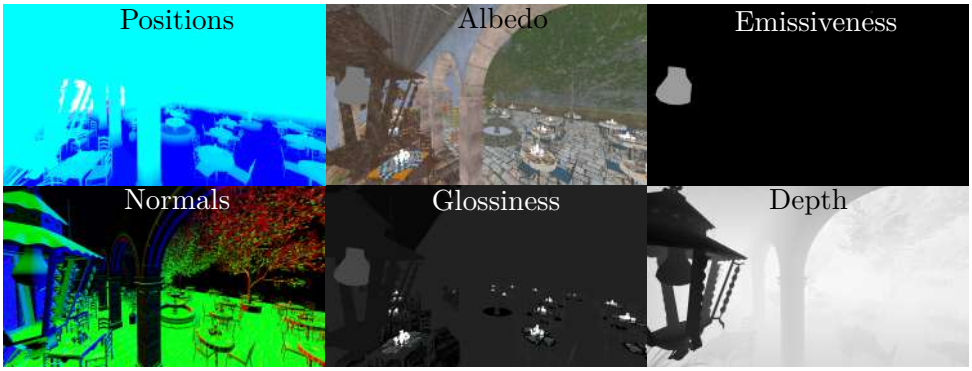


Figure 3.2: Geometry and material attributes stored inside GBuffer.

Deferred shading starts by rendering the opaque geometry of the scene into the GBuffer. The whole graphics pipeline is called as described above only with one difference to the forward rendering. During GBuffer computation, the lighting is not computed. This pass only renders object attributes into each render target. Depending on the implementation of the later deferred rendering pass and the used lighting model, the GBuffer usually contains world space positions and normals. Some deferred renderers also store depth values. GBuffer also contains material attributes such as albedo, glossiness, and the emissive term based on the illumination model.

After the GBuffer is computed, the renderer would usually do other work needed for the final image. For example, it would compute shadow maps.

This is why the method is called deferred shading; it postpones the shading until the very end of the rendering routine.

Once the GBuffer has information about opaque objects and all the extra rendering work is done, the renderer can start shading the final image. This is done simply by rendering a quad over the whole screen. Each fragment on the screen is then shaded using its corresponding data stored inside GBuffer.

Notice that only data about opaque objects were stored inside GBuffer and later shaded. This is because GBuffer cannot store information about visibility for translucent objects, which is the main disadvantage of the method. However, since deferred shading is effective for opaque geometry, and most scenes usually do not have many translucent objects, most modern engines use deferred shading in conjunction with forward rendering.

Also, since we render the image into an HDR framebuffer, we need to convert it to LDR for the image to be displayed on the screen. We use G3D's tone mapping and gamma correction solution, which is applied after we pass the image to the framework for display.

3.2 Shadows

Shadows are one of the most important effects used in graphical applications. The reason is that they give the viewer more information about the scene. Mainly it gives the viewer information about the depth and positions of objects on a 2D image. The approach G3D uses to compute shadows in a scene is called variance shadow maps [DL06] together with naive shadow mapping. Variance shadow maps are discussed in later part of the thesis since the DDGI algorithm makes use of the same principle for visibility between probes. Therefore, in this section will mainly discuss the underlining principle behind shadow mapping. We will also discuss one of our shadow modes for creating pixel perfect sharp shadows.

Shadow maps solve the visibility function by keeping information about what each light in the scene sees. Furthermore, the information is then compared with the camera's view to determine what part of the view is in shadow.

Each light in the scene carries a depth texture into which the scene is drawn whenever it changes. Later, when the camera's view is rendered, each visible fragment projected into each light's space. Then the fragment's distance to the light is compared with the depth value in its corresponding location in the depth map. If the fragment is further away, it can assumed that the fragment is occluded by some object the light sees. Therefore, it is in shadow.

Naive shadow maps have many problems from being sharp and therefore not physically accurate to having issues with visible artifacts. The relevant artifacts for our application are either caused by imprecise representation of depth by floating-point values or by the low resolution of depth maps.

The artifacts caused by floating-point values cause self-shadowing problems, which are usually dealt with by offsetting the fragment from its surface using a slight bias. However, this solution is imperfect and usually leads to

a phenomenon called peter panning, where the shadow does not correctly connect with the occluder.

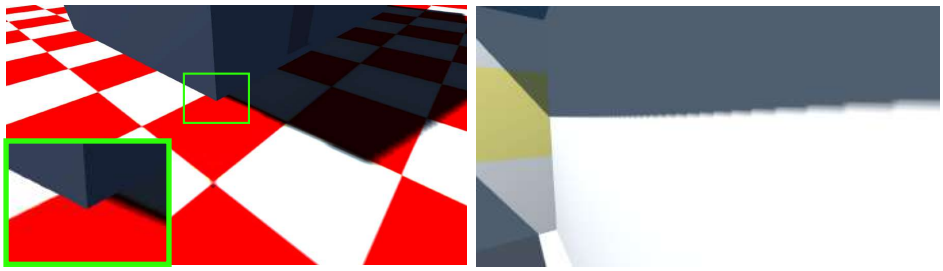


Figure 3.3: Shadow mapping artifacts. The artifact on the left is caused by small shadowing bias causing a phenomenon called peter panning. The right image shows shadow aliasing caused by low resolution shadow maps. Depth map for the light source had resolution 1024×1024.

The second artifact manifests itself by visible aliasing on the edges of a shadow. This artifact happens when an object is far away from a light source. With increasing distance, the objects become increasingly smaller in the depth map, and therefore there are simply not enough texels to accurately represent them. Both of these artifacts are shown in Figure 3.3.

In addition to shadow mapping we also support simple sharp ray-traced shadows calculated from camera as shown in Figure 3.4. To compute shadowing we make use of world position data stored inside GBuffer, which we use as origins for our rays.

Figure 3.4: Direct illumination with environment map utilizing one sample ray-traced shadows cast from camera.

Therefore, whenever the shadowing is computed the renderer generates a ray origin for every world position in GBuffer and a ray direction towards each light source. This way we get a texture of rays for every light source in the scene. Then we pass these rays to a Optix raytracer for any hit occlusion cast which returns a boolean texture. Each texel in this textere therofore is either zero or one, where one means the ray had a hit. Texture is then used during rendering as a mask which indicates which fragments should be shaded by direct lighting.

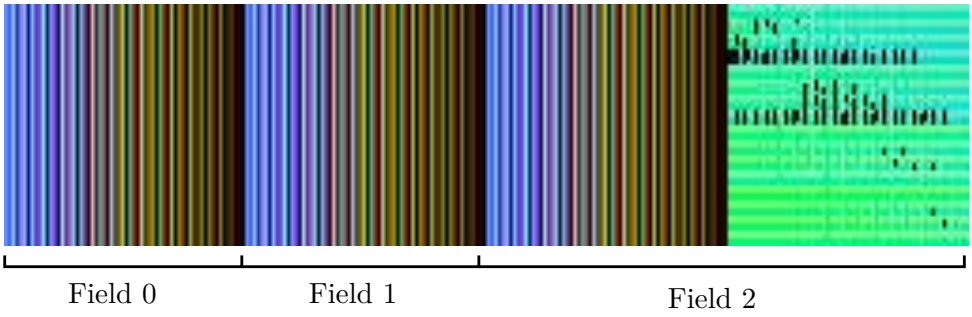


Figure 3.5: Ray atlas of directions. There are three fields placed in the scene. Fields 0 and 1 are DDGI irradiance fields. Field 2 is a shadowing field, so it has directions for probes and directions toward the only light in the scene. Each field had 64 probes, and for each probe, the field casts 64 rays.

3.3 Field Update

Ray casting and sampling the scene's geometry is an important part of the DDGI algorithm. We looked for a framework that already supports GPU ray tracing, such as Unity engine and G3D. We settled on G3D not only since it contains an Optix ray casting but also because it uses OpenGL, with which we are more familiar.

The G3D's Optix bounding volume hierarchy is the main class we interact with for ray casting. Its ray-cast method takes two buffers or textures as an input, one for origins and the second for directions. It then returns an array of buffers (or textures) that contains the scene's attributes. We decided to use textures since they provide more straightforward access from shaders and can be easily visualized, see Figure 3.5. We detail the generation of these textures in Sections 4.2 and 5.1, where we describe how we generate rays for one field. The array of textures is also explained in the following chapter.

Our extensions require updates of multiple fields, possibly in each frame. Therefore doing a cast for each field in each frame, where each texture does not contain many rays, is inefficient. The Optix engine's scheduler would do much better work if we gave him all those rays at once. Thus, instead of doing multiple casts in each frame, we use a large unified ray atlas that contains rays from every field in the scene.

We gather rays from each field and batch them into our ray atlas in each frame. The fields are placed into the atlas next to each other. We offset each field in the atlas based on the number of rays generated by fields before it. Therefore, the width of the atlas is the sum of all of the rays per probe across all fields. The height is the maximum number of probes between all fields. When we gather rays from a field, we already know how many rays will be generated, so we pass it the location where the field will write its ray data. We then pass the atlas as a whole to the Optix BVH for ray casting.

Chapter 4

Dynamic Diffuse Global Illumination Algorithm Overview

To approximate diffuse global illumination for a dynamic virtual scene, Majercik et al. [MGNM19] proposed to compute the light transfer by recurrently updating the irradiance field every frame. This is achieved by using information gathered by rays cast from each irradiance probe placed in a virtual scene. Since the ray casting is independent of the rendering, it avoids denoising or prefiltering high-resolution spherical textures.

Our probe placement strategy and probe representation for DDGI are detailed in Section 4.1. The geometry data and material attributes gathered by ray casting are saved into a structure similar to the G-buffer called a surfel buffer. The surfel buffer is then used for ray shading by direct and indirect light. Lastly, the computed illumination contributions are used to update data inside the probes. To summarize, the algorithm executes four steps in each frame:

1. Generate m rays from each probe, creating $m \times n$ rays, where n is the number of probes in the scene. The ray generation is further detailed in Section 4.2.

2. Cast and trace the generated rays into the scene. Each ray gathers attributes from the scene's geometry into a G-buffer-like structure of surfels. Section 4.3 gives a more in-depth description of the ray casting and the surfel buffer.

3. Shade rays by direct and indirect illumination using data inside the surfel buffer. The ray shading method is detailed in Section 4.4.

4. Update irradiance and distance probe data for each of m probes using the shaded rays and their hit distances. The update procedure is detailed in Section 4.5.

The resulting irradiance probe field is then used to compute the indirect illumination contribution for the final image visible from the camera. This is done in the same way as in ray shading by indirect illumination, but instead of probe rays, the fragments seen from the camera are used.

The whole algorithm was implemented as a set of OpenGL compute shaders for each of the above passes. We avoid overhead caused by memory transfers between the device and the host memory by keeping all necessary data necessary for computation and simple decisions on the GPU device. Where relevant, we discuss the implementation details such as the format of used data structures, memory access strategy, and parallel algorithms. We also define used parameters for compute shader dispatch parameters such as workgroup sizes and computational grid dimensions if they are not trivial.

4.1 Probe Placement and Representation

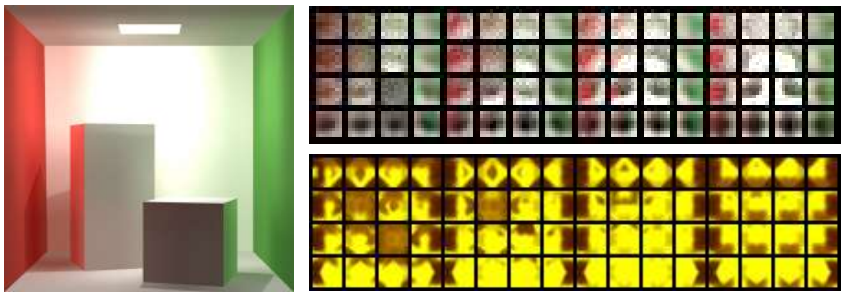


Figure 4.1: On the first image is a standard Cornell box scene where we placed a $4 \times 4 \times 4$ probe grid. The second image is our atlas of octahedral irradiance maps for each of the probes placed in the scene, where each map has a resolution of 8×8. The final image shows the distance atlas where each map has a resolution of 16x16. Each entry in the distance atlas contains a distance and a squared distance. The guard borders are set to be black to illustrate the environment maps better.

The irradiance probes are placed in the scene's bounding volume at vertices of a uniform 3D grid. Using the uniform grid, we can leverage the fast queries and effective interpolation between probes for accurate sampling of the dynamic scene. The original paper always uses the power of the two resolution strategy for the grid to achieve faster probe queries. On the other hand, we opted for a more general solution where the grid can use an arbitrary resolution. This provides us with a bit more flexibility in probe placement at the cost of a slight performance decrease due to slow modulo operations on the GPU. Inside the application, grid properties such as resolution or probe spacing are fully customizable to better sample a given scene.

As shown in Figure 4.1 the probes' data is stored inside two texture atlases. The first atlas holds irradiance maps of each probe, and the second contains distance maps used for visibility testings during irradiance sampling from the field. Traditionally such maps are stored as cube maps, where we would store the values inside each of the cube's sides. However, this is incredibly memory inefficient for maps that do not need much precision. In order to save space, the original authors proposed to use octahedral representation for environment maps [CDE+14] instead of standard cube maps. The implementation allows

for separate settings of maps' side lengths for each atlas for scenes where we need higher precision.

4.2 Generating Rays

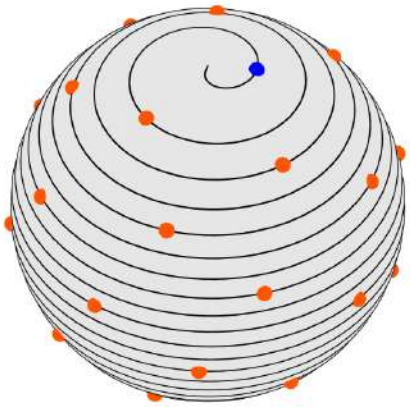


Figure 4.2: In order to get uniform distribution of spherical directions for each ray we use Fibonacci spherical mapping. [KISS15]

For each of the n probes placed in the scene, m rays are generated, yielding $m \times n$ rays. These rays share a common origin with their probe's center position. Since we would ideally want to cast the rays uniformly from the whole surface of our spherical probe, we need a uniform distribution.

We use the same uniform distribution as the original paper, which proposed to use Fibonacci spherical distribution. The distribution returns a spherical Fibonacci point set SF uniformly spread across the surface of a unit sphere based on the desired total number of points. Each point P in SF set on the unit sphere is defined using spherical coordinates:

$$P(\phi, \theta) = (\cos(\phi), \sin(\theta), \cos(\phi), \sin(\theta), \cos(\theta))^T \tag{4.1}$$

The point with index i of a point set SF for n samples is given as [KISS15]:

$$SF_i^n = P(\phi_i, \cos^{-1}(z_i)) \tag{4.2}$$

$$\phi_i = 2\pi \frac{i}{\Phi}, z_i = 1 - \frac{2i+1}{n}, i \in \{0, \ldots, n-1\} \tag{4.3}$$

where $\Phi = (\sqrt{5}+1)/2$ is the golden ratio.

Now that we have origins and directions, we need to tell G3D's raytracer the minimum and maximum distances a ray can travel. To do so, we use the alpha channels of our two textures. The alpha channel of an origin holds the minimum distance, which we set to a slight offset to avoid self shadowing. On the other hand, the direction's alpha channel contains the maximum distance.

Since we do not care how far a ray should travel, the maximum distance of our rays is set to infinity.

We generate and batch ray origins and directions in each frame into two textures and later pass it to G3D's Optix implementation for casting. For reference these textures are shown in Figure 4.3.

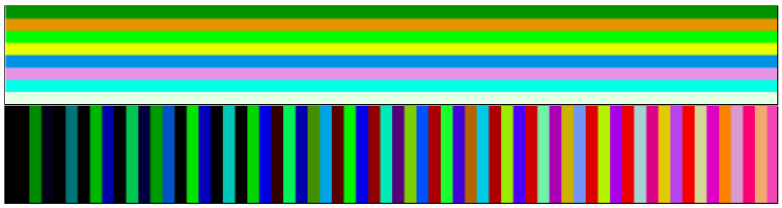


Figure 4.3: Images demonstrate the generated ray textures for a field with eight probes and 64 rays per probe. Top texture stores ray origins. Each entry in the texture contains the corresponding probe center and a minimum ray distance. The bottom texture contains ray directions sampled using Fibonacci spherical mapping. The alpha channel contains the maximum distance a ray can travel, set to infinity.

4.3 Probe Ray Casting

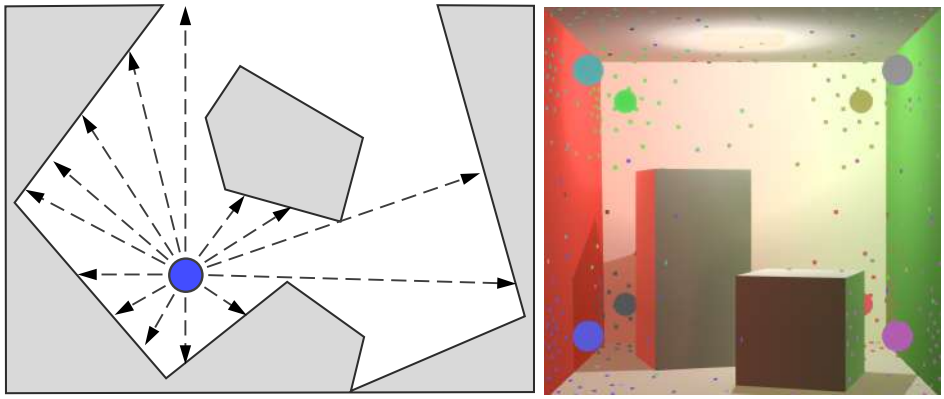


Figure 4.4: Each probe casts m rays in m uniformly sampled spherical directions. The rays then sample the geometry attributes at their points of intersections with the scene and store them inside a surfel buffer.

From each probe, we cast m rays, which were generated in the previous pass. We ignore backface culling to avoid visibility errors from probes enclosed inside geometry. Each ray gathers data about the scene's geometry into a G-buffer-like structure of surfels. The structure for one field is a set of textures with the exact dimensions as our ray textures, where each entry in the surfel buffer represents information about surfels' attributes. The structure contains information about world-space positions, normals, and material attributes: albedo, specular reflectance, emissive and transmissive terms. We also get

information about ray misses from the ray cast, which is encoded into the normal texture as zero vectors.

Using this information, we then calculate the distances to hit points of each ray. We also encode any backfaces hit by the rays into the distance values. This is done using a sign of cosine of an angle between a normal of a surfel and a ray's direction.

4.4 Ray Shading

Figure 4.5: Indirect illumination contribution in the Sponza scene. On the left image is the indirect illumination sampled from a camera. On the right image is the final image.

For clarity, we separated the ray shading into two standalone passes: direct and indirect illumination passes. However, the indirect illumination does not have to be a standalone pass, and its contributions could be sampled directly from the atlases inside a unified illumination pass.

First, we start with the indirect illumination, where we shade the rays by diffuse indirect illumination leveraging the probe data accumulated in previous frames. Section 4.4 describes the indirect illumination pass and the interpolation weights used to handle visibility errors such as light leaks and shadow leaks.

After we sample the irradiance field for indirect illumination, we begin the second pass, where we compute the direct illumination for each ray from the light sources in the scene. Once we get each light's direct contribution, we add indirect illumination from the previous pass. The Direct illumination is described in Section 4.4.

Please note that these passes are not limited only to shading rays. Since the surfel buffer is analogous to a GBuffer, we use the same passes to render indirect contribution and the final image inside the renderer. The only difference is that instead of sampling the scene using arbitrary samples from probes inside the field, we use fragmens seen from the camera, see Figure 4.5.

Diffuse Indirect Illumination

Once the probes collect their data from the scene, we can shade the rays by diffuse indirect illumination. We sample each ray's hit location from the surfel buffer. If the hit surfel exists, we find eight closest probes to that surfel

forming a grid cage around it (see Figure 4.6). Due to the grid structure of the irradiance field, we can therefore encapsulate every point in the scene in a grid cage.

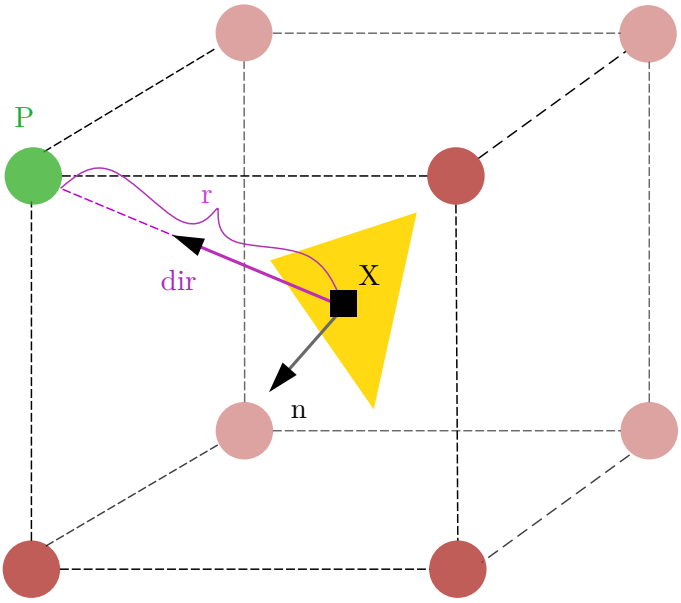


Figure 4.6: Depiction of the eight probe grid cage around a surfel X. Each probe P is sampled using the surfel's normal n in world space. Each probe's contribution is weighted based on its visibility using direction dir from the sampled point X to the current probe P. r represents the mean distance from X to P.

After the probe cage is formed, we iterate over every probe in the cage. We sample the irradiance from the probe's environment map in the direction of the surfel's normal. More specifically, we take the normal of the surfel point, encode it into octahedral texture coordinates, and sample the corresponding irradiance texture. The sampled irradiance is then added to the total irradiance sum of the cage.

The sampled irradiance from probes on its own would not yield correct results since it does not account for visibility. To ensure that the indirect light appears continuous and accounts for dynamic geometry and lighting, Majercik et al. [MGNM19] describe various methods to smooth it and cull unwanted contributions. The authors use the following interpolation weights to blend the irradiance from the closest eight probes:

Wrap diffuse shading. The first applied weight is acquired using a method called Wrap diffuse shading [SNY11]. Wrap shading is a technique commonly used as a cheap approximation of subsurface scattering or as a more expressive base-shading model. Furthermore, the method works well as a heuristic to cull indirect contributions from probes that are not mutually visible to the surfel. The wrap shading is defined as follows [SNY11]:

$$f(\theta, a) = ((\cos\theta + a)/(1 + a))^{1+a} \tag{4.4}$$

where θ is an angle between surfel's normal and direction to the active probe and a is a parameter in the range $[0, 1]$. In our case we use $a = 1$. For reference, see Figure 4.7.

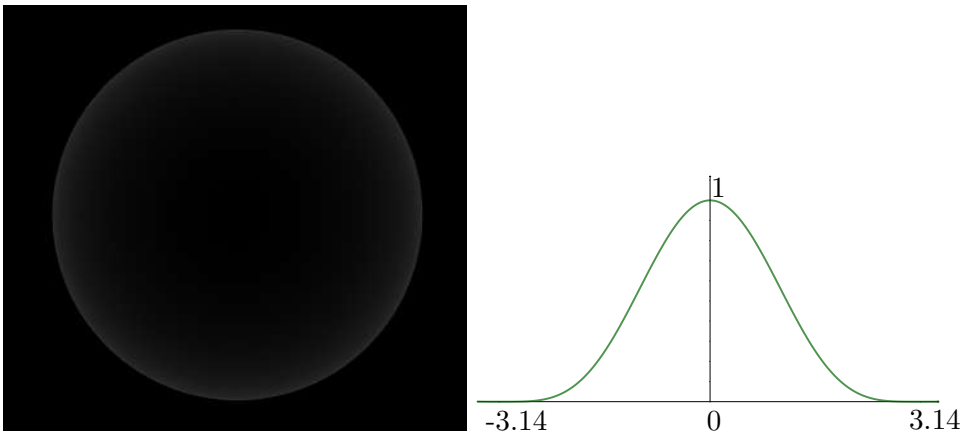


Figure 4.7: The graph shows the wrap shading function curve for $a = 1$. The image shows the wrap diffuse shading on a sphere, where the light is placed on the opposite side.

Chebyshev moment visibility test. In CG, Variance-biased Chebyshev interpolant [DL06] is used to counter shadow aliasing and produce physically accurate shadows by approximating how they soften on their edges. DDGI uses the variance shadow maps as a practical heuristic to counter light leaks by filtering out probe contributions that are occluded by geometry. To calculate The Chebyshev interpolant, we sample moments M_1 and M_2 stored in the active probe's depth map.

$$M_1 = E(x) \tag{4.5}$$

$$M_2 = E(x^2) \tag{4.6}$$

The moments are sampled from the depth atlas, similar to how we sample irradiance. However, since we are sampling visibility in a particular direction, we do not use the normal's direction for sampling but rather a direction from the current probe to the surfel. Once we sample the moments M_1 and M_2, we use them to obtain the mean μ and variance σ^2:

$$\mu = M_1 \tag{4.7}$$

$$\sigma^2 = M_2 - M_1^2 \tag{4.8}$$

Finally, we calculate the Chebyshev interpolant, which is described by Chebychev's inequality theorem: *Let there be a random variable drawn from a distribution with mean μ and variance σ^2. Then for $r > \mu$* [DL06]:

$$P(x >= r) <= p_{max}(r) \equiv \frac{\sigma^2}{\sigma^2 + (r - \mu)} \tag{4.9}$$

Where r is the exact distance from the current probe to the evaluated sample.

Log perception weight. To account for human perception, specifically how human eyes are sensitive to contrast in low-light conditions, we crush any weights smaller than some threshold c. The log perception weight is defined as:

$$f(w) = \begin{cases} \frac{w^2}{c^2} & \text{if } w < c \\ 1 & \text{otherwise} \end{cases} \tag{4.10}$$

This way, we eliminate any small weights, but keep the curve continuous.

Trilinear weight. The standard trilinear interpolation to smooth the contributions of probes based on their distance from the surfel point.

Biasing. Some scenes also require a slight global normal bias to eliminate self-shadowing caused by the Chebyshev weight. Our shadowing bias b is defined as follows:

$$b = (0.8\boldsymbol{n} + 0.3\boldsymbol{v}) * k_n \tag{4.11}$$

where $\boldsymbol{n}$ is a surfel's normal, $\boldsymbol{v}$ is a view direction and k_n is normal bias multiplier constant in range $[0, 1]$. However, using the normal bias brings its own set of problems namely for large k_n light leaks may appear on thin geometry.

Lastly, the algorithm does not obey the law of energy preservation, which means that values inside our irradiance atlas can start to diverge. The problem is most prominent in scenes with bright surfaces where the scene gradually explodes with energy. Therefore we apply energy preservation constant in the range $[0, 1]$ whenever we return sampled irradiance [MGNM19].

Direct Illumination

To calculate direct illumination for each sample, we use the surfel buffer as a standard G-buffer as if we were rendering the final image. The pass is analogous to the standard deferred rendering, where we render into a texture of the same size as the surfel buffer. We compute direct light contributions from each light source in the scene using G3D's illumination model for each texel in the texture.

For shadowing queries, we project surfel's world space coordinates from the surfel buffer to light space to check if the point is in shadow using shadow maps. Once the contribution of the direct light is computed, we sample the indirect illumination texture from the previous pass and add its contribution.

4.5 Updating Probes

The final step is to update the irradiance and distance maps. We gather the irradiance value E_e for each texel from all sampled points. Then we use them to update the value in the texel using the linear interpolation. We linearly

interpolate the new value with the old one present in the atlas using *hysteresis* parameter α as the interpolant:

$$E_{e_{new}}(\boldsymbol{\omega}) = lerp(E_{e_{old}}(\boldsymbol{\omega}), \frac{1}{w_{sum}} \sum_{probeRays} max(0, \boldsymbol{\omega} \cdot \boldsymbol{r}) * L_e, \alpha) \tag{4.12}$$

$$w_{sum}(\omega) = \sum_{probeRays} max(0, \boldsymbol{\omega} \cdot \boldsymbol{r}) \tag{4.13}$$

where E_e is irradiance, $\boldsymbol{\omega}$ is the probe's texel direction in octahedral coordinates, $\mathbf{r}$ is the ray's direction, L_e is the rays radiance, and w_sum is the sum of all weights for a given texel direction to normalize to contribution.

The distance map is updated in the same way. For each texel in the distance map, we compute a mean of distances between the probe center and hit locations of the rays. Furthermore, we also compute the mean of squared distances needed for visibility sampling during indirect illumination sampling.

On the contrary, for irradiance update, we do not use $\boldsymbol{\omega} \cdot \boldsymbol{r}$ weight for depth values since it will not reduce the light leaks as much. To make visibility sampling more accurate and, therefore better reduce the light leaks, the original paper recommends using weight γ, which is defined as:

$$\gamma = (\boldsymbol{\omega} \cdot \boldsymbol{r})^{k_s} \tag{4.14}$$

where k_s is a constant in range $[1, 50]$, $\boldsymbol{\omega}$ is the probe's texel direction in octahedral coordinates, $\boldsymbol{r}$ is a ray direction. For reference on how k_s helps with light leak reduction see Figure 4.8.

Figure 4.8: The images show light leak reduction and k_s values used to sharpen the values in the depth atlas.

Lastly, we update the old distance and the squared distance means by using the temporal hysteresis parameter:

$$D_{new}(\boldsymbol{\omega}) = lerp(D_{old}(\boldsymbol{\omega}), \frac{1}{w_{sum}} \sum_{probeRays} max(0, \gamma) * l, \alpha) \tag{4.15}$$

$$D^2_{new}(\boldsymbol{\omega}) = lerp(D^2_{old}(\boldsymbol{\omega}), \frac{1}{w_{sum}} \sum_{probeRays} max(0, \gamma) * l^2, \alpha) \tag{4.16}$$

where D is the distance mean value and l is the ray's length.

The hysteresis values close to 1 change the texture map very slowly, improving stability at lower accuracy when objects move in the scene. On the other hand, values close to 0.9 (and lower) lead to rapid reactions to changes in the scene. However, it also leads to noticeable flickering. Furthermore,

the flickering can occur even with the hysteresis parameter close to 1. This is due to the low amount of sampling rays, which leads to rapid changes in irradiance maps if there is an exceptionally bright surface in the scene.

The naive implementation of these updates can be slow because of each map's size in the atlas or the number of rays to be processed. Therefore the compute shaders that do these updates are tailored to address these factors.

In order to optimize the sum of the contributions of all rays across the sphere, we use shared memory. The size of the shared memory is directly proportional to the number of rays per probe. The size of a single workgroup G_{size} varies based on the number of rays per probe. To ensure that we always use all threads inside every warp, we require the ray count per probe to be multiple of a warp size, 32 threads. However we limit the size of a single workgroup to a maximum of 64.

We use all three dimensions for the computational grid of size D. We set up the computation grid proportionally to the field's uniform grid resolution R and the side length of octahedral maps L. The computational grid is defined as follows:

$$D_x = R_x \times R_y, D_y = R_z, D_z = \frac{L^2}{G_{size}} \tag{4.17}$$

D_x and D_y are self-explanatory. We want to have workgroups for each probe. What D_z effectively splits each environment map in the atlas into multiple groups. Therefore if we had 8×8 resolution maps and 64 rays per probe, we would have one group for each map. On the other hand, 64×64 maps would lead to 64 groups per map. These groups are placed below each other, so if we reach 64 groups per map, each group handles one row of a map. This approach limits us to environment maps of sidelength of 64 texels. However, we think that this size is more than enough in general since, most of the time, DDGI does not need more than 8×8 maps. Since we limit the

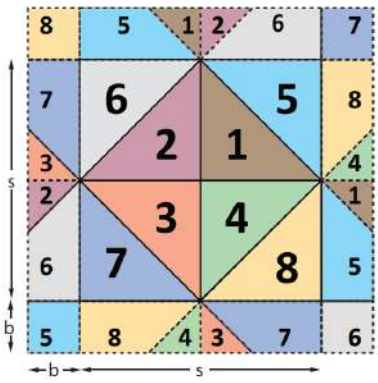


Figure 4.9: Octahedral enviroment map of size $s \times s$ extended by a border of width b [ED08].

workgroup size to 32 or 64 threads, we need to define the number of batches we need to perform to load ray data into the shared memory. Therefore we need to spread the workload over our limited number of rays. Thus if we had two warps per group and 128 rays per probe, each thread must load

data about two rays into the shared memory. However, 64 rays per probe are usually enough to get a stable sampling of a scene.

Now that the atlases are updated, we still have one problem. Atlases of octahedral environment maps have an issue with color bleeding when bi-linear sampling or mipmapping is used. This issue and its solution are described in a paper by Engelhardt et al. [ED08]. To summarize, the solution is to extend each octahedral map by a border the width of one pixel. After the atlas updates, we also added a step: to copy texels on borders of each environment map into their guard bands. The pattern of how the texels are copied is shown in Figure 4.9.

Chapter 5

Dynamic Global Shadowing

Figure 5.1: San Miguel scene where we computed shadowing using only our ambient occlusion method. The probe density for the scene was set to $16 \times 16 \times 16$ with 16×16 occlusion maps.

We were tasked to reconstruct shadowing using probe volumes placed in a scene as part of this project. We used DDGI as a starting point and repurposed it to compute global ambient occlusion. The method works similarly to DDGI because we have an occlusion field defined by a uniform probe grid of ray-tracing sampling probes with octahedral environment maps.

These probes in each frame cast a fixed number of rays into a scene to sample its geometry, serving as origin locations for shadow rays directed into lights placed in the scene.

Then we compute the visibility of each surfel from each light using information gathered using the cast shadow rays. The visibility is then used to update probe data inside our occlusion volume. To summarize, the method makes the following steps each step

- Generate primary and shadow rays and cast them into the scene. We generate primary rays from each probe in the scene in each frame. We

also generate shadow rays based on hit locations found in the previous frame. Section 5.1 gives a more in-depth description of we generate the primary rays and the way we spread the ray generation over multiple frames.

- Compute visibility and probe update. Using the occlusion data gathered from shadow rays, we determine the visibility of each surfel from each light. Using the visibility information, we update each probe's environment map. After the field is updated, we can use it to sample shadowing inside our occlusion volume. The visibility function evaluation, probe update, and how we sample the occlusion volume for the rendering of the final image are detailed in Section 5.2.

5.1 Ray Generation and Casting

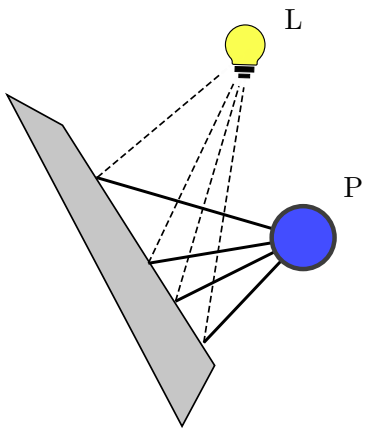


Figure 5.2: Shadow casts are spread over two frames. We cast primary rays from probe P in the first frame, which sample geometry in a scene. We generate secondary shadow rays from the hitpoints directed towards light receiver L in the next frame.

We spread the ray generation for shadowing probes over two frames, see Figure 5.2. In the first frame, we generate $m \times n$ primary rays, as we would for the irradiance field. This is done in each frame using probe centers as origins and spherical Fibonacci mapping for ray directions. We then cast these rays into the scene and gather information about the scene's geometry: world-space positions and normals. Based on information encoded in normals and the ray directions, we determine if a ray missed a scene or hit backface and compute distances for probe visibility queries.

Then we generate up to $m \times n$ shadow rays for each light in the scene. These rays have origins at the primary rays' hit locations and are directed toward light receivers. Primary rays that missed the scene or hit a back face generate a dummy ray with zero max distance in their alpha channel. This way, we indicate to G3D that we do not wish to trace these rays. However,

even though they will not be traced, we still use them by encoding ray miss information into them. The generated rays are then batched with the original primary rays in a ray texture atlas to be cast in the next frame.

5.2 Occlusion Probes Update and Sampling

Figure 5.3: On the left image we show the occlusion sampling with the cosine weight and on the right we use wrap diffuse shading. Note that there is no normal biasing. Notice the much more pronounced shadow leaking artifacts next to the door in the wrap shading image.

Once our shadow rays are cast from sampled surfels we determine the visibility $V(\boldsymbol{x}, \boldsymbol{l})$ of each surfel $\boldsymbol{x}$ from each light l. The visibility function $V(\boldsymbol{x}, l)$ is defined as:

$$V(\boldsymbol{x}, \boldsymbol{l}) = \begin{cases} 1 & \boldsymbol{x} \text{ is visible from } \boldsymbol{l} \\ 0 & \text{otherwise} \end{cases} \tag{5.1}$$

To evaluate the $V(\boldsymbol{x}, \boldsymbol{l})$ for $\boldsymbol{x}$ we check if the shadow ray to $\boldsymbol{l}$ had any hit along its way. To do so, we use the miss information encoded into normals as a zero vector. Therefore, if we detect any non-zero normal conclude that the $\boldsymbol{x}$ is not visible from $\boldsymbol{l}$.

During DDGI sampling, we want to know the amount of irradiance a surfel receives in the direction of its normal. Therefore, the probe update reflects that by spreading the irradiance values evenly over the environment map. However, we cannot do that for shadowing since our occlusion rays are not sampled uniformly but rather concentrated towards each light in the scene. Hence when we update an occlusion map, we do not us the probe's uniformly distributed rays, but rather the mean of the visibility contributions is computed using the directions of the shadow rays instead. The occlusion environment maps thus reflect the visibility mean towards each light in the scene, which is defined as follows:

$$V_{e_{new}}(\boldsymbol{\omega}) = lerp(V_{e_{old}}(\boldsymbol{\omega}), \frac{1}{w_{sum}} \sum_{lights} \sum_{probeRays} max(0, \boldsymbol{\omega} \cdot \boldsymbol{l}) * V(\boldsymbol{x}, \boldsymbol{l}), \alpha) \tag{5.2}$$

$$w_{sum}(\omega) = \sum_{lights} \sum_{probeRays} max(0, \boldsymbol{\omega} \cdot \boldsymbol{l}) \tag{5.3}$$

To evaluate the $V(\boldsymbol{x}, \boldsymbol{l})$ for $\boldsymbol{x}$ we check if the shadow ray to $\boldsymbol{l}$ had any hit along its way. To do so, we use the miss information encoded into normals as a zero vector. Therefore, if we detect any non-zero normal can conclude that the $\boldsymbol{x}$ is not visible from $\boldsymbol{l}$.

Chapter 6

Extensions

This chapter describes our extensions to the DDGI algorithm to solve some of its problems. These extensions mainly target the uniform grid representation that the DDGI and our shadowing algorithm use for probe placement. First, in Section 6.1 we present our geometry aware strategy to determine probes that are not actively participating in global illumination or shadowing. In Section 6.2 we describe field cascades to locally increase probe density for medium to large scenes where the uniform grid can become too sparse for accurate sampling.

6.1 Geometry Aware Probe Update

As mentioned in Section 4.5 the update of each texture atlas can be relatively slow. For outdoor scenes where the geometry is sparse, the uniform 3D grid structure of the fields can yield a high amount of probes that are not actively sampled and do not need to be updated every frame. However, we would want to keep the uniform grid structure of our fields since it interacts well with dynamic geometry.

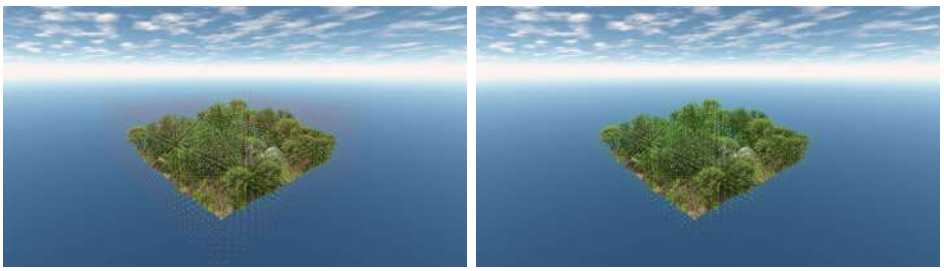


Figure 6.1: Probe culling comparison. The left image is our forest scene where we placed a $32 \times 16 \times 32$ probe grid. The same probe grid is on the right after we culled all inactive probes.

Furthermore, many probes can be placed inside geometry in scenes with large objects. However, some probes cannot be considered inactive in a traditional sense. This is because they may still be close enough to geometry and therefore are included in a probe cage during indirect light sampling. Unfortunately, the inclusion of these probes during sampling causes artifacts

to appear. Typically, DDGI deals with these artifacts by offsetting a surfel from the surface using a large enough normal bias, which causes its own problems. Due to these issues, samples from such probes are not desirable, and we would want a way to detect them and potentially cull their contribution, see Figure 6.2.

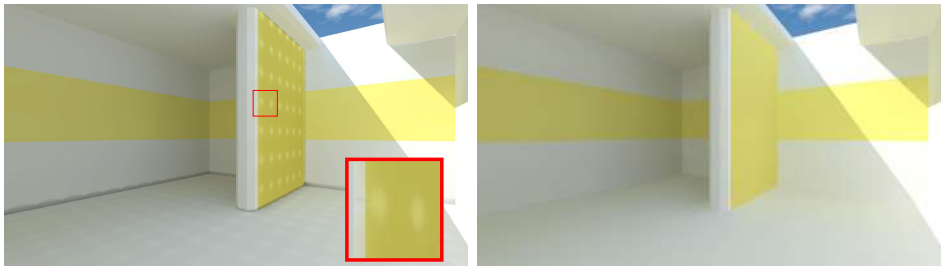


Figure 6.2: Dead probe contribution culling comparison. On the left image are visible artifacts caused by probes that are oversampling small areas inside geometry. The right image shows the DDGI's result when we cull contributions from dead probes.

We propose a fast and straightforward extension that directly leverages ray-traced data gathered by the probes to detect and flag inactive probes.

6.1.1 Inactive Probe Detection

The probe detection works as a local heuristic which tells us if a probe would be included in some probe cage grid during indirect illumination pass. We do this by computing distances from each probe to its samples. If a sample was a miss, it is flagged by setting its distance to a slightly larger value than the length of a probe cage's diagonal r. We also compute if a sample was a backface and encode it into the distance value.

Relevant Samples

Once we have our encoded distances, we start computing the usage of each probe P by counting its relevant samples. A relevant sample is such a sample that has a distance smaller or equal to a length of a probe cage's diagonal or is smaller than zero. The final usage of a probe P is computed as follows:

$$Usage(P) = \sum_{N_P}^{1} R(d_{i_P}) \tag{6.1}$$

$$R(d) = \begin{cases} 1 & \text{if } d \leq r \\ -1 & \text{if } d < 0 \\ 0 & \text{otherwise} \end{cases} \tag{6.2}$$

where d_{i_P} is a distance from a probe P to a sample i and r is the length of a diagonal, see Figure 6.3.

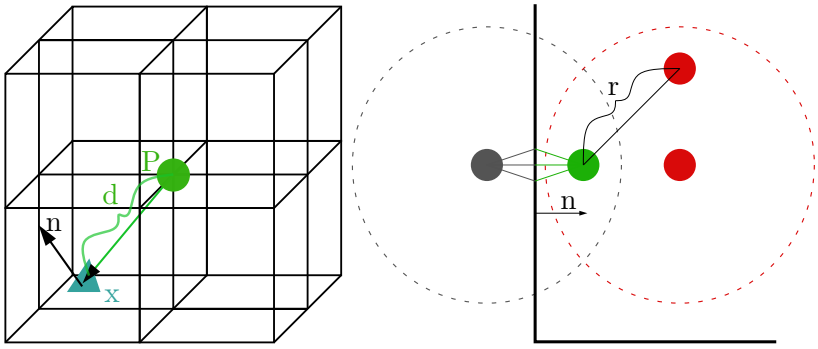


Figure 6.3: We are finding relevant samples and probe states. The left image shows a relevant sample x, used during sampling by one of the probe cages that probe P is part of. The second image shows our three types of probes. The ALIVE (green) probe is a probe with a positive number of samples. SLEEPING (red) probes that have no relevant samples in their radius. Lastly, the DEAD (black) samples are mostly backfaces.

We use the parallel reduction algorithm in our compute shader for the sum of relevant samples. The compute shader also uses shared memory for better memory access during the computation. Parallel reduction refers to algorithms that take an array of elements and produce a single value. There are many problems where this algorithm applies, such as the sum of elements or finding a minimum or maximum value.

Each probe gets one or more one-dimensional workgroups of 32 or 64 threads based on the number of rays each probe has. These groups then get a shared memory of the same as a single workgroup. Each thread writes into the shared memory the relevancy of each sample based on the distance stored in the distance texture 6.2. Then in each step, we half the number of threads accessing the shared memory while we sum the values inside the shared memory as shown in Figure 6.4.

Once the first thread in the workgroup has the sum of all values for the workgroup, we make the final sum. In case of each probe having only one group, we write the workgroup's result into a buffer that is the same size as the number of probes in the field. On the other hand, if a probe had more than one workgroup, we do an atomic sum of their results and then store the final result.

Probe State

Now that we know probes' usage values, we can update their state and hysteresis values. A probe can be flagged by one of three flags:

- ALIVE: A probe had a positive number of relevant samples. Since probes might exist almost at the rim of r, we keep active probes in this state for some minimum of frames. This is done to reduce possible flickering between states due to random sampling.

  Active probes are updated every frame if not stated otherwise. Further-

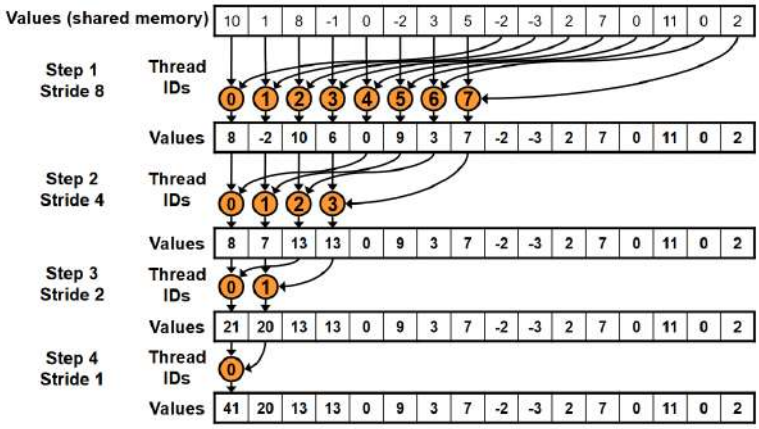


Figure 6.4: Parallel accessing [H+07].

more, we logarithmically increase their hysteresis every update until it reaches a user-defined maximum.

- SLEEPING: These are probes that had usage equal to zero. Sleeping probes are updated every couple of frames to keep their data updated. More specifically, in the implementation, they are updated every tenth frame. Similarly, as alive probes, their hysteresis parameter changes every iteration, although their hysteresis parameter logarithmically falls off longer they stay in this state.

- DEAD: Probe is dead if more than half of its relevant samples were backfaces. Such probes are updated in their first frame with probes starting hysteresis. Once they are updated will not be updated again as long as they remain in this state. This is because they most likely got into this state for one of two reasons. First, the probe was placed inside static geometry on scene initialization, and it will never leave this state, so there is no point in updating the probe. Second, it got occluded by a dynamic object, and it will most likely leave this state soon.

  Furthermore, we check for dead probes during indirect illumination passes. If a probe was flagged as dead, we would cull its contribution to the final irradiance value sampled from the probe cage it was part of.

6.2 Field Cascades

A single irradiance field for accurate light propagation approximation in a large scene requires a high-density probe grid. However dense irradiance grid would lead to an excessive amount of memory. Furthermore, much of the memory would be wasted due to the high number of probes that may not be used during sampling. To avoid this issue, we use field cascades inspired by Kaplanyan et al. [KD10] nested grids. The cascades are a level of detail

technique where we have multiple levels of irradiance fields centered around the camera with decreasing probe density. This way, we always have a high density of irradiance probes near the viewer's vicinity, which provides a more accurate sampling of the scene. Lower resolution grids of higher-level cascades then cover distant parts of the scene. The highest level cascade is built as a standard probe grid enveloping the scene.

In further sections, we detail the extension using only the irradiance field. However, even though we present it for irradiance fields, the extension would work the same way for our shadowing field.

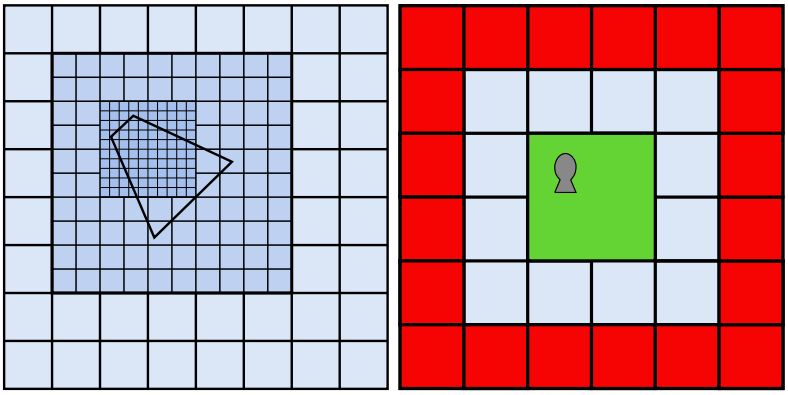


Figure 6.5: Field cascades. On the left image is a representation of the field cascades nestled into each other. These cascades are centered on the viewer's position and allow for higher fidelity the closer a sample is to the viewer. On the right is a since cascade, where the green area demonstrates the volume where the viewer can freely move without displacing the grid. The red is probe cages that are not sampled during indirect illumination sampling but are rather used as buffers with low hysteresis.

6.2.1 Representation and Update

The cascades are represented as standalone grids nested inside each other. Since moving the grids with the viewer continuously would cause artifacts. Thus we use discreet steps to move the grids. Each cascade has a closed area, where the viewer can move freely without offsetting the grid, see Figure 6.5. In each frame, we check if the camera left the area, and if we detect that it did, we decide on an offset direction. To offset the grid, we are using axis-aligned unit directions. We compute a new grid origin for each of these offsets and choose the one that yielded the grid's starting position closest to the camera and move the grid accordingly.

However, moving the grid by discrete steps does not solve the problem that suddenly a probe has data that do not match its location. To solve this issue, we offset the indices of the probes in the opposite direction of the grid movement. Thus remapping data inside the probe atlas to moved probes. This way, whenever we move the grid, we keep sampling the correct data for a particular location.

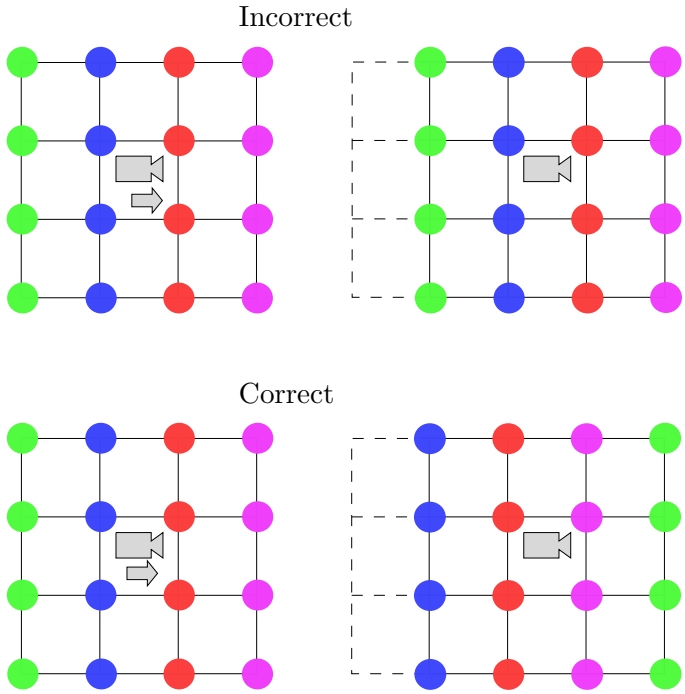


Figure 6.6: Grid indices offeseting. In the top image is the grid state if we did not offset the probes in the opposite direction of the grid's movement. Although probes were moved, they have incorrect values from the previous position. The bottom images show the correct probe positions.

The only exceptions are probes on the cascade's edges, which can suddenly appear on the opposite side of the grid when we move the viewer. However, this is not an issue. We do not sample these probes but rather use them as buffering probes with low hysteresis parameters. So when the grid is moved, these probes already have correct data about the scene to which can be passed on.

Currently, the default size of the cascaded fields is based on the size of the highest cascade and their count. However, we give the user the ability to customize the size of each cascade volume together with probe density and probe spacing. The cascades are then sorted by their size, so we always start sampling from the smallest grid. We also give the user the option to customize other field attributes, such as the number of rays per probe, irradiance map sizes, and depth map sizes.

The cascades are updated independently of each other. Thus, a cascade works as a standalone irradiance field that performs each DDGI step independently with no knowledge about other cascades. The only difference is in ray casting, where we batch every cascade in the pyramid into one unified atlas of rays. This batching is done by the renderer and was detailed in Section 3.3.

6.2.2 Final Image Sampling

We sample the irradiance cascade pyramid as an array of irradiance fields during the final image rendering. Therefore, we start the sampling at the lowest level for each sample and continue to the highest cascade. On the way to the largest cascade, we gather each field's indirect contribution as was described in Section 4.4. However, we also need to determine if a given field is used and how much.

To do so, we weigh each cascade based on a given sample's location. We first compute the normalized coordinates of the sample inside the grid. Then the final weight is the geometrical mean of these normalized coordinates in each axis; see Algorithm 1. Each weight is then scaled based on how far we are in the pyramid. Lastly, we weigh the current field's contribution and add it to the final sum. The final indirect illumination contribution of the whole pyramid is thus computed as a mean of all fields in the pyramid; see Algorithm 2.

Algorithm 1 Get Field Weight. Compute the weight of a field for a given sample as a geometrical mean across all axes.

Input: F ▷ irradiance field
$\boldsymbol{x}$ ▷ sample
Output: Field Weight
$\boldsymbol{o} \leftarrow FieldOrigin(F)$ ▷ Origin of the field
$\boldsymbol{s} \leftarrow FieldStep(F)$ ▷ Space between probes
$w \leftarrow 1$
$\boldsymbol{x} \leftarrow (\boldsymbol{x} - \boldsymbol{o})/s$ ▷ Normalized coordinates of the sample inside the field
for $i <= 3$ **do**
  $a \leftarrow \boldsymbol{x_i}$
  if a < 1 **then** ▷ Is the sample inside the field
    $w \leftarrow w * clamp(a, 0, 1)$
  else
    $\boldsymbol{R} \leftarrow FieldResolution(F)$
    if $a > \boldsymbol{R}_i - 1$ **then**
      $w \leftarrow w * clamp(R_i - 1 - a, 0, 1)$
    end if
  end if
end for
return w

Algorithm 2 Sample Irradiance Fields

Input: $F_1 \dots F_N$, ▷ irradiance fields
$\boldsymbol{x}$ ▷ sample
Output: Irradiance mean E

```
w_sum ← 0
E_sum ← 0
for f <= N And w_sum <= 1 do
    w ← 1
    E ← 0
    if f < N Or w_sum > 0.9 then
        w ← GetFieldWeight(F_f, x)
    end if
    if w > 0 then                          ▷ Is the field going to be used?
        E ← SampleIrradianceField(F_f, x)
        w ← w * clamp(1 − w_sum, 0, 1)
        E_sum ← E_sum + wE
        w_sum ← w_sum + w
    end if
end for
return E_sum / w_sum                       ▷ normalize the result
```

Chapter 7

Results

We evaluated our implementation of the DDGI algorithm and our ambiance occlusion method and proposed extensions on multiple scenes in terms of performance and quality of produced images. This section is structured as follows. We detail our quality comparisons in Section 7.1. The produced images by the DDGI method are compared to a reference image from a brute-force path-tracer. We also show a perceptual error between the reference image and the image produced by our implementation. The error was measured using Nvidia's FLIP algorithm [ANAM+20] which incorporates principles of the human eye to show the perceptual difference between two images using magma color space. Ray-tracing throughput, timings, and other performance evaluations are detailed in Section 7.2.

7.1 Qualitative Results

In this section, we show qualitative results of our implementation of the DDGI algorithm together using our extensions. First, we demonstrate how the method handles shadow and light leaks by itself. In order to evaluate these artifacts, we prepared an outdoor scene with a closed section with a door opening. The scene is enclosed in a $16 \times 8 \times 16$ grid, where each probe has a resolution of 8×8 and a 16×16 distance map. A spotlight then illuminates the scene placed far from the scene, simulating sunlight. Figure 7.1 shows this scene where we gradually added the visibility weights. No tone mapping or gamma correction is used to brighten the image artificially.

However, this scene was constructed using water-tight geometry. A scene like San Miguel is constructed using production techniques which means that there are walls that are represented only as planes. DDGI has problems with such geometry since its visibility weights cannot accurately cull contributions, and therefore the scene would need normal biasing. In Figure 7.2 we show light leaks caused by the infinitely thin walls and how our probe culling extension handles them without any normal biasing. The scene is enveloped by a probe grid $32 \times 8 \times 32$. The light leak is caused by a probe behind a one-sided wall that primarily samples the skybox environment map. In the figure, we also show the positions of the probes in the area for better reference where the troublesome probes are. Please note that the light leak

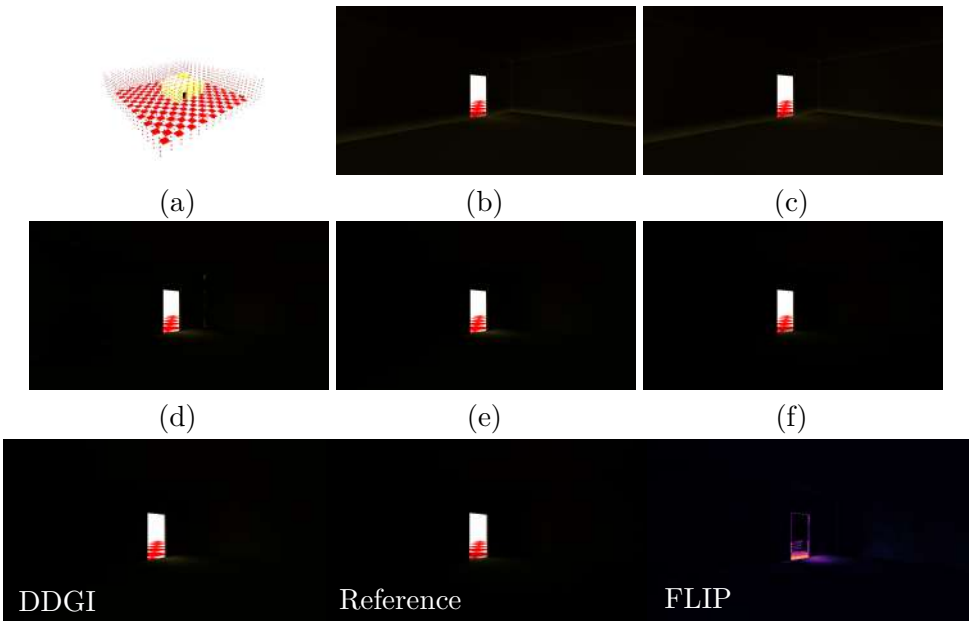


Figure 7.1: Irradiance visibility sampling comparison. Scene (a) is a closed room where light enters through a small door opening. Images (b) to (e) show the interior view of the room, illustrating the gradual reduction of light-leaking by added sampling weights. On image (b) are no weights applied, (c) backface weight, (d) Chebyshew weight, and finally (e) with normal biasing. The last three images are images (e) and (f) together with FLIP error.

on the right is not caused by DDGI but the inaccurate shadow mapping.

Figure 7.2: Staircase in San Miguel scene. We placed a $32 \times 8 \times 8$ irradiance field into the scene. The left image demonstrates light leaks caused by probes behind a one-sided wall. The right image shows global illumination with our probe culling enabled.

To see how the light propagates, we created a closed scene with a ceiling opening from which the light floods it, see Figure 7.3. A wall separates the room into two parts to create an occluded section in the scene. See how the DDGI method correctly makes a shadowed area in the corner of the room occluded by the wall. The scene was discretized by a $16 \times 16 \times 16$ probe grid, with 8×8 probe resolution and 16x16 distance map resolution.

To further demonstrate how DDGI handles ambient occlusion, we prepared an utterly closed scene with two occluders and one spotlight, as shown in

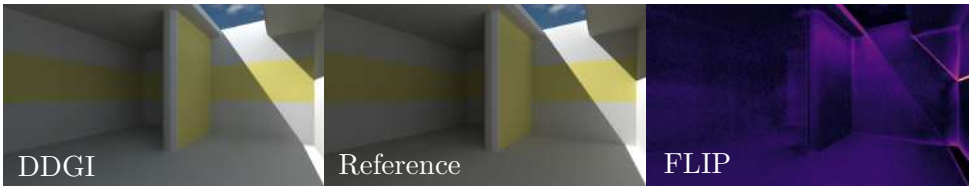


Figure 7.3: Our global illumination testing scene. The first Image demonstrates the global illumination produced by DDGI. The next is our ground truth reference. The final image shows the perceptual image error produced by FLIP. The probes were placed in a $16 \times 16 \times 16$ resolution, where each probe had an 8×8 irradiance map and 16×16 depth map.

Table 7.1. The table shows the scene with different grid configurations and environment map resolutions. Notice that to approximate correct ambient occlusion, the method can compensate for the lack of probes by environment map size.

The color bleeding effect introduced by the diffuse light transfer is illustrated in Figure 7.4. The scene is a closed box with two colored objects and a dynamically translating spotlight. We set the probe density to $2 \times 2 \times 2$, where each probe had an 8×8 irradiance map and 16x16 distance map. Notice that the approximated color bleeding matches almost perfectly the ground truth.

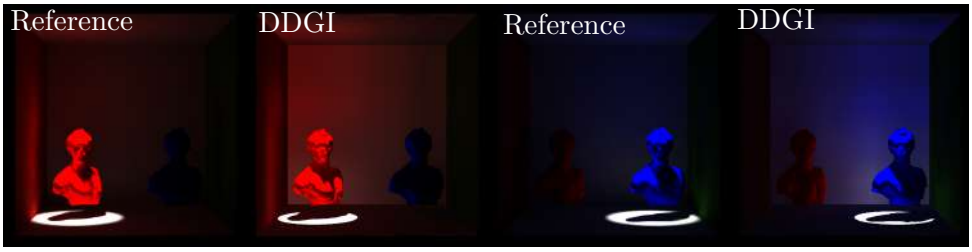


Figure 7.4: Comparison between the DDGI method and path-traced reference. The images illustrate color bleeding in the scene, with a dynamically translating light source introduced by the diffuse indirect light reflected from the two objects placed in the scene.

In Table 7.2 we further show the diffuse transfer captured by DDGI. The table shows our forest scene with a spotlight high in the air simulation sunlight. We demonstrate the amount of diffuse light transfer based on the grid configuration and size of environment maps. Notice that the green color bleeding in the bottom corner increases with the irradiance map size. Please note that DDGI does not cause the light leaks in the foliage but rather the faulty geometry used to create this scene. Figures 7.5 and 7.6 show the DDGI method on scenes with more elaborate geometry. Notice the diffuse light on the left wall reflected from the floor in Figure 7.5. In Figure 7.6, we can see noticeable light leaks on the leaves of the tree. This is due to the dense foliage geometry, which would require a much denser irradiance field to be sampled accurately. Also, notice how the scenes tend to be over illuminated by the indirect illumination produced by the method. Next, Figure 7.8 shows the

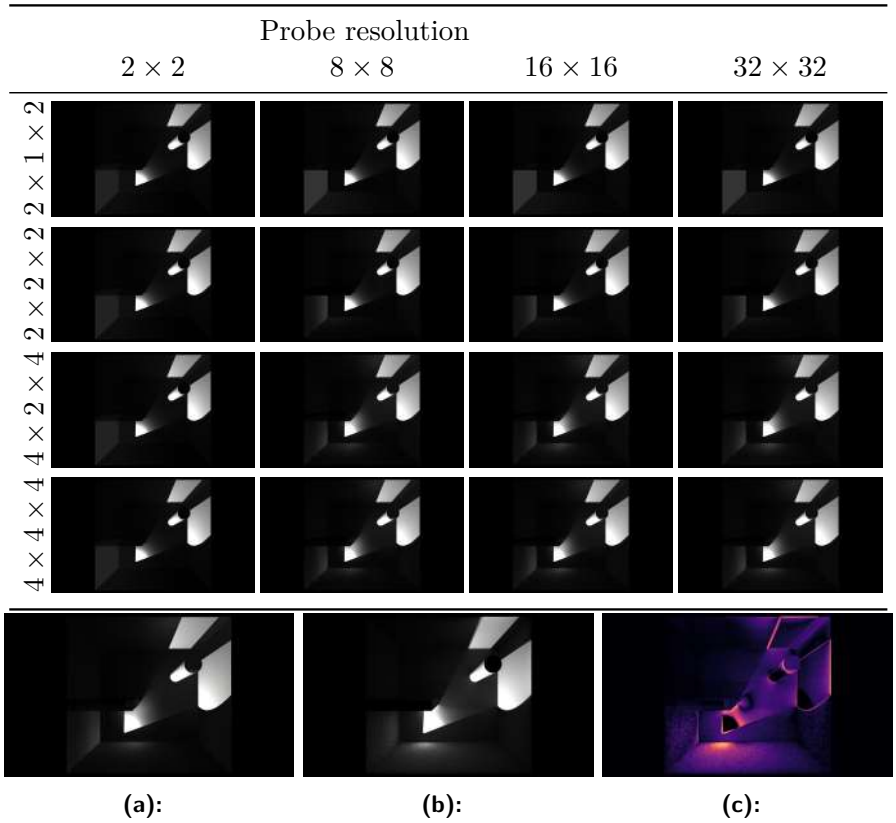


Table 7.1: The table illustrates the global illumination quality with respect to grid density and probe resolution size in a closed scene with a spotlight. The scene with $2 \times 2 \times 2$ probe grid resolution and 32×32 probe resolution (a) is then compared to our path-traced reference (b). The last image (c) shows the FLIP perceptual error between the compared images (a) and (b), where (a) has probe density $4 \times 4 \times 4$ and 32×32 resolution.

Figure 7.5: Sibenik global illumination comparison. The first Image demonstrates the global illumination produced by DDGI. The next image is our ground truth reference. The last image shows the perceptual image error produced by FLIP. The grid resolution was set to $16 \times 8 \times 12$ with 16×16 probe resolution.

Sponza scene with cascaded irradiance fields. Since we did not have a large enough scene that would generally be the target for this extension, we decided to use a coarsely discretized Sponza scene. Every cascade in the scene has $8 \times 8 \times 8$ probes. The figure shows three images where we gradually add lower cascades. Each image also shows probes of each cascade, where in the first picture are only shown probes of the largest cascade, and in the last are

	Probe resolution		
	8×8	16×16	32×32
$8 \times 8 \times 8$			
$16 \times 8 \times 16$			
$16 \times 16 \times 16$			

Table 7.2: Probe density and environment maps resolution comparison.

Figure 7.6: San Miguel global illumination comparison. The first image demonstrates the global illumination produced by DDGI. The next image is our ground truth reference. The last image shows the perceptual image error produced by FLIP. The grid resolution was set to $16 \times 8 \times 15$ with 64×64 probe resolution.

shown probes of the whole pyramid.

Lastly, we present dynamic global ambient occlusion produced by our shadowing field. In Figure 7.9 we demonstrate our global ambient occlusion on the Sponza Scene with only sunlight. The field grid density was set to $16 \times 16 \times 16$ where each probe had 64 rays and 16×16 occlusion maps.

We used our closed scene with two occluders to show the soft shadows the method can produce. There we placed two lights directed at one of the occluders. The Figure 7.10 shows the scene. Each image uses an increasing sharpening value used on the sampled occlusion to emphasize the shadows. For reference to the shape of the shadows, we include an image with shadow mapping.

Figure 7.7: Sponza global illumination comparison. The first image demonstrates the global illumination produced by DDGI. The following image is our ground truth reference. The last image shows the perceptual image error produced by FLIP. The grid resolution was set to $16 \times 16 \times 16$ with 8×8 probe resolution.

Figure 7.8: The images depict a Sponza scene with upto three irradiance field cascades. Each cascade had $8 \times 8 \times 8$ probe grid with 8×8 irradiance maps and 16 depth maps.

Figure 7.9: Dynamic Global Shadowing in Sponza scene. For the shadowing we used probe $16 \times 16 \times 16$ probe grid.

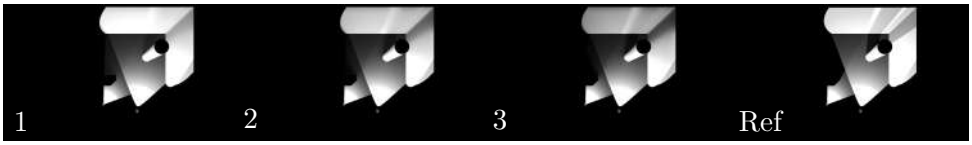
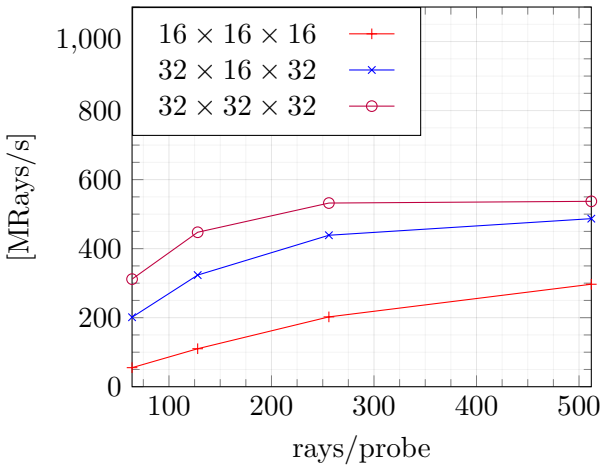


Figure 7.10: Ambient occlusion box scene. There are two spotlights placed in the scene. The images show the shadows produced by our shadowing field, where we sharpened the sampled values by a power shown on each picture. For reference, the last image shows shadows produced by shadow mapping. The AO field's density was set to $32 \times 8 \times 32$, where each probe had an occlusion atlas of size 16×16.

7.2 Performance Results

The performance was measured on the following machine. OS: Windows 10; Processor: AMD Ryzen 2700X, 4100 MHz, 8 cores, 16 Logical processors; RAM: 32 GB; GPU: RTX 3080. Times in milliseconds were measured using G3D's GPU profiler.

First, we present the raytracing throughput of the DDGI algorithm in Table 7.3 without our probe culling extension. The throughput of the DDGI method was measured in the Sibenik scene (72862 triangles) with a single irradiance field using multiple probe grid configurations and the number of rays per probe. This measurement's irradiance and depth atlas contained maps with sizes 8×8.



Rays per Probe	$16 \times 16 \times 16$	$32 \times 16 \times 32$	$32 \times 32 \times 32$
64	55.2	201.3	311.7
128	110.1	323.4	447.5
256	202.3	438.8	532.1
512	296.9	486.9	537.1

Table 7.3: Ray-trace throughput [MRays/s] of the irradiance field update with respect to probe density and number of rays per probe without probe culling.

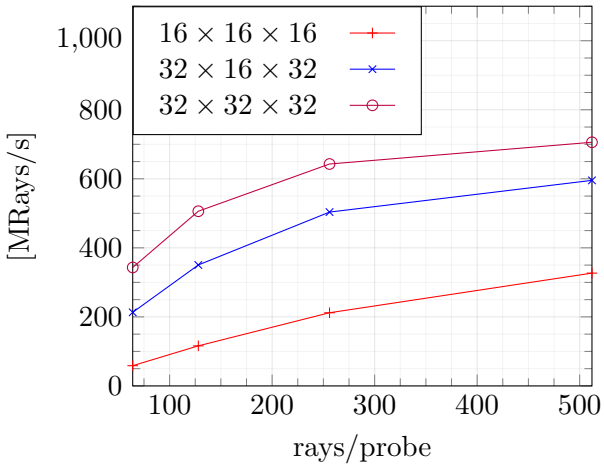
As the graph shows, the throughput starts to fall off the more rays each probe has. For reference, in Table 7.4 we present timings of the DDGI algorithm for grid configuration $32 \times 16 \times 32$, and two ray counts 64 and 512.

The table suggests that the bottleneck shown in the graph is not caused by ray-tracing but by the slow probe update that we tried to address with the geometry-aware update extension. In Table 7.5 we show the ray-tracing throughput with the same setup and the extension enabled. We also include reference timings of the probe update together with the probe culling step in Table 7.6.

Next, we measured the speed up of the probe atlas update in the San Miguel scene(10M triangles) based on the number of probes placed in the scene. Table 7.7 shows the average times of the update with and without

Irradiance field pass	64 rays time[ms]	512 rays time[ms]
Ray generation	0.06	0.425
Ray cast	2.245	4.357
Indirect shade	0.228	2.228
Direct shade	0.211	2.632
Probe irradiance update	0.057	1.858
Probe depth update	0.056	2.101

Table 7.4: Table of timings [ms] of the irradiance field update without probe culling.



Rays per Probe	16 × 16 × 16	32 × 16 × 32	32 × 32 × 32
64	58.6	213.2	343.2
128	116.1	350.4	506.1
256	212.0	503.6	642.9
512	326.8	595.5	705.9

Table 7.5: Ray-trace throughput [MRays/s] of the irradiance field update with respect to probe density and number of rays per probe with probe culling enabled.

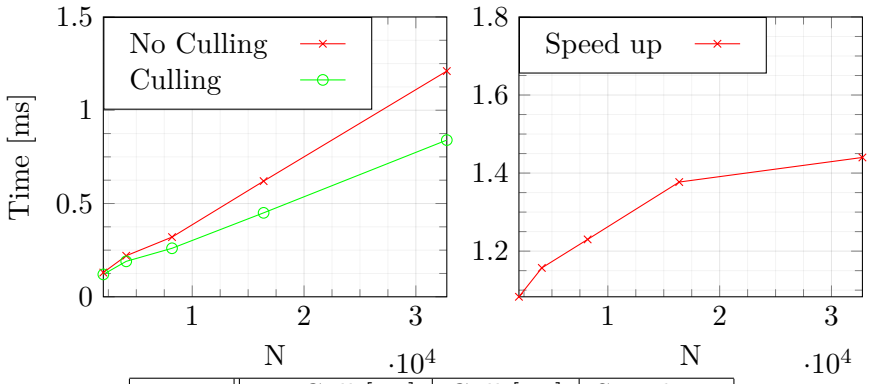
culling. It also shows the average speedup of the update the extension brings for San Miguel. Each probe in the scene had irradiance maps of size 8×8 and depth maps 16×16.

We also measured how well our extension prunes the uniform grid field on various scenes. Table 7.9 presents the amounts of active, inactive, and dead probes in different scenes and grid configurations. Notice that the Sponza scene had only three inactive probes in configuration $16 \times 16 \times 16$. In Sponza, there is not much open space where a probe could become inactive in this configuration. Therefore most probes find some number of relevant samples.

Next in Table 7.8 we show timings of each pass in our shadowing field. The measurement was taken on Sibenik Scene with one light outside of the cathedral. Around the cathedral we placed $32 \times 16 \times 32$ field, where each probe had 16×16 occlusion map.

Irradiance field pass	64 rays time[ms]	512 rays time[ms]
Indirect shade	0.201	1.297
Direct shade	0.153	1.244
Probe irradiance update	0.053	0.892
Probe depth update	0.049	1.014
Mark inactive probes	0.044	0.226

Table 7.6: Table of timings [ms] of the irradiance field update with probe culling enabled.



N	No Cull [ms]	Cull [ms]	Speed Up
2048	0.13	0.12	1.083
4096	0.22	0.19	1.157
8192	0.32	0.26	1.230
16384	0.62	0.45	1.377
32768	1.21	0.84	1.440

Table 7.7: Timing measurements of DDGI probe atlas update in San Miguel scene. We show timings comparison with and without probe culling for N probes in the scene on the left graph. On the right is shown the speed up the culling provides.

Shadowing field pass	time[ms]
Primatry ray generation	0.06
Shadow ray generation	0.07
Ray cast	2.58
Ray occlusion	0.11
Probe occlusion update	0.5

Table 7.8: Sibenik scene. Table of timings [ms] of the shadowing field update without probe culling.

Scene	Density	Alive	Sleeping	Dead
GI Box	$16 \times 16 \times 16$	2636	480	980
	$32 \times 32 \times 32$	17606	9772	6090
San Miguel	$16 \times 16 \times 16$	2049	559	788
	$32 \times 16 \times 32$	7375	6001	3008
	$32 \times 32 \times 32$	15237	12363	5168
Sponza	$16 \times 16 \times 16$	3667	3	426
	$32 \times 16 \times 32$	13008	1286	2090
	$32 \times 32 \times 32$	24221	3887	4660
Forest	$16 \times 16 \times 16$	2941	1135	20
	$32 \times 16 \times 32$	8888	7370	126
	$32 \times 32 \times 32$	17470	15109	189

Table 7.9: Culling

Chapter 8

Discussion and Future Work

As mentioned in the methods overview and results, the method DDGI has some drawbacks, which we further discuss in Section 8.1. Two further possible extensions to improve the performance of the method are discussed in Section 8.2.

8.1 DDGI Drawbacks

As mentioned in Section 4.5, the method can suffer from noticeable flickering. This problem can be caused by a low hysteresis parameter or an insufficient quantity of sampling rays in a scene with bright materials. Unfortunately, the only way to stabilize the image is to increase the number of sampling rays, increasing the number of samples to process in the probe update. Unfortunately, this could lead to infeasible number of rays cast in each frame.

Another drawback of the method is a noticeable illumination delay caused by the recurrent computation of indirect light between frames. This problem is most pronounced in mostly static scenes where for example we suddenly occlude large part of the scene; however, it is not apparent in dynamic scenes with many moving objects.

As was shown in Figures 7.5 the method can struggle with light leaks in scenes with dense geometry, such as foliage, and not water-tight geometry. These problems we were addressed with our extensions. However, in case of light leaks on fine geometry, there is so far we can go. Even a dense irradiance field cannot accurately approximate high-frequency ambient occlusion. A better and much cheaper approach is to use ambient occlusion methods, which work best for such situations as SSAO. SSAO would create shadows on the dense geometry and therefore reduce light leaks. It would also darken corners in a scene, making it look more natural. Of course, the same problem with occlusion applies to our shadowing field, which cannot accurately capture high-frequency occlusion.

8.2 Future Work

Even though the approximation of diffuse transfer produces images that are sometimes almost indistinguishable from the ground truth, the probe update in each frame is still relatively costly, even with our update extension. We update every probe in each frame with an initial set ray budget even though they are inactive, which leads to an infeasible amount of ray casts. An adaptive approach would be more appealing, where we would change the budget of each probe based on its surroundings. One approach would be to use our data about active probes and reduce the ray budget of inactive probes with time. Another way would be to use data about each probe's contribution. A probe with a low contribution to the overall global illumination, such as probes placed in geometry or dark areas, does not need the same ray budget as a probe in a well-illuminated area.

We could take inspiration from the paper of K. Vardis et al. [VVP21], which describes an illumination-driven technique to optimize automatic probe placement methods for light baking, e.g., uniform 3D grids or tetrahedral grids. More specifically, their use of YCoCg color space to determine which probes to disable. To reduce the number of rays a probe emits, we would compute the cost based on absolute percentage errors, e.g., SMAPE. Since the DDGI method already computes illumination between frames, we calculate the errors of current and previous frames. Probes with minor errors or probes placed inside geometry would gradually reduce their budget to a set minimum.

Another possible extension would be to use more elaborate probe placement to break the uniform grid placement strategy. This way, we would reduce artifacts caused by the grid structure and increase performance by reducing the required probes in the scene. However, this approach would possibly lessen the dynamic aspect of the method. Inspiration could be taken from the paper by Wang et al. [WKKN19] or Sedláček's approach [Sed19] for placing sparse radiance probes [SL17]. The solution by Sedláček uses a voxelization of a scene together with a few simple rules to avoid putting irradiance probes inside geometry and to avoid probe overlap. However, as already mentioned, this approach might limit the method to static or only partially dynamic scenes due to the costly voxelization of the scene.

Chapter 9

Conclusion

We showed our implementation of a recent real-time global illumination method called DDGI, which complements traditional rasterization by hardware-accelerated ray-tracing.

We also presented our extensions, such as geometry aware probe update, which culls passive probes from the update and sampling process. The extension was mainly focused on addressing performance issues with high-density grids. However, the method can also have an emerging behavior of culling irradiance contributions of probes closed inside geometry that would still be sampled. Therefore, it reduces artifacts caused by the grid placement strategy.

We also extended the algorithm by cascaded fields. This extension increased the visual fidelity around the viewer without unnecessarily wasting memory on dense fields.

Furthermore, we presented our modified DDGI algorithm, which reconstructs low-frequency soft shadows using shadow rays.

We showed our findings and results from our implementation of the technique. We also discussed possible extensions such as adaptive scaling of probes' ray budget and possible dynamic probe placement tactics.

Bibliography

[ANAM+20] Pontus Andersson, Jim Nilsson, Tomas Akenine-Möller, Magnus Oskarsson, Kalle Åström, and Mark D Fairchild, *Flip: A difference evaluator for alternating images.*, Proc. ACM Comput. Graph. Interact. Tech. **3** (2020), no. 2, 15–1.

[BS08] Louis Bavoil and Miguel Sainz, *Screen space ambient occlusion*, NVIDIA developer information: http://developers. nvidia. com **6** (2008), no. 2.

[BWP+20] Benedikt Bitterli, Chris Wyman, Matt Pharr, Peter Shirley, Aaron Lefohn, and Wojciech Jarosz, *Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting*, ACM Transactions on Graphics (TOG) **39** (2020), no. 4, 148–1.

[CDE+14] 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 **3** (2014), no. 2.

[DL06] William Donnelly and Andrew Lauritzen, *Variance shadow maps*, Proceedings of the 2006 symposium on Interactive 3D graphics and games, 2006, pp. 161–165.

[ED08] Thomas Engelhardt and Carsten Dachsbacher, *Octahedron environment maps.*, VMV, 2008, pp. 383–388.

[GTGB84] Cindy M Goral, Kenneth E Torrance, Donald P Greenberg, and Bennett Battaile, *Modeling the interaction of light between diffuse surfaces*, ACM SIGGRAPH computer graphics **18** (1984), no. 3, 213–222.

[H+07] Mark Harris et al., *Optimizing parallel reduction in cuda*, Nvidia developer technology **2** (2007), no. 4, 70.

[HH04] Shawn Hargreaves and Mark Harris, *Deferred shading*, Game Developers Conference, vol. 2, 2004, p. 31.

[HMY12] Takahiro Harada, Jay McKee, and Jason C Yang, *Forward+: Bringing deferred lighting to the next level.*, Eurographics (Short Papers), 2012, pp. 5–8.

[HSA91] Pat Hanrahan, David Salzman, and Larry Aupperle, *A rapid hierarchical radiosity algorithm*, Proceedings of the 18th annual conference on Computer graphics and interactive techniques, 1991, pp. 197–206.

[ICG86] David S Immel, Michael F Cohen, and Donald P Greenberg, *A radiosity method for non-diffuse environments*, Acm Siggraph Computer Graphics **20** (1986), no. 4, 133–142.

[Jen96] Henrik Wann Jensen, *Global illumination using photon maps*, Eurographics workshop on Rendering techniques, Springer, 1996, pp. 21–30.

[Kaj86] James T Kajiya, *The rendering equation*, Proceedings of the 13th annual conference on Computer graphics and interactive techniques, 1986, pp. 143–150.

[KD10] Anton Kaplanyan and Carsten Dachsbacher, *Cascaded light propagation volumes for real-time indirect illumination*, Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games, 2010, pp. 99–107.

[KISS15] Benjamin Keinert, Matthias Innmann, Michael Sänger, and Marc Stamminger, *Spherical fibonacci mapping*, ACM Transactions on Graphics (TOG) **34** (2015), no. 6, 1–7.

[LW93] Eric P. Lafortune and Yves D. Willems, *Bi-directional path tracing*, PROCEEDINGS OF THIRD INTERNATIONAL CONFERENCE ON COMPUTATIONAL GRAPHICS AND VISUALIZATION TECHNIQUES (COMPUGRAPHICS '93, 1993, pp. 145–153.

[MGNM19] Zander Majercik, Jean-Philippe Guertin, Derek Nowrouzezahrai, and Morgan McGuire, *Dynamic diffuse global illumination with ray-traced irradiance fields*, Journal of Computer Graphics Techniques Vol **8** (2019), no. 2.

[MM02] Vincent CH Ma and Michael D McCool, *Low latency photon mapping using block hashing*, Graphics Hardware, Citeseer, 2002, pp. 89–98.

[MMK+21] Zander Majercik, Thomas Mueller, Alexander Keller, Derek Nowrouzezahrai, and Morgan McGuire, *Dynamic diffuse global illumination resampling*, ACM SIGGRAPH 2021 Talks, 2021, pp. 1–2.

[MMM17] Morgan McGuire, Michael Mara, and Zander Majercik, *The G3D innovation engine*, 01 2017, `https://casual-effects.com/g3d`.

[MSW10] Oliver Mattausch, Daniel Scherzer, and Michael Wimmer, *High-quality screen-space ambient occlusion using temporal coherence*, Computer Graphics Forum, vol. 29, Wiley Online Library, 2010, pp. 2492–2503.

[PBD+10] Steven G Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, et al., *Optix: a general purpose ray tracing engine*, Acm transactions on graphics (tog) **29** (2010), no. 4, 1–13.

[PDC+05] Timothy J Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan, *Photon mapping on programmable graphics hardware*, ACM SIGGRAPH 2005 Courses, 2005, pp. 258–es.

[RDGK12] Tobias Ritschel, Carsten Dachsbacher, Thorsten Grosch, and Jan Kautz, *The state of the art in interactive global illumination*, Computer graphics forum, vol. 31, Wiley Online Library, 2012, pp. 160–188.

[Sed19] Šimon Sedláček, *Real-time global illumination using irradiance probes*.

[SL17] Ari Silvennoinen and Jaakko Lehtinen, *Real-time global illumination by precomputed local reconstruction from sparse radiance probes*, ACM Transactions on Graphics (TOG) **36** (2017), no. 6, 1–13.

[SNY11] Peter-Pike Sloan, Derek Nowrouzezahrai, and Hong Yuan, *Wrap shading*, Journal of Graphics, GPU, and Game Tools **15** (2011), no. 4, 252–259.

[SSMW09] Daniel Scherzer, Michael Schwärzler, Oliver Mattausch, and Michael Wimmer, *Real-time soft shadows using temporal coherence*, International Symposium on Visual Computing, Springer, 2009, pp. 13–24.

[VVP21] Konstantinos Vardis, Andreas Alexandros Vasilakis, and Georgios Papaioannou, *Illumination-driven light probe placement*.

[WKKN19] Yue Wang, Soufiane Khiat, Paul G Kry, and Derek Nowrouzezahrai, *Fast non-uniform radiance probe placement and tracing*, Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, 2019, pp. 1–9.

[ŽBS+05] Jiří Žára, Bedřich Beneš, Jiří Sochor, Petr Felkel, et al., *Moderní počítačová grafika*, Computer press, 2005.

Appendix A
User Manual

All controls are listed in this appendix. Prepared scenes can be loaded from inside the application.

A.1 User Interface

The application provides a basic user interface, as shown in Figure A.1. The toolbar in the bottom left is used to open widgets to control the application, such as debug window(top window), Debug camera, Scene Editor, and many more. However, the debug window is the main widget used to control our application. It provides control over different shadowing modes and DDGI. Furthermore, tabs on the top of the window give control over the implemented DDGI method parameters, dynamic shadowing field, and even the used path tracer.

The scene can be changed by selecting one from the drop list next to the scene in the scene editor.

To unlock the camera, change the camera in the Camera drop list to Debug Camera or press F2. Then the camera is controlled by WSAD and the mouse while holding the left mouse button.

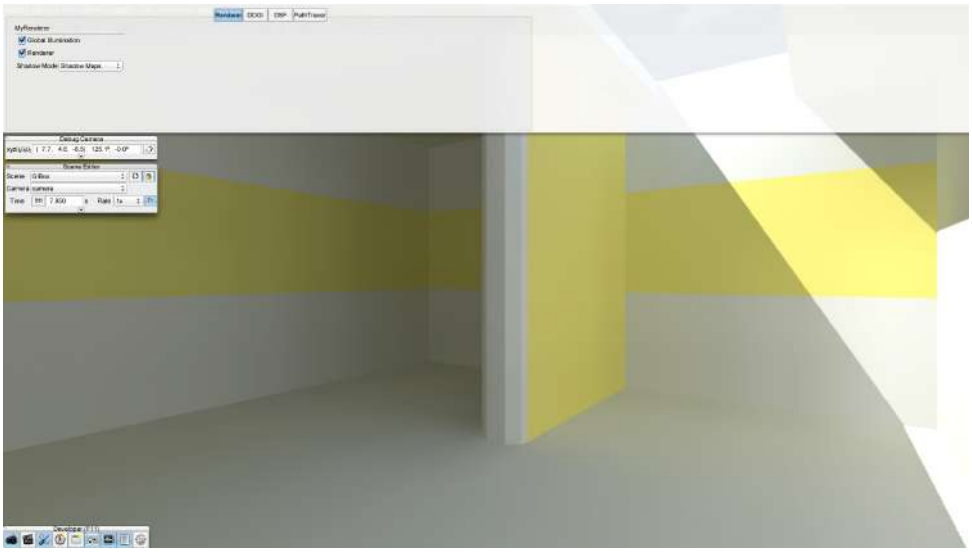


Figure A.1: User Interface

Appendix B

Dependencies

G3D Innovation Engine[1]

[1] http://casual-effects.com/g3d/www/index.html

Appendix C

Electronic Contents