Diffuse Global Illumination using Surfels

Master's thesis in Computer science and engineering

Hampus Ekberg

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

Diffuse Global Illumination using Surfels

Hampus Ekberg

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Diffuse Global Illumination using Surfels
Hampus Ekberg

© Hampus Ekberg, 2022.

Supervisor: Erik Sintorn, Department of Computer Science and Engineering
Examiner: Ulf Assarsson, Department of Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Description of the picture on the cover page (if applicable)

Typeset in LaTeX
Gothenburg, Sweden 2022

Diffuse Global Illumination using Surfels
Subtitle
Hampus Ekberg
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Simulating global illumination is an important part of rendering realistic-looking scenes. Diffuse global illumination is a subset of global illumination which focuses on the diffuse reflections of light. Methods for solving diffuse global illumination usually require either pre-computations of the light or a lot of raycasts every frame. This thesis explores an alternative approach inspired by *Global Illumination Based on Surfels* to see if it is possible to reduce the number of rays cast each frame while maintaining a similar visual quality as previous methods, thus reducing the computation cost. This exploration was accomplished by implementing the alternative approach and comparing both performance and visual quality results to a pre-existing diffuse global illumination solution. The results show that it is possible to limit number of rays while keeping a similar visual quality, but the implementation as described in the thesis has other computation bottlenecks that end up overriding the gains from reducing number of rays in many cases.

Keywords: Computer, science, computer science, engineering, project, thesis.

Acknowledgements

I would like to thank my supervisor Erik Sintorn for the very helpful insights and discussions. Big thank you also to Therese for all the support and encouragement throughout not only the thesis-writing but through all my years at Chalmers.

Hampus Ekberg, Gothenburg, June 2022

Contents

1 Introduction **1**
1.1 Rendering Equation . . . 2
1.2 Problem Statement . . . 2

2 Previous Works **5**

3 Theory **7**
3.1 GIBS - Global Illumination Based On Surfels . . . 7
3.1.1 Surfels . . . 7
3.1.1.1 Placement . . . 7
3.1.1.2 Management . . . 8
3.1.2 Acceleration Structure . . . 9
3.1.3 Irradiance cache . . . 10
3.1.4 Light application . . . 12
3.2 DDGI - Dynamic Diffuse Global Illumination . . . 13
3.2.1 Irradiance caches . . . 13
3.2.2 Light application . . . 14

4 Implementation **15**
4.1 Surfel representation . . . 15
4.2 Sorting surfels . . . 16
4.2.1 Counting surfels . . . 16
4.2.2 Prefix Sum . . . 17
4.2.3 Finalization . . . 17
4.3 Computing irradiance . . . 18
4.4 Rendering the scene . . . 19
4.5 Surfel placement . . . 20

5 Results **23**
5.1 Visual quality . . . 23
5.2 Performance . . . 25

6 Discussion **27**
6.1 Ethical considerations . . . 28
6.2 Conclusions . . . 29

Bibliography **31**

1

Introduction

In the world of computer games, there is a constantly growing demand for more realistic graphics. The available hardware is pushed to its limits in order to deliver the player the best possible visuals at an intractable framerate. A very important aspect of rendering a photorealistic image is the simulation of light interacting with the scene.

The way we humans see things is through photons being emitted from a light source, bouncing on a surface or, more likely, multiple surfaces, to finally hit our eyes. In essence computer graphics is not much different, but instead of photons hitting the eyes we want to know how many photons hit a virtual camera as they bounce around in a three-dimensional scene. Billions of photons hit our eyes every second, fully simulating all of them is unfeasible, especially given the tight time constraints involved in games. Instead, simplifications have to be made in order to approximate realistic lighting.

The light traveling directly from a light source to a single surface and then to the camera, the so-called direct light, can be calculated quite simply. We know how many photons travel in a given direction from each light source, so calculating the number of photons hitting a certain surface location is trivial. From there, since we are only concerned with one possible path when it comes to direct lighting, we can easily approximate how many photons bounce from the given surface to the camera.

Simulating indirect light, the light bouncing on multiple surfaces before hitting the camera, is more complex. Instead of being concerned with one path for each pixel and light source, there are instead an infinite number of paths that any given photon could have taken before reaching the camera. Simulating indirect light is important for rendering realistic scenes, see Figure 1.1.

The problem of simulating not only the direct light but also the indirect light is called *Global Illumination*. Global Illumination includes all sorts of light effects such as refraction and caustics, but I will in this thesis focus only on the diffuse reflection of light, commonly known as *Diffuse Global Illumination*. Diffuse reflection is a type of reflection that evenly spreads out light in the hemisphere oriented along the normal of the surface where the light hits, see Figure 1.2.

Figure 1.1: Two images depicting renders of the Crytek scene. The image on the left was rendered only considering the direct light from the sun. The image on the right was rendered with direct light as well as indirect light

1.1 Rendering Equation

How best to calculate indirect light is a problem that has been studied for decades. The concepts of light from a physical point of view are fairly well-understood. In 1986 Kajiya formulated an equation known as the *Rendering Equation*, which describes the total light emitted from any given point in any given direction [12]. Multiple forms of this equation exists. One such version, taken from the book Real-Time Rendering by Akenine-Möller et al., can be seen in Equation 1.1 [2].

$$L_o(p, v) = L_e(p, v) + \int_{l \in \Omega} f(l, v) L_o(r(p, l), -l)(n \cdot l)^+ dl \tag{1.1}$$

Where $\boldsymbol{L_o(p, v)}$ is how much radiance (flux per steradian per square meter) is leaving a point $\mathbf{p}$ in direction $\mathbf{v}$,$\boldsymbol{L_e(p, v)}$ is the emitted radiance at a point $\mathbf{p}$ in direction $\mathbf{v}$, $\mathbf{n}$ is the normal vector, $\boldsymbol{\Omega}$ is the hemisphere oriented along the normal vector, $\mathbf{f(l,v)}$ is the so-called *Bidirection reflectance distribution function*, $\mathbf{r(p,l)}$ is a ray casting function which casts a ray in direction $\mathbf{l}$ from position $\mathbf{p}$ and returns the position of the first surface point hit. The equation will be explained further in later sections, but for now we can see that the equation is infinitely recursive. In order to calculate $\boldsymbol{L_o(p, v)}$, we need to calculate multiple $\boldsymbol{L_o(r(p, l), -l)}$, further cementing the difficulty of simulating light. Despite its age, this equation still lies at the heart of most rendering algorithms.

1.2 Problem Statement

Due to the expensive nature of the diffuse global illumination problem, simplifications to the problem have to be made in order for a solution to be fast enough for use

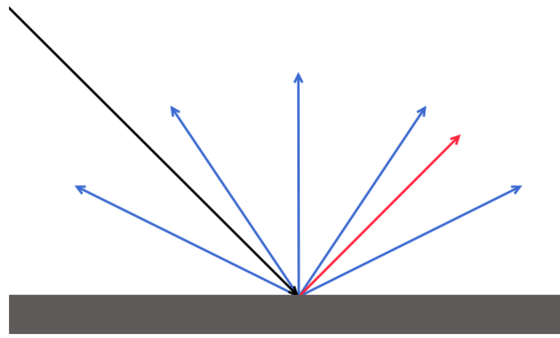

Figure 1.2: Incident light (black) hits a surface resulting in both diffuse (blue) and specular (red) reflections. Most surface materials would result in a combination of both specular and diffuse reflections, but this thesis will focus on computations that consider all surfaces as fully diffuse

in games. These simplifications all come with different trade-offs. Some solutions only work for static environments while others work for dynamic ones. For some solutions, a pre-computation step is required before the algorithm can be used, and some require artist intervention in order to get the desired result.

One state-of-the-art solution for diffuse global illumination that has seen use in several modern games is DDGI, or RTXGI as it is sometimes called, based on the research by Majercik et al. [14] [15] [6] The general idea of the algorithm is to place out irradiance probes in a grid throughout the scene. The job of these probes is to keep track of the irradiance from all directions. The way this is accomplished is that rays are cast in multiple directions from each probe. The light at the hit location of all the rays are then calculated and accumulated inside the storage of the probe which it was cast from. The irradiance is stored as a very low-resolution, spherically mapped 2D texture in order to describe irradiance from multiple directions around the sphere. When rendering a surface, the nearby probes are sampled in the direction of the surface normal, and then interpolated to calculate the indirect light at a certain surface point for the normal of that surface. This indirect light calculation is also done during the light accumulation step to effectively approximate infinite bounces.

Using DDGI it is possible to calculate irradiance for all points and all directions in space. A drawback of this is that there is a lot of potentially wasted information. Excluding volumetric effects, which are outside the scope of this thesis, the only place where the irradiance information is needed is on surfaces of the scene. Even if a probe is near a surface, only the light coming in towards the surface is needed, so storing irradiance for all directions can be unnecessary. This information leads to wasted memory. It also means a lot of potentially unnecessary computations are being made to figure out irradiance for positions and directions that might not be relevant. As the algorithm is designed to work for dynamic scenes though, the computations might become useful if an object moves into the area of the probe.

When it comes to real-time rendering, computation time is a crucial metric. Games need to be able to render at least 30 frames per second to feel responsive, and preferably a lot more. Computing the irradiance involves a lot of raycasting, which is a very time-consuming operation. Recent advances in hardware have sped this operation up drastically, but even so, needless raycasting is wasted computation time which could be better spent elsewhere.

This thesis aims to find an alternative approach to dynamic diffuse global illumination, which requires less raycasting to achieve a similar visual quality as DDGI.

The algorithm described by Brinck et al. [4] called *Global Illumination Based On Surfels*, or *GIBS*, appears to suit these criteria. Instead of computing irradiance for all points in space, it only computes irradiance where it is needed, on visible (or recently visible) surfaces. Instead of computing irradiance for all directions, it computes irradiance only in the normal direction of the surface. This does mean that if an object moves, the information of the indirect light at the new position might not be calculated as it would be for DDGI, which could have detrimental effects on visual quality.

I will, through implementing a slimmed down version of GIBS and evaluating it against an already implemented version of DDGI, show that the implementation in fact does require fewer raycasts in order to get a mostly similar visual quality, but that some other costs increase. The trade-offs implicit with the approach of only calculating what is currently needed (GIBS) and calculating for potential future needs (DDGI) are also analyzed.

2
Previous Works

Diffuse Global Illumination is an area that has been actively researched for decades. This section will give a brief overview of some of the relevant advances.

At the foundation of most global illumination research lies Kajiya's Rendering Equation [12]. The equation describes how the outgoing light at a surface point in a specific direction depends on the incoming light from all directions in the hemisphere oriented around the normal vector at that point.

Radiosity was one of the first algorithms used for calculating diffuse global illumination. Originally designed for simulating heat transfer, it was adapted to the problem of diffuse global illumination by Goral et al. [7] [8] The algorithm works by discretizing surfaces to one or more smaller surfaces, called patches, calculating the visibility between all the patches and then bouncing light between patches based on the visibility. When radiosity first appeared it was not designed to be used in real-time applications, but with advances in hardware and some adaptations it has seen real-time uses [16] [22].

An approach used early on in games like Quake and Half-Life 2 as a solution for diffuse global illumination is lightmapping [1] [19]. Lightmapping works by laying out all the surfaces of a scene into a flat texture that stores indirect light. The indirect light is computed as a pre-computation step, meaning a more expensive algorithm such as Radiosity or Photonmapping could be used. One obvious trade-off with this approach is that the indirect light will not adapt to changes in the environment or changes in the illumination.

Greger et al. [9] introduced *Irradiance Volumes*, an alternative representation of irradiance which allows calculation of irradiance for any point in space as opposed to only surfaces. The basic idea is to place a grid of light probes in the world and compute irradiance in those. The light probes can be represented as either textures mapping to a sphere or using spherical harmonics [20]. Much like lightmapping, the irradiance probes are mostly used for precomputed irradiance for games. Unlike lightmapping, the irradiance probes enable the application of light to dynamic objects. Irradiance probes have also been used to handle static scenes but dynamic lights in real-time [18].

Pfister et al. [21] present a way of discretizing surfaces using a set of discs referred to as *surfels*. Jendersie et al. [11] use the concept of these surfels in their

global illumination algorithm, where the surfels act as sampling points for direct light, and transport the light through precomputed links to irradiance caches. As both the surfel placement and the transport links are precomputed, the light will not react to dynamic objects moving around. The direct light is however sampled from the surfels in real-time, so the algorithm correctly responds to dynamic lighting.

Recent advances in hardware specifically designed for raycasting have improved the viability of fully dynamic, real-time, diffuse global illumination algorithms. Majercik et al. [14] improve on previous irradiance probe algorithms by introducing occlusion information as well as a method for updating the irradiance caches dynamically during runtime. They further improve the method by introducing a more efficient method of querying indirect light as well as other optimizations [15]. This method is now commercially known as RTXGI [6] and is able to simulate diffuse global illumination during real-time in a completely dynamic scene with dynamic lights. The method does however require hardware capable of evaluating a very large amount of raycasts every second in order to run at a responsive framerate.

Another approach for fully dynamic real-time diffuse global illumination was introduced by Barré-Brisebois et al. [3] as part of a new experimental renderer [23]. The approach is centered around discretizing the scene as well as storing irradiance in surfels rather than classic irradiance probes. Unlike previous uses of surfels mentioned [11], the surfels are in this case placed and recycled dynamically during the run-time of the program instead of being placed as a pre-computation step.

3
Theory

This chapter is mostly devoted to the theory behind the GIBS algorithm by Brinck et al. [4]. As the algorithm will be evaluated against DDGI, the chapter will present the theory behind that as well. As both approaches share a lot of similarities, the sections about DDGI will mostly highlight the differences.

3.1 GIBS - Global Illumination Based On Surfels

The GIBS algorithm aims to approximate a solution to Equation 1.1, which it accomplishes by making a few simplifications to the problem. The first simplification is, instead of solving the equation for every pixel, which for a standard resolution of 1920x1080 would result in more than 2 million pixels, to discretize the scene into surfels and solve the rendering equation for each surfel. Another simplification is to accumulate the irradiance over time rather than solving it all fully every frame. The integration part of the equation is approximated numerically by raycasting a number of times every frame and averaging the result in a specific manner.

GIBS is a continuation of the work presented in Barré-Brisebois et al. [3], an experimental hybrid renderer utilizing raytracing, compute shaders, and rasterization to render different global illumination effects.

3.1.1 Surfels

Surfels, an abbreviation of *Surface Element*, are a way of describing a part of a surface. They are typically represented as disks with a position, a normal vector and a radius. As the name suggests, GIBS makes heavy use of surfels both as a location for storing irradiance, and as a way of dividing the global illumination problem into more manageable pieces.

3.1.1.1 Placement

Unlike the algorithm by Jendersie et al. [11] where surfel placement is pre-computed, the GIBS algorithm dynamically places surfels during runtime as the need arises. This is accomplished by dividing the rendered image in 16x16 pixel tiles, finding the pixel that has the least coverage by the current set of placed surfels in each tile, and if the pixel has a low enough coverage there is a chance that the algorithm places a surfel at that location.

The reason there is only a chance of a surfel being placed, as opposed to doing it everywhere coverage is low enough, is to avoid surfels too close to each other. In cases where all pixels have zero coverage, surfels would be placed in each 16x16 tile of the image. Depending on the distance to the surface this might be fine, but if the surface being discretized is too close to the camera, a lot of surfels will be placed on a small area of the surface, overlapping each other.

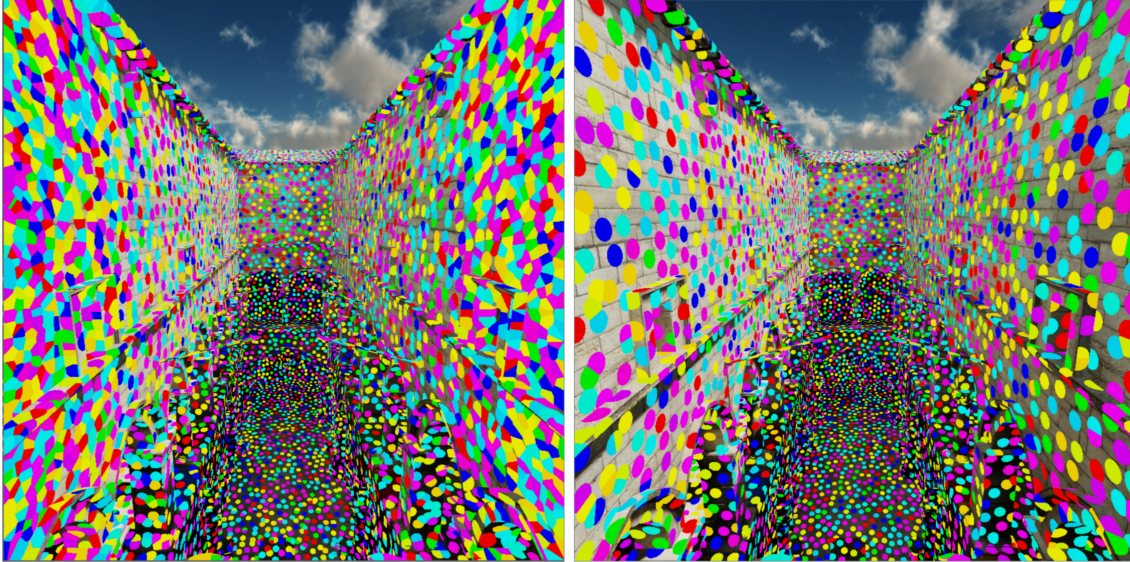

Figure 3.1: Two images showing surfels placed on the Sponza scene. Image on the left shows an algorithm that always places surfels if the coverage is low enough. Note the overlapping surfels on the close walls. Image on the right shows an algorithm that has a chance of spawning surfels when coverage is low enough. There are still some overlapping surfels, but not nearly as many.

While the rendered image is used for the placement of surfels, the surfels are not actually placed in screen space. As the irradiance is accumulated over multiple frames in the surfels, placing them in screen space would mean having to reset everything every time the camera moves or rotates. Instead, the surfels are placed at the position represented by the pixel and can thus persist across camera movements.

3.1.1.2 Management

All the surfel memory exists on the GPU, and is accessed and manipulated solely by compute and fragment shaders on the GPU. This enables a large amount of parallelism and prevents having to sync data between GPU and CPU, which would introduce a lot of waiting and other costs in terms of memory bandwidth. Memory is preallocated to only handle a certain amount of surfels at a time, which means there are no dynamic memory allocations.

The surfels persist once placed, but in order to fully adapt to dynamic scenes, surfels will have to be able to be removed if they are no longer relevant. This is done with the help of a stack of available surfels. When a surfel is placed, the system takes an

ID from the top of the surfel stack and uses the memory associated with that ID to represent the surfel. When a surfel is no longer needed, the ID of the surfel is added to the top of the surfel stack.

To decide when to remove, or rather recycle, a surfel, a heuristic is used based on: Amount of surfels currently placed, time since the surfel was last seen, distance to the surfel, and the surfel coverage of the surrounding area. If a surfel passes the recycling heuristic, there is a random chance that it is recycled. A random chance is used for the same reason it is used for placement. If an area has too much coverage and a completely deterministic metric was used, all the surfels in that area would be recycled, but only some of them need to be recycled in order to lower the coverage.

In order to maintain a constant level of quality for the lighting across all visible surfaces, the surfels automatically grow and shrink in size based on how far away they are from the camera. Far away surfels are larger, while close surfels are smaller. The idea is that surfels should have a constant size in screen space no matter how far away they are. With surfels growing, the recycling algorithm can decide that there are too many surfels in one spot and recycle them, meaning that there will be more surfels closer to the camera where they make the most difference.

In order to properly support fully dynamic scenes where objects can move around, the surfels are designed to follow whichever object it was placed on. This is accomplished by, instead of storing position and normal of the surfel in world space, storing position and normal in a space local to the object as well as a transform ID. The transform ID is then used to transform the local position of the surfel to a world position every frame.

3.1.2 Acceleration Structure

When calculating the surfel coverage and applying the indirect light, which will be described in Section 3.1.4, the algorithm needs to know which surfels are near a certain position. A naïve approach of iterating through every single surfel and calculating the distance is unfeasible due to the large number of surfels. Instead, an acceleration structure is used, which allows for querying nearby surfels given a position.

Barré-Brisebois et al. [3] use a 3D uniform grid, a simple data structure which divides the space into a three-dimensional grid of uniformly sized cells and stores lists of surfels contained in each cell. This works well for cases where the scene is quite small and has a pre-defined extent, as it is then possible to find a good balance between number of cells and cell sizes. As GIBS is designed to work for scenes of any size, including huge open-world scenes, a traditional uniform grid would require either cells that are too large, removing the benefit of the acceleration structure, or too much memory to represent the many cells required to fill the space. Instead, Brinck et al. [4] recommend a hybrid approach where the area around the camera uses a regular uniform grid, and for further away surfels a similar type of grid is

used, but instead of the cells being uniformly sized in world space, they are uniformly sized in a space defined by the perspective projection of the camera. This has the effect that cells further away are larger, which makes sense since the further away surfels will be larger, and detail is less important for objects far away from the camera, see Figure 3.2.

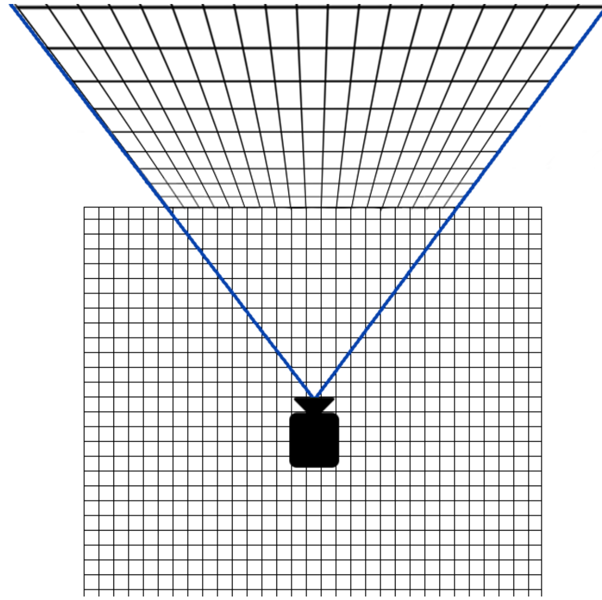

Figure 3.2: An illustration of the acceleration structure used in GIBS. Near the camera is a traditional uniform grid, and further away is a grid distorted by perspective

Since the surfels may have moved from frame to frame, the surfels have to be sorted into the grid every frame.

3.1.3 Irradiance cache

One of the purposes of the surfels is to compute, and store, the incoming light for the area of the surface they represent. For each surfel we want to solve the integral of the rendering equation (Equation 1.1, integral copied below for simplicity).

$$\int_{\boldsymbol{l}\in\Omega} \boldsymbol{f}(\boldsymbol{l},\boldsymbol{v})\boldsymbol{L}_o(\boldsymbol{r}(\boldsymbol{p},\boldsymbol{l}),-\boldsymbol{l})(\boldsymbol{n}\cdot\boldsymbol{l})^+ \boldsymbol{dl} \tag{3.1}$$

No analytic solutions to the rendering equation are known to exist, so the best approach is generally to estimate a solution by taking multiple random samples, and using a form of Monte Carlo integration. In practice this is done by raycasting in a random direction of the hemisphere, calculating the outgoing light at the hit location in the direction of the ray, and tracking a scaled moving average of the result. More details about the math behind the integration is provided in Section 4.3.

If the ray being sent from a surfel hits a surface, calculating the outgoing light is done in the same way that the surface would be shaded when doing the rendering,

by multiplying the albedo color of the surface with all the direct and indirect light. The fact that the indirect light is also sampled creates a feedback loop, effectively emulating the calculation of infinite bounce lighting. If the ray instead misses all surfaces, then the assumption is that the ray hit the sky, and a sky texture is sampled and used as the outgoing light.

The calculated outgoing light from the raycast then has to be integrated with the existing irradiance stored in the surfel. The authors use a modified exponential moving average estimator to keep track of the average result. It works by keeping track of a short-term average, as well as a long-term average. The speed at which the long-term average changes with each new result depends on how large the difference between the short-term average, and the long-term average is. A benefit of this approach is that it is able to both quickly react to changes in the lighting, and also eventually converge to a stable solution if the lighting is stable.

Rather than trying to integrate the irradiance all at once, which would require huge amount of rays cast from each surfel every frame, GIBS integrates over multiple frames by only casting one ray from each surfel on average. This has the drawback that indirect light calculations could lag behind the current lighting situation, but it is a necessary trade-off to make as the alternative would be far too costly in terms of computations.

The directions that rays are cast in can make a large difference in how quickly the irradiance at a surfel location converges to a stable irradiance. In cases where a majority of irradiance is coming from a limited set of directions, a random direction is unlikely to coincide with one of the directions where most of the light is coming from. Continuously integrating irradiance where a majority of the samples do not contribute to the irradiance, and very few of the samples contribute a lot, causes spikes and noise in the irradiance on the off chance that the right direction was chosen and a slow decline in irradiance otherwise.

To remedy this problem, GIBS uses ray guiding, a type of importance sampling, to cast more rays in directions where light is coming from. This is done by keeping track of the radiance from previous raycasts in a small 2D texture mapped to the hemisphere along the normal direction. When selecting a direction to cast a ray in, directions that have previously contributed a lot to the irradiance are more likely to be selected. The sample is then weighted based on the probability that the direction was chosen, so that each direction contributes equally to the average irradiance.

Another optimization that GIBS has to reduce the time it takes for irradiance to converge, is a system where more rays are cast from some surfels than others. Surfels that are currently visible, and surfels where the short-term average irradiance differs from the long-term average, are provided more rays. In order to maintain a consistent frame rate, a ray budget is used. This ray budget limits how many rays are cast each frame, and every surfel is then given a part of the budget weighted by visibility and the irradiance insecurity. On average, one ray is cast per surfel.

Figure 3.3: Rendered image of a mostly enclosed box where the only light is coming from a small opening in one of the walls. Illustrates a scene where calculating indirect light is hard due to the small area of directly lit surfaces. Image rendered with my implementation which does not use ray guiding of ray directions and subsequently has some noise and irregularities.

3.1.4 Light application

Applying the light to surfaces when rendering is done by accessing the grid cell that contains the world position of the pixel being rendered, looping through the cell's list of surfels, and accumulating a weighted irradiance. The irradiance is weighted based on the method by Lehtinen et al. [13], which uses a smoothstep distance attenuation function together with a Mahalanobis metric that takes the surfel's orientation compared to the pixel's orientation into account [3].

One problem with only considering the distance and orientation when accumulating light is that there could potentially be a wall or some other object blocking the path between the surfel and the pixel being rendered. This would potentially lead to an issue known as light leaking, where light travels through walls. The solution that GIBS uses is to keep track of the depth around each surfel and to then do depth testing when applying the light to make sure the pixel's position is visible from the surfel's position, thus preventing light leaks.

Depth is tracked with a depth texture, which is updated as a part of the irradiance integration. When a ray is cast and hits a surface inside the diameter of the surfel, depth is updated to include the distance from the surfel to the hit location of the ray. The depth is stored in a 4x4 texture mapped to the hemisphere of the surfel, where each texel stores both a moving average of the depth, and a moving average of depth squared, which allows the recreation of mean as well as the variance. Inspired by the work of Donnely et al. [5], Chebychev's inequality is then used with the mean and variance when applying irradiance to get a smooth depth test.

Figure 3.4: An example of light leaking through the walls to an otherwise dark room.

3.2 DDGI - Dynamic Diffuse Global Illumination

DDGI by Majercik et al. [14] [15] is another algorithm for solving dynamic diffuse global illumination in real-time. The algorithm is quite similar to GIBS in that irradiance caches are placed in a scene, and the irradiance stored in them is integrated dynamically during run-time by sampling the area around the caches through raytracing. There are also many differences, and this section will highlight some of those differences in order to give an overview of the DDGI algorithm.

3.2.1 Irradiance caches

One major difference between GIBS and DDGI is the way the irradiance caches are set up. Instead of surfels, DDGI uses a grid of irradiance probes as the basis for the algorithm, which means irradiance needs to be stored differently. The surfels used in GIBS represent a part of a specific surface with a normal direction, so tracking irradiance from only one direction is sufficient. The irradiance probes however represent a point in space, not attached to a specific object or surface, so the irradiance probes need to track incoming light from all directions.

The DDGI algorithm stores irradiance in small two-dimensional textures, one for each probe, where each texel represents a direction around a sphere. The texture size could be configured, but Majercik et al. [14] specify using 6x6 textures for irradiance which includes a 1 texel border to allow for bilinear interpolation, so essentially a 4x4 texture. These textures are stored in large texture atlases to ensure efficient memory usage.

Just like GIBS, irradiance calculation is done using a version of Monte Carlo integration over multiple frames. For DDGI, the rays are cast around the whole sphere as opposed to just a hemisphere. Multiple rays are cast from a probe each frame, and a weighted average of the resulting irradiance is calculated for each texel of the irradiance storage. The weight used is based on the dot product between the

direction of the ray, and the direction represented by the particular texel. These weighted averages from the raytracing are then added to the moving average of each texel through a linear interpolation between the current average and the new sample with a factor the authors call *hysteresis*. In order to allow the irradiance probes to converge faster, this hysteresis factor can be changed either on a probe level, or on a texel level based on recent magnitude changes detected, or on scene-dependent factors such as large geometry being moved.

DDGI Also includes a probe classification system which classifies probes based on if they are near static object, near dynamic objects, or not near anything. It also tracks if the classification recently changed. These classifications influence how many rays are being cast from each probe. It might completely stop casting rays from probes that are currently not near any objects, and for probes that recently changed classification it might cast more rays to allow the probe to converge faster.

3.2.2 Light application

As the probes are already placed on a grid with a known configuration, there is no need for any extra acceleration structures to find the irradiance caches, as is the case for GIBS. Instead, when querying the indirect light, the algorithm can access the eight probes constituting the corners of a 3D box around the query position, and compute a weighted average irradiance by sampling each probe's irradiance texture in the direction of the surface normal.

The weights used when calculating the irradiance are based on the distance from the query point to the probe, how the probe is positioned relative to the query position and normal, and if the query point is visible from the probe. Visibility is computed in the same way as for GIBS, by tracking mean distance and mean distance squared for multiple directions in a small texture that is updated during the raycasting step.

4

Implementation

This chapter describes an implementation of the GIBS algorithm and motivates the decisions made along the way. The implementation mostly follows the original, but certain elements have been removed or changed, either due to not being needed for the evaluation or for simplicity's sake.

The algorithm was implemented using DirectX 12 (DX12) on top of a fairly basic forward-shaded renderer, utilizing a depth prepass to avoid overdraw. Almost all the data used for the GIBS algorithm is stored on the GPU to avoid synchronization costs from communication between GPU and CPU. This means that any operations on surfels or the acceleration structure has to be implemented to run on the GPU and utilize parallelism in order to be efficient. All the data is allocated at the start of the programs runtime for a reasonable max number of surfels, as opposed to dynamically allocating during run-time.

4.1 Surfel representation

At its core, a surfel is represented by a position, a normal direction, and a radius. The implementation assumes a constant size of surfels as the scenes it is intended for are not large enough to require dynamically sized surfels based on range. With sizes being constant, there is no need for storing radius. Instead, the surfel radius is implicit. The normal and position still has to be stored somewhere, and on top of that there is also other data for each surfel used in the implementation which requires storage.

The implementation has storage for each surfel for the following data:

- Position and normal in both object-space and world-space, with object space being persistent, and world space being recalculated every frame.
- Object ID for the object the surfel is placed on. Zero is reserved to mean that the surfel has not yet been placed. Surfels that have not been placed are treated as inactive.
- The data used for representing stored irradiance as prescribed by Barré-Brisebois et al. [3], including RGB values for mean, short-term mean, and variance as well as floating point values for variance-based blend reduction and inconsistency. More detailed description is provided in Section 4.3.

To keep track of unplaced surfels, a stack is initialized containing all surfel IDs. When a new surfel is needed, the program decrements a stack counter and retrieves

the ID pointed to by the counter.

4.2 Sorting surfels

The first step of the algorithm is to sort and place all the surfels in the acceleration structure. This has to be done every frame, as surfels could have moved or new surfels could have been placed. Brinck et al. [4] use a special version of a uniform grid as an acceleration structure in order to support large scenes. Since this thesis aims to evaluate the algorithm for smaller and more controlled scenes, the implementation uses a basic grid that divides the scene in a fixed number of uniformly sized three-dimensional cells.

The placement of surfels in the acceleration structure is done in three steps. First, a count is made to see how many surfels should be placed in each grid tile. Secondly, a prefix sum is calculated on the counts to get an offset to the data used by each tile. Lastly, the ID of each surfel belonging to a tile is added to that tile's data. All of these parts are implemented as compute shaders.

For the purpose of storing and using surfels in the grid, a few data buffers are used. First, an array of integers is used to store surfel IDs belonging to a certain tile. Then, an array of two integers for each tile in the grid: *Offset*, an offset to the beginning of the tile's data in the array of IDs, and *OffsetToEnd*, which at the end of the sorting represents the offset to the end of the tile's data. Before that, it also represents the number of surfels that the tile should contain.

4.2.1 Counting surfels

Counting the surfels belonging to each tile is fairly trivial. First, *OffsetToEnd* is cleared to 0 for all tiles. Then, a compute shader is dispatched with one thread for each surfel that calculates which tile the surfel belongs to based on its position. This is also where the surfel's position and normal are converted to world space from object space. The transformation is done by retrieving the object ID and multiplying both position and normal with the transformation matrix associated with the object.

The implementation assumes that the grid starts at a position of zero in all dimensions, so tile coordinates are simply calculated by dividing the world-space position of the surfel with the tile size and then flooring the resulting position to a vector of integers representing the coordinates of the tile. The shader then does an atomic add operation of 1 to the *OffsetToEnd* variable of the tile, which at this point of the algorithm represents a running count of the number of surfels belonging to the tile.

An optimization which Brinck et al. [4] suggest is to also place each surfel in each neighboring tile, which sacrifices memory for less computation time later down the road when irradiance is applied. Therefore, the shader also increments the count of all neighboring tiles.

4.2.2 Prefix Sum

The reason for counting how many surfels are in each tile is done in order to subdivide the array of surfel IDs, so that each tile receives a slice of the array. With the number of surfels in each tile known, the offset to each tile's slice of the array can be calculated by computing the prefix sum of all the counts.

Computing the prefix sum of all entries in an array with a sequential algorithm is trivial. You can simply iterate through the array while keeping a running count of the total sum so far. As all operations on the acceleration structure are to be executed on the GPU however, a sequential approach would not work. Instead, a parallel algorithm for prefix sum is needed.

In the shading language used by DX12, there is a function called *WaveActivePrefixSum*. It computes a prefix sum efficiently, but it only does so for one wave at a time. Wave size depends on hardware, but there are usually either 32 or 64 threads in a wave. The number of tiles in a grid depends on how the grid is configured, but a grid that actually serves a purpose of meaningfully dividing the scene will always have more than 64 tiles.

The implementation makes use of *WaveActivePrefixSum* to recursively compute local prefix sums for larger and larger groups. The initial step computes prefix sum for groups the size of a wave, in the usual case 32, by simply calling *WaveActivePrefixSum* with every count value as the input. The next step then computes the offset to each group by calling *WaveActivePrefixSum* with the size of each group of 32 as the input, calculated by taking the local offset + count of the last item in the group. This can be repeated as many times as needed, growing the group for which prefix sum has been calculated for by a factor of 32 each iteration, until one group encompasses the entire range and a prefix sum for the entire range has been computed.

Once the prefix sum is calculated, the *Offset* and *OffsetToEnd* of each tile are both set to the calculated prefix sum for that tile.

4.2.3 Finalization

The final step of the sorting works in almost the same way as the first step, the counting. A compute shader is dispatched with one thread for each surfel, and the coordinates of the tile are calculated in the same way. At this point, the world-space position of the surfel is already stored from the counting step, so the matrix multiplication can be skipped.

Instead of increasing a count variable however, the finalization compute shader atomically increments the *OffsetToEnd* of all the relevant tiles (the tile that contains the surfel plus all neighboring tiles) while keeping track of the old value. The ID of the surfel is then written to the ID array using the old value of *OffsetToEnd* as the index. This ensures that each surfel in a tile receives a unique index, and that *OffsetToEnd* points to the last index plus one for each tile after the shader is executed,

which makes iterating through the IDs of surfels inside any tile easy.

4.3 Computing irradiance

Once the surfels have been sorted into the acceleration structure, the irradiance can be computed. Just like the original, the implementation calculates irradiance over multiple frames, but some optimizations for making irradiance converge faster have been excluded from the implementation due to time constraints.

The raytracing is implemented using the DirectX Raytracing (DXR) library built in to DX12. In order to know what a scene looks like, DXR requires two things, one or more Bottom Level Acceleration Structures (BLAS), and one Top Level Acceleration Structure (TLAS). A BLAS contains a representation of the triangles of a mesh. A TLAS represents an entire scene by connecting one or more BLAS with transformation matrices. In the implementation, a BLAS is generated for each mesh as soon as the mesh is loaded by sending the vertices and indices of the mesh to a DXR function for building acceleration structures. At program startup, and any time an object has moved in the scene, a TLAS is generated by a call to the same function, but passing in a transformation matrix and a reference to a BLAS for each object in the scene. The raytracing is then done through a Ray Generation shader.

In order to solve the irradiance integral (the integral from Equation 1.1, as seen below), a modified Monte Carlo integration scheme is used.

$$\int_{\boldsymbol{l}\in\boldsymbol{\Omega}} \boldsymbol{f}(\boldsymbol{l},\boldsymbol{v})\boldsymbol{L}_o(\boldsymbol{r}(\boldsymbol{p},\boldsymbol{l}),-\boldsymbol{l})(\boldsymbol{n}\cdot\boldsymbol{l})^+\boldsymbol{dl} \tag{4.1}$$

Monte Carlo integration is a method where an integral is estimated by taking N random samples, averaging the result divided by the probability density function for taking that each sample, and multiplying with the area being integrated over. In the case of the rendering equation, integration is done on all directions of a hemisphere. Sampling a random direction in a hemisphere can be seen as taking a random point on one half of the surface of a unit sphere. That means the area being integrated over is half of the surface area of a unit sphere, $\boldsymbol{2\pi}$. A Monte Carlo integration of the integral in the rendering equation can be seen in Equation 4.2.

$$\frac{\boldsymbol{2\pi}}{\boldsymbol{N}}\sum^{\boldsymbol{N}} \boldsymbol{f}(\boldsymbol{l},\boldsymbol{v})\boldsymbol{L}_o(\boldsymbol{r}(\boldsymbol{p},\boldsymbol{l}),-\boldsymbol{l})(\boldsymbol{n}\cdot\boldsymbol{l})^+ \tag{4.2}$$

$\boldsymbol{l}$ is a random direction sampled uniformly from the hemisphere. Sampling uniformly is not the best solution however, as more light generally is received from the normal direction than from the edges of the hemisphere. This can be seen from the equation in that the irradiance from a direction $\boldsymbol{l}$ is scaled by $(\boldsymbol{n}\cdot\boldsymbol{l})$. The implementation instead uses the concept of importance sampling, and samples in a cosine-distributed direction instead of a uniformly distributed direction. The irradiance equation can then be simplified to Equation 4.3.

$$\frac{\boldsymbol{\pi}}{\boldsymbol{N}}\sum^{\boldsymbol{N}} \boldsymbol{f}(\boldsymbol{l},\boldsymbol{v})\boldsymbol{L}_o(\boldsymbol{r}(\boldsymbol{p},\boldsymbol{l}),-\boldsymbol{l}) \tag{4.3}$$

The reason $(\boldsymbol{n} \cdot \boldsymbol{l})$ disappeared is that for Monte Carlo integration, the sample has to be divided by the probability density function of the distribution. For a cosine distribution, the probability density function is $\boldsymbol{cos(\theta)}$, where $\boldsymbol{\theta}$ is the angle between $\boldsymbol{n}$ and $\boldsymbol{l}$, which is equivalent to $(\boldsymbol{n} \cdot \boldsymbol{l})$ as long as both vectors are of unit length as they are in this case.

As the irradiance should respond to dynamic changes in the scene, the obvious approach of averaging by keeping track of the number of samples and dividing the total sum by that number will not work. After a while, new samples would stop mattering as there are too many previous samples. Instead, the implementation uses the same moving average calculation as Barré-Brisebois et al. [3], the code of which can be found in their paper. The way averages are tracked is by linearly interpolating between the old average and a new sample by some factor. Both a short-term average and a long-term average are kept. The change factor of the short-term average is constant, but the change factor of the long-term average depends on how different the short-term average is from the long-term average. The long-term average is then used as the irradiance in the surfel.

New samples are computed by generating a cosine-distributed direction, casting a ray in that direction from the surfel using DXR, and computing the outgoing light at the location where the ray stopped. If the ray missed everything, a sky texture is sampled as the outgoing light, and if the ray hit a surface, the direct light on the surface as well as the light from nearby surfels is treated as the outgoing light.

4.4 Rendering the scene

After the irradiance computation step, the scene is ready to be rendered. As the GIBS algorithm is based on the idea of placing surfels using screen-space information, it is not enough to only render the final image to be displayed on the screen. Textures containing information needed for the placement of surfels also have to be rendered.

The implementation renders five textures each frame using a pipeline with a vertex shader and a pixel shader, see Figure 4.1. First, since it is a forward renderer, one texture is used as the forward shaded final image. Then, two textures with four 16-bit floats each are used to store the position and normal in object-space. The alpha channel of the position texture is also used to represent surfel coverage. The fourth texture contains 16-bit unsigned integers that represent object ID of the object being rendered, or 0 for the sky. Finally, a depth texture is rendered in order to enable depth culling.

The forward shaded final image includes the indirect light calculated by the GIBS algorithm. The irradiance from surfels is applied by calculating which tile the pixel belongs to, iterating through all the surfels in the tile, and tracking a weighted average of the irradiance from those surfels.

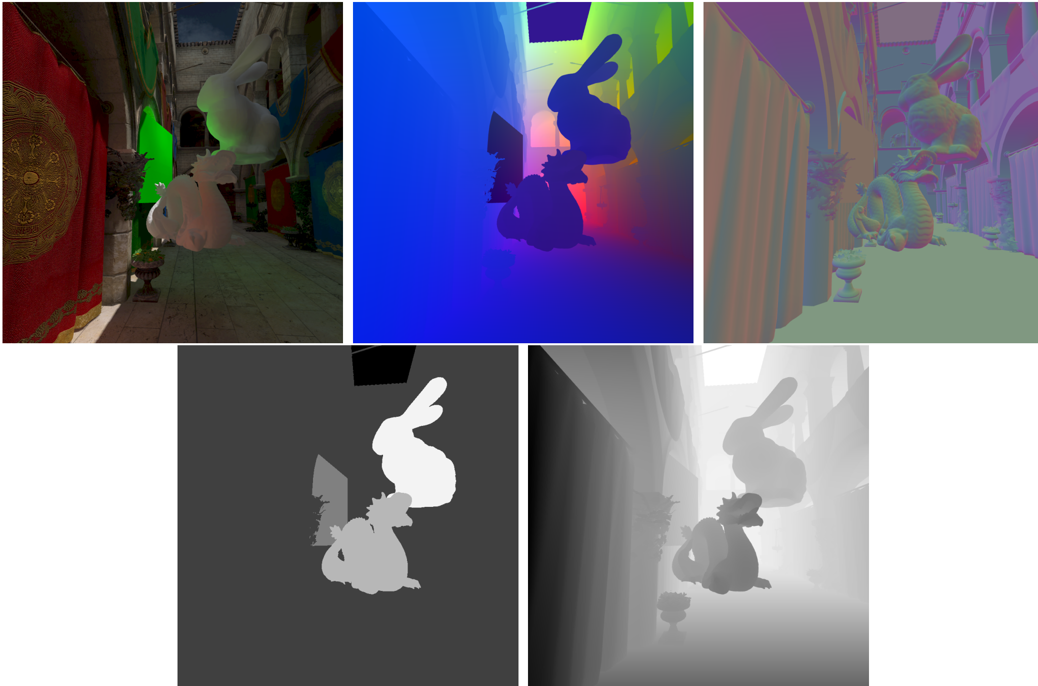

Figure 4.1: All the textures being rendered every frame. From left to right, top to bottom: the forward shaded final image, object-space position, object-space normal, transform ID and depth. Bunny and Dragon model from Morgan McGuire's Computer graphics archive [17].

The weight given to a surfel's irradiance is based on the distance to the pixel, as well as the difference between the surfel's normal vector and the normal vector of the surface being rendered. The implementation uses the weight algorithm presented by Lehtinen et al. [13]. The authors introduce a distance function which treats surfels as further away if the pixel does not lie on the tangential plane of the surfel. They also present a way of translating that distance smoothly to a weight which is 1 when the distance is 0, and 0 when the distance is the same as the radius of the surfel or more.

The weighted average of irradiance from the surfels is then just multiplied with the albedo color of the surface and added to the final image. As the renderer uses physically based materials, there are also some other modifications to the output based on the BRDF of the material, but the exact modifications are not detailed in this thesis. The total weight of all the surfels iterated upon is also rendered to the alpha channel of the position texture as the surfel coverage to be used when placing surfels.

4.5 Surfel placement

Placing surfels is done as the final step of the algorithm and utilizes the information from the rendered textures to guide placement. The placement of surfels is done by first dividing the screen in smaller tiles. Then, for each tile, the texel with the lowest surfel coverage is found. If the surfel coverage of that texel is lower than a threshold,

and a random check is passed, a surfel is placed at the location represented by the texel.

In the implementation, this is accomplished by first dispatching a compute shader with one thread for each texel, divided into thread groups of 16x16 threads. Finding the texel with the minimum coverage then starts by constructing an integer in each thread from an OR operation between the following two integers.

- The thread ID within the group, an integer between 0 and 255
- The surfel coverage (a float), multiplied by some large number, converted to an integer, and bitshifted 8 bits to the left.

The reason for the bitshift by 8 bits is so that the thread ID, which requires 8 bits to represent 0-255, can fit in the low bits while the coverage is in the higher bits of the resulting integer.

Each thread then calls *InterlockedMin* with the combined integer as the value and some group-shared memory as the destination. This is a function in the shading language of DX12 which performs a min operation atomically, replacing the destination value with the passed in value if the passed in value is lower than what is currently at the destination. As the coverage is in the high bits of the integer passed in, it will be the most influential part of the integer, and the group ID will only influence the result if the coverage is exactly the same. After every thread has performed the atomic min operation, the thread ID of the thread with the lowest coverage will be in the low 8 bits of the group-shared memory used as the destination.

Each thread then does three checks, and if all of them are passed a surfel is placed. First a check is made to see if the thread ID of the thread matches the thread ID of the thread with the minimum coverage. Then, a check is made to see if the coverage is less than some threshold. Lastly, it does a random check which has a certain percentage chance of passing.

For the implementation, a few different thresholds of coverage were tested, but a coverage threshold of one resulted in adequately spaced out surfels. Using coverage threshold of more than one resulted in two or more surfels being placed in the same exact place sometimes. A threshold of less than one does not guarantee all pixels to be fully covered by surfels.

The reason for the random check is as explained in Section 3.1.1.1, to minimize overlapping surfels. Too low of a chance and the placement of surfels will be too slow and visible, see Figure 4.2. Too high of a chance and surfels are more likely to be placed on top of each other. Brinck et al. [4] use a percentage chance that varies depending on the projected size of the surfel to be placed. The reason for this is that the overlapping is only an issue when placing surfels that are near the camera, meaning they would have a large projected size. For the implementation, adequate results for the purpose of the thesis were had with a fixed chance of 8%, no matter the projected size. The actual surfel placement is done by first decrementing the surfel stack counter, mentioned in Section 4.1, and retrieving the ID of the surfel

Figure 4.2: What the scene looks like when surfels haven't covered the entirety of the screen yet. If the surfel placement is too slow this will be visible.

Figure 4.3: The scene with different percentage chance of placing a surfel. From left to right: 8%, 15% and 25%. Higher percentage should lead to more overlapping surfels.

pointed to by the counter. Once the surfel ID is retrieved, the position, normal, and object ID are sampled from the respective texture and placed in the surfel data storage indexed by the surfel ID. The inconsistency value of that surfel is also set to one to make sure that the newly placed surfel quickly has its irradiance computed.

5

Results

In order to evaluate if it is possible to get a similar visual quality for diffuse indirect light as with the DDGI approach while being more restrictive with raycasting, a few tests were conducted. Essentially, the first set of tests aim to see how visual quality for both of the approaches varies with different ray budgets. Then the performance was tested for configurations of similar visual quality.

The implementation described in this thesis is evaluated against the RTXGI sample application [6], an implementation of DDGI. All the tests were executed on an Nvidia GTX 1660 Super at a resolution of 1920x1080 pixels. The scene used for all tests is Crytek Sponza, downloaded from a collection of sample models maintained by The Khronos Group [10]. The only direct light in the scene is a directional light.

5.1 Visual quality

The idea behind the first test is to keep ray budget as a constant between RTXGI and the implementation and compare the resulting visual quality. In order to get some varied images from the one scene, three different configurations of camera position and light direction were tested, see Figure 5.1. Three different ray budgets

Figure 5.1: Ground truth images generated by a path tracer included in the RTXGI sample application. From left to right: Hallway, Overview, Lion

were tested, 50000, 100000 and 200000 rays per frame. To enforce this budget in the RTXGI application, a minimally viable grid size was found through experimentation, and a rays per probe variable was set accordingly. The minimal grid size found which still had acceptable results was 20x10x10, so 2000 probes. With 2000 probes, each probe is set to cast 25, 50, or 100 rays per frame depending on the test. For the implementation of GIBS, it was found that around 50000 surfels is enough to cover the entirety of the Sponza scene, so it seemed reasonable to cast 1, 2 and 4 rays per surfel per frame for the tests, as that would correspond to computing indirect light for the entire scene just like DDGI is doing.

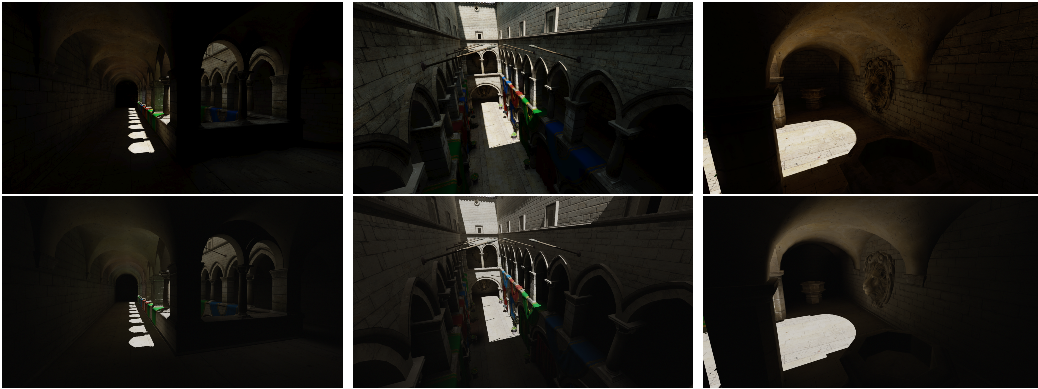

Figure 5.2: Visual quality for a ray budget of 50000. Top shows the implementation described in the thesis, bottom shows RTXGI sample application

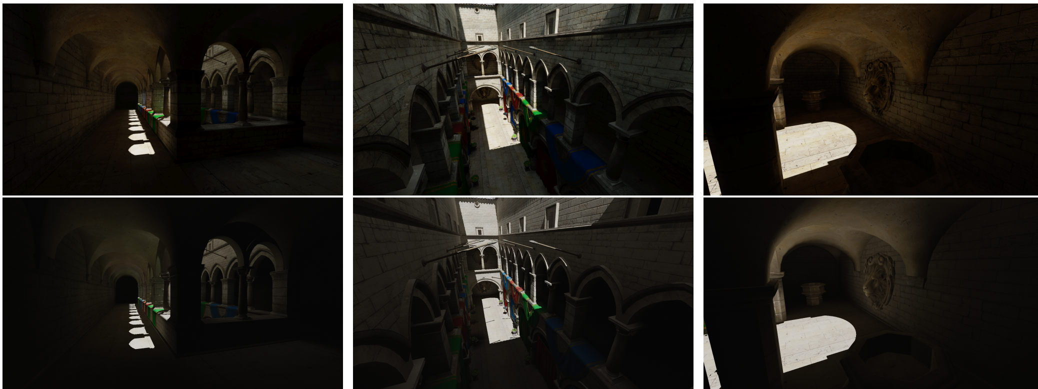

Figure 5.3: Visual quality for a ray budget of 100000. Top shows the implementation described in the thesis, bottom shows RTXGI sample application

It is not too apparent from still images, but 25 rays per probe is just not enough for DDGI as there was a lot of flickering. A ray budget of 100000 or 200000 does not seem to make too much of a difference when it comes to the visual quality, and there was no flickering for either implementation. Both implementations also seem to mostly match in terms of the indirect light calculations. Most of the differences between them come from being part of different renderers with different tonemapping.

Neither solution looks exactly like the ground truth, but one notable area where the implementation described in the thesis looks more like the ground truth is on the pillars in the Hallway configuration. Both ground truth and the implementation capture the red, blue, and green indirect light from the sunlight bouncing on the banners. The reason this indirect light does not show up for the DDGI approach is that no probe happens to be placed near the pillar's surface. This problem could potentially be hidden by making the grid denser or moving the probes, but the problem is inherent to separating irradiance caches from surfaces. Another artifact visible in the DDGI images is a darkened area around one of the plants on the left side of the Overview configuration. This is caused by one of the light probes being placed

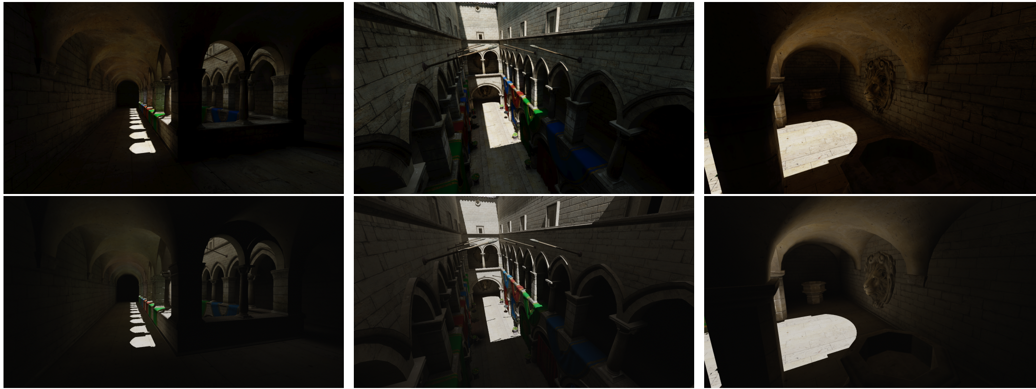

Figure 5.4: Visual quality for a ray budget of 200000. Top shows the implementation described in the thesis, bottom shows RTXGI sample application

inside the plant, and could potentially be avoided by configuring the grid differently.

Other than these two artifacts, most of the differences between these different configurations are in how fast the light converges and how it looks as it is converging, not necessarily how it looks when it has converged which this test covers. Both of those things are hard to convey in image form, but for the tests an attempt was made to keep the approaches being compared fairly similar in converge time for all ray budgets.

5.2 Performance

The next test is about keeping visual quality a constant and measuring computation time. From the previous section it was discovered that 25 rays per probe was not good enough in terms of visual quality, so that configuration will be excluded from this test. The other two comparisons with ray budgets of 100000 and 200000 were compared again, but now in terms of performance. The tests were only conducted for the Overview configuration as that turned out to be the heaviest one performance wise. The tests were conducted by starting each application through Nsight, waiting a few seconds for the lighting to converge, and then doing a GPU trace in Nsight. The case where the camera has not yet moved is the perfect case for the

Time elapsed (ms)				
	DDGI 50 rays/probe	Thesis impl. 2 rays/surfel	DDGI 100 rays/probe	Thesis impl. 4 rays/surfel
Raycasting	2.4	0.32	4.450	0.58
Application	1.750	4.28	1.75	4.46
Misc	0.25	0.25	0.24	0.25
Total	4.4	4.85	6.44	5.29

implementation as that means the minimal amount of surfels has been placed. Another test was conducted to instead focus on the worst case, where surfels have been

placed on every surface. This was done by flying the camera around the entirety of the scene before taking performance metrics.

Time elapsed (ms)	
	Thesis impl. 4 rays/surfel, moved camera
Raycasting	3.1
Application	12.2
Misc	0.25
Total	15.55

6 Discussion

A complete, physically accurate simulation of photons in real-time is extremely far out, so for now real-time global illumination is all about making trade-offs. This thesis focuses primarily on one such trade-off, to compute only what is needed at any given time, or to compute more than necessary right now in case it is needed later.

The tests conducted to evaluate the solution are not perfect. One of the differences between both approaches is the way a dynamic object is lit by indirect light as it is moving through the scene, but that is hard to illustrate with just pictures. For the DDGI approach, dynamic objects should be able to receive the proper light no matter where it is placed in the scene. For the surfel-based approach however, the irradiance caches move with the dynamic objects. If they are quickly moved into a location with different illumination, some time is needed for the surfels on the object to update. In my testing this did not seem to be a very noticeable issue, the light converges very fast thanks to the mean estimator used by GIBS as well as the idea of sending more rays from a surfel if the irradiance is uncertain, which it will be if moved to a new area.

Another potential drawback inherent to the GIBS approach, is the fact that the surfels do not exist until after they are needed the first time. They are placed as the final step on surfaces being rendered, meaning that the very first frame a surfel is needed it can not be there. Even a few frames after, it might not be there due to the probability. The effect of this can be seen in Figure 4.2. This seems like something that can be hidden quite well through proper configuration of the probability to place surfel as well as allocating enough rays to new surfels so that they converge fast.

While I was able to lower the number of raycasts needed to calculate indirect light through implementing the use of surfels from GIBS, my implementation suffered from other issues instead. The major issue is how indirect light was queried when rendering the scene, which ended up being a major performance bottleneck as evident by the performance metrics in Section 5.2. The cost was manageable when the camera had not yet moved prior to measuring, but that is obviously not very useful. The issue stems from the fact that my implementation has to iterate through every surfel in the grid of whatever pixel is being rendered, which for complex surfaces could be very many. As can be seen from Figure 6.1, the act of flying around the entirety of the scene drastically increases the number of surfels present in each

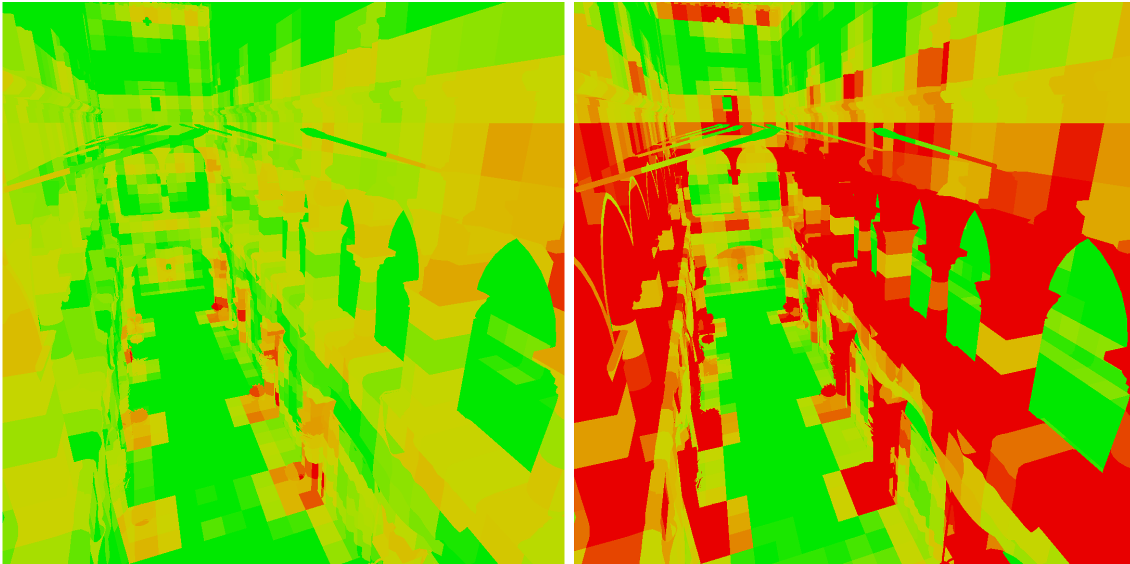

Figure 6.1: The grid complexity before and after flying around the scene. Flying around places surfels from more angles as well as on backsides of for example the pillars. Bright green grid tiles contain 20 or fewer surfels, bright red contain 70 or more

grid tile. The perhaps naïve approach of iterating through all these for every pixel is very costly. Brinck et al. [4] prescribe iterating through $\boldsymbol{N}$ surfels in each tile, which seems to imply not all surfels are iterated. Attempts of mirroring that approach resulted in a lot of flickering as the acceleration structure sort is inherently unstable, and only considering $\boldsymbol{N}$ surfels means that different surfels will impact the light for every frame. Another solution for this could be to treat the surfels as light sources and apply the indirect light in a deferred manner as described by Stachowiak [23].

6.1 Ethical considerations

With advancements in computer graphics, the difference between reality and for example a game becomes smaller. This could have detrimental effects on issues such as game addiction where people spend an unhealthy amount of time in front of a computer screen. Too much time in front of a computer screen can be bad for peoples eyes as well as body with increased risk of blood clots through sitting down for long periods of time.

More realistic lighting and graphics when it comes to Virtual Reality could lead to people neglecting real life to spend most of their time in a virtual world instead.

On the other hand, the immersion that comes with more realistic graphics in games can lead to them being more enjoyable to play, as long as the playing time is kept at reasonable levels.

6.2 Conclusions

In this thesis, an approach to diffuse global illumination that primarily focuses on computing only what is necessary was implemented. This implementation was then evaluated against an approach which computes a fair bit extra in case it is needed.

The result shows that a surfel-based approach is capable of producing images of similar quality of indirect light using fewer raycasts, and that the inherent drawback of the approach with dynamic objects entering new illumination situations can be mostly hidden with proper configurations.

While the implementation of the surfel-based approach decreased the cost of raytracing, the cost of querying the indirect light was increased a lot more. The problem could potentially be solved through for example shading the surfels in a deferred manner.

Bibliography

[1] Michael Abrash. Quake's lighting model: surface caching. https://www.bluesnews.com/abrash/chap68.shtml, 2000.

[2] Tomas Akenine-Mller, Eric Haines, and Naty Hoffman. *Real-Time Rendering, Fourth Edition*. A. K. Peters, Ltd., USA, 4th edition, 2018.

[3] Colin Barré-Brisebois, Henrik Halen, Graham Wihlidal, Andrew Lauritzen, Jasper Bekkers, Tomasz Stachowiak, and Johan Andersson. Hybrid rendering for real-time ray tracing. *Ray Tracing Gems Chapter 25*, 2019.

[4] Andreas Brinck, Xiangshun Bei, Henrik Halén, and Kyle Hayward. Global illumination based on surfels. https://advances.realtimerendering.com/s2021/SIGGRAPH%20Advances%202021%20-%20Surfel%20GI.pdf, 2021.

[5] William Donnelly and Andrew Lauritzen. Variance shadow maps. volume 2006, pages 161–165, 01 2006.

[6] NVIDIA GameWorks. Rtx global illumination. https://github.com/NVIDIAGameWorks/RTXGI, 2020.

[7] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modeling the interaction of light between diffuse surfaces. *SIGGRAPH Comput. Graph.*, 18(3):213–222, jan 1984.

[8] Donald Greenberg, Michael Cohen, and Kenneth Torrance. Radiosity: A method for computing global illumination. *The Visual Computer*, 2:291–297, 09 1986.

[9] Gene Greger, Peter Shirley, P.M. Hubbard, and D.P. Greenberg. The irradiance volume. *Computer Graphics and Applications, IEEE*, 18:32–43, 04 1998.

[10] Khronos Group. Khronos group gltf sample models, 2018. https://github.com/KhronosGroup/glTF-Sample-Models/tree/master/2.0/Sponza.

[11] Johannes Jendersie, David Kuri, and Thorsten Grosch. Real-time global illumination using precomputed illuminance composition with chrominance compression. *Journal of Computer Graphics Techniques (JCGT)*, 5(4):8–35, December 2016.

[12] James T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, aug 1986.

[13] Jaakko Lehtinen, Matthias Zwicker, Emmanuel Turquin, Janne Kontkanen, Frédo Durand, François X. Sillion, and Timo Aila. A meshless hierarchical representation for light transport. In *ACM SIGGRAPH 2008 Papers*, SIGGRAPH '08, New York, NY, USA, 2008. Association for Computing Machinery.

[14] Zander Majercik, Jean-Philippe Guertin, Derek Nowrouzezahrai, and Morgan McGuire. Dynamic diffuse global illumination with ray-traced irradiance fields. *Journal of Computer Graphics Techniques (JCGT)*, 8(2):1–30, June 2019.

[15] Zander Majercik, Adam Marrs, Josef Spjut, and Morgan McGuire. Scaling probe-based real-time dynamic global illumination for production. *Journal of Computer Graphics Techniques (JCGT)*, 10(2):1–29, May 2021.

[16] Sam Martin. Enlighten real-time radiosity. 08 2011.

[17] Morgan McGuire. Computer graphics archive, July 2017. `https://casual-effects.com/data`.

[18] Morgan McGuire, Michael Mara, Derek Nowrouzezahrai, and David Luebke. Real-time global illumination using precomputed light field probes. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, page 11, February 2017. I3D 2017.

[19] Jason Mitchell, Gary McTaggart, and Chris Green. Shading in valve's source engine. In *ACM SIGGRAPH 2006 Courses*, SIGGRAPH '06, page 129–142, New York, NY, USA, 2006. Association for Computing Machinery.

[20] Tatarchuk Natalya. Irradiance volumes for games. *GDC*, 2005.

[21] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. Surfels: Surface elements as rendering primitives. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, page 335–342, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[22] Brennan Rusnell. Radiosity for computer graphics. 01 2007.

[23] Tomasz Stachowiak. Stochastic all the things: Raytracing in hybrid real-time rendering. `https://www.ea.com/seed/news/seed-dd18-presentation-slides-raytracing`, 2018.